Laborator 07: Parcurgerea grafurilor. Aplicații (1/2)

Obiective laborator

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

Importanţă – aplicaţii practice

Grafurile sunt utile pentru a modela diverse probleme și au numeroase aplicații practice:

- Rețele de calculatoare (ex: stabilirea unei topologii fără bucle / arbore de acoperire)
- Pagini Web (ex. algoritmi de căutare - Google PageRank)
- Rețele sociale (ex. sugestii de prietenie pe Facebook)
- Hărți cu drumuri (ex. drum minim între două localități)
- Modelare grafică (ex. arbori de parționare)
- Rețele de transport (ex. flux)

Grafuri

Puteți consulta capitolul "Elementary Graph Algorithms" din "Introduction to Algorithms" [0] pentru mai multe definiții formale. Această secțiune sumarizează principalele notații folosite în laboratoarele de PA.

Definiții

> Un **graf** G se definește ca fiind o pereche (V, E), unde **V = {node / un nod oarecare din graf}**, iar **E = {(x, y) / (x, y) muchie in graf}**.
>
> Un graf este **neorientat** dacă relațiile dintre noduri sunt **bidirecționale**: oricare ar fi (x, y) în E, există și (y, x) în E. Relațiile se numesc **muchii**.
>
> Un graf este **orientat** dacă relațiile dintre noduri sunt **unidirecționale**: (x, y) este în E nu implică neapărat (y, x) în E. Relațiile se numesc **arce**.
>
> O **componentă conexă (CC)** 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. Pentru grafuri orientate, o componentă conexă se numește **componentă tare conexă (CTC)**.
>
> Un graf **aciclic** este un graf (orientat/neorientat) care nu conține cicluri.

Reprezentare

Problemele care se modelează folosind grafuri, de obicei, presupun explorarea spațiului. O parcurgere explorează fiecare nod al grafului, exact o singură dată, pornind de la un nod ales, numit în continuare nod sursă (EN: **source**). Modul de reprezentare ar grafului, poate influența performanța unei parcurgeri/unui algoritm.

Un graf poate fi modelat în mai multe moduri (folosind mai multe notații):

- printr-o pereche de mulțimi $G = (V, E)$
  - V = {v / v este un nod în graf} = mulțimea nodurile grafului (EN: nodes / vertices)

        - E = {e / $e = (x, y)$ este o muchie în graf între nodurile x și y} = mulțimea muchiile/arcelor (EN: edges), fiecare muchie stabilind o relație de vecinătate între doua noduri.

- printr-o pereche $G = (nodes, a)$
    - $nodes$ = {node / node este un nod în graf}
    - $a[x][y] = 0/1$
        - **1** = există muchia/arcul (x, y)
        - **0** = **NU** există muchia/arcul (x, y)

- printr-o pereche de mulțimi $G = (nodes, adj)$
    - $nodes$ = {node / node este un nod în graf}
    - adj = {$adj[node]$ / unde $adj[node]$ este lista de adiacență a lui node} = reprezentarea grafului ca liste de adiacențe
        - $adj[node] = \ldots, neigh, \ldots \Rightarrow$ există muchie/arc (node, neigh)

Reprezentarea în memorie a grafurilor se face, de obicei, cu **liste de adiacență**. Se pot folosi însă și alte structuri de date, care vor fi introduse pe parcurs.

Cele mai uzuale notații din laboratoarele de grafuri sunt descrise în Precizări laboratoare 07-12 [https://ocw.cs.pub.ro/courses/pa/skel_graph] (ex. n, m, adj, adj_trans, (x, y), etc).

Colorare

Algoritmii de parcugere se pot folosi de o colorare a nodurilor:

- **white** (alb) = nod care nu a fost încă vizitat (nu este în coadă)
- **gray** (gri) = nod care este în curs de vizitare (a fost adăugat în coadă)
- **black** (negru) = nod care a fost complet vizitat (node scos din coadă și pentru care s-a vizitat tot subarborele)

Algoritmi de parcurgere

Problemă: Să se parcurgă un graf dat. Fiecare nod se parcuge (exact) o singură dată. Algoritmi:

- **BFS**
- **DFS**

BFS - Parcurgerea în lățime

Parcurgerea în lățime **(Breadth-first Search - BFS)** este un algoritm de căutare în graf, în care, atunci când se ajunge într-un nod oarecare **node**, nevizitat, se vizitează toate nodurile nevizitate adiacente lui (notate pe rand cu **neigh**), apoi toate vârfurile nevizitate adiacente vârfurilor adiacente lui node, etc.

Atenție! BFS depinde de nodul de start **source**. Plecând din acest nod, se vor vizita toate nodurile accesibile. De exemplu, într-un graf neorientat, aceste noduri accesibile formează o componentă conexă; în urma aplicării algoritmului BFS asupra fiecărei componente conexe a grafului, se obține un arbore de acoperire a întregului graf (prin eliminarea muchiilor pe care nu le folosim la parcurgere). Pentru a putea reconstitui acest arbore, se păstrează pentru fiecare nod dat identitatea părintelui său. În cazul în care nu exista o funcție de cost asociată muchiilor, BFS va determina și drumurile minime de la rădăcină la oricare nod.

Pentru implementarea BFS se folosește o coadă.

Algoritm

BFS

```
// do a BFS traversal from source
//
// source    = the source for the BFS traversal
// nodes     = list of all nodes from G
// adj[node] = the adjacency list of node
//             example: adj[node] = {..., neigh, ...} => edge (node, neigh)
BFS(source, G=(nodes, adj)) {
    // STEP 0: initialize results
    // d[node] = distance from source to node
    // p[node] = parent of node in the BFS traversal started from source
    // [optional] color[node] = white/gray/black
    //                              * white = not yet visited
    //                              * gray  = visit in progress
    //                              * black = visited
    foreach (node in nodes) {
        d[node] = +oo;                          // distance not yet computed
        p(node) = null;                         // parent not yet found
        // [optional] color[node] = white;
    }

    // STEP 1: initialize a queue
    q = {}

    // STEP 2: add the source(s) into q
    d[source] = 0;                              // distance from source to source
    p[source] = null;                           // the source never has a parent (because it's the root of the traversal)
    q.push(source);
    // [optional] color[source] = gray;

    // STEP 3: start traversal using the node(s) from q
    while (!q.empty()) {                        // while still have nodes to explore
        // STEP 3.1: extract the next node from queue
        node = q.pop();

        // [optional] STEP 3.2: print/use the node

        // STEP 3.3: expand/visit the node
        foreach (neigh in adj[node]) {          // for each neighbour
            if (d[node] + 1 < d[neigh]) {       // a smaller distance <=> color[neigh] == white
                d[neigh] = d[node] + 1;         // update distance
                p[neigh] = node;                // save parent
                q.push(neigh);                  // add neigh to the queue of nodes to be visited
                // [optional] color[neigh] = gray;
            }
        }

       // [optional] color[node] = black;
    }
}
```

Liniile cu **[optional]** se referă la logica de colorare menționată anterior, care se poate omite (dacă nu se dorește acest rezultat).

Complexitate

- cu liste de adiacență: $O(n + m)$ sau $O(|V| + |E|)$
- cu matrice de adiacență: $O(n^2)$ sau $O(|V|^2)$

DFS - Parcurgerea în adâncime

Parcurgerea în adâncime **(Depth-First Search - DFS)** pornește de la un nod dat (**node**), care este marcat ca fiind în curs de procesare. Se alege primul vecin nevizitat al acestui nod (**neigh**), se marchează și acesta ca fiind în curs de procesare, apoi și pentru acest vecin se caută primul vecin nevizitat, și așa mai departe. În momentul în care nodul curent nu mai are vecini nevizitati, se marchează că fiind deja procesat și se revine la nodul anterior. Pentru acest nod se caută primul vecin nevizitat. Algoritmul se repetă până când toate nodurile grafului au fost procesate.

În urma aplicării algoritmului DFS asupra fiecărei componente conexe a grafului, se obține pentru fiecare dintre acestea câte un arbore de acoperire (prin eliminarea muchiilor pe care nu le folosim la parcurgere). Pentru a putea reconstitui acest arbore, păstram pentru fiecare nod dat identitatea părintelui sau.

Pentru fiecare nod se vor reține:

- $start[node]$ = timestamp-ul / timpul descoperirii
- $finish[node]$ = timestamp-ul / timpul finalizării
- $p[node]$ = părintele din parcugerea DFS a lui node

Spre deosebire de BFS, pentru implementarea DFS se folosește o stivă (abordare **LIFO** în loc de **FIFO**). În practică, stiva nu va fi reținută explicit - ci ne vom baza pe recursivitate.

Algoritm

| DFS

```
// do a DFS traversal from all nodes
//
// nodes     = list of all nodes from G
// adj[node] = the adjacency list of node
//              example: adj[node] = {..., neigh, ...} => edge (node, neigh)
//
DFS(G=(nodes, adj)) {
    // STEP 0: initialize results
    // p[node]     = parent of node in the BFS traversal started from source
    // start[node] = the timestamp (the order) when we started visiting the node subtree
    // finish[node] = the timestamp (the order) when we finished visiting the node subtree
    // [optional] color[node] = white/gray/black
    //                              * white = not yet visited
    //                              * gray  = visit in progress
    //                              * black = visited
    foreach (node in nodes) {
        p[node] = null;                             // parent not yet found
        // [optional] color[node] = white;
    }

    timestamp = 0;                                  // the first timestamp before the DFS traversal

    foreach (node in nodes) {
        if (p[node] == null) {                      // or [optional] color[node] == white
            DFS_RECURSIVE(node, G, p, timestamp)
        }
    }
}

DFS_RECURSIVE(node, G=(node, adj), p, ref timestamp) {
    start[node] = ++timestamp;                      // start visiting its subtree
    // [optional] color[node] = gray;

    for (neigh in adj[node]) {                      // for each neighbour
        if (p[neigh] == null) {                     // or [optional] color[neigh] = white;
            p[neigh] = node;                        // save parent
            DFS_RECURSIVE(neigh, G, p, timestamp);  // continue traversal
        }
    }

    finish[node] = ++timestamp;                     // finish visiting its subtree
    // [optional] color[node] = black;
}
```

Liniile cu **[optional]** se referă la logica de colorare menționată anterior, care se poate omite (dacă nu se dorește acest rezultat).

Complexitate

- cu liste de adiacență: $O(n + m)$ sau $O(|V| + |E|)$

- cu matrice de adiacență: $O(n^2)$ sau $O(|V|^2)$

Tipuri de muchii/arce în parcurgerea DFS

> **Arborele de parcurgere DFS** cuprinde toate nodurile din graf împreună cu muchiile/arcele pe vizitate de DFS. Dacă graful nu este conex / tare conex, se obțin mai mulți arbori care formează o **pădure de arbori DFS**.

Arborele DFS nu este unic - depinde de ordinea în care nodurile sunt stocate în listele de adiacență.

Exemplu:

- dacă în lista lui 1 avem nodurile 2 și 3, atunci când se va vizita 1, prima oară se încearcă vizitarea lui 2, apoi a lui 3.
- dacă în lista lui 1 avem nodurile 3 și 2, atunci când se va vizita 1, prima oară se încearcă vizitarea lui 3, apoi a lui 2.

Putem folosi o parcurgere DFS pentru a clasifica tipurile de muchii (toate muchiile din graf) relativ la arborele DFS curent.

> **tree-edge** (**T**) / muchie de arbore = muchie **(x, y)** care conectează un nod x de copilul său y din arbore.
>
> **back-edge** (**B**) / muchie înapoi = muchie **(x, y)** care conectează un nod x de un strămoș y (ambele noduri sunt în curs de vizitare).

Pentru **graf orientat**, mai există încă 2 tipuri de muchii (arce):

> **forward-edge** (**F**) / muchie înainte /muchie de înaintare = muchie **(x, y**) care nu este **tree-edge** și care conectează un nod **x** de un descendent **y** .

În graful neorientat aceasta nu are sens. Într-un graf neorientat, **(x, y)** și **(y, x)** reprezintă același lucru. Dacă x ar fi strămoș a lui y (x nu este părintele lui y, căci altfel **(x, y)** ar fi **tree-edge**), mai întâi se va încerca din y să se ajungă în x (moment în care spunem că muchia **(y, x)** este **back-edge**), ulterior la revenirea din recursivitate se va încerca din x să se ajungă în y (moment în care ar trebui să spunem că muchia **(x, y)** este **forward-edge**). Deoarece există o singură muchie **(x, y)** sau **(y, x)**, nu putem pune 2 categorii, așa că rămâne prima categorie găsită: **back-edge**.

Într-un graf **orientat** muchiile **(x, y)** și **(y, x)** sunt diferite, deci pot avea tipuri diferite!

> **cross-edge** (**C**) / muchie de traversare = muchie **(x, y)** care conectează un nod x de un nod y din alt subarbore.

În graful neorientat aceasta nu are sens. Într-un graf neorientat, dacă **(x, y)** ar fi **cross-edge**, înseamnă că aceasta conectează pe x de un nod y din alt subarbore (care a fost deja vizitat!). Dacă nodul y a fost deja vizitat, iar graful este neorientat, atunci s-ar fi **înaintat** pe muchia **(y, x)** din momentul vizitării nodului y. Acest lucru ar face muchia **(y, x)** un **tree-edge**. Prin urmare obținem o contradicție, deci nu putem avea cross-edge într-un graf neorientat.

Într-un graf **orientat** muchiile **(x, y)** și **(y, x)** sunt diferite, deci pot avea tipuri diferite!

Mai sus s-a catalogat o muchie **(x, y)** atunci când dintr-un nod în curs de vizitare x (culoare **GRAY**) se încearcă trecerea într-un nod y.

În funcție de culoare lui y, observăm care este tipul muchiei:

```
 * **(x, y)** -> **(GRAY, WHITE)** => **tree-edge**
 * **(x, y)** -> **(GRAY, GRAY)** => **back-edge**
 * **(x, y)** -> **(GRAY, BLACK)** => **forward-edge** sau **cross-edge**
```

Aplicații parcurgeri

- Componente Conexe
- Sortarea Topologică
- Componente Tare-Conexe
- Componente Biconexe

În acest laborator vom studia doar problema sortare topologică.

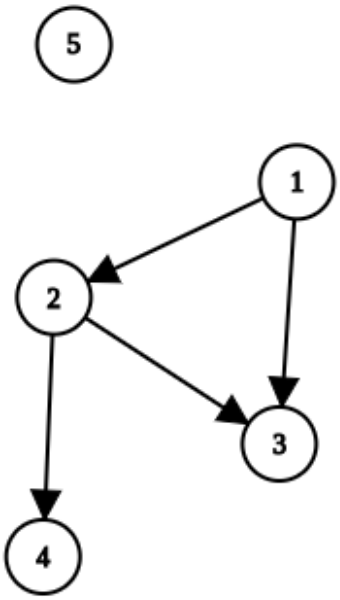
TopSort - Sortarea Topologică

Problemă

> O **sortare topologică** într-un **graf orientat aciclic** reprezintă o aranjare/permutare a nodurilor din graf care ține cont de arce.

Orientarea muchiilor corespunde unei relatii de ordine de la nodul sursa catre cel destinație: dacă (x, y) este un arc, x trebuie să apară înaintea lui y în înșiruire.

Daca graful ar fi ciclic, nu ar putea exista o astfel de insiruire (nu se poate stabili o ordine intre nodurile care alcatuiesc un ciclu).



În figura anterioară avem un graf cu:

- $n = 5$ $m = 4$
- $arce$: $(1, 2); (1, 3); (2, 3); (2, 4);$

Toate sortările topologice valide sunt:

- cele date de ordinea relativa a primelor 4 noduri: (1,2,3,4)
  - $topsort = [1, 2, 3, 4, 5]$
  - $topsort = [1, 2, 3, 5, 4]$
  - $topsort = [1, 2, 5, 3, 4]$
  - $topsort = [1, 5, 2, 3, 4]$

    - $topsort = [5, 1, 2, 3, 4]$
- cele date de ordinea relativa a primelor 4 noduri: (1,2,4,3)
    - $topsort = [1, 2, 4, 3, 5]$
    - $topsort = [1, 2, 4, 5, 3]$
    - $topsort = [1, 2, 5, 4, 3]$
    - $topsort = [1, 5, 2, 4, 3]$
    - $topsort = [5, 1, 2, 4, 3]$

Explicație pentru $topsort = [1, 2, 3, 4, 5]$:

- deoarece avem arcele $1 \rightarrow 3$ si $1 \rightarrow 2$, 1 trebuie să apara înainte lui 2 și 3
- deoarece avem arcul $2 \rightarrow 3$ si $2 \rightarrow 4$, 2 trebuie să apara înainte lui 3 și 4
- 5 nu depinde de nimeni, poate să apară oriunde

Algoritmi

Sunt doi algoritmi cunoscuti pentru sortarea topologică.

TopSort - DFS: sortare descrescătoare după timpul de finalizare

Algoritm TopSort cu DFS:

- se face o parcurgere DFS pentru determinarea timpilor de finalizare
- se sortează descrescător in functie de timpul de finalizare
- permutarea de noduri obținută este o sortare topologică

Optimizare: Pentru a evita sortarea nodurilor in functie de timpul de finalizare, se poate folosi o stiva ce retine aceste noduri in ordinea terminarii parcurgerii (sau un vector care la final este inversat).

Complexitate

- $T(n) = O(n + m)$

TopSort - BFS: algoritmul lui Kahn

Algoritm TopSort cu BFS:

- se initializeaza coada de la BFS cu toate nodurile din graf care au grad inten **0**
- se porneste parcurgerea BFS
    - la fiecare pas se vizitează un nod **node**
    - se șterg toate muchiile care pleacă din **node**: $(node, neigh)$
        - $neigh$ este adaugat in coada doar daca devine un nod cu grad intern **0**
- se verifica la finalul parcugerii daca mai sunt muchii ramase in graf
    - **daca** inca mai exista muchii nește rse, atunci graful conține cel puțin un ciclu - nu se poate determina o sortare topologică
    - **altfel**, ordinea in care s-au scos nodurile din coada reprezinta o sortare topologica

Optimizare: Pentru a evita ștergerea propriu-zisă a muchiilor din graf, se poate modifica gradul intern al fiecărui nod (care poate fi reținut într-un vector $in\_degree[node]$).

Complexitate

- $T(n) = O(n + m)$

Concluzie

Ambele variante au aceeasi complexitate.

- Algoritmul bazat pe DFS nu verifica daca graful este ciclic: presupune corectitudinea inputului. Este relativ mai simplu de implementat.
- Algoritmul bazat pe BFS se poate folosi pentru a detecta daca graful este aciclic; in caz afirmativ, gaseste o sortare topologica valida.

TLDR

- Cele mai uzuale moduri de reprezentare a unui graf sunt: liste de adiacentă și matrice de adiacentă.
- Cele doua moduri uzuale de parcurgere a unui graf sunt: **BFS** și **DFS**.
- O aplicație importantă a parcurgerilor este **Sortarea topologică** - o modalitate de aranjare a nodurilor în funcție de muchiile dintre ele. În functie de nodul de start al DFS, se pot obține sortări diferite, păstrând însă proprietatile generale ale sortarii topologice.

Exerciții

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

Prin citirea acestor precizari vă asigurați ca:

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

Scheletul de laborator se găsește pe pagina pa-lab::skel/lab07 [https://github.com/acs-pa/pa-lab/tree/main/skel/lab07].

Începând cu acest laborator, fiecare problemă are restricții concrete: dimensiuni pentru input și timp maxim de execuție. Pentru a vedea dacă o soluție (idee) intră în timp înainte de a o implementa, va trebui să îi calculați complexitatea și să aproximați timpul de execuție folosind tutorialul pa-lab::docs/Complexity [https://github.com/acs-pa/pa-lab/tree/main/docs/complexity.md].

BFS

Se dă un graf **neorientat** cu **n** noduri și **m** muchii. Se mai dă un nod special **source**, pe care îl vom numi sursa.

Se cere să se găsească **numărul minim de muchii** ce trebuie parcurse de la **source** la **toate** celelalte noduri.

Restricții și precizări:

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

Rezultatul se va returna sub forma unui vector **d** cu **n** elemente.

Convenție:

- **d[node]** = numărul minim de muchii ce trebuie parcurse de la **source** la nodul **node**
- **d[source] = 0**
- **d[node] = -1**, dacă nu se poate ajunge de la **source** la **node**

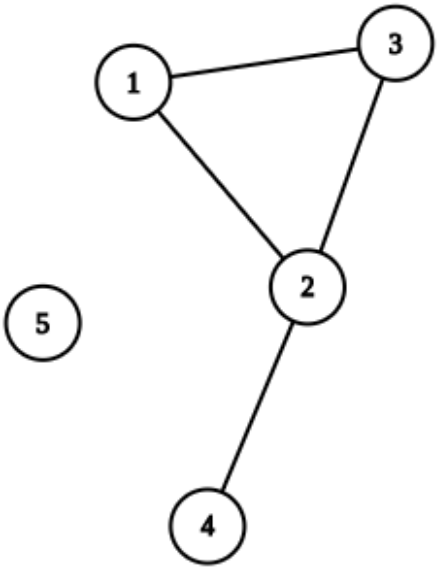
$n = 5\ m = 4\ source = 3$

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

Răspuns:

node	1	2	3	4	5
d	1	1	0	2	-1

Explicație: Graful dat este cel din figura urmatoare.



- **d[3] = 0** pentru că 1 este sursa
- **d[1] = d[2] = 1** pentru că există muchie directă de la 2 la fiecare nod
- **d[4] = 2** pentru că trebuie să parcurgem 2 muchii $(3 - 2 - 4)$
- **d[5] = -1** pentru că nu se poate ajunge de la 3 la 5

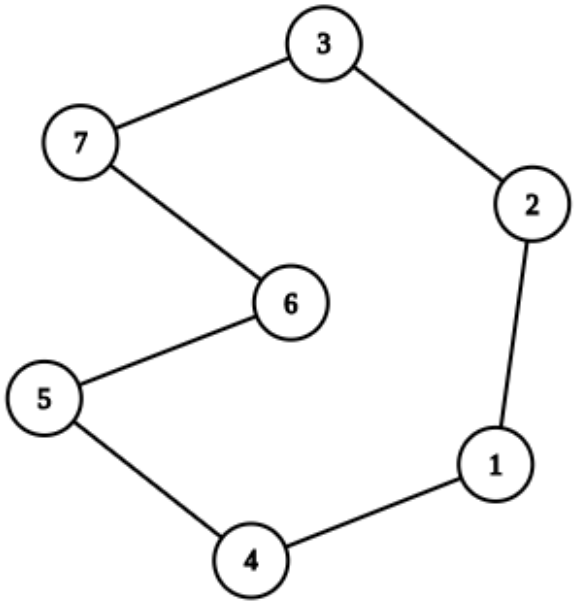
$n = 7\ m = 7\ source = 1$

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

Răspuns:

node	1	2	3	4	5	6	7
d	0	1	2	1	2	3	3

Explicație: Graful dat este cel din figura urmatoare.



- **d[1] = 0** pentru că 1 este sursa
- **d[2] = d[4] = 1** pentru că există muchie directă de la 2 la fiecare nod
- **d[3] = d[5] = 2** pentru că trebuie să parcurgem 2 muchii ($1-2-3$, $1-4-5$)
- **d[6] = d[7] = 3** pentru ca trebuie să parcurgem 3 muchii ($1-2-3-7$ sau $1-4-5-6$)

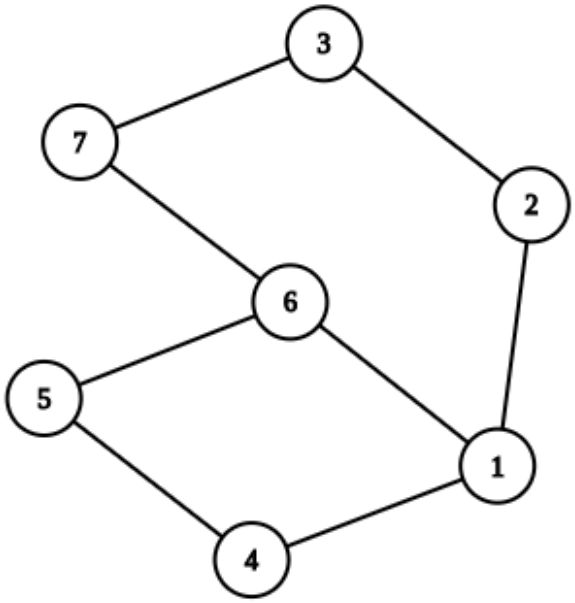
$n = 7$ $m = 8$ $source = 1$

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

Răspuns:

node	1	2	3	4	5	6	7
d	0	1	2	1	2	1	2

Explicație: Graful dat este cel din figura urmatoare.



- **d[1] = 0** pentru că 1 este sursa
- **d[2] = d[4] = d[6] = 1** pentru că există muchie directă de la 2 la fiecare nod
- **d[3] = d[5] = d[7] = 2** pentru că trebuie să parcurgem 2 muchii ($1 - 2 - 3$, $1 - 4 - 5$, $1 - 6 - 7$)

Topological Sort

Se dă un graf **orientat** aciclic cu **n** noduri și **m** arce. Se cere să se găsească **o sortare topologica** validă.

Restricții si precizari:

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

Rezultatul se va returna sub forma unui vector **topsort** cu **n** elemente.

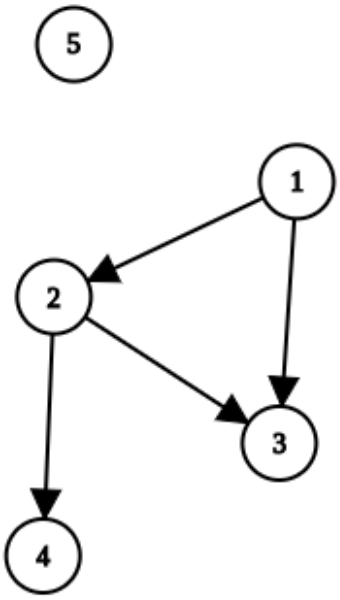
Vectorul **topsort** va reprezenta o permutare a multimii $1, 2, 3, \ldots, n$ reprezentand sortarea topologica gasita.

$n = 5$ $m = 4$

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

Răspuns: $topsort = [1, 2, 3, 4, 5]$

Explicație: Graful dat este cel din figura următoare.



- deoarece avem arcele $1 \rightarrow 3$ si $1 \rightarrow 2$, 1 trebuie să apara înainte lui 2 și 3
- deoarece avem arcul $2 \rightarrow 3$ si $2 \rightarrow 4$, 2 trebuie să apara înainte lui 3 și 4
- 5 nu depinde de nimeni, poate să apară oriunde

Toate sortările topologice valide sunt:

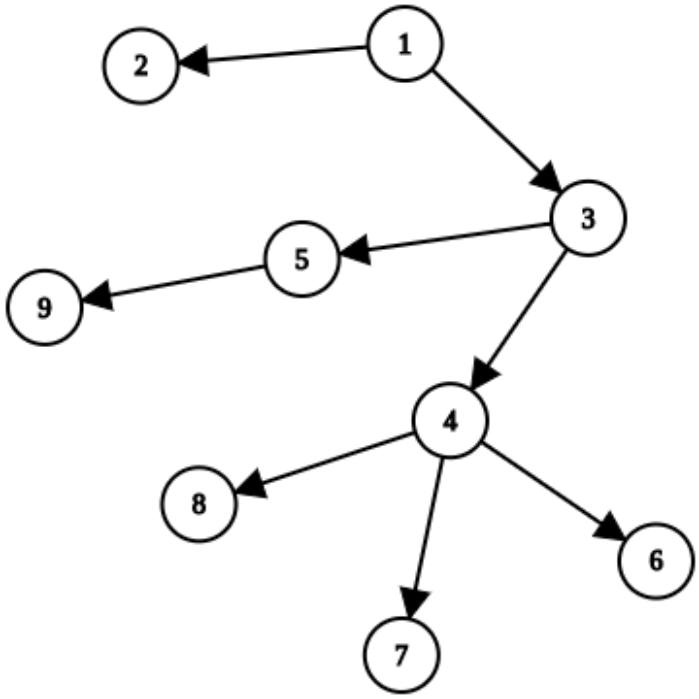
- cele date de ordinea relativa a primelor 4 noduri: 1,2,3,4
  - $topsort = [1, 2, 3, 4, 5]$
  - $topsort = [1, 2, 3, 5, 4]$
  - $topsort = [1, 2, 5, 3, 4]$
  - $topsort = [1, 5, 2, 3, 4]$
  - $topsort = [5, 1, 2, 3, 4]$
- cele date de ordinea relativa a primelor 4 noduri: (1,2,4,3)
  - $topsort = [1, 2, 4, 3, 5]$
  - $topsort = [1, 2, 4, 5, 3]$
  - $topsort = [1, 2, 5, 4, 3]$
  - $topsort = [1, 5, 2, 4, 3]$
  - $topsort = [5, 1, 2, 4, 3]$

$n = 9$ $m = 8$

$arce : (1, 2); (1, 3); (3, 4); (3, 5); (5, 9); (4, 6); (4, 7); (4, 8);$

Răspuns: $topsort = [1, 2, 3, 4, 6, 7, 8, 5, 9]$

Explicație: Graful dat este cel din figura următoare.



Se observă din desen că soluția menționată este validă.

BONUS

B1 Determinați componentele conexe ale unui graf neorientat. Puteți testa implementarea pe infoarena la problema dfs [https://infoarena.ro/problema/dfs].

B2 Rezolvați problema muzeu [https://infoarena.ro/problema/muzeu] pe infoarena.

Extra

Rezolvați problema arbore3 [https://infoarena.ro/problema/arbore3] pe infoarena.
Rezolvați problema Pokemon GO AWAY [https://www.hackerrank.com/contests/test-practic-pa-2017-v2-meeseeks/challenges/test-2-pokemon-go-away-grea] de la test PA 2017. Cu ce problemă seamana?
Rezolvați problema insule [https://infoarena.ro/problema/insule] pe infoarena.
Rezolvați problema tsunami [https://infoarena.ro/problema/tsunami] pe infoarena.
Rezolvați problema berarii2 [https://infoarena.ro/problema/berarii2] pe infoarena.

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/Breadth-first_search [https://en.wikipedia.org/wiki/Breadth-first_search]

[2] https://en.wikipedia.org/wiki/Depth-first_search [https://en.wikipedia.org/wiki/Depth-first_search]

[3] https://en.wikipedia.org/wiki/Topological_sorting [https://en.wikipedia.org/wiki/Topological_sorting]