Laborator 07 - Grafuri

Responsabili:

- Mihai Bivol (2013)
- Sorina Sandu (2013)
- Claudia Cârdei (2012)

Obiective

În urma parcurgerii acestui laborator, studentul va fi capabil:

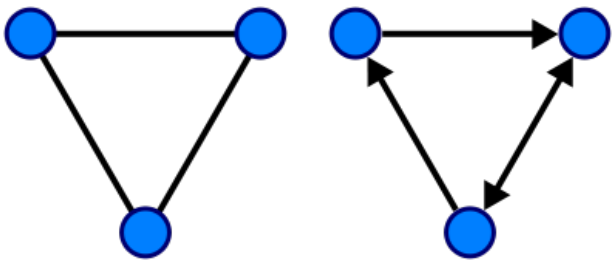
- să înțeleagă operațiile de parcurgere a grafurilor și diferențele dintre ele.
- să implementeze parcurgerile pe grafuri având la dispoziție structurile de date studiate.
- să evalueze complexitatea parcurgerii grafurilor.
- să găsească soluțiile unor probleme folosind algoritmii de parcurgere

Ce este un graf

Un graf este o pereche de mulţimi `G = (V, E)`. Mulțimea `V` conține nodurile grafului `(vertices)`, iar mulțimea `E` conține muchiile `(edges)` sale, fiecare muchie stabilind o relație de vecinătate între cele două noduri. Mulţimea `E` este inclusă în mulţimea `VxV`.

Diferenţa între graf orientat şi graf neorientat

Dacă pentru orice element al mulţimii `E`, `e = (u, v)`, elementul `e' = (v, u)` aparţine de asemenea mulţimii `E`, atunci spunem că graful este **neorientat**. În caz contrar, graful este **orientat**. În cazul grafului orientat, muchiile se mai numesc şi arce.



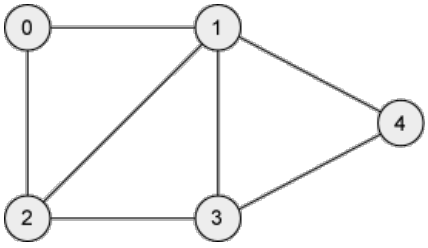
Reprezentările grafurilor în memorie

În funcţie de problemă şi de tipul grafurilor, avem 2 reprezentări: liste de adiacenţă sau matrice de adiacenţă.

Liste de adiacenţă

Reprezentarea prin liste de adiacenţă constă într-un tablou `Adj` cu `|V|` liste, una pentru fiecare vârf din `V`. Pentru fiecare `u` din `V`, lista de adiacenţă `Adj[u]` conţine referinţe către toate vârfurile `v` pentru care există muchia `(u, v)` în `E`. Cu alte cuvinte, `Adj[u]` este formată din totalitatea vârfurilor adiacente lui `u` în `G`.

Această reprezentare este preferată pentru grafurile rare (`|E|` este mult mai mic decât `|V|x|V|`).



Pentru graful de mai sus, lista de adiacenţă este următoarea:

- **0**: 1→2
- **1**: 0→2→3→4
- **2**: 0→1→3
- **3**: 1→2→4
- **4**: 1→3

Matrice de adiacenţă

Reprezentarea prin matrice de adiacenţă a unui graf constă într-o matrice `A[i][j]` de dimensiune `|V|x|V|` astfel încât:

- `A[i][j] = 1`, dacă muchia `(i,j)` aparţine lui `E`
- `A[i][j] = 0`, în caz contrar.

Această reprezentare este preferată pentru grafurile dense (`|E|` este aproximativ egal cu `|V|x|V|`).

Pentru graful de mai sus, matricea de adiacenţă este următoarea:

	0	**1**	**2**	**3**	**4**
0	0	1	1	0	0
1	1	0	1	1	1
2	1	1	0	1	0
3	0	1	1	1	0
4	0	1	0	1	0

Parcurgerea grafurilor

Parcurgerea în lăţime

Parcurgerea în lățime (**Breadth-first Search - BFS**) presupune vizitarea nodurilor în următoarea ordine:

- nodul sursă (considerat a fi pe nivelul 0)
- vecinii nodului sursă (aceștia constituind nivelul 1)
- vecinii încă nevizitați ai nodurilor de pe nivelul 1 (aceștia constituind nivelul 2)
- vecinii încă nevizitați ai nodurilor de pe nivelul 2
- s.a.m.d.

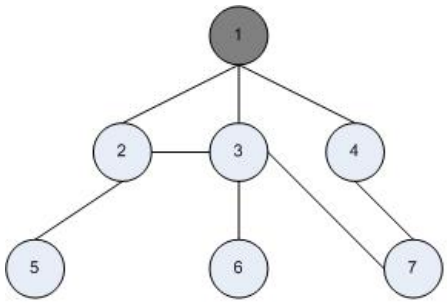
Caracteristica esențială a acestui tip de parcurgere este, deci, că se preferă explorarea **în lățime**, a nodurilor de pe același nivel (aceeași depărtare față de sursă) în detrimentul celei **în adâncime**, a nodurilor de pe nivelul următor.

Pași de execuție

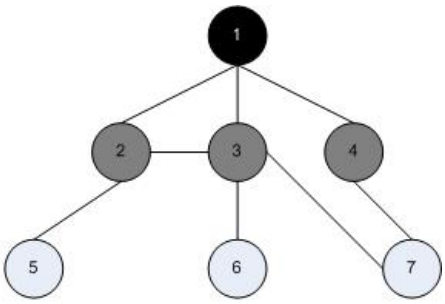
- colorarea nodurilor. Pe parcurs ce algoritmul avansează, se colorează nodurile în felul următor:
  - **alb** - nodul este nedescoperit încă
  - **gri** - nodul a fost descoperit și este în curs de procesare
  - **negru** - procesarea nodului s-a încheiat
- păstrarea informațiilor despre distanța până la nodul sursa.
  - entru fiecare nod în `d[u]` se reține distanța până la nodul sursă (poate fi util în unele probleme)
- obținerea arborelui BFS.
  - în urma aplicării algoritmului BFS se obține un arbore de acoperire (prin eliminarea muchiilor pe care nu le folosim la parcurgere). Pentru a putea reconstitui acest arbore, se

păstrează pentru fiecare nod dat informația despre părintele său în `p[u]`.

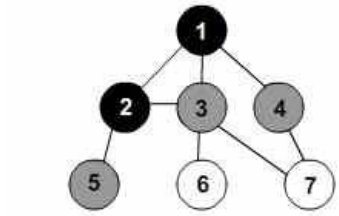
Pentru implementarea BFS se utilizează o coadă (`Q`) în care inițial se află doar nodul sursă. Se vizitează pe rând vecinii acestui nod şi se pun și ei în coada. În momentul în care nu mai există vecini nevizitați, nodul sursă este scos din coadă.



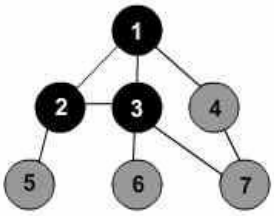
Q: 1



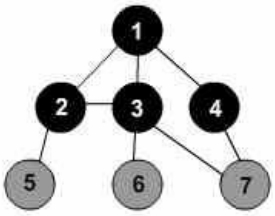
Q: ~~1~~,2,3,4



Q: ~~2~~, 3, 4, 5

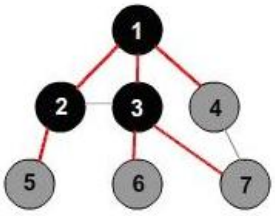


Q: ~~3~~, 4, 5, 6, 7



Q: ~~4~~, 5, 6, 7

Arborele obținut în urma execuției este următorul:



Q: ~~3~~, 4, 5, 6, 7

Pseudocod BFS

```
// inițializări
pentru fiecare nod u din V
{
    culoare[u] = alb
    d[u] = infinit
    p[u] = null
}
culoare[sursa] = gri
d[sursa] = 0
enqueue(Q,sursa) // punem nodul sursă în coada Q

// algoritmul propriu-zis
cât timp coada Q nu este vidă
{
    v = dequeue(Q) // extragem nodul v din coadă
    pentru fiecare u dintre vecinii lui v
        dacă culoare[u] == alb
        {
            culoare[u] = gri
            p[u] = v
            d[u] = d[v] +1
            enqueue(Q,u) // adăugăm nodul u în coadă
        }
    culoare[v] = negru // am terminat de explorat toți veci
}
```

Dacă graful are mai multe componente conexe, algoritmul, în forma dată, va parcurge doar componenta din care face parte nodul sursă. Pe grafuri cu mai multe componente conexe se va aplica în continuare algoritmul pentru fiecare nod rămas nevizitat și astfel se vor obține mai mulți arbori, câte unul pentru fiecare componentă.

Complexitate BFS

`O(|E|+|V|)` - unde `|E|` este numărul de muchii, iar `|V|` este numărul de noduri.

- **explicație**: în cazul cel mai defavorabil, vor fi explorate toate muchiile și toate nodurile. Acesta se obține atunci când nodurile sunt liniarizate.

Parcurgerea în adâncime

Parcurgerea în adâncime (**Depth-First Search - DFS**) presupune explorarea nodurilor în următoarea ordine:

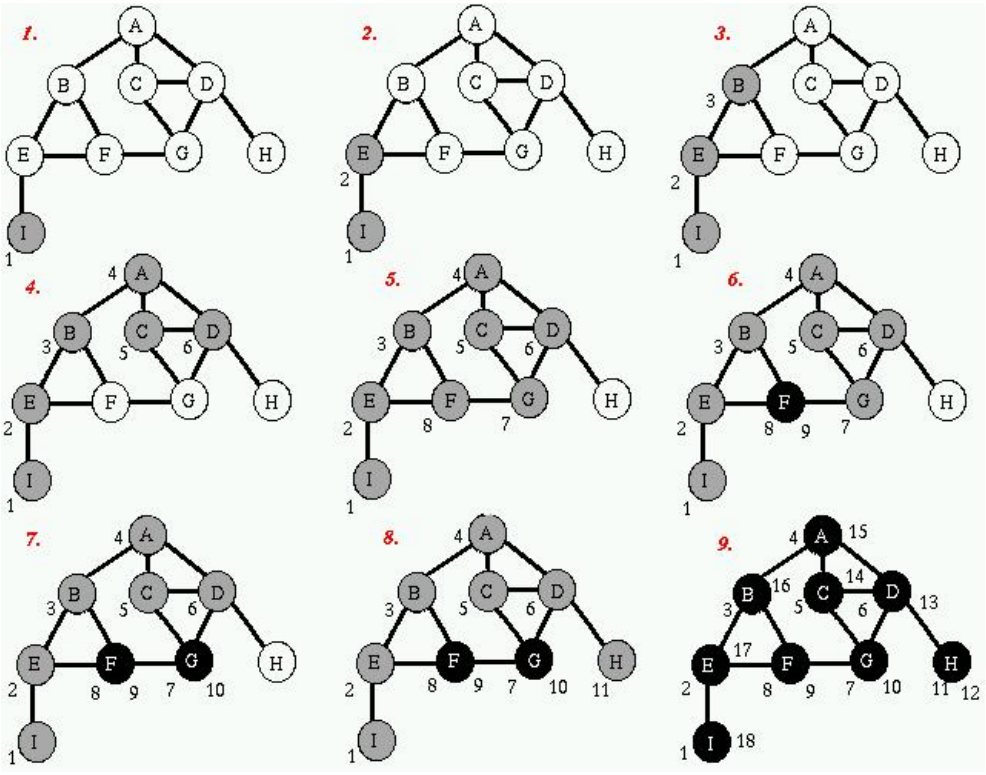
- nodul sursă
- primul vecin nevizitat al nodului sursă (îl vom numi `V1`)
- primul vecin nevizitat al lui `V1` (îl vom numi `V2`)
- primul vecin nevizitat al lui `V2`
- s.a.m.d.
- în momentul în care am epuizat vecinii unui nod `Vn`, continuăm cu următorul vecin nevizitat al nodului anterior, `Vn-1`

Așadar, spre deosebire de BFS, acest tip de parcurgere pune prioritate pe explorarea **în adâncime** (la distanțe tot mai mari față de nodul sursă), în detrimentul celei **în lățime** (pe același nivel).

Pași de execuție

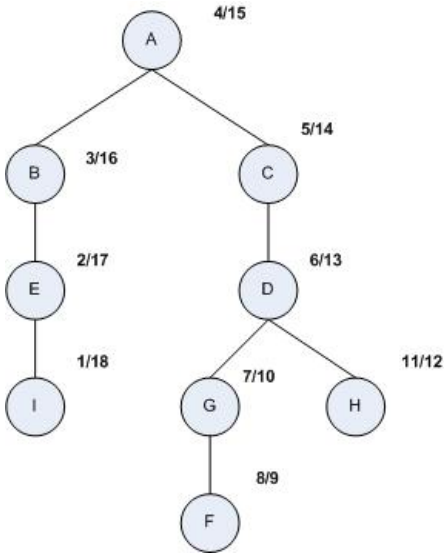
- colorarea nodurilor. Pe parcurs ce algoritmul avansează, se colorează nodurile in felul următor:
  - **alb** - nodul este nedescoperit încă
  - **gri** - nodul a fost descoperit și este în curs de procesare
  - **negru** - procesarea nodului s-a încheiat
- păstrarea informațiilor despre timp. Fiecare nod are două momente de timp asociate:
  - `tDesc[u]` - momentul descoperirii nodului (și a schimbării culorii din alb în gri)
  - `tFin[u]` - momentul în care procesarea nodului s-a încheiat (și culoarea acestuia s-a schimbat din gri în negru)
- obținerea arborelui DFS.
  - 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 informația despre părintele său în `p[u]`.

Un exemplu de aplicare al DFS este următorul:



Nodul de pornire este I, iar pentru simplificare vecinii sunt aleși în ordine alfabetică. În stânga nodului este notat `tDesc`, iar în dreapta `tFin`. Dacă se afișează nodurile, în urma parcurgerii se obține următorul output: **I, E, B, A, C, D, G, F, H**

Arborele obținut în urma parcurgerii este următorul:



Pseudocod DFS

```
//inițializări
pentru fiecare nod u din V
{
    culoare[u] = alb
    p[u] = NULL
    tDesc[u] = 0
    tFin[u] = 0
}
contor_timp = 0


 vizitare(nod) // functie de vizitare a nodului
 {
    contor_timp = contor_timp + 1
    tDesc[nod] = contor_timp
    culoare[nod] = gri
```

```
    printeaza nod;
 }
//algoritmul propriu-zis
DFS( nod)
{
    stiva s;

    viziteaza nod;
    s.introdu(nod);

    cat timp stiva s nu este goala
    {
        nodTop=nodul din varful stivei

        vecin=afla primul vecin nevizitat al lui nodTop.
        daca vecin exista
            {
               p[v] = nodTop
               viziteaza v;
               s.introdu(v);
             }
        altfel
        {
          contor_timp = contor_timp + 1
          tFin[nodTop] = contor_timp
          culoare[nodTop] = negru
          s.scoate(nodTop);
        }
    }



}
```

Complexitate DFS

La fel ca în cazul BFS, complexitatea este `O(|E|+|V|)` - unde `|E|` este numărul de muchii, iar `|V|` este numărul de noduri.

- **explicație**: în cazul cel mai defavorabil, vor fi explorate toate muchiile si toate nodurile.Acesta este cand graful este liniarizat.

Aplicaţii parcurgeri

Aflarea distanței minime între două noduri

Dacă toate muchiile au același cost, putem afla distanța minimă între două noduri `A` și `B` efectuând o parcurgere `BFS` din nodul `A` și oprindu-ne atunci când nodul `B` a fost descoperit. Reamintindu-ne că nivelul unui nod este analog distanței, în muchii, față de sursă, și că `BFS` descoperă un nod de pe nivelul `N` numai după ce toate nodurile de pe nivele inferioare au fost descoperite, este ușor de văzut că nivelul nodului `B` în parcurgere corespunde distanței minime între `A` și `B`.

Pentru a reține distanța și drumul exact de la `A` la `B`, se vor reține pentru fiecare nod `d[x]` (distanța de la `sursă` la `x`) și `p[x]` (părintele lui `x` în drumul de la sursă spre `x`). În momentul descoperirii unui nod `y` al cărui părinte este `x`, se vor face următoarele atribuiri:

```
d[y] = d[x] + 1
p[y] = x
```

sursa având `d[A] = 0` și `p[A] = NULL`.

Observații:

- dacă parcurgerea BFS se încheie fără ca nodul B să fi fost descoperit, nu există drum între A și B și deci distanța între acestea este infinită.
- Algoritmul funcționează corect numai în situații de cost uniform

(toate muchiile au același cost). Pentru grafuri cu muchii de costuri diferite, sunt necesari algoritmi mai avansați, cum ar fi Dijkstra sau Bellman-Ford.

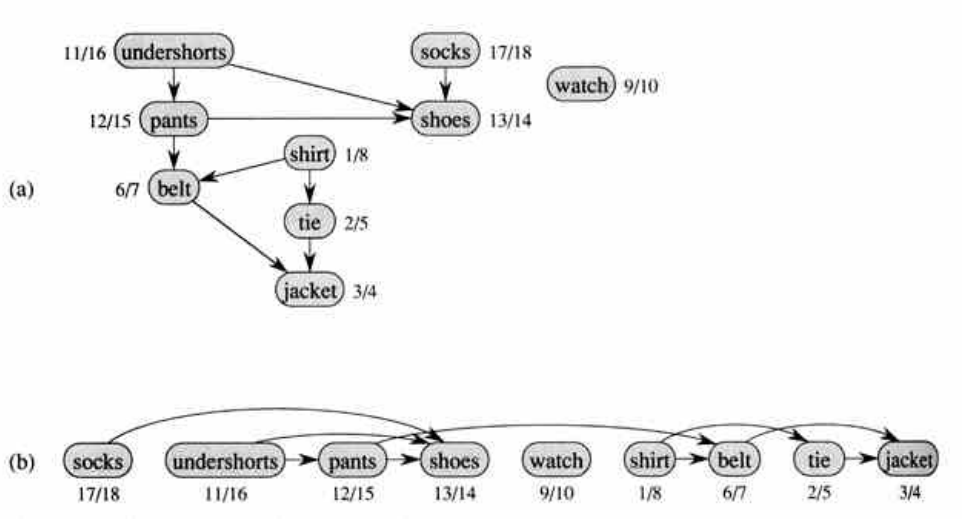
Sortarea topologică

Se dă un graf orientat aciclic. Orientarea muchiilor corespunde unei relații de ordine de la nodul sursă către cel destinație. O sortare topologică a unui astfel de graf este o ordonare liniară a vârfurilor sale astfel încât, dacă `(u,v)` este una dintre muchiile grafului, `u` trebuie să apară înaintea lui `v` în înșiruire. Dacă graful ar fi ciclic, nu ar putea exista o astfel de înșiruire (nu se poate stabili o ordine între nodurile care alcătuiesc un ciclu).

Sortarea topologică poate fi văzută și ca plasarea nodurilor de-a lungul unei linii orizontale astfel încât toate muchiile să fie direcționate de la stânga la dreapta.

Un exemplu: Profesorul Bumstead își sortează topologic hainele înainte de a se îmbrăca.

- fiecare muche `(u, v)` înseamna că obiectul de îmbrăcăminte `u` trebuie îmbrăcat înaintea obiectului de îmbrăcaminte `v`. Timpii de descoperire `(tDesc)` și de finalizare `(tFin)` obținuți în urma parcurgerii DFS sunt notați lângă noduri.
- același graf, sortat topologic. Nodurile lui sunt aranjate de la stânga la dreapta în ordinea descrescătoare a `tFin`. Observați că toate muchiile sunt orientate de la stânga la dreapta. Acum profesorul Bumstead se poate îmbrăca liniștit.



Așa cum se observă din poză, sortarea topologică constă în sortarea nodurilor descrescător după timpii de finalizare. Demonstrația acestei afirmații se face simplu, arătând că nodul care se termină mai târziu trebuie să fie efectuat înaintea celorlalte noduri finalizate.

Importanţă

Grafurile sunt utile pentru a modela diverse probleme şi se regăsesc implementaţi în multiple aplicaţii practice:

- reţele de calculatoare (ex: stabilirea unei topologii fără bucle)
- pagini Web (ex: Google PageRank)
- rețele sociale (ex: calcul centralitate)
- hărţi cu drumuri (ex: drum minim)
- modelare grafică (ex: prefuse, graph-cut)

Aplicaţii

- se va porni de la scheletul de cod oferit: **lab07-tasks.zip**

1. Se citesc din fisierul `graf.in` M muchii ale unui graf *neorientat* cu N varfuri. Folosind scheletul de cod dat implementati urmatoarele cerinte:

- TODO1.1: Studiati reprezentarea grafului in scheletul de cod atasat.
- TODO1.2: (1p) Adaugati muchiile in structura grafului. In scheletul de cod graful este reprezentat ca un vector de liste de adiacenta.
- TODO1.3: (3p) Implementati parcurgerea BFS a grafului pornind dintr-un nod ales.
- TODO1.4: (3p) Implementati parcurgerea DFS a grafului pornind dintr-un nod ales.
- Datele se vor citi dintr-un fișier cu următorul format.
- pe prima linie: numărul de varfuri `N` și de muchii `M` ale grafului
- pe următoarea linie: nodul de start BFS si nodul de start DFS
- pe următoarele `M` linii: perechi de noduri (u,v) pentru care exista muchii in graf

- **hints:** În cadrul scheletului, s-au folosit următoarele clase din STL:
- Vector pentru care aveţi nevoie de următoarele funcţii:
  - v.size() - întoarce numărul de elemente din vector
  - v[i] (operatorul `[]`) - întoarce elementul de pe poziția i
  - Un vector se poate sorta cu ajutorul funcției sort din STL.
  - Exemplu de sortare: `std::sort(v.begin(),v.end());`
  - Exemplu de parcurgere și afișare a elementelor dintr-un vector STL folosind indecsi:

```
for (int i = 0; i < v.size(); i ++)
{
    std::cout << v[i] << std::endl;
}
```

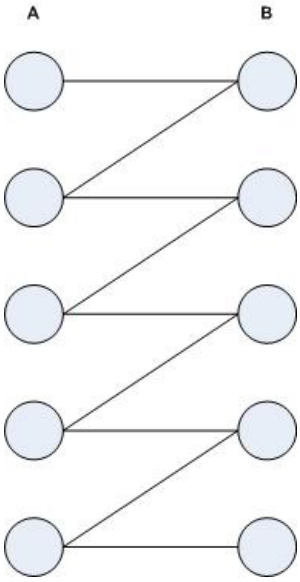
- Exemplu de parcurgere și afișare a elementelor dintr-un vector STL folosind iteratori:

```
for (std::vector<int>::iterator it=v.begin();it!=v.end();it+
{
    std::cout << *it << std::endl;
}
```

- Exemplu de parcurgere și afișare a elementelor dintr-un vector STL folosind funcția copy din STL.

```
std::ostream_iterator<int> out_it(std::cout,std::endl);
std::copy(v.begin(),v.end(),out_it);
```

2.**[4p]** Se numește graf bipartit un graf `G = (V, E)` în care mulțimea nodurilor poate fi împărțită în două mulțimi disjuncte `A` și `B` astfel încât `V = A U B` și `E este inclus în A x B` (orice muchie leagă un nod din `A` cu un nod din `B`).



Folosind codul de la problema 1, determinaţi dacă un graf este bipartit.

- Pentru a determina dacă un graf este bipartit sau nu, una din metode constă în efectuarea de parcurgeri `BFS` și atribuirea de etichete

nodurilor conform cu paritatea nivelului acestora în parcurgere (`A` pentru nodurile de pe nivel par, `B` pentru nodurile de pe nivel impar). Atunci când se adaugă vecinii nevizitați ai unui nod în coadă, se vor verifica de asemenea etichetele vecinilor deja vizitați: dacă se descoperă că unul din aceștia are aceeași etichetă ca cea atribuită nodului curent, graful are o muchie între noduri de pe același nivel și deci nu poate fi bipartit. În caz contrar (s-a realizat parcurgerea `BFS` fără a apărea această situație), graful este bipartit și nodurile sunt etichetate cu mulțimea din care fac parte.
- In scheletul de cod, sectiunea aferenta acestei probleme este marcata cu TODO2.1

3. **[4p]** Consideram ca fisierul de intrare al primei probleme, reprezinta `N` cursuri ale unei programe analitice, între care există `M` condiționări de forma `A-B`, cu semnificația `cursul A trebuie să preceadă cursul B`. (In acest caz graful este *orientat*.) Fiind date regulile de precendență (aflate în fișierul `graf.in`), propuneți o ordonare coerentă de studiere a materiilor.

- Structura codului ce trebuie implementată este marcată cu secțiunea TODO3.1
- Se consideră că cursul 0 reprezintă **singurul** curs de bază, fără de care nici un alt curs nu se poate desfășura.

Interviu

Această secțiune nu este punctată și încearcă să vă facă o oarecare idee a tipurilor de întrebări pe care le puteți întâlni la un job interview (internship, part-time, full-time, etc.) din materia prezentată în cadrul laboratorului.

Cum multe din companiile mari folosesc date stocate sub forma de grafuri (Facebook Open Graph, Google Social Graph si Page Rank etc) la angajare vor dori sa vada ca stiti grafuri:

- cum se reprezeinta grafurile
- cum functioneaza si cum se implementeaza parcurgerile (BFS, DFS)
- algoritmi mai avansati pentru grafuri precum Dijkstra si A* (cu care va veti familiariza la materiile de algoritmica din anul 2: Analiza Algoritmilor si Proiectarea Algoritmilor)

Puteti cauta mai multe intrebari pe http://www.careercup.com/ si pe http://www.glassdoor.com/

Resurse

[1] - BFS

[2] - Distanţa minimă

[3] - DFS

[4] - Sortare topologică

[5] - Dijkstra

[6] - Bellman - Ford