Teoria Grafurilor şi Combinatorică

Curs 9: Traversarea grafurilor

23 noiembrie 2020

Curs actualizat în 5 decembrie 2020. În acest curs sunt prezentaţi algoritmii fundamentali de traversare a grafurilor – traversarea în adâncime şi traversarea în lăţime, – şi aplicaţii ale acestora. Aceşti algoritmi sunt concepuţi să rezolve problema următoare:

Se dau un graf $G = (V, E)$ şi un nod $s \in V$, numit **sursa** căutării.

Să se determine (1) mulţimea de noduri la care se poate ajunge din s, adică $S_s = \{x \in V \mid s \rightsquigarrow x\}$, şi (2) o mulţime $\{\pi_x \mid x \in S_s\}$ unde π_x este o cale de la s la x în G.

Cele mai cunoscute strategii de traversare a grafurilor sunt **traversarea în adâncime** (engl. depth first search sau **DFS**) şi **traversarea în lăţime** (engl. breadth first search sau **BFS**). Ambele strategii rezolvă problema de mai sus calculând reprezentarea cu predecesori a unui arbore T_s cu rădăcina s, care are proprietăţile următoare:

1. Mulţimea de noduri a lui T_s este S_s, adică $V(T_s) = S_s$.
2. Pentru fiecare $x \in S_s$, lista de noduri pe ramura de la s la x în T_s este o cale de la s la x în G. Deci T_s este o reprezentare compactă a mulţimii de căi $\{\pi_x \mid x \in S_s\}$ din enunţul problemei.

Vom prezenta implementarea unui API java pentru traversarea grafurilor. Acest API constă din clasa abstractă `GraphSearch` cu subclasele `BFS` şi `DFS`:

`public abtract class GraphSearch`		
`boolean`	`visited(int x)`	Există drum de la sursă la `x`?
`Iterable<Integer>`	`path(int x)`	Un drum de la sursă la `x`

`public class DFS`	`extends GraphSearch`
`DFS(IGraph G,int s)`	Constructor de traversare în adâncime a lui `G` pornind de la nodul sursă `s`

`public class DFS`	`extends GraphSearch`
`BFS(IGraph G,int s)`	Constructor de traversare în lăţime a lui `G` pornind de la nodul sursă `s`

În implementare vom folosi două clase pentru colecţii iterabile: `Queue<E>` pentru o coadă de elemente de tip `E`, şi `Stack<E>` pentru o stivă de elemente de tip `E`. Ambele clase sunt din arhiva de clase java `algs4.jar`.

public	class Queue<E>	implements Iterable<E>
	Queue<E>()	Crează o coadă fără elemente
void	enqueue(E e)	Adaugă elementul e la sfârşitul cozii
void	E dequeue()	Returnează primul element din coadă, şi îl şterge din coadă
int	size()	Numărul de elemente din coadă
boolean	isEmpty()	Este coada vidă?

public	class Stack<E>	implements Iterable<E>
	Stack<E>()	Crează o stivă fără elemente
void	push(E e)	Pune elementul e în vârful stivei
E	pop()	Scoate din stivă ultimul element adăugat
int	size()	Numărul de elemente din stivă
boolean	isEmpty()	Este stiva vidă?

Reamintim că o clasă este iterabilă dacă implementează interfaţa `Iterable`, care ne permite să scriem bucle for-each ca să iterăm toate elementele. De exemplu, putem avea cod client de forma

```
Queue<Integer> Q = ...;
Stack<Integer> S = ...;
for (int e : Q) System.out.println(e);
for (int e : S) System.out.println(e);
```

Elementele unei cozi sunt iterate de la primul la ultimul element inserat (First-In First-Out), iar elementele unei stive sunt iterate de la ultimul la primul element inserat (Last-In First-Out).

1 Traversarea în adâncime de la un nod sursă

Cu această strategie, vizitarea unui nod x se face astfel:

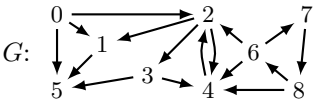
1. Se marchează x ca nod vizitat.
2. Se vizitează recursiv toţi vecinii nevizitaţi ai lui x. Pentru fiecare vecin y care se vizitează se setează $\mathtt{p}[y] = x$ pentru a reţine faptul că traversarea la y s-a făcut de la x.

Arborele T_s calculat cu această strategie se numeşte arborele DFS cu rădăcina s. O implementare a acestei strategii este clasa `DFS` ilustrată mai jos:

```
public class DFS {
     private boolean[] visited;
     private int[] p;
     private final int s;
```

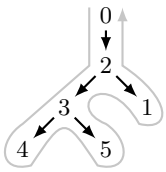
```
   public DFS(IGraph G, int s) {
      visited = new boolean[G.V()];
      p = new int[G.V()];
      this.s = s;
      dfs(G,s);
   }
   private void dfs(IGraph G, int i) {
      visited[i] = true;
      for (int j : G.adj(i))
          if (!visited[j]) {
             p[j] = i;    // reţine că s-a traversat muchia i−j
             dfs(G,j);
          }
   }
   public boolean visited(int x) { return visited[x]; }
   public Iterable<Integer> path(int x) {
      Stack<Integer> S = new Stack<Integer>();
      if (visited(x))
         while(x!=s) { S.push(x); x=p[x]; }
         S.push(s);
         return S;
   }
}
```

Exemplul 1. Fie digraful reprezentat cu liste de adiacenţă



$\texttt{adj}[0] = [2, 1, 5]$, $\texttt{adj}[1] = [5]$, $\texttt{adj}[2] = [3, 1, 4]$,
$\texttt{adj}[3] = [4, 5]$, $\texttt{adj}[4] = [2]$, $\texttt{adj}[5] = []$,
$\texttt{adj}[6] = [4, 2]$, $\texttt{adj}[7] = [8]$, $\texttt{adj}[8] = [6]$.

Traversarea în adâncime cu $\texttt{DFS}(G, 0)$ vizitează nodurile 0,2,3,4,5,1 în această ordine şi setează $\texttt{p}[0] = \texttt{p}[6] = \texttt{p}[7] = \texttt{p}[8] = -1$, $\texttt{p}[2] = 0, \texttt{p}[1] = \texttt{p}[3] = 2$, $\texttt{p}[4] = \texttt{p}[5] = 3$, adică reprezentarea arborelui DFS cu rădăcina 0 ilustrat mai jos:



nodul 0 nu are părinte; 0 este nod rădăcină.

nodul 2 are părintele 0.

nodurile 3 şi 1 au părintele 2.

nodurile 4 şi 5 au părintele 3.

Linia curbă indică ordinea de vizitare a nodurilor de către DFS. Cu API-ul clasei `DFS` putem consulta componenţa şi structura acestui arbore:

x	`visited`(x)	`G.path`(x)
0	`true`	[0]
1	`true`	[0,1]
2	`true`	[0,2]
3	`true`	[0,2,3]
4	`true`	[0,2,3,4]
5	`true`	[0,2,3,5]
6	`false`	`null`
7	`false`	`null`
8	`false`	`null`

□

Drumurile găsite de căutarea în adâncime nu sunt cele mai scurte posibile. De exemplu, `G.path(5)` găseşte drumul $[0, 2, 3, 5]$, dar drumul cel mai scurt de la 0 la 5 este $[0, 5]$. Dacă vrem să găsim drumuri de lungime minimă, putem folosi traversarea în lăţime.

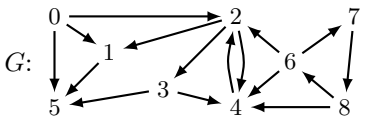
2 Traversarea în lăţime de la un nod sursă

Traversarea în lăţime de la un nod sursă s se face în runde: în prima rundă vizităm s, iar în fiecare rundă următoare vizităm vecinii nevizitaţi ai nodurilor vizitate în runda precedentă. Arborele T_s calculat cu această strategie se numeşte arborele BFS cu rădăcina s.

Această strategie se poate implementa cu o coadă în care se reţin nodurile în ordinea în care urmează să le vizităm vecinii nevizitaţi. Pentru lucrul cu cozi, folosim clasa `Queue<E>` din `algs4.jar`. O implementare a acestei strategii este clasa `BFS` care diferă de `DFS` doar prin faptul că foloseşte metoda `bfs()` în loc de `dfs()` ca să traverseze graful `G` pornind de la sursă.

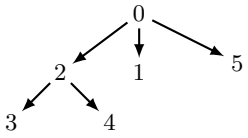
```
private void bfs(IGraph G, int s) {
   visited[s] = true;       // vizitează nodul sursă
   Queue<Integer> Q = new Queue<Integer>();
   Q.enqueue(s);            // şi il pune in coadă
   while(!Q.isEmpty()) {
      int v = Q.dequeue();
      for(int w : G.adj(v))
         if(!visited[w]) {
            p[w] = v;
            visited[w] = true;
            Q.enqueue(w);
         }
   }
}
```

Exemplul 2. Fie digraful reprezentat cu liste de adiacenţă



$\texttt{adj}[0] = [2, 1, 5], \texttt{adj}[1] = [5], \texttt{adj}[2] = [3, 1, 4],$
$\texttt{adj}[3] = [4, 5], \texttt{adj}[4] = [2], \texttt{adj}[5] = [],$
$\texttt{adj}[6] = [4, 2, 7], \texttt{adj}[7] = [8], \texttt{adj}[8] = [4, 6].$

BFS de la sursa 0 vizitează nodurile 0,2,1,5,3,4 în această ordine şi setează $\texttt{p}[0] = \texttt{p}[6] = \texttt{p}[7] = \texttt{p}[8] = -1$, $\texttt{p}[2] = \texttt{p}[1] = \texttt{p}[5] = 0$, $\texttt{p}[3] = \texttt{p}[4] = 2$, adică reprezentarea arborelui BFS cu rădăcina 0 ilustrat mai jos:



□

Pentru fiecare nod $x \in S_s$ în arborele BFS cu rădăcina s al unui graf G, lista de noduri pe ramura de la s la x este un drum de lungime minimă de la s la x în G.

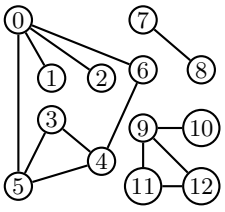
2.1 Ordini de traversare a nodurilor în adâncime

Un rol important în multe aplicaţii îl joacă relaţiile de ordine definite de traversarea în adâncime a tuturor nodurilor unui graf. Pentru un graf $G = (V, E)$, aceste ordini de traversare se definesc în raport cu următorul proces: se fixează o enumerare $[x_1, x_2, \ldots, x_n]$ a nodurilor din V şi se execută

```
for (int s = 1; s <= n; s++)
     if (!vizitat[x_s]) dfs(G,x_s);
```

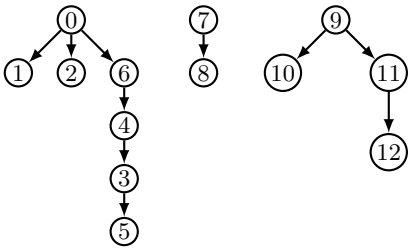
DFS de la un singur nod sursă produce un arbore DFS cu rădăcina s, în timp ce traversarea tuturor nodurilor lui G produce o pădure de arbori DFS.

Exemplul 3. Fie graful neorientat reprezentat cu listele de adiacenţă



$\texttt{adj}[0] = [1, 2, 5, 6], \texttt{adj}[1] = [0], \texttt{adj}[2] = [0],$
$\texttt{adj}[3] = [4, 5], \texttt{adj}[4] = [3, 5, 6], \texttt{adj}[5] = [0, 3, 4],$
$\texttt{adj}[6] = [0, 4], \texttt{adj}[7] = [8], \texttt{adj}[8] = [7],$
$\texttt{adj}[9] = [10, 11, 12], \texttt{adj}[10] = [9], \texttt{adj}[11] = [9, 12],$
$\texttt{adj}[12] = [9, 11].$

Pentru enumerarea nodurilor $[0, 1, 2, \ldots, 11, 12]$, traversarea în adâncime produce o pădure cu trei arbori DFS:



□

Ordinile de traversare în adâncime sunt:

Preordine: $x <_{\text{pre}} y$ dacă una din condiţiile următoare are loc:

- x, y sunt în arbori DFS diferiţi iar arborele DFS în care apare x este construit înaintea arborelui DFS în care apare y, sau
- x, y sunt în acelaşi arbore DFS şi x este deasupra lui y, sau
- x, y sunt în acelaşi arbore DFS şi x este în stânga lui y.

Relaţia de preordine se poate calcula în felul următor: Fiecare nod x se adaugă în o coadă înaintea apelului recursiv al lui `dfs(G,x)`. Avem $x <_{\text{pre}} y$ dacă x apare înaintea lui y în această coadă.

De pildă, enumerarea în preordine a nodurilor din cei trei arbori DFS din Exemplul 7 este [0,1,2,6,4,3,5,7,8,9,10,11,12].

Postordine: $x <_{\text{post}} y$ dacă una din condiţiile următoare are loc

- x, y sunt în arbori DFS diferiţi iar arborele DFS în care apare x este construit înaintea arborelui DFS în care apare y, sau
- x, y sunt în acelaşi arbore DFS şi x este sub y, sau
- x, y sunt în acelaşi arbore DFS şi x este în stânga lui y.

Relaţia de preordine se poate calcula în felul următor: Fiecare nod x se adaugă în o coadă după apelul recursiv al lui `dfs(G,x)`. Avem $x <_{\text{post}} y$ dacă x apare înaintea lui y în această coadă.

De pildă, enumerarea în postordine a nodurilor din cei trei arbori DFS din Exemplul 7 este [1,2,5,3,4,6,0,8,7,10,12,11,9].

Postordine inversă: Avem $x <_{\text{revpost}} y$ dacă $y <_{\text{post}} x$. Deci postordinea inversă se poate calcula punând nodurile în o stivă după apelul recursiv al lui `dfs()`. Avem $x <_{\text{revpost}} y$ dacă x apare deasupra lui y în stivă.

De pildă, enumerarea în postordine inversă a nodurilor din cei trei arbori DFS din Exemplul 7 este [9,11,12,10,7,8,0,6,4,3,5,2,1].

Aceste ordini se pot calcula şi accesa cu API-ul clasei `DepthFirstOrder`.

`public class`	`DepthFirstOrder`	
`DepthFirstOrder(`	`IGraph G)`	Constructor pentru ordinile de traversare în adâncime
`Iterable<Integer>`	`pre()`	Noduri in preordine
`Iterable<Integer>`	`post()`	Noduri in postordine
`Iterable<Integer>`	`reversePost()`	Noduri in postordine inversă

Figura 1: API pentru ordinile de traversare în adâncime a unui digraf.

```
public class DepthFirstOrder {
   private boolean[] visited;
   private Queue<Integer> pre;
   private Queue<Integer> post;
   private Stack<Integer> reversePost;

   public DepthFirstOrder(IGraph G) {
      pre = new Queue<Integer>();
      post = new Queue<Integer>();
      reversePost = new Stack<Integer>();
      visited = new boolean[G.V()];
      for (int s = 0; s < G.V(); s++)
          if (!visited[s]) dfs(G,s);
   }
   private void dfs(IGraph G, int i) {
      pre.enqueue(i);
      visited[i] = true;
      for (int j : G.adj(i))
           if (!visited[j]) dfs(G,j);
      post.enqueue(i);
      reversePost.push(i);
   }
   public Iterable<Integer> pre() { return pre; }
   public Iterable<Integer> post() { return post; }
   public Iterable<Integer> reversePost() { return reversePost; }
}
```

3 Aplicaţii ale traversării în adâncime

3.1 Detecţia componentelor conexe

Traversarea în adâncime se poate folosi pentru detecţia componentelor conexe ale unui graf neorientat. Clasa `CC` ilustrată mai jos oferă un API pentru calculul lor cu DFS.

`public`	`class CC`	
	`CC(Graph G)`	Constructor pentru componentele conexe ale grafului neorientat `G`
`boolean`	`connected(int i,int j)`	Există drum de la `i` la `j`?
`int`	`count()`	Numărul de componente conexe
`int`	`id(int v)`	Identificatorul de componentă al nodului `v` (între 0 şi `count()-1`)

```
public class CC {
   private boolean[] visited;
   private int[] id;
   private int count;

   public CC(Graph G) {
      visited = new boolean[G.V()];
```

```
      id = new int[G.V()];
      for(int s=0; s<G.V(); s++)
         if (!visited[s]) { dfs(G,s); count++; }
   }
   private void dfs(Graph G, int i) {
        visited[i] = true;
        id[i] = count;
        for (int j : G.adj(i))
            if (!visited[j]) dfs(G,j);
   }
   public boolean connected(int i, int j) {
      return id[i] == id[j];
   }
   public int count() { return count; }
   public int id(int v) { return id[v]; }
}
```

Această traversare foloseşte un contor `count` setat să indice numărul arborelui DFS în curs de construire (pornind de la 0) şi folosit pentru defini identificatorul de componentă al tuturor nodurilor din arborele DFS.

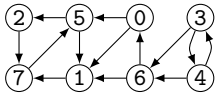
3.2 Detecţia ciclurilor în grafuri orientate

Presupunem că G este un digraf şi ne propunem să rezolvăm problema următoare: „Are G un ciclu? Dacă da, să se găsească un ciclu în G."

Un digraf poate avea un număr exponenţial de cicluri, deci este rezonabil să ne mulţumim cu găsirea unui singur ciclu, atunci când există unul. Multe aplicaţii, inclusiv cea a planificării activităţilor, se pot realiza doar pentru digrafuri fără cicluri, numite **DAG** (engl. directed acyclic graph).

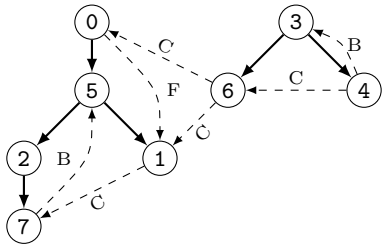
Pornim de la observaţiile următoare:

- Traversarea în adâncime a tuturor nodurilor unui digraf produce o pădure de arbori de căutare în adâncime. Arcele lui G care apar ca muchii în această pădure de arbori se numesc **muchii de arbore** (engl. tree edges). Celelelte arce sunt de 3 feluri, după cum arată dacă le adăugăm la pădurea de arbori de căutare:
  1. **Muchii de întoarcere**: sunt arcele $u \to v$ de la un nod u la un predecesor de-al lui în un arbore de căutare în adâncime.
  2. **Muchii de salt înainte**: sunt arcele $u \to v$ de la un nod u la un succesor de-al lui în un arbore de căutare în adâncime.
  3. **Muchii transversale**: sunt arce $u \to v$ de două feluri: (1) între noduri pe ramuri diferite din acelaşi arbore, sau (2) de la un nod u la un nod v într-un arbore construit anterior.
- G are un ciclu dacă şi numai dacă există o muchie de întoarcere.
- Mai multe detalii despre aceste a tipuri de muchii şi proprietăţile lor găsiţi în subsecţiunea Classification of edges a secţiunii 22.3 din [**?**].

Exemplul 4. Fie digraful G:  reprezentat cu

$\texttt{adj}[0] = [5, 1]$, $\texttt{adj}[1] = [7]$, $\texttt{adj}[2] = [7]$, $\texttt{adj}[3] = [6, 4]$,
$\texttt{adj}[4] = [3, 6]$, $\texttt{adj}[5] = [2, 1]$, $\texttt{adj}[6] = [0, 1]$, $\texttt{adj}[7] = [5]$.

Traversarea în adâncime a lui G produce doi arbori de căutare:



Celelelte arce au fost desenate cu linie întreruptă: muchiile de întoarcere au fost etichetate cu B, cele de salt înainte cu F, şi cele transversale cu C. □

Aşadar, putem reduce detecţia unui ciclu la detecţia unei muchii de întoarcere în un arbore de căutare în adâncime. În acest scop, vom folosi clasa `Stack<E>` din `algs4.jar` pentru stiva de apeluri recursive ale metodei `dfs()`.

Clasa `DCycle` pe care o prezentăm aici implementează API-ul

`public`	`class DCycle`	
	`DCycle(Digraph G)`	Constructor de detecţie a ciclurilor
`boolean`	`hasCycle(E e)`	Are `G` un ciclu?
`Iterable<Integer>`	`cycle()`	Nodurile din un ciclu (dacă există unul)

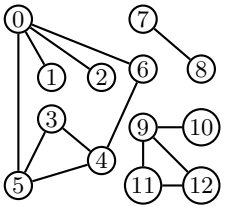
```
public class DCycle {
   private boolean[] visited;
   private int[] p;
   private Stack<Integer> cycle;
   private boolean[] inStack;

   public DCycle(Digraph G) {
      inStack = new boolean[G.V()];
      p = new int[G.V()];
      visited = new boolean[G.V()];
      for (int s=0; s<G.V(); s++)
          if (!visited[s])
             dfs(G,s);
   }
   private void dfs(Digraph G, int i) {
        inStack[i] = visited[i] = true;
        for (int j : G.adj(i))
            if(hasCycle()) return;
            else if (!visited[j]) {
```

```
                p[j] = i;
                dfs(G,j);
            }
            else if (inStack[j]) {
                cycle = new Stack<Integer>();
                for (int k = i; k != j; k = p[k]) cycle.push(k);
                cycle.push(j);
                cycle.push(i);
            }
            inStack[i] = false;
    }
    public boolean hasCycle() { return cycle != null; }
    public Iterable<Integer> cycle() { return cycle; }
}
```

3.3 Detecţia ciclurilor în grafuri neorientate

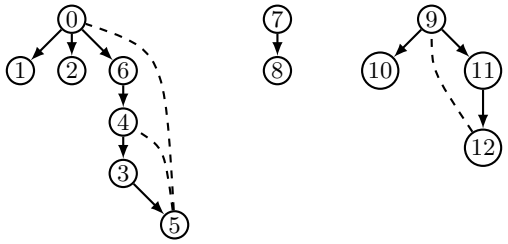
Presupunem că G este un graf neorientat. Traversarea în adâncime a lui G produce o pădure de arbori, câte un arbore pentru fiecare componentă conexă a lui G.

Exemplul 5. Fie graful neorientat reprezentat cu listele de adiacenţă



$\texttt{adj}[0] = [1, 2, 5, 6], \texttt{adj}[1] = [0], \texttt{adj}[2] = [0],$
$\texttt{adj}[3] = [4, 5], \texttt{adj}[4] = [3, 5, 6], \texttt{adj}[5] = [0, 3, 4],$
$\texttt{adj}[6] = [0, 4], \texttt{adj}[7] = [8], \texttt{adj}[8] = [7],$
$\texttt{adj}[9] = [10, 11, 12], \texttt{adj}[10] = [9], \texttt{adj}[11] = [9, 12],$
$\texttt{adj}[12] = [9, 11].$

Traversarea în adâncime produce trei arbori de căutare care, împreună cu celelalte muchii din graful G (desenate cu linie întreruptă), arată astfel:



Toate muchiile suplimentare (cele întrerupte) sunt de la un nod la un predecesor al părintelui său în arbore, motiv pentru care se numesc **muchii de întoarcere**. În literatură, arborii de căutare în adâncime împreună cu muchiile de întoarcere se numesc **palmieri**. Fiecare muchie de întoarcere închide o ramără arborelui de căutare într-un un ciclu. În acest exemplu sunt trei muchii de întoarcere:

5−0, care determină ciclul $[0, 6, 4, 3, 5, 0]$,
5−4, care determină ciclul $[4, 3, 5, 4]$,
12−9, care determină ciclul $[9, 11, 12, 9]$. □

Arborii palmier ne dau un criteriu de detecţie a ciclurilor: Când traversarea de la un nod i cu părintele k descoperă un vecin $j \neq k$ al lui i deja vizitat, ştim că există o muchie de întoarcere de la j care formează un ciclu în G.

Pe baza acestei observaţii, vom extinde API-ul clasei `CC` să detecteze componentele conexe cu ciclu, şi cicluri în ele (dacă există):

`public`	`class CC`	
	...	
`boolean`	`hasCycle(int c)`	Există ciclu în componenta cu identificatorul `c`?
`Iterable<Integer>`	`cycle(int c)`	Nodurile din un ciclu al componentei conexe `c` (dacă există unul)

```
public class CC {
   private boolean[] visited;
   private int[] id;
   private int[] p;
   private int count;
   private Stack<Integer>[] cycle;

   public CC(Graph G) {
      visited = new boolean[G.V()];
      p = new int[G.V()];
      cycle = (Stack<Integer>[]) new Stack[G.V()];
      id = new int[G.V()];
      for(int s=0; s<G.V(); s++)
         if (!visited(s)) { dfs(G,s,-1); count++; }
   }
   private void dfs(Graph G, int i, int k) {
        visited[i] = true;
        id[i] = count;
        for (int j : G.adj(i))
            if (!visited[j]) {
               p[j]=i;
               dfs(G,j,i);
            } else if((j!=k) && !hasCycle(count)) {
               cycle[count] = new Stack<Integer>();
               cycle[count].push(j);
               for (int l=i;l!=j;l=p[l])
                   cycle[count].push(l);
               cycle[count].push(j);
            }
   }
   boolean hasCycle(int c) { return (cycle[c] != null); }
   Iterable<Integer> cycle(int c) { return cycle[c]; }
   public boolean connected(int i, int j) {
      return id[i] == id[j];
   }
   public int count() { return count; }
   public int id(int v) { return id[v]; }
}
```

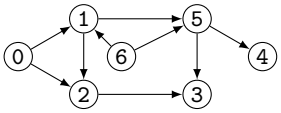
3.4 Sortarea topologică

O **sortare topologică** a unui digraf G este o enumerare a tuturor nodurilor lui G astfel încât, dacă $x \to y$ este un arc în G atunci x apare înaintea lui y în enumerare. În general, un digraf are o sortare topologică dacă şi numai dacă nu are cicluri, iar o sortare topologică este enumerarea nodurilor în postordine inversă.

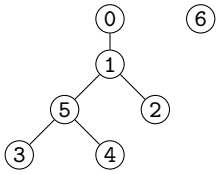
Clasa `Topologic` are un API de creare a unei sortări topologice a nodurilor unui digraf, dacă acesta este DAG.

`public class Topologic`		
`Topologic`	`(Digraph G)`	Crează un obiect pentru sortarea topologică a nodurilor unui digraf `G`
`boolean`	`isDag()`	Este aciclic?
`Iterable<Integer>`	`order()`	Nodurile sortate topologic.

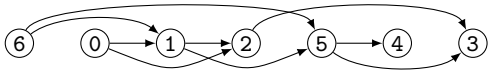
```
public class Topologic {
   private Iterable<Integer> order;   // ordinea topologică

   public Topologic(Digraph G) {
      public DCycle dc = new DCycle(G);
      if (!dc.hasCycle()) {
         DepthFirstOrder dfo = new DepthFirstOrder(G);
         order = dfo.reversePost();
      }
   }
   public Iterable<Integer> isDag() { return order != null; }
   public Iterable<Integer> order() { return order; }
}
```



Exemplul 6. Digraful reprezentat cu listele de adiacenţă `adj`$[0] = [1, 2]$, `adj`$[1] = [5, 2]$, `adj`$[2] = [3]$, `adj`$[3] =$ `adj`$[4] = []$, `adj`$[5] = [3, 4]$, `adj`$[6] = [1, 5]$ este DAG. Traversarea în adâncime pornind mereu de la cel mai mic nod nevizitat încă, produce doi arbori DFS:



Sortarea topologică produsă cu clasa `Topologic` este $[6, 0, 1, 2, 5, 4, 3]$. Dacă repoziţionăm nodurile digrafului în ordine topologică pe o linie imaginară orizontală, observăm că toate arcele digrafului sunt orientate spre dreapta:



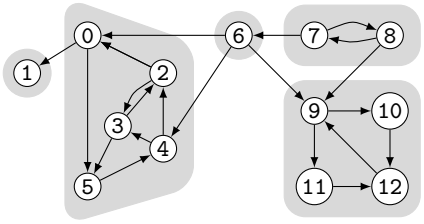
□

3.5 Detecţia componentelor tare conexe

Fie $G = (V, E)$ un digraf. Reamintim faptul că o componentă tare conexă a lui G este o clasă de echivalenţă a relaţiei de echivalenţă

$$x \sim_{sc} y \text{ dacă şi numai dacă } x \rightsquigarrow y \text{ şi } y \rightsquigarrow x.$$

De exemplu, digraful



are 5 componente tari: $\{0, 2, 3, 4, 5\}$, $\{1\}$, $\{6\}$, $\{7, 8\}$ şi $\{9, 10, 11, 12\}$.

În continuare vom descrie un algoritm de calcul al componentelor tare conexe ale unui digraf, şi implementarea acestuia în o clasă Java cu următorul API:

`public class SCC`		
	`SCC(Digraph G)`	Constructor de detecţie a componentelor tare conexe
`boolean`	`stronglyConnected(int v,int w)`	Sunt `v` şi `w` conectate tare?
`int`	`count()`	Numărul de componente tari
`int`	`id(int v)`	Identificatorul componentei tari a nodului `v` (între 0 şi `count()` − 1)

Acesta este algoritmul lui Sharir-Kosaraju care se bazează pe faptul evident că, dacă G este un digraf, atunci G şi inversul lui G au aceleaşi componente tari, şi operează în trei paşi:

1. Calculează postordinea inversă a nodurilor din graful invers G^r.
2. Traversează toate nodurile lui G în adâncime, însă în ordinea calculată în pasul 1.
3. Toate nodurile din un arbore de căutare în adâncime calculat în felul acesta formează o componentă tare conexă a lui G.

```
public class SCC {
   private boolean[] visited;
   private int[] id;
   private int count;
   public SCC(Digraph G) {
      visited = new boolean[G.V()];
      id = new int[G.V()];
      DepthFirstOrder order = new DepthFirstOrder(G.reverse());
      for(int s : order.reversePost())
          if (!visited[s]) { dfs(G,s); count++; }
   }
```

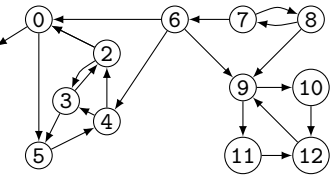
```
   private void dfs(Digraph G, int i) {
      visited[i] = true;
      id[i] = count;
      for (int j : G.adj(i))
          if (!visited[j]) dfs(G,j);
   }
   public boolean stronglyConnected(int v, int w) {
      return id[v] == id[w];
   }
   public int id(int v) { return id[v]; }
   public int count() { return count; }
}
```

Algoritmul lui Kosaraju determină componentele tari ale unui digraf G cu n noduri şi m arce în timp liniar $\Theta(n+m)$.

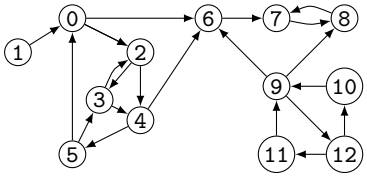
Observaţi asemănările şi diferenţele dintre implementările claselor `CC` de la pagina 8 pentru detecţia componentelor conexe şi `SCC` pentru detecţia componentelor tare conexe.

Exemplul 7. Vom ilustra cum putem calcula cu algoritmul lui Kosaraju componentele tare conexe ale digrafului

G:



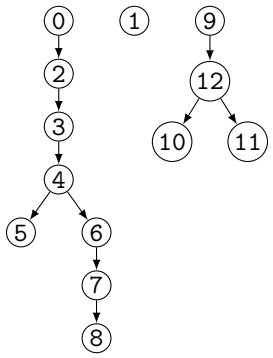
Graful invers este G^r:



Putem presupune că reprezentarea cu liste de adiacenţă a digrafului G^r este

$$\begin{array}{llll} \texttt{adj}[0] = [2,6], & \texttt{adj}[1] = [0], & \texttt{adj}[2] = [3,4], & \texttt{adj}[3] = [2,4], \\ \texttt{adj}[4] = [5,6], & \texttt{adj}[5] = [0,3], & \texttt{adj}[6] = [7], & \texttt{adj}[7] = [8], \\ \texttt{adj}[8] = [7], & \texttt{adj}[9] = [6,8,12], & \texttt{adj}[10] = [9], & \texttt{adj}[11] = [9], \\ \texttt{adj}[12] = [10,11]. & & & \end{array}$$

Traversarea în adâncime a nodurilor lui G^r enumerate în ordinea $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$ produce o pădure cu trei arbori DFS:



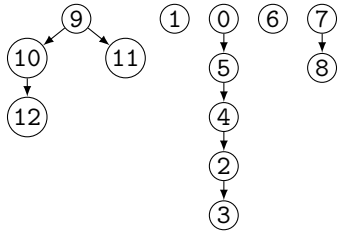
Enumerarea în postordine inversă a nodurilor grafului G^r este

$$[9, 12, 11, 10, 1, 0, 2, 3, 4, 6, 7, 8, 5].$$

Putem presupune că reprezentarea cu liste de adiacenţă a lui G este

$$\begin{array}{llll} \texttt{adj}[0] = [1, 5], & \texttt{adj}[1] = [], & \texttt{adj}[2] = [0, 3], & \texttt{adj}[3] = [2, 5], \\ \texttt{adj}[4] = [2, 3], & \texttt{adj}[5] = [4], & \texttt{adj}[6] = [0, 4, 9], & \texttt{adj}[7] = [6, 8], \\ \texttt{adj}[8] = [7, 9], & \texttt{adj}[9] = [10, 11], & \texttt{adj}[10] = [12], & \texttt{adj}[11] = [12], \\ \texttt{adj}[12] = [9]. & & & \end{array}$$

Traversarea în adâncime a nodurilor lui G enumerate în ordinea $[9, 12, 11, 10, 1, 0, 2, 3, 4, 6, 7, 8, 5]$ produce pădurea de arbori DFS



Deducem că digraful G are componentele tare conexe

$$\{9, 10, 11, 12\},\ \{1\},\ \{0, 5, 4, 2, 3\},\ \{6\} \text{ şi } \{7, 8\}.$$

3.6 Drumuri elementare de lungime maximă de la o sursă într-un DAG

Problema găsirii unor drumuri elementare de lungime maximă de la un nod sursă într-un graf este următoarea:

Se dau un graf G cu n noduri, m muchii, şi un nod sursă $s \in V(G)$.

Să se găsească drumuri elementare de lungime maximă la $s \overset{\pi}{\rightsquigarrow} x$ pentru toate nodurile din $\{x \in V(G) \mid s \rightsquigarrow x\}$.

Un drum elementar de la s la un nod x este de forma $[x_1, \ldots, x_r]$ cu $x_1 = s, x_r = x$ şi $\langle x_1, \ldots, x_r \rangle$ o r-permutare de noduri din $V(G)$. O metodă naivă şi foarte ineficientă de rezolvare a acestei probleme ar putea fi ca, pentru toate cele $n-1$ noduri $x \in V(G) - \{s\}$, pornind de la $r = n$ la $r = 2$:

- Se generează toate cele $P(n-2, r-2)$ permutări $\langle x_1, \ldots, x_r \rangle$ cu $x_1 = s$ şi $x_r = x$ şi se verifică dacă $[x_1, \ldots, x_r]$ este un drum în G.
- Primul drum găsit în acest fel este un drum elementar de lungime maximă de la s la x.

Pentru grafuri arbitrare, această problemă este NP-completă. Dacă G este DAG, problema se poate rezolva uşor cu ajutorul sortării topologice:

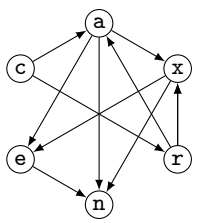
1. Se calculează o sortare topologică $[x_0, x_1, \ldots, x_k]$ a nodurilor la care se poate ajunge din s. Observaţi că $x_0 = s$.
2. Pentru $0 \leq i \leq k$, fie
   - $\mathtt{L}[i]$: lungimea celui mai lung drum elementar de s la x_i, şi
   - $\mathtt{d}[i]$: o coadă care reţine nodurile unui drum elementar de lungime maximă de la s la x_i.

   Valorile lui $\mathtt{L}[i]$ şi $\mathtt{d}[i]$ se pot calcula astfel:
   - Se setează valorile iniţiale $\mathtt{L}[0] = 0$, $\mathtt{d}[0] = [s]$, $\mathtt{L}[i] = -\infty$ şi $\mathtt{d}[i] = []$ pentru $1 \leq i \leq k$.
   - Se actualizează valorile iniţiale în felul următor:

     for $i = 0$ to k **do**
       for $j \in \mathtt{adj}[i]$ **do**
         if $\mathtt{L}[j] < \mathtt{L}[i] + 1$ **then**
           $\mathtt{L}[j] = \mathtt{L}[i] + 1$;
           $\mathtt{d}[j] = \mathtt{d}[i]$ cu nodul j adăugat la sfârşit;
         endif
3. În final, fiecare element $\mathtt{d}[i]$ va conţine nodurile unui drum elementar de lungime maximă de la s la x_i.

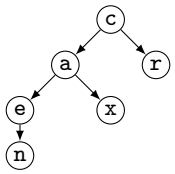
Algoritmul se termină în $k+m+1$ paşi, deci are complexitate liniară $O(n+m)$, unde n este numărul de noduri şi m este numărul de muchii.

Exemplul 8. Să se calculeze drumuri elementare de lungime maximă de la nodul sursă `c` în digraful



reprezentat cu listele de adiacenţă
$\mathtt{adj}[\mathtt{a}] = [\mathtt{e}, \mathtt{x}, \mathtt{n}]$, $\mathtt{adj}[\mathtt{c}] = [\mathtt{a}, \mathtt{r}]$, $\mathtt{adj}[\mathtt{x}] = [\mathtt{e}, \mathtt{n}]$,
$\mathtt{adj}[\mathtt{r}] = [\mathtt{a}, \mathtt{x}]$, $\mathtt{adj}[\mathtt{e}] = [\mathtt{n}]$, $\mathtt{adj}[\mathtt{n}] = []$.

Arborele de căutare în adâncime cu sursa c este

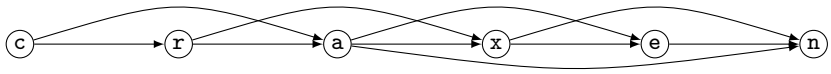


iar traversarea în postordine inversă a acestui arbore produce sortarea topologică $[\mathtt{c}, \mathtt{r}, \mathtt{a}, \mathtt{x}, \mathtt{e}, \mathtt{n}]$. În acest exemplu avem $k = 5$.

Valorile iniţiale ale tablourilor L şi d sunt

$\mathtt{L}[0] = 0, \mathtt{d}[0] = [\mathtt{c}]$, şi
$\mathtt{L}[i] = -\infty$ şi $\mathtt{d}[i] = []$ pentru $1 \leq i \leq 5$.

Evoluţia valorilor elementelor tabloului d este ilustrată mai jos:



d[0]	d[1]	d[2]	d[3]	d[4]	d[5]
[c]	[]	[]	[]	[]	[]
[c]	[c, r]	[c, a]	[]	[]	[]
[c]	[c, r]	[c, r, a]	[c, r, x]	[]	[]
[c]	[c, r]	[c, r, a]	[c, r, a, x]	[c, r, a, e]	[c, r, a, n]
[c]	[c, r]	[c, r, a]	[c, r, a, x]	[c, r, a, x, e]	[c, r, a, x, n]
[c]	[c, r]	[c, r, a]	[c, r, a, x]	[c, r, a, x, e]	[c, r, a, x, e, n]

Suport de curs

Implementări java ale claselor prezentate în acest curs sunt disponibile pe site-ul

`https://staff.fmi.uvt.ro/~mircea.marin/lectures/EduGraph/`

4 Concluzii

Algoritmii de traversare au fost concepuţi ca, pentru un graf şi un nod s să determine mulţimea de noduri la care se poate ajunge din s, precum şi căi de legătură de la s la aceste noduri. Cei mai cunoscuţi algoritmi de traversare sunt traversarea în adâncime (DFS) şi traversarea în lăţime (BFS).

- Traversarea în lăţime a fost concepută să găsească drumuri de lungime minimă de la un nod s la toate nodurile la care se poate ajunge din s.
- Traversarea în adâncime a tuturor nodurilor induce trei relaţii de ordine pe noduri: preordine, postordine, şi postordine inversă.

- Traversarea în adâncime are aplicaţii în
  1. Detecţia componentelor conexe
  2. Detecţia ciclurilor
  3. Sortarea topologică
  4. Determinarea componentelor tare conexe
  5. Determinarea unor drumuri elementare de lungime maximă în digrafuri aciclice.
  6. Determinarea nodurilor critice, punţilor şi componentelor biconexe ale unui graf neorientat (vezi Exerciţiul 8).