École Polytechnique

CSC_41011

Les bases de la programmation et de l'algorithmique

Jean-Christophe Filliâtre

graphes (1/2)

en suivant les liens sur `fr.wikipedia.org`,

peut-on aller de la page

Pomme

Pour les articles homonymes, voir Pomme (homonymie).

La **pomme** est le fruit du pommier. Elle est comestible et a un goût sucré ou acidulé selon les variétés. Elle fait partie des fruits les plus consommés dans le monde.

Généralement, on distingue trois types de pommes alimentaires :

- les pommes à cidre ;
- les pommes de table ou pommes à couteau ;
- les pommes à cuire qui appartiennent à un des deux premiers types mais supportent bien la cuisson.

Pomme rouge Fuji

à la page

Linguistique

La **linguistique** est la discipline s'intéressant à l'étude du langage. Elle se distingue de la grammaire, dans la mesure où elle n'est pas prescriptive mais descriptive.

On trouve des témoignages de réflexions sur le langage dès l'Antiquité avec des philosophes comme Platon. Cependant il faut attendre le xx[e] siècle pour voir se dégager une approche scientifique autour des faits de langues. Ferdinand de Saussure est réputé avoir grandement contribué à cet effort de rationalisation, notamment avec son influent *Cours de linguistique générale* (1916) qui est devenu un classique dans ce domaine [1,2] et "a imposé la conception structurale du langage qui domine largement la linguistique contemporaine en dépit des conflits d'écoles."[3]

on peut répondre à cette question avec un **parcours de graphe**
qui s'écrit en cinq lignes de code

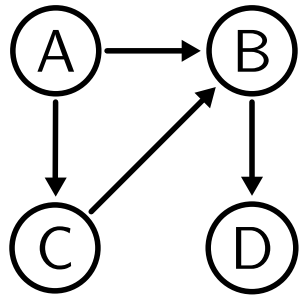
Définition

Un **graphe orienté** est un ensemble de **sommets** V
et un ensemble d'**arcs** $E \subseteq V \times V$.

exemple

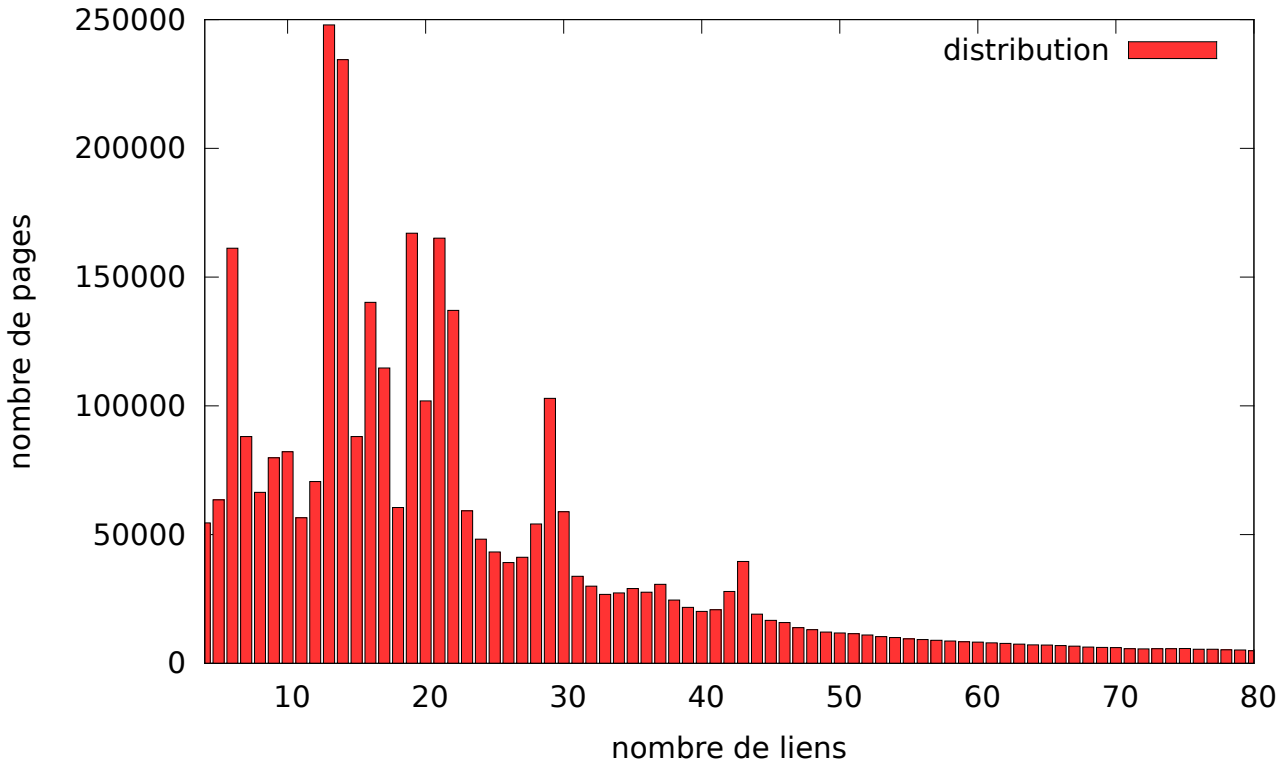
$$V = \{A, B, C, D\}$$

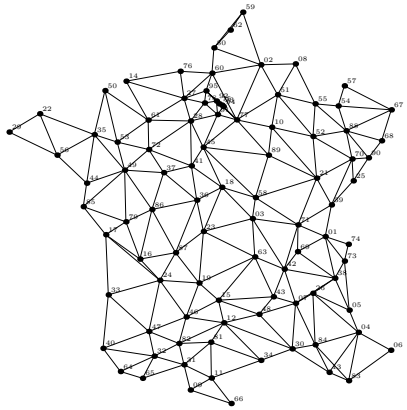
$$E = \{(A, B), (A, C), (C, B), (B, D)\}$$



l'ensemble des pages de `fr.wikipedia.org`

$$|V| = 6\,042\,266$$
$$|E| = 142\,747\,529$$





réseau routier

villes / routes entre ces villes

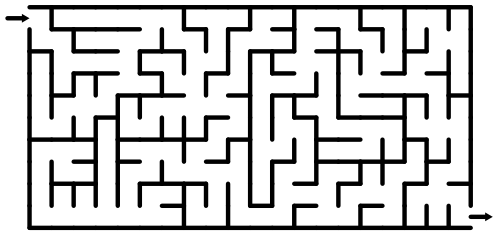
réseau informatique

machines / connexions entre ces machines

réseau social

personnes / contacts

mathématiciens / coauteurs



labyrinthe

salles / portes

carte

régions, pays, etc. / contiguïté

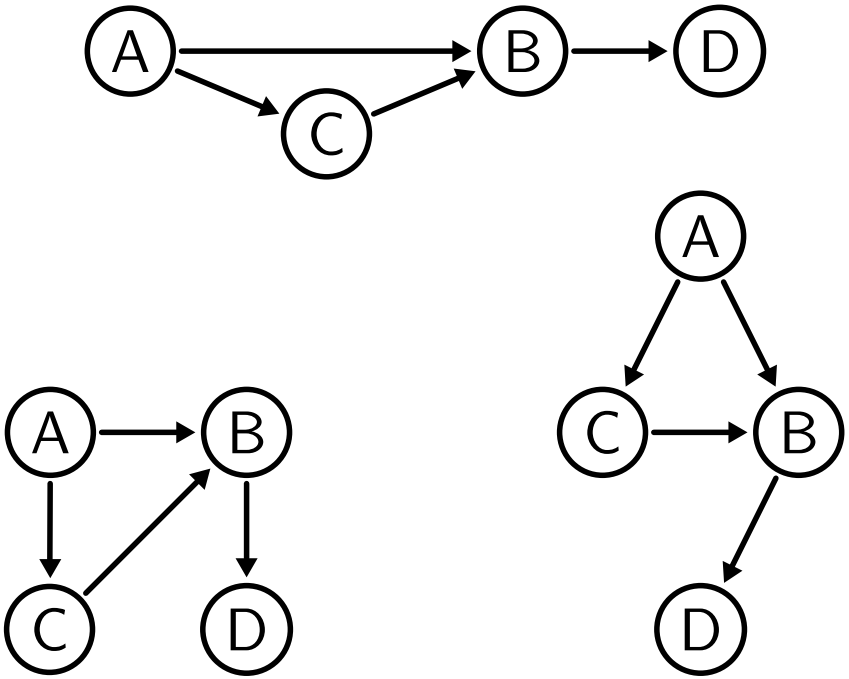
jeu

configurations / coups valides

(un ou plusieurs joueurs)

(graphe possiblement infini)

peu importe le dessin

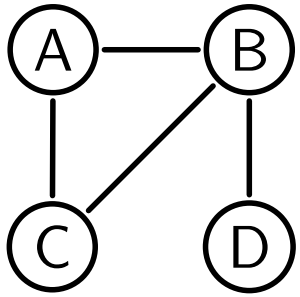


seuls importent V et E

si E est symétrique, on parle de **graphe non orienté**

$$V = \{A, B, C, D\}$$

$$E = \{ (A,B), (B,A), (A,C), (C,A), (C,B), (B,C), (B,D), (D,B) \}$$



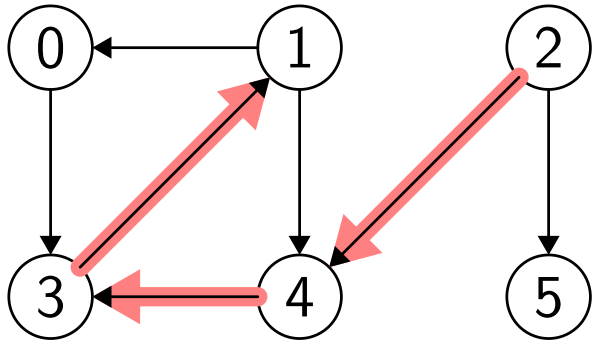
(cours d'aujourd'hui : uniquement des graphes orientés)

on note $u \to v$ pour $(u, v) \in E$

Définition

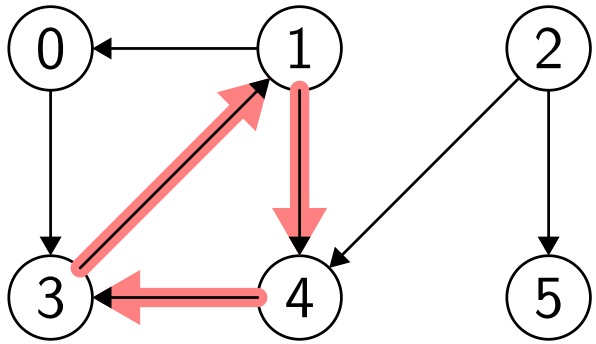
Un **chemin** de longueur n du sommet x_0 au sommet x_n est une séquence $x_0 \to x_1 \to \cdots \to x_{n-1} \to x_n$. On note $x_0 \to^\star x_n$.

Un **chemin simple** est un chemin sans répétition d'arc.

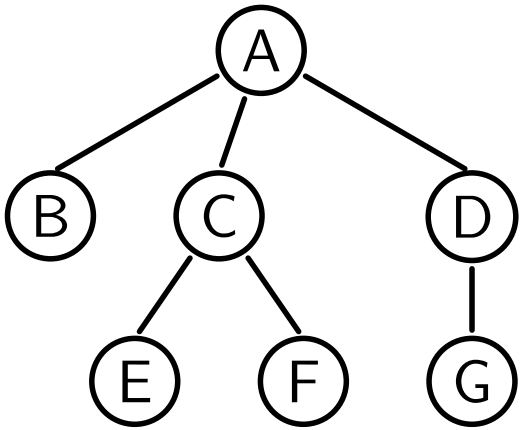


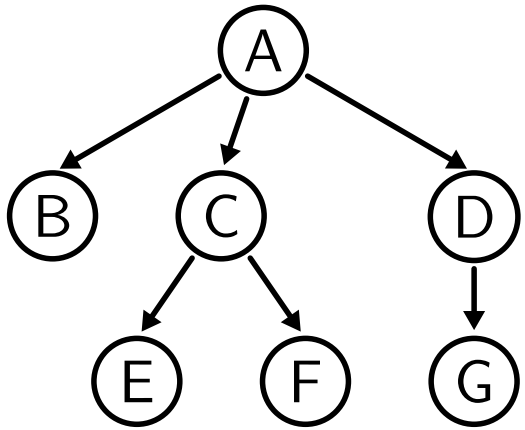
Définition

Un **cycle** est un chemin simple de u à u de longueur ≥ 2.



un arbre est un cas particulier de graphe

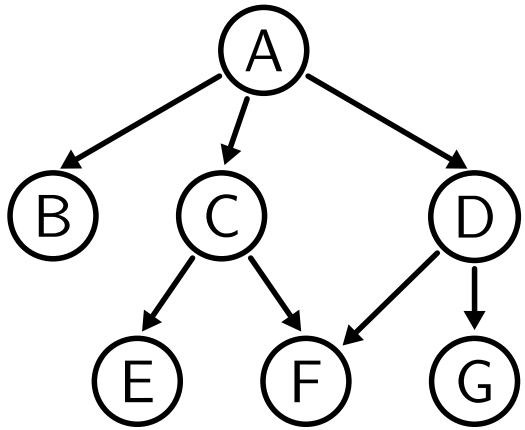




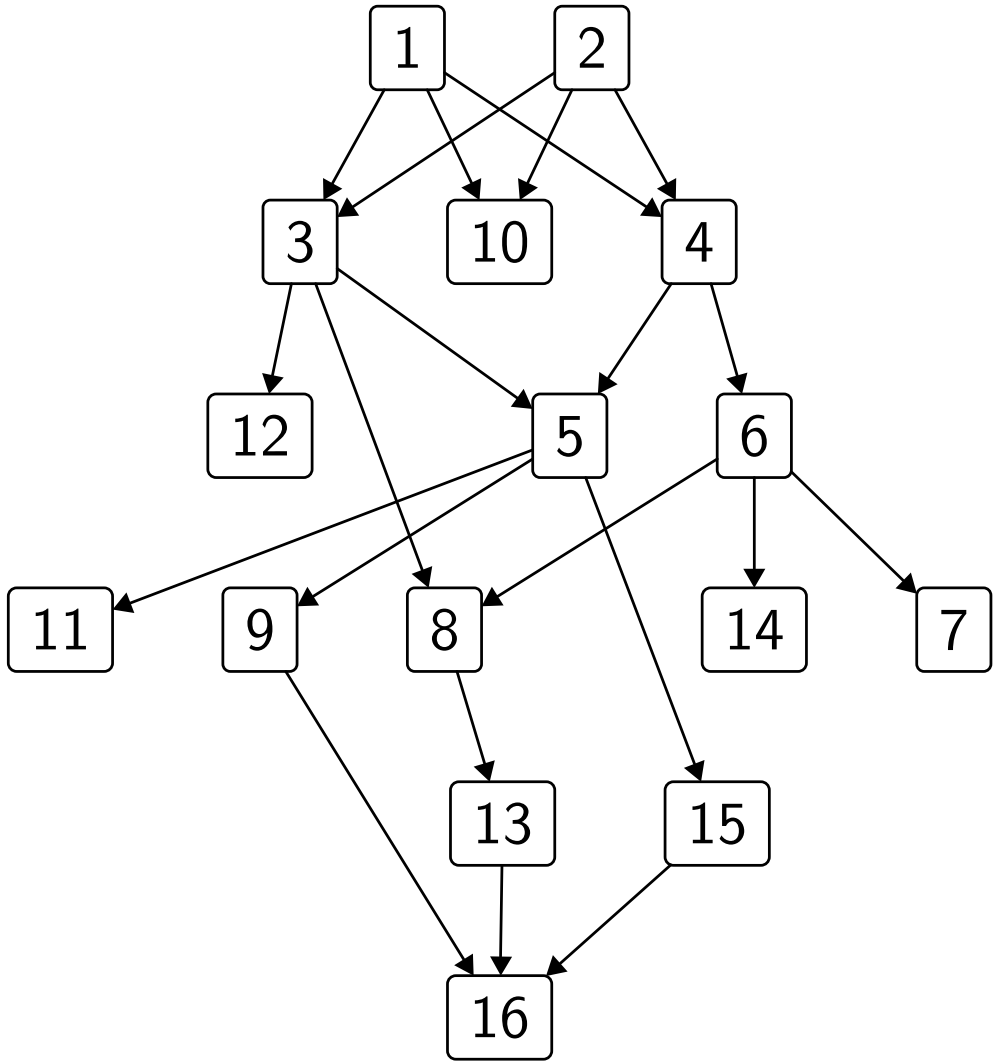
en particulier, c'est un graphe sans cycle

Définition

Un graphe orienté acyclique est appelé un **DAG**
(pour *Directed Acyclic Graph*).



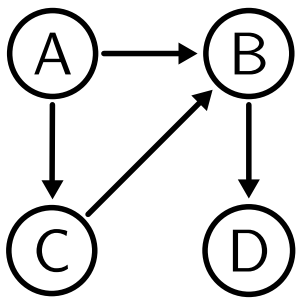
dépendances entre les chapitres du poly de CSC_41011



représentation en machine

```
class Graph<V> { // V est le type des sommets
          Graph()                 // graphe vide
     void addVertex(V u)          // ajouter un sommet u
     void addEdge(V u, V v)       // ajouter l'arc u->v
     void removeEdge(V u, V v) // supprimer u->v

  boolean hasEdge(V u, V v)       // est-ce que u->v existe ?
   Set<V> vertices()              // tous les sommets
   Set<V> successors(V u)         // tous les voisins de u

          ...
}
```

si $V = \{0, 1, \ldots, N-1\}$, on peut représenter un graphe par sa **matrice d'adjacence**



	A	B	C	D
A	false	true	true	false
B	false	false	false	true
C	false	true	false	false
D	false	false	false	false

```
class AdjMatrix {
  private int N;                    // les sommets sont 0,...,N-1
  private boolean[][] m;
  ...
```

solution 1 : matrice d'adjacence

un graphe à N sommets, sans arc

```
AdjMatrix(int N) {
  this.N = N;
  this.m = new boolean[N][N]; // initialisé à false
}
```

tester/ajouter/supprimer un arc est immédiat

```
boolean hasEdge(int u, int v) { return this.m[u][v];  }
   void addEdge(int u, int v) { this.m[u][v] = true;  }
void removeEdge(int u, int v) { this.m[u][v] = false; }
```

(bien sûr, il faudrait tester $0 \leq u, v < N$ proprement)

parcourir tous les sommets

```
for (int u = 0; u < N; u++)
  ...
```

parcourir tous les voisins de `u`

```
for (int v = 0; v < N; v++)
  if (this.m[u][v])
    ...
```

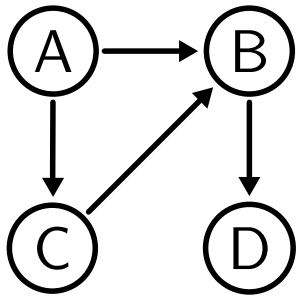
- en **espace** : $\approx N^2$ octets

  exemple : $N = 1024 \Rightarrow 1$ Mo

- en **temps**

ajouter $u \to v$	supprimer $u \to v$	tester $u \to v$	parcourir tous les sommets	parcourir les voisins de u
$O(1)$	$O(1)$	$O(1)$	$O(N)$	$O(N)$

à chaque sommet, on associe la **liste de ses voisins**



$A \mapsto B, C$
$B \mapsto D$
$C \mapsto B$
$D \mapsto$

par exemple avec une table de hachage

```
class AdjList<V> {
  private HashMap<V, LinkedList<V>> adj;
```

```
AdjList() {
  this.adj = new HashMap<V, LinkedList<V>>();
}
```

```
void addVertex(V u) {
  this.adj.put(u, new LinkedList<V>()); // O(1)
}
void addEdge(V u, V v) {
  this.adj.get(u).add(v);                // O(1)
}
boolean hasEdge(V u, V v) {
  return this.adj.get(u).contains(v);   // plus cher (liste)
}
void removeEdge(V u, V v) {
  this.adj.get(u).remove(v);   // plus cher (liste)
}
```

parcourir tous les sommets

```
for (V u: this.adj.keySet())
  ...
```

parcourir tous les voisins de u

```
for (V v: this.adj.get(u))  // moins cher
  ...
```

- en **espace** : $\approx 48V + 40E$ octets

  exemple : $V = 1024, E = 2V \Rightarrow 128$ ko

- en **temps**

ajouter $u \to v$	supprimer $u \to v$	tester $u \to v$	parcourir tous les sommets	parcourir les voisins de u
$O(1)$	$O(\delta(u))$	$O(\delta(u))$	$O(V)$	$O(\delta(u))$

où $\delta(u)$ est le nombre de voisins de u

le poly propose une alternative utilisant
un **ensemble** plutôt qu'une liste

```
class AdjSet<V> {
  private HashMap<V, HashSet<V>> adj;
```

maintenant optimal

ajouter $u \to v$	supprimer $u \to v$	tester $u \to v$	parcourir tous les sommets	parcourir les voisins de u
$O(1)$	$O(1)$	$O(1)$	$O(V)$	$O(\delta(u))$

- matrice d'adjacence
  - adapté aux graphes denses
    (plus compact que les listes/ensembles pour $N < 150$)
  - nécessite $V = \{0, 1, \ldots, N-1\}$

- listes/ensembles d'adjacence
  - adapté aux graphes creux
  - V n'a pas besoin d'être fixé

aujourd'hui, on cherche à répondre aux questions suivantes

- quels sont tous les sommets atteignables depuis u ?
- existe-t-il un chemin de u à v ?
- quel est le plus court chemin de u à v ?
- existe-t-il un cycle partant de u ?

on y répond avec des algorithmes de **parcours** de graphe

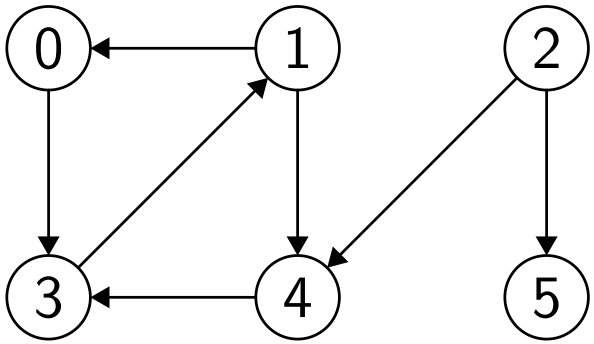
parcours en profondeur

le **parcours en profondeur** (en anglais *depth-first search* ou DFS)

applique le principe du rebroussement (*backtracking*) :

- tant qu'on peut progresser en suivant un arc, on le fait
- sinon, on fait machine arrière

il peut exister des cycles



pour éviter de « tourner en rond », on doit **marquer** les sommets déjà vus

on choisit pour cela une table de hachage

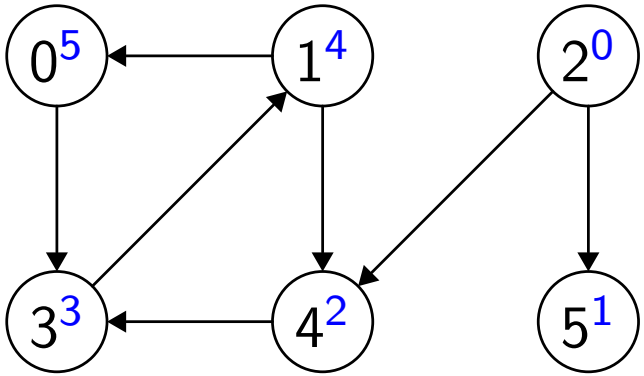
```
class DFS<V> {
  private Graph<V> g;
  private HashSet<V> visited;

  DFS(Graph<V> g) {
    this.g = g;
    this.visited = new HashSet<V>();
  }
```

le parcours en profondeur est écrit récursivement

```
void dfs(V v) {
  // si v a déjà été atteint, on ne fait rien
  if (this.visited.contains(v)) return;
  // sinon, on le marque comme atteint
  this.visited.add(v);
  // puis on lance un parcours sur ses voisins
  for (V w : this.g.successors(v))
    dfs(w);
}
```

(il est crucial de marquer `v` **avant** de considérer ses voisins)

en partant du sommet 2

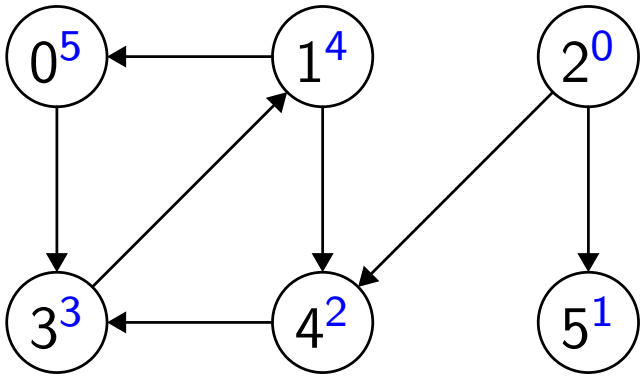


`dfs(2)`

`visited :`
2

(les exposants indiquent l'ordre de découverte)

en partant du sommet 2

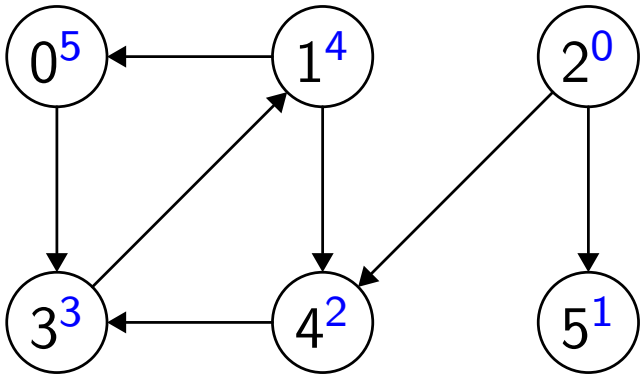



```
dfs(2)
  dfs(5)
```

```
visited :
 2, 5
```

(les exposants indiquent l'ordre de découverte)

en partant du sommet 2

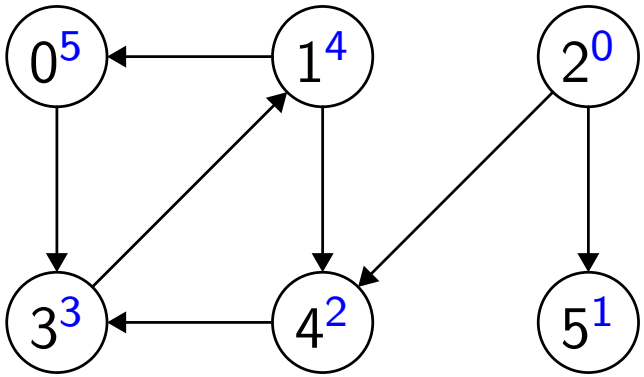



```
dfs(2)
  dfs(4)
```

`visited` :
2, 5, 4

(les exposants indiquent l'ordre de découverte)

en partant du sommet 2

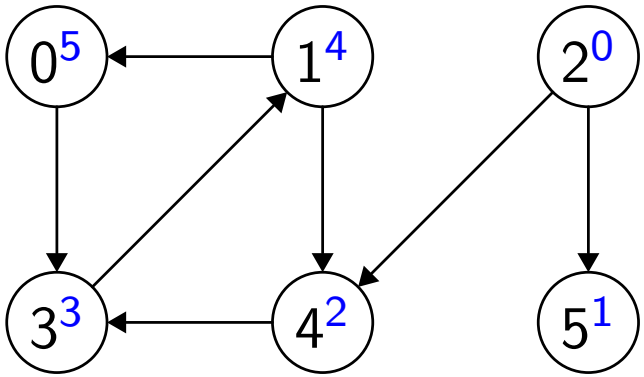



```
dfs(2)
  dfs(4)
    dfs(3)
```

```
visited :
 2, 5, 4, 3
```

(les exposants indiquent l'ordre de découverte)

en partant du sommet 2

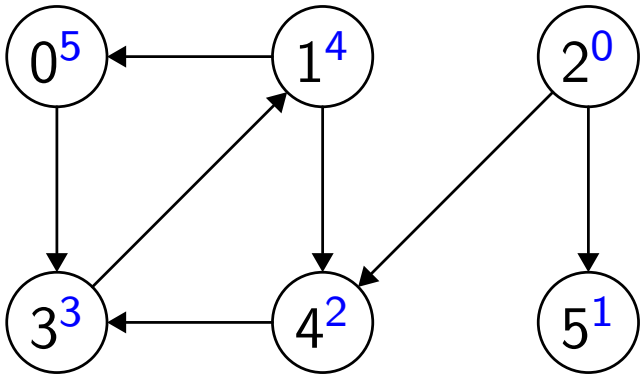



```
dfs(2)
  dfs(4)
    dfs(3)
      dfs(1)
```

```
visited :
 2, 5, 4, 3, 1
```

(les exposants indiquent l'ordre de découverte)

en partant du sommet 2

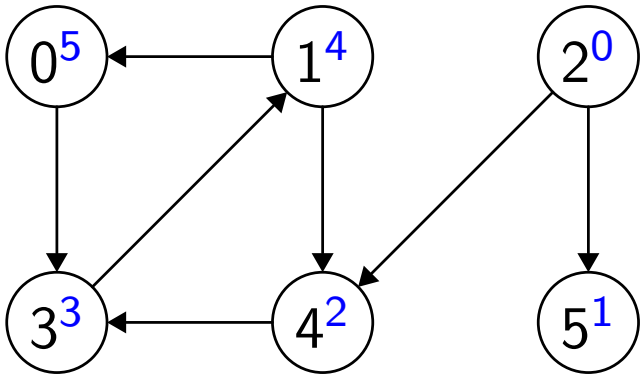



```
dfs(2)
  dfs(4)
    dfs(3)
      dfs(1)
        dfs(4)
```

```
visited :
 2, 5, 4, 3, 1
```

(les exposants indiquent l'ordre de découverte)

en partant du sommet 2

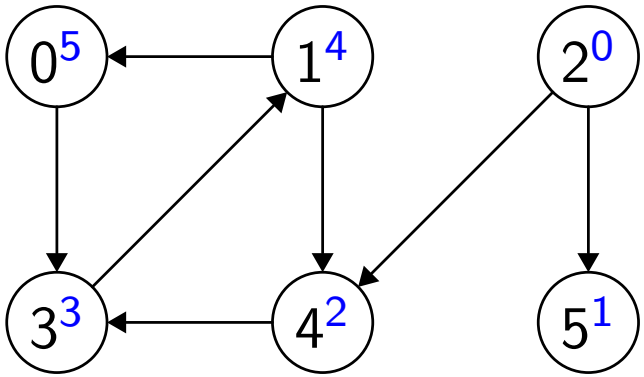



```
dfs(2)
  dfs(4)
    dfs(3)
      dfs(1)
        dfs(0)
```

```
visited :
 2, 5, 4, 3, 1, 0
```

(les exposants indiquent l'ordre de découverte)

en partant du sommet 2

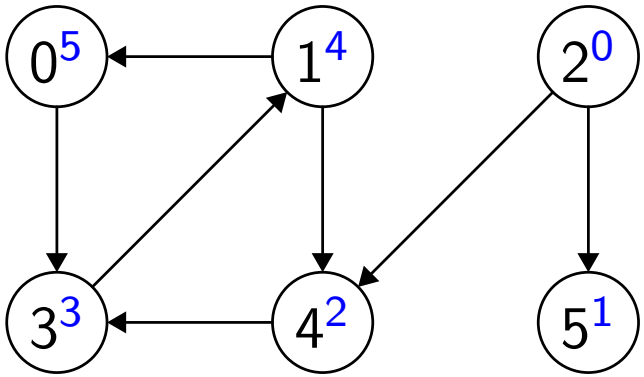



```
dfs(2)
  dfs(4)
    dfs(3)
      dfs(1)
        dfs(0)
          dfs(3)
```

```
visited :
 2, 5, 4, 3, 1, 0
```

(les exposants indiquent l'ordre de découverte)

en partant du sommet 2

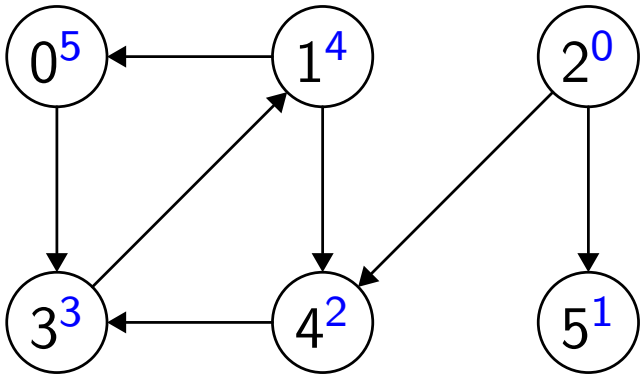



```
dfs(2)
  dfs(4)
    dfs(3)
      dfs(1)
        dfs(0)
```

```
visited :
 2, 5, 4, 3, 1, 0
```

(les exposants indiquent l'ordre de découverte)

en partant du sommet 2

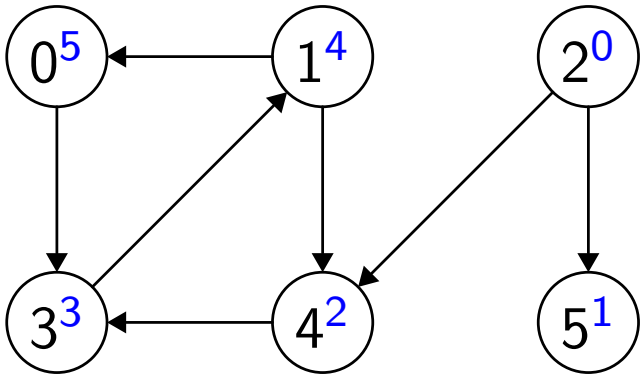



```
dfs(2)
  dfs(4)
    dfs(3)
      dfs(1)
```

```
visited :
 2, 5, 4, 3, 1, 0
```

(les exposants indiquent l'ordre de découverte)

en partant du sommet 2

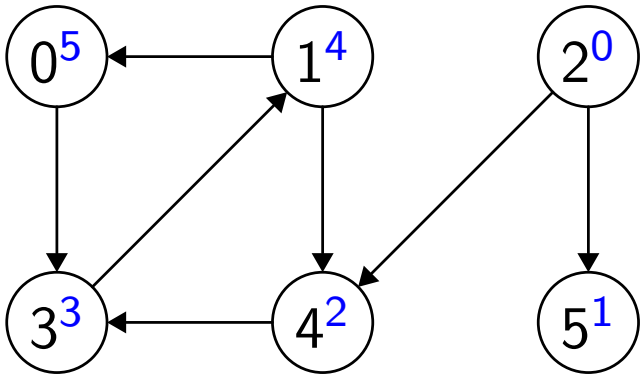



```
dfs(2)
  dfs(4)
    dfs(3)
```

```
visited :
 2, 5, 4, 3, 1, 0
```

(les exposants indiquent l'ordre de découverte)

en partant du sommet 2

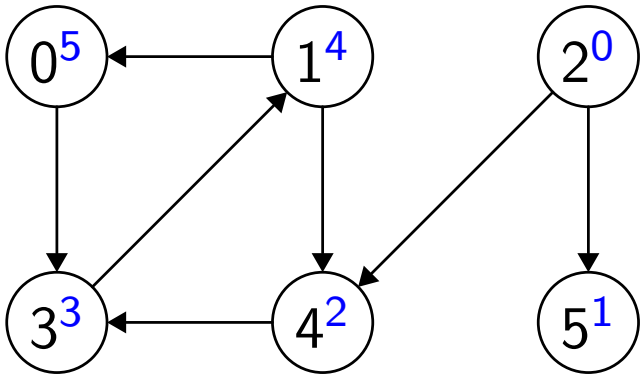



```
dfs(2)
  dfs(4)
```

```
visited :
 2, 5, 4, 3, 1, 0
```

(les exposants indiquent l'ordre de découverte)

en partant du sommet 2



`dfs(2)`

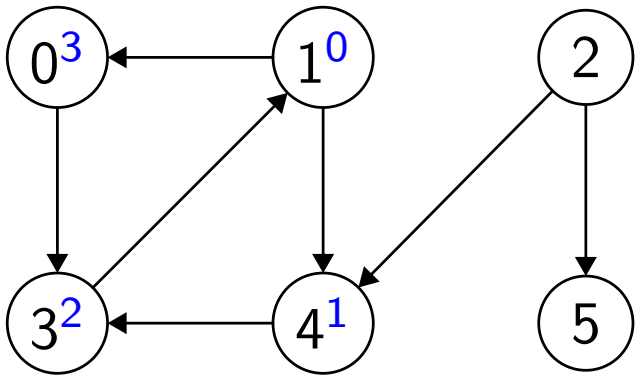
`visited` :
2, 5, 4, 3, 1, 0

(les exposants indiquent l'ordre de découverte)

chaque arc $\texttt{v} \rightarrow \texttt{w}$ est considéré au plus une fois,
la première fois que `dfs` est appelée sur `v`

d'où une complexité $O(E)$, optimale

en partant cette fois du sommet 1



$\texttt{visited} = \{1, 4, 3, 0\}$

il n'y a pas de chemin $1 \rightarrow^\star 2$, ni de chemin $1 \rightarrow^\star 5$

`dfs(u)` détermine l'ensemble des sommets accessibles depuis la source `u`

ce sont exactement les sommets dans `visited` au final

Propriété

Après `dfs(u)`, si $u \to^\star v$ alors $v \in$ `visited`.

par récurrence sur la longueur du chemin

- longueur 0 : alors $v = u$ et c'est immédiat
- longueur $n > 0$: on a $u \to^{n-1} w \to v$
  par hypothèse de récurrence, $w \in$ `visited`
  $\Rightarrow$ `dfs(w)` a été appelé
  $\Rightarrow$ l'arc $w \to v$ a été examiné
  $\Rightarrow$ `dfs(v)` a été appelé
  $\Rightarrow v \in$ `visited`

Propriété

L'appel à `dfs(u)` n'ajoute à `visited` que des sommets `v` tels que $u \rightarrow^\star v$.

par récurrence sur le nombre d'appels à `dfs`

- cas de base
  `dfs(u)` ajoute `u` à `visited` (éventuellement) et on a bien $u \rightarrow^\star u$

- sinon : `dfs(u)` appelle `dfs(w)` avec $u \rightarrow w$
  par hypothèse de récurrence, `dfs(w)` n'ajoute que des sommets tels que $w \rightarrow^\star v$, donc que des sommets tels que $u \rightarrow^\star v$

on peut répondre facilement à la question

existe-t-il un chemin de `u` à `v` ?

```
boolean existsPath(V u, V v) {
  this.visited.clear();
  dfs(u);
  return this.visited.contains(v);
}
```

exhiber le chemin

et si on veut exhiber le chemin de u à v qui a été trouvé ?

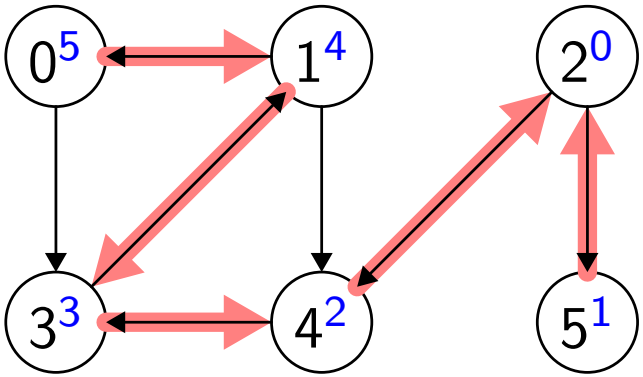
solution : au lieu d'un simple booléen, `visited`(v) donne le sommet u tel que l'arc $u \to v$ a permis d'atteindre v

on remplace

```
  private HashSet<V> visited;
```

par

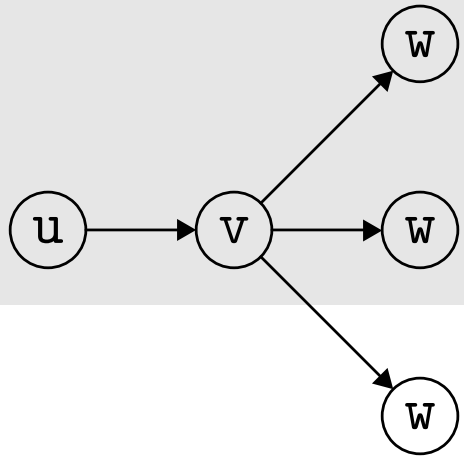
```
  private HashMap<V, V> visited;
```



$$\texttt{visited} = \{ \\ 2 \mapsto \texttt{null}, \\ 5 \mapsto 2, \\ 4 \mapsto 2, \\ 3 \mapsto 4, \\ 1 \mapsto 3, \\ 0 \mapsto 1, \\ \}$$

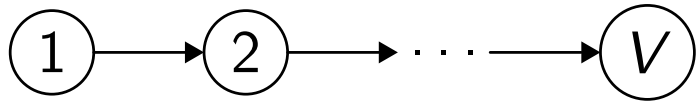
les arcs dans `visited` forme un arbre, appelé **arbre couvrant**
(voir poly page 203)

```
// DFS à partir du sommet v, en arrivant du sommet u
void dfs(V u, V v) {
  if (this.visited.containsKey(v)) return;
  this.visited.put(v, u);
  for (V w : this.g.successors(v))
    dfs(v, w);
}
```




```
// DFS à partir du sommet v
void dfs(V v) {
  dfs(null, v);
}
```

la taille de pile de `dfs` peut atteindre $O(V)$

sur un graphe de la forme 

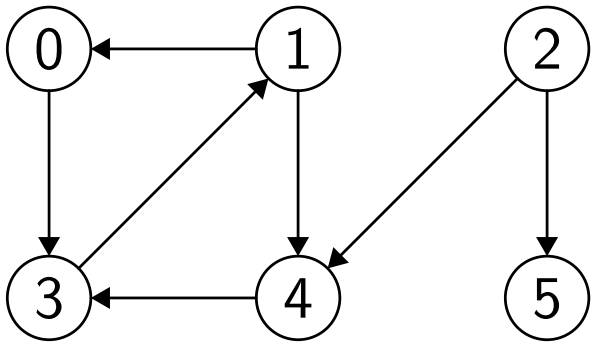
d'où `StackOverflowError` pour V de l'ordre de quelques milliers

on peut cependant se passer de récursivité, avec une **pile explicite**

```
void dfs(V v) {
  Stack<V> stack = new Stack<V>();
  stack.push(v);
  while (!stack.isEmpty()) {
    v = stack.pop();
    if (this.visited.contains(v)) continue;
    this.visited.add(v);
    for (V w : this.g.successors(v))
      stack.push(w);
  }
}
```

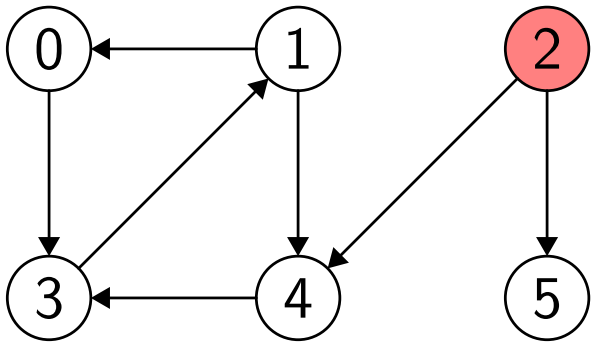
(les voisins `w` ne sont pas examinés dans le même ordre,
mais ce n'est pas grave)

exemple

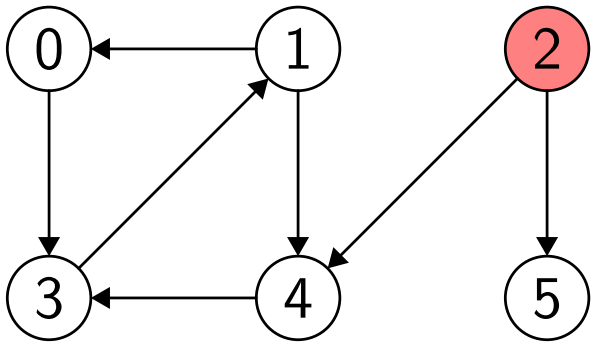


visited	stack
	2

exemple

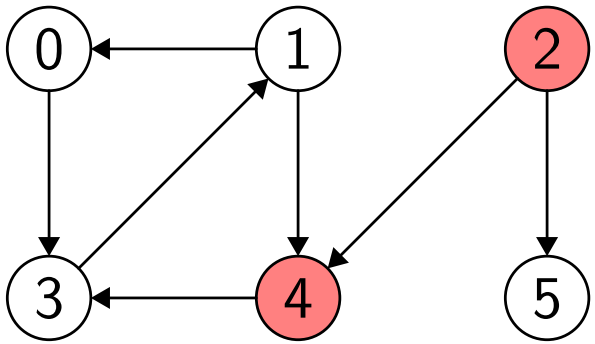


visited	stack
2	

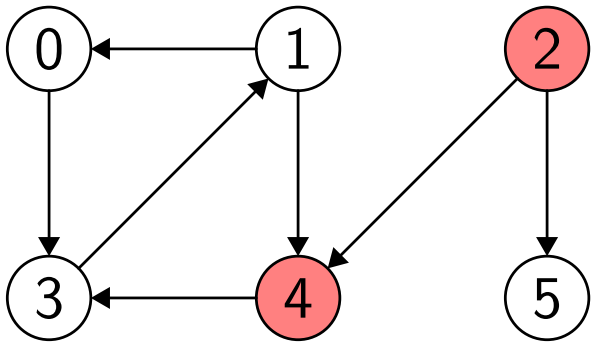


visited	stack
2	5
	4

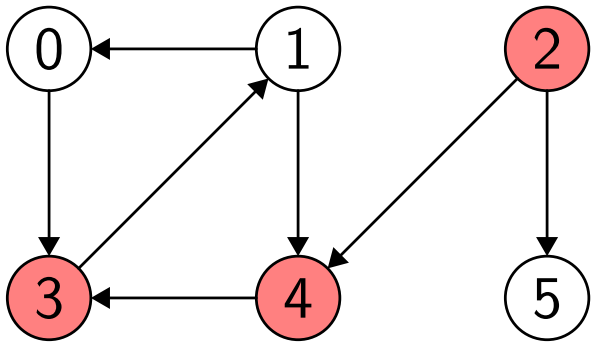
exemple



visited	stack
2	5
4	

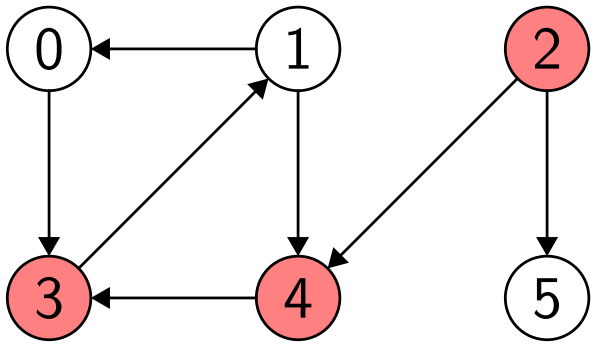


visited	stack
2	5
4	3



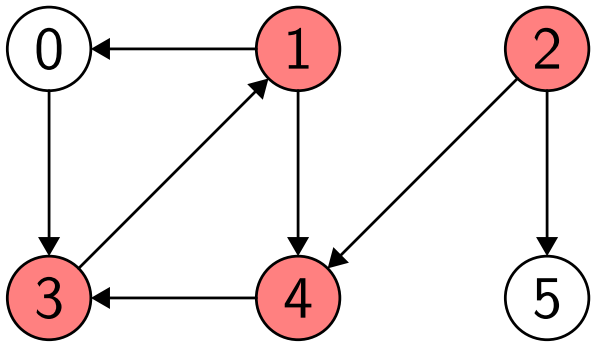
visited	stack
2	5
4	
3	

exemple

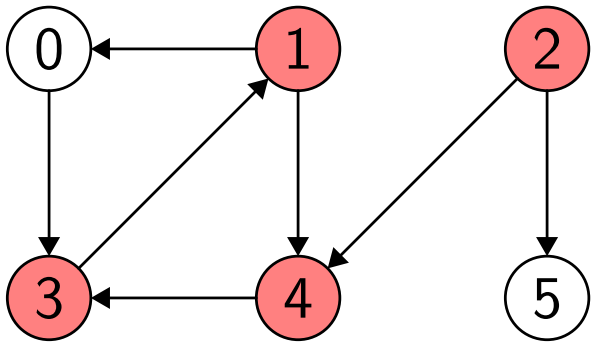


visited	stack
2	5
4	1
3	

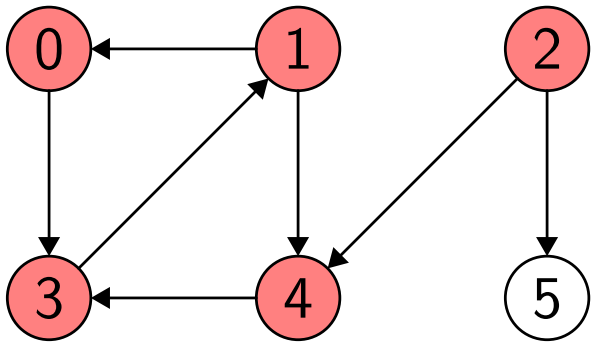
exemple



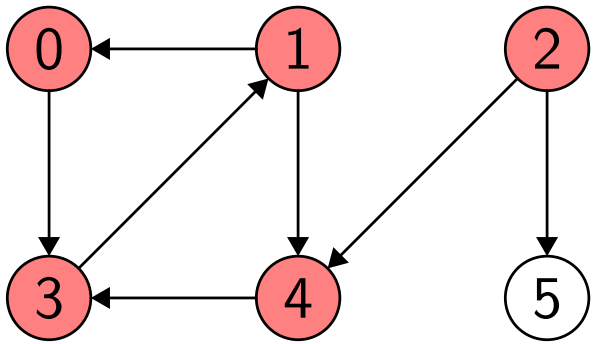
visited	stack
2	5
4	
3	
1	



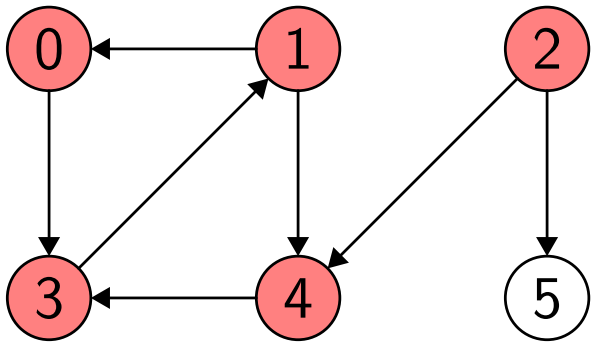
visited	stack
2	5
4	4
3	0
1	



visited	stack
2	5
4	4
3	
1	
0	

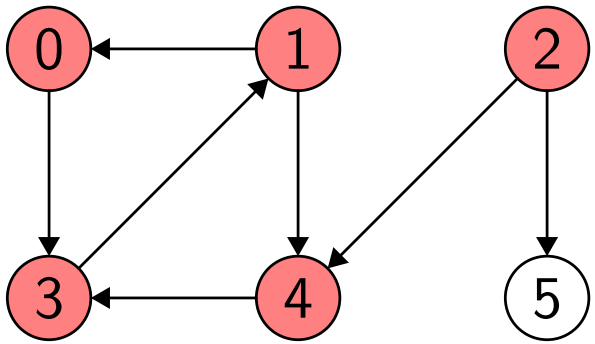


visited	stack
2	5
4	4
3	3
1	
0	



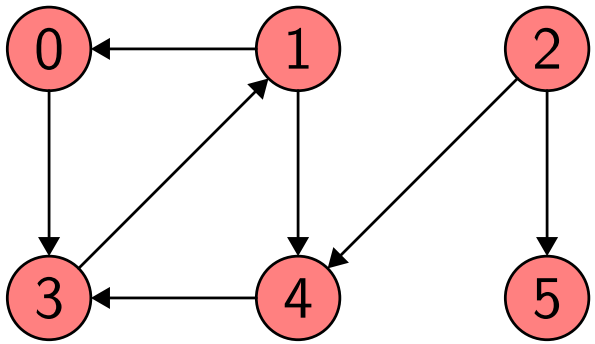
visited	stack
2	5
4	4
3	
1	
0	

exemple



visited	stack
2	5
4	
3	
1	
0	

exemple



visited	stack
2	
4	
3	
1	
0	
5	

on sait donc répondre à la question

existe-t-il un chemin de `u` à `v` ?

et même le construire, le cas échéant

pour répondre à la question

existe-t-il un **cycle** partant de u ?

il faut modifier le parcours en profondeur

deux états (non atteint / atteint) ne suffisent plus
il faut **trois états** (non atteint / en cours de visite / terminé)

cf exercice 110 du poly

quel est le **plus court** chemin de u à v ?
(en nombre d'arcs)

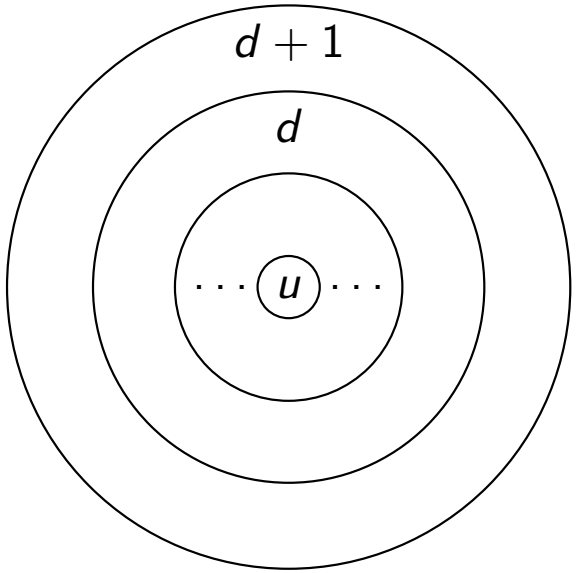
parcours en largeur

parcourir le graphe ≪ en cercles concentriques ≫ à partir de u

- d'abord les sommets à distance 1
- puis les sommets à distance 2
- etc.

à chaque instant, on a

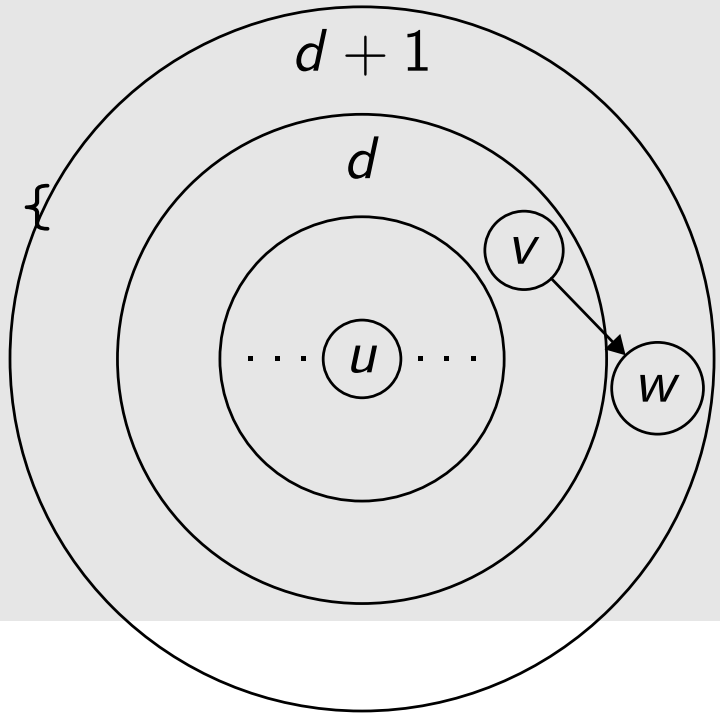
- des sommets à distance d en cours d'exploration
- des sommets à distance $d + 1$ à explorer ensuite



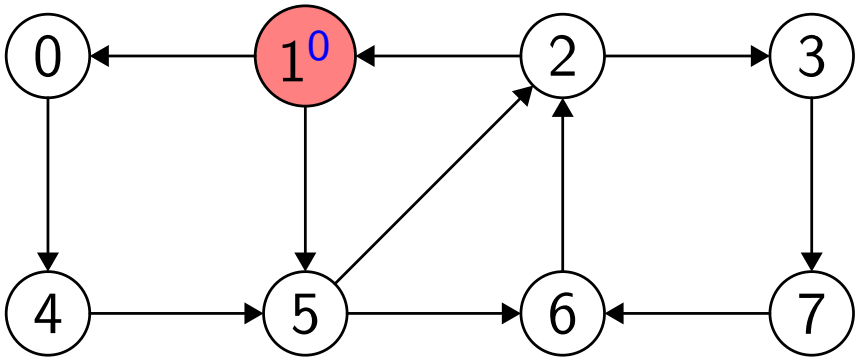
d'où l'idée d'utiliser une **file**

```
class BFS<V> {          // en anglais, Breadth-First Search
  private Graph<V> g;
  private HashSet<V> visited;

  BFS(Graph<V> g) {
    this.g = g;
    this.visited = new HashSet<V>();
  }
```

```
void bfs(V u) {
  Queue<V> q = new LinkedList<V>(); // une file
  q.add(u);
  visited.add(u);
  while (!q.isEmpty()) {
    V v = q.poll();
    for (V w : this.g.successors(v))
      if (!this.visited.contains(w)) {
        q.add(w);
        this.visited.add(w);
      }
  }
}
```

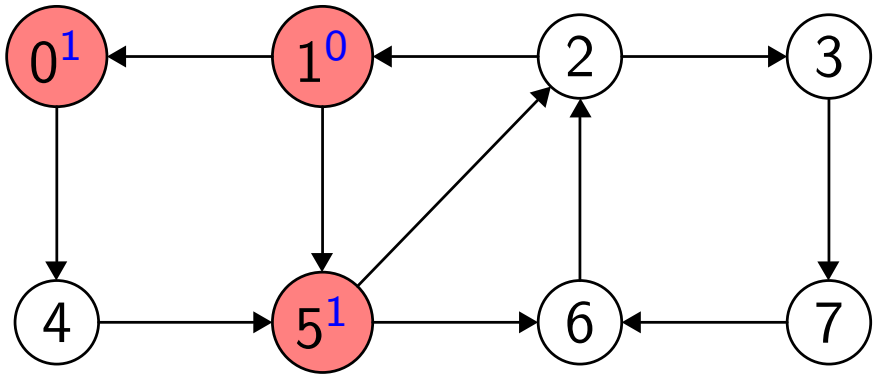


(invariant : dans la file $\Rightarrow$ dans `visited`)

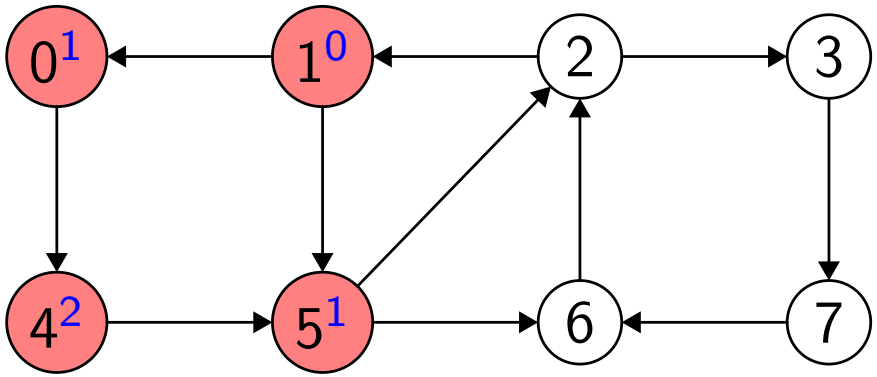


visited	queue ↑
1	1

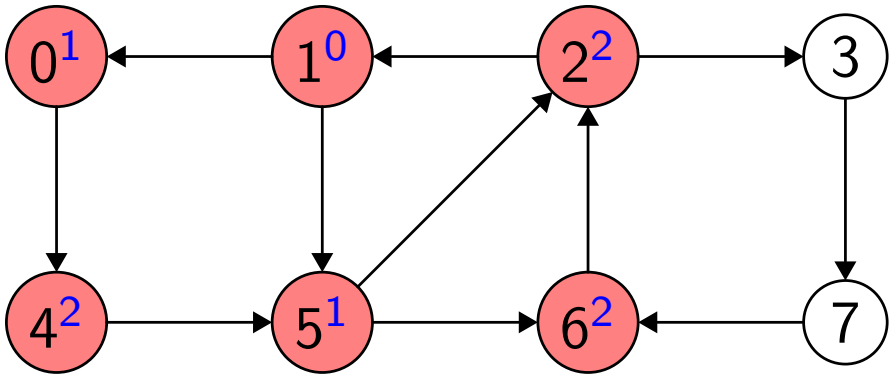
exemple



visited	queue ↑
1	0
0	5
5	

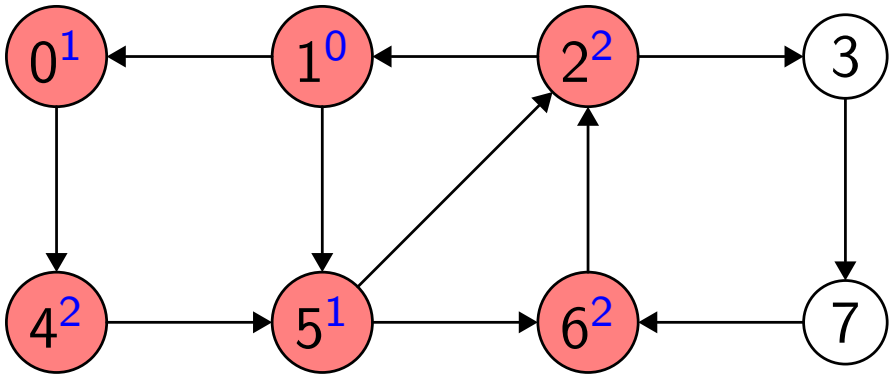


visited	queue ↑
1	5
0	4
5	
4	



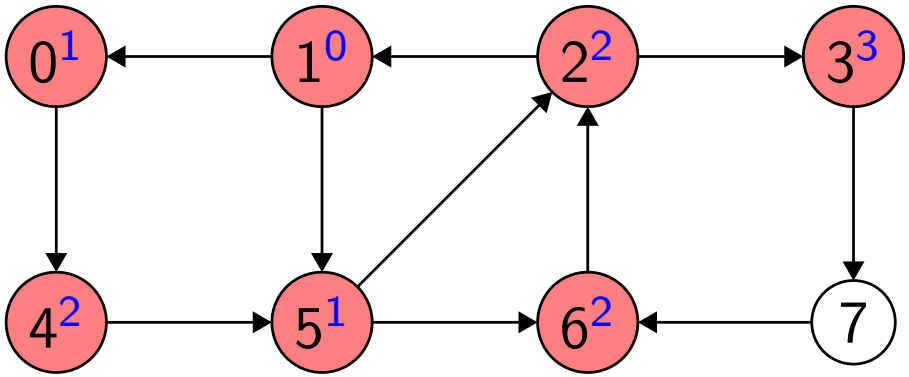
visited	queue ↑
1	4
0	2
5	6
4	
2	
6	

exemple



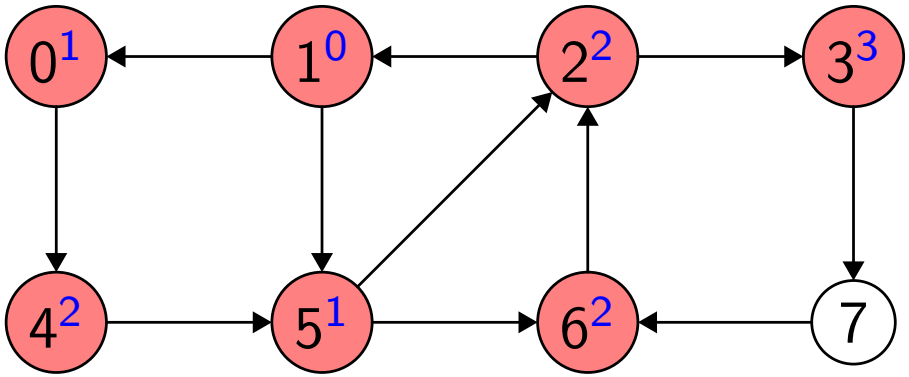
visited	queue ↑
1	2
0	6
5	
4	
2	
6	

exemple



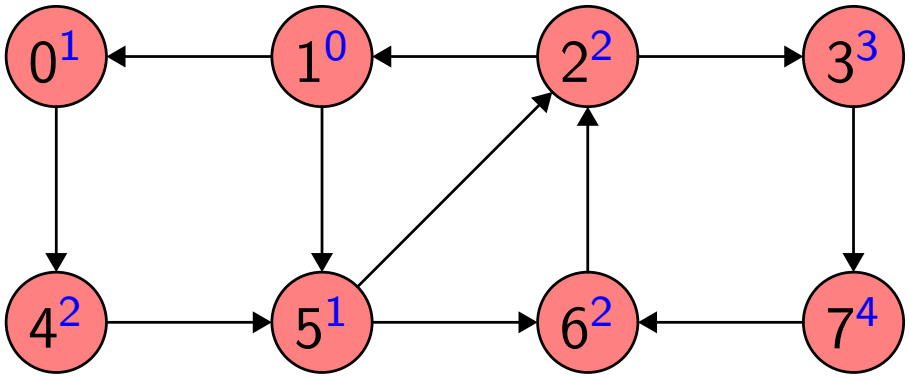
visited	queue ↑
1	6
0	3
5	
4	
2	
6	
3	

exemple

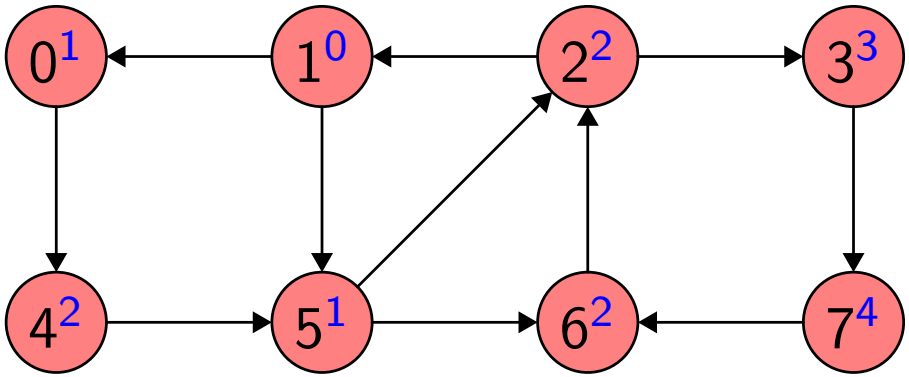


visited	queue ↑
1	3
0	
5	
4	
2	
6	
3	

exemple



visited	queue ↑
1	7
0	
5	
4	
2	
6	
3	
7	



visited	queue ↑
1	
0	
5	
4	
2	
6	
3	
7	

- chaque sommet est mis dans la file au plus une fois
- chaque arc est considéré au plus une fois

d'où une complexité $O(E)$

si on souhaite calculer la **distance** de u à v

on remplace

```
  private HashSet<V> visited;
```

par une table donnant la distance des sommets déjà atteints

```
  private HashMap<V, Integer> distance;
```

si on souhaite construire le plus court chemin trouvé

on procède comme pour DFS
(une table donnant le prédécesseur dans le chemin)

cf exercice 105 du poly

avec *Simple English Wikipedia* ($V = 151\,625$, $E = 4\,497\,295$)

cherchons les chemins de la page *'Alan Turing'* à la page *'Albert Einstein'*

avec DFS, on trouve un chemin de longueur 4077 :

```
Alan_Turing -> Enigma_(machine) -> Cypher -> ...
... ->
Poincaré_conjecture -> Henri_Poincaré -> Albert_Einstein
```

avec BFS, on trouve un chemin de longueur 3 :

```
Alan_Turing -> Cyanide -> Atom -> Albert_Einstein
```

c'est le plus court
(mais il peut y avoir d'autres chemins de longueur 3)

autres applications

problème : étant donné un DAG, énumérer ses sommets dans un ordre compatible avec l'adjacence

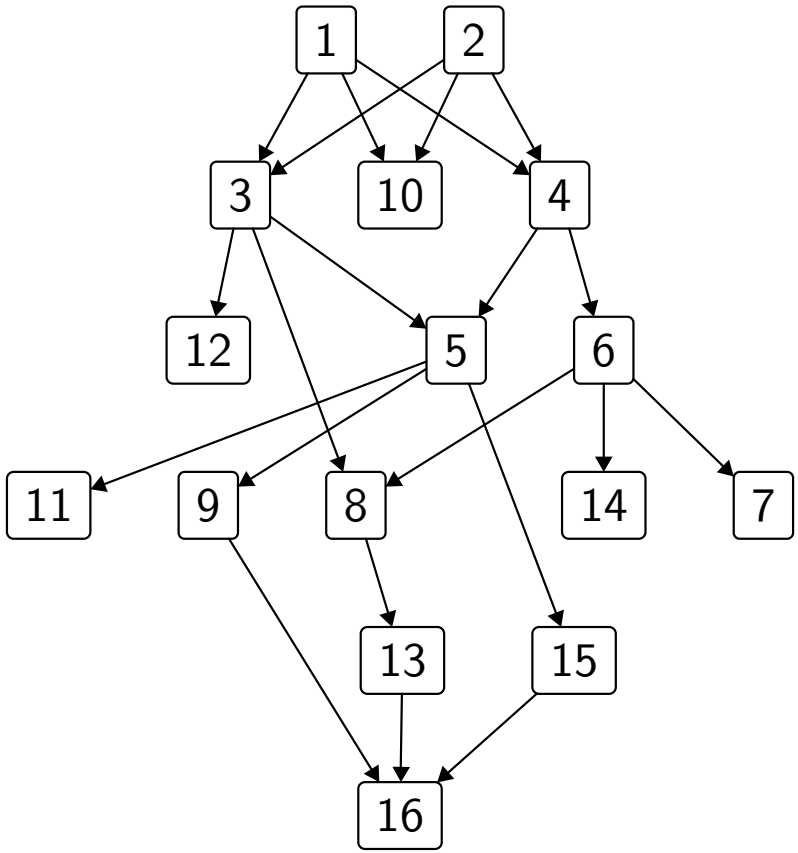
c'est-à-dire, si $u \rightarrow v$, alors u doit apparaître avant v dans l'énumération

1. créer une liste pour le résultat
2. pour chaque sommet v non encore visité
   - 2.1 faire un parcours en profondeur à partir de v
   - 2.2 ajouter chaque sommet au **début** de la liste **après** avoir parcouru ses voisins

sur les chapitres du poly, on obtient par exemple

```
2 1 10 4 6 14 7 3
12 8 13 5 15 11 9 16
```

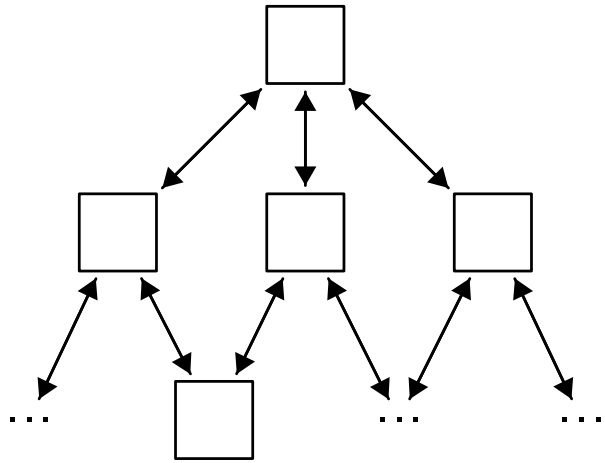
note : ce n'est pas la seule solution



trouver la **plus courte** solution d'un casse-tête tel que

où on effectue des déplacements

on se déplace dans le graphe des états possibles du jeu



inutile de construire ce graphe ; seule importe l'adjacence (`successors`)

le graphe pourrait même être infini (pour d'autres jeux)

le parcours en profondeur trouvera une solution, s'il y en a une

elle sera sûrement inutilement compliquée

le parcours en **largeur**, en revanche,
trouvera une solution optimale en nombre de déplacements

les parcours de graphes

- répondent à des questions comme
  - existe-t-il un chemin ?
  - quel est le plus court chemin ?
- sont simples à programmer
- sont efficaces (linéaires en la taille du graphe)
- ont de multiples applications,
  en particulier comme ingrédient d'autres algorithmes

le TD de cette semaine

trouver la solution la plus courte

- **lire le poly**, chapitres
  15. définition et représentation
  16.1 parcours de graphes

  il y a des **exercices** dans le poly
  suggestions : ex 105 p 186, ex 110 p 190

- **bloc 10** : graphes (2/2)

dans deux semaines