Graphes sans circuit

F.M.

2024/2025

Graphes sans circuits, sources et puits

Soit G un graphe **orienté**, **simple** et **connexe**. Rappelons que :

- une **source** de G est un sommet sans prédécesseur,
- un **puits** de G est un sommet sans successeur.

Si G est sans source alors G contient au moins un circuit.
Contraposée : **si G est sans circuit alors G possède une source.**

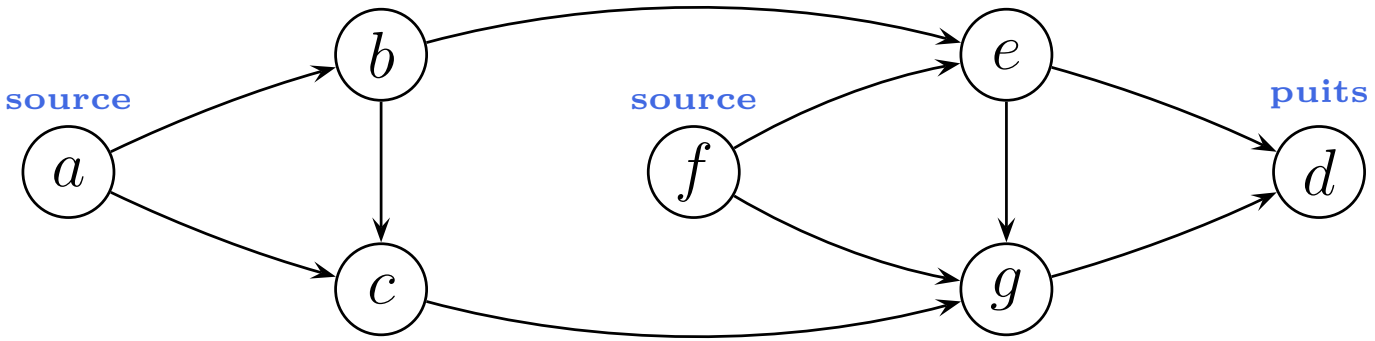
Si G est sans puits alors G contient au moins un circuit.
Contraposée : **si G est sans circuit alors G possède un puits.**

Soit s un sommet d'un graphe G sans puits. Alors s possède un successeur qui possède lui aussi un successeur, etc. Partant de s, on construit ainsi un chemin élémentaire $\mathcal{C}$ d'origine s et de longueur maximale (il est impossible d'ajouter un sommet n'appartenant pas à $\mathcal{C}$). Soit p le but de $\mathcal{C}$ et x un successeur de p. Le sommet x appartient nécessairement à $\mathcal{C}$ et G contient un circuit composé de l'arc (p, x) et de la portion du chemin $\mathcal{C}$ allant de x à p.

Dans la suite, l'acronyme DAG (Directed Acyclic Graph) désigne un graphe orienté sans circuit, simple et connexe.

2 / 35

Graphe sans circuit ou DAG, sources et puits



3 / 35

Propriétés

Soit G un DAG.

Tout chemin de G est élémentaire.

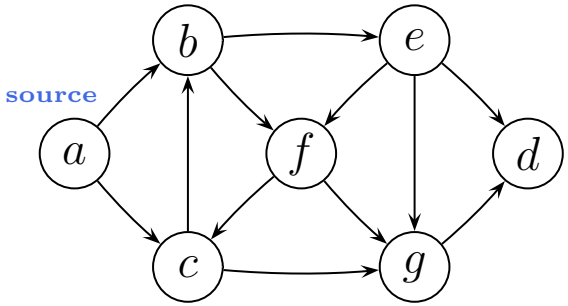
Tout sommet de G admet une source parmi ses descendants et un puits parmi ses antécédents.

Si G ne contient qu'une et une seule source (resp. un seul puits) alors cette source (resp. ce puits) est aussi une racine (resp. *une anti-racine*).

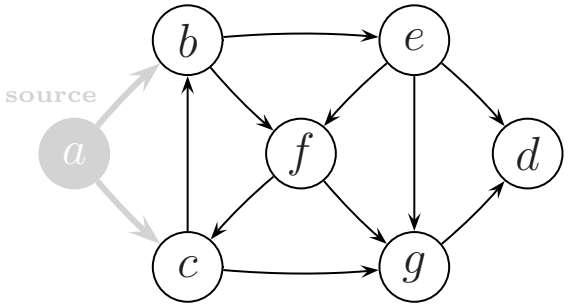
Tout sous-graphe de G est un DAG. En particulier, si s est une source (ou un puits) de G alors $G - s$ est un DAG.

4 / 35

DAG et sous-graphes

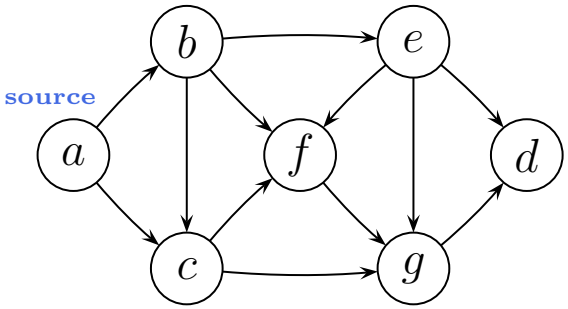


un graphe G

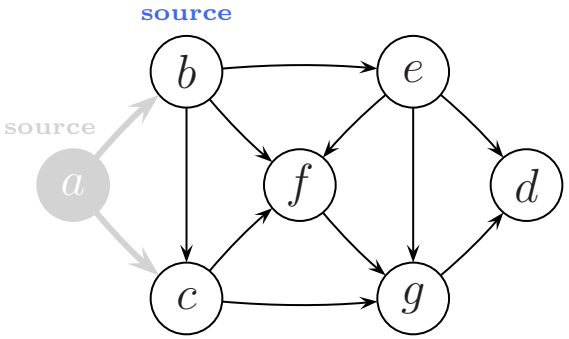


on supprime a

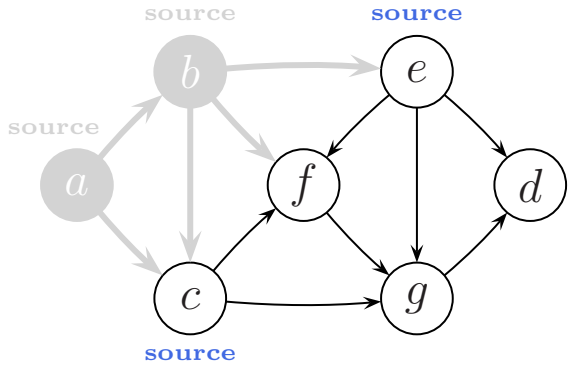
$G - a$ n'a pas de source
$\Rightarrow G$ n'est pas un DAG



un autre graphe G



on supprime a et
b devient une source



on supprime aussi b et
c, e deviennent des sources

Que se passe-t-il si on poursuit les suppressions ?

5 / 35

Algorithme de base

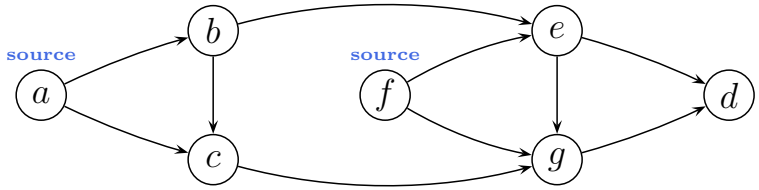
Il s'agit d'**un algorithme qui vérifie si, oui ou non, un graphe G est sans circuit**. Il repose sur le fait qu'un DAG admet (au moins) une source et que tout sous-graphe d'un DAG est un DAG.

tant que G *contient des sources* **faire**
  $s \leftarrow$ *choisir une source de* G ..... ☞ **choix arbitraire**
  $G \leftarrow G - s$ ..... ☞ **suppression de s**
fin tant que
si G *est vide* **alors** ecrire ("le graphe est un DAG")
sinon ecrire ("le graphe contient des circuits")

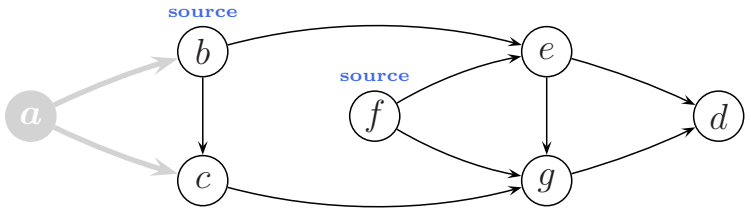
On appelle **candidat** tout sommet qui peut être choisi en début d'itération. Les candidats sont donc les sources du sous-graphe induit par l'ensemble des sommets non supprimés.

6 / 35

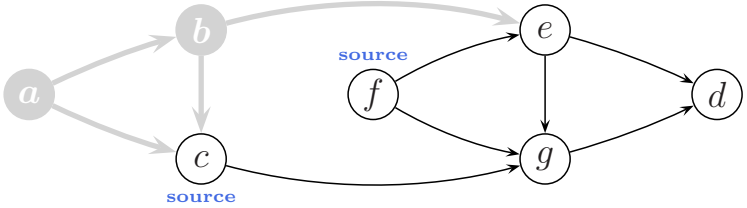
Illustration



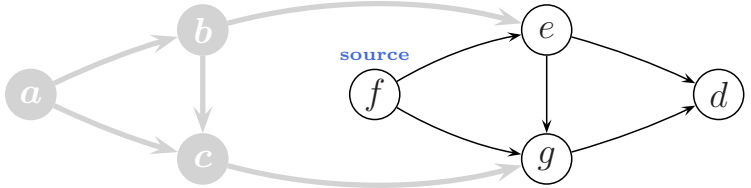
ité. 0 ou init. : candidats $C = \{a, f\}$



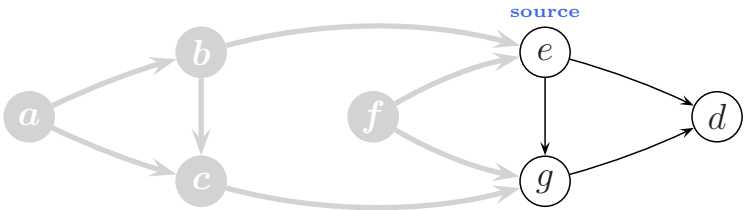
ité. 1 : suppr. $a \Rightarrow C = \{b, f\}$



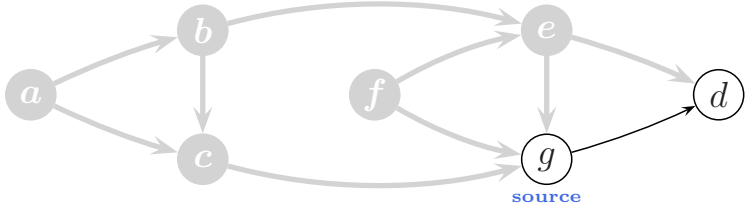
ité. 2 : suppr. $b \Rightarrow C = \{c, f\}$



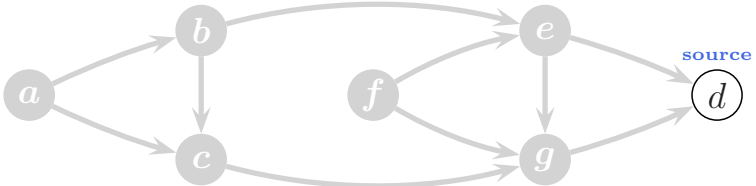
ité. 3 : suppr. $c \Rightarrow C = \{f\}$



ité. 4 : suppr. $f \Rightarrow C = \{e\}$



ité. 5 : suppr. $e \Rightarrow C = \{g\}$



ité. 6 : suppr. $g \Rightarrow C = \{d\}$

ité. 7 : suppr. $d \Rightarrow$ c'est un DAG

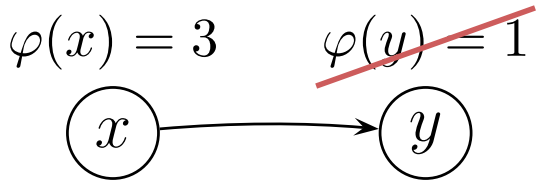
7 / 35

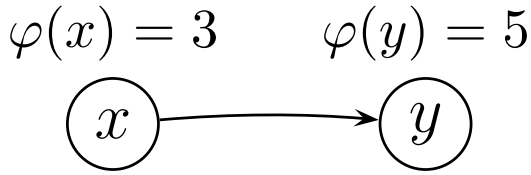
Ordre topologique

Soit $G = (S, A)$ un graphe d'ordre n. Un **ordre topologique** pour G est une bijection $\varphi : S \mapsto \{1, \ldots, n\}$ vérifiant :

$$\forall (x, y) \in A, \quad \varphi(x) < \varphi(y)$$

Le nombre $\varphi(x)$ est appelé **numéro topologique** du sommet x.





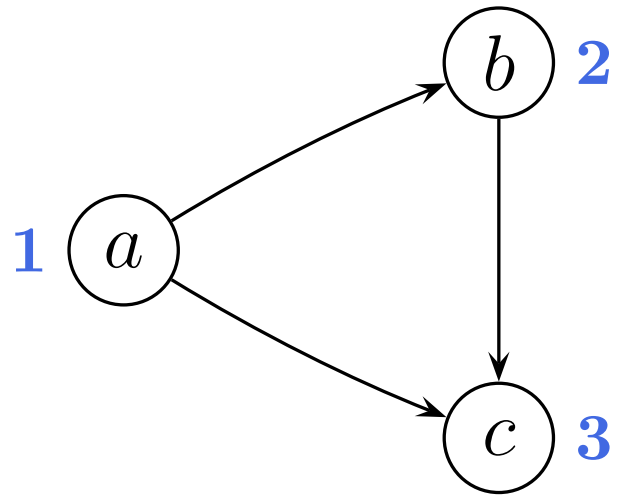
On dit qu'**une liste L est un ordre topologique** pour G si L est la liste des sommets rangés dans **l'ordre croissant de leur numéro topologique**.

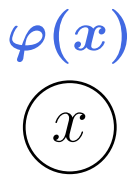
Le rang de tout sommet x dans cette liste (c'est aussi un n-uplet) est égal à $\varphi(x)$.

8 / 35

Exemple 1

$$\varphi : \{a, b, c\} \mapsto \{1, 2, 3\}$$





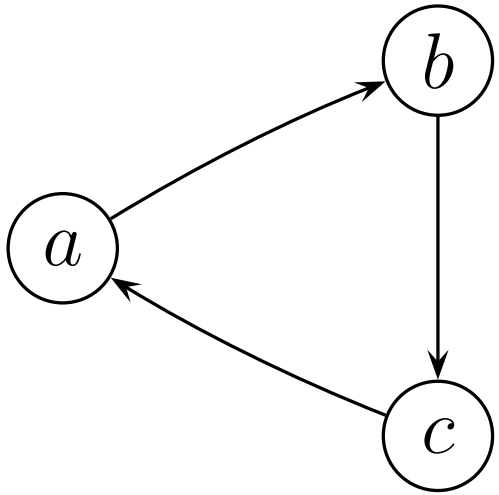
la liste $L = (a, b, c)$ est un ordre topologique pour ce graphe

le numéro topologique d'un sommet correspond à son rang dans L

9 / 35

Exemple 2

$$\varphi : \{a, b, c\} \mapsto \{1, 2, 3\}$$

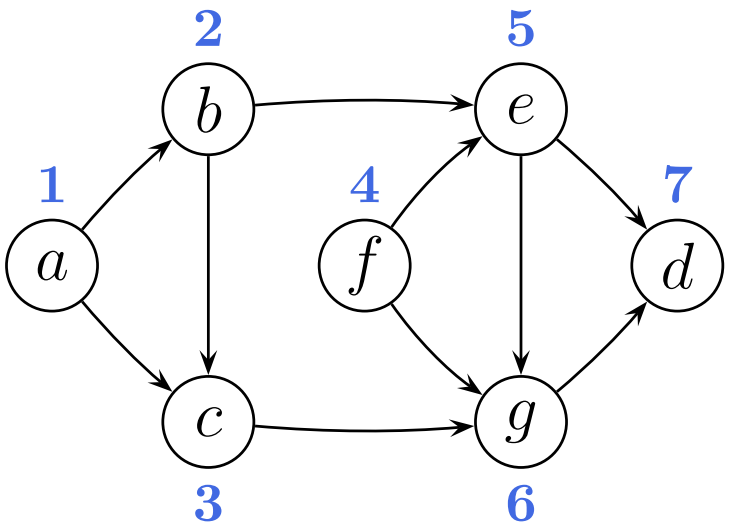


il n'existe pas d'ordre topologique
pour les graphes contenant des circuits

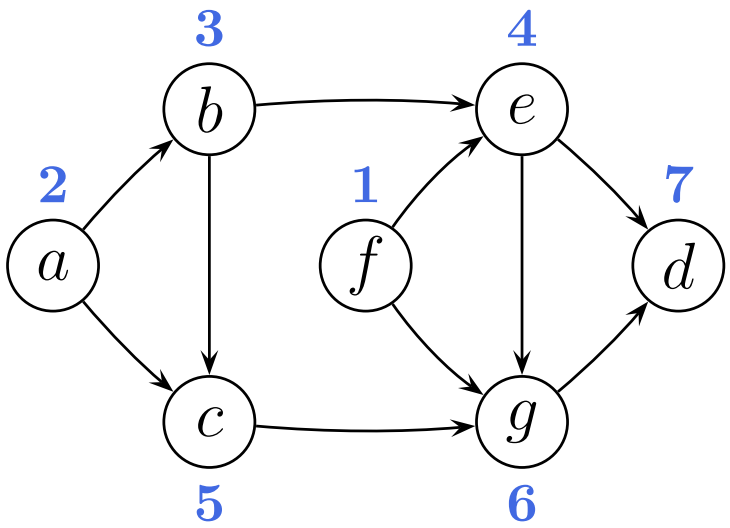
10 / 35

Exemple 3

$$\varphi : \{a, b, c, d, e, f, g\} \mapsto \{1, 2, 3, 4, 5, 6, 7\}$$



(a, b, c, f, e, g, d)



(f, a, b, e, c, g, d)

Un DAG admet au moins un ordre topologique (il n'y a pas unicité)

11 / 35

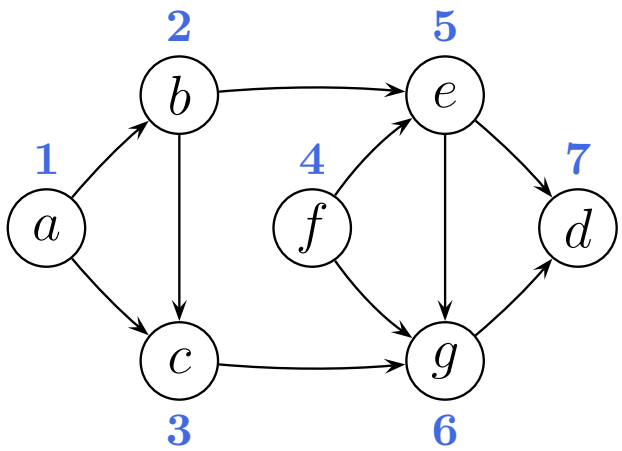
Propriétés évidentes

Soient G un DAG d'ordre n et φ un ordre topologique pour G.

$\varphi(x) = 1 \Rightarrow x$ est une source de G. La réciproque est fausse puisqu'un graphe peut contenir plusieurs sources.

$\varphi(x) = n \Rightarrow x$ est un puits de G. La réciproque est fausse puisqu'un graphe peut contenir plusieurs puits.

Tout chemin $(s_1, s_2, \ldots, s_k)$ de G est élémentaire et vérifie : $\varphi(s_1) < \varphi(s_2) < \cdots < \varphi(s_k)$.



$\varphi(a) = 1 \Rightarrow a$ est une source
f est une source $\not\Rightarrow \varphi(f) = 1$

$\varphi(d) = 7 \Rightarrow d$ est un puits

(a, b, e, d) est un chemin
$\Rightarrow \varphi(a) < \varphi(b) < \varphi(e) < \varphi(d)$

12 / 35

Propriété fondamentale

Un graphe orienté G admet un ordre topologique si et seulement si G ne contient pas de circuit.

($\Rightarrow$ par contradiction) Soit φ un ordre topologique pour G. Hypothèse : G contient un circuit $(s_1, s_2, \ldots, s_k)$ où $s_1 = s_k$. Alors l'ordre topologique implique que $\varphi(s_1) < \varphi(s_2) < \cdots < \varphi(s_k) = \varphi(s_1)$, ce qui aboutit à la contradiction $\varphi(s_1) < \varphi(s_1)$. L'hypothèse est donc fausse.

($\Leftarrow$ par récurrence sur le nombre n de sommets) Si $n = 1$, l'implication est vraie. Supposons que tout DAG d'ordre n possède un ordre topologique. Soit G' un DAG d'ordre $n + 1$. Alors G' contient un puits p et le sous-graphe $G = G' - p$ est un DAG d'ordre n. Par hypothèse de récurrence, G possède un ordre topologique φ et, en posant $\varphi(p) = n + 1$, on obtient alors un ordre topologique pour G'. En effet, pour chaque prédécesseur x de p, l'arc (x, p) vérifie $\varphi(x) < \varphi(p)$.

13 / 35

Construction d'un diagramme de topologie

Soit G un DAG et φ un ordre topologique pour G.

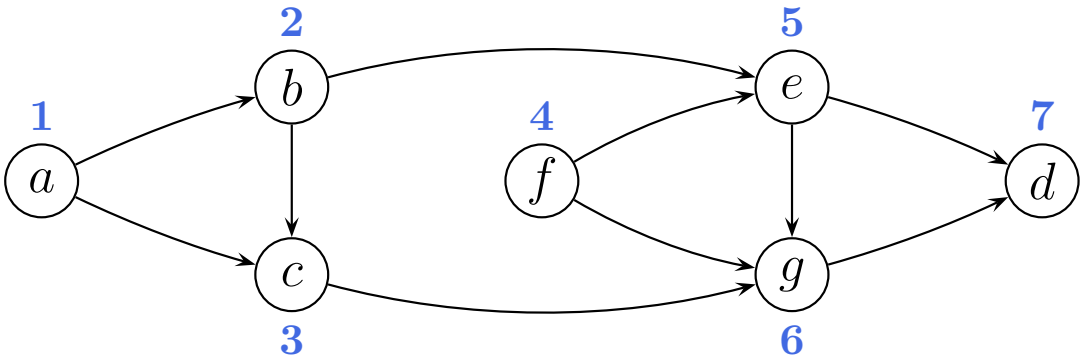
On place les sommets de G le long d'une ligne horizontale dans un ordre topologique : le sommet de numéro topologique 1 d'abord, puis celui de numéro topologique 2, etc. On ajoute ensuite les arcs de G.

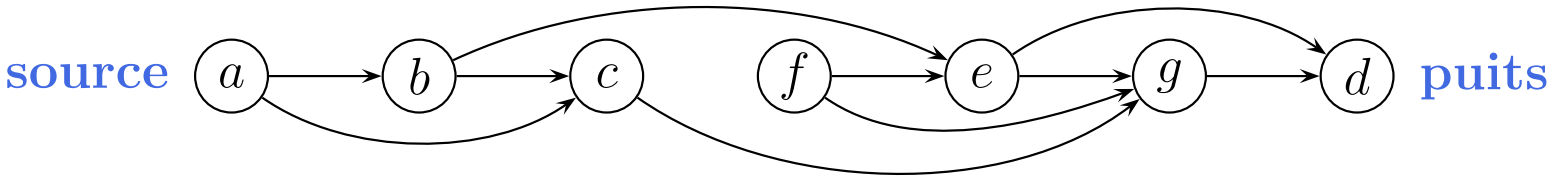
Le diagramme ainsi obtenu est appelé **diagramme de topologie** du graphe G.

le diagramme de topologie
permet de visualiser la structure d'un DAG

14 / 35

Illustration





tous les arcs sont orientés de gauche à droite

Les DAG permettent de modéliser des réseaux de distribution d'électricité, des réseaux de distribution d'eau, des réseaux de transport de personnes ou de marchandise, etc. Problème associé : gestion de la circulation (optimiser un flux de circulation par exemple).

Les DAG sont aussi utilisés pour modéliser des projets décomposés en de multiples tâches. Problème associé : organisation et gestion de projet.

15 / 35

Application : organisation et gestion de projet

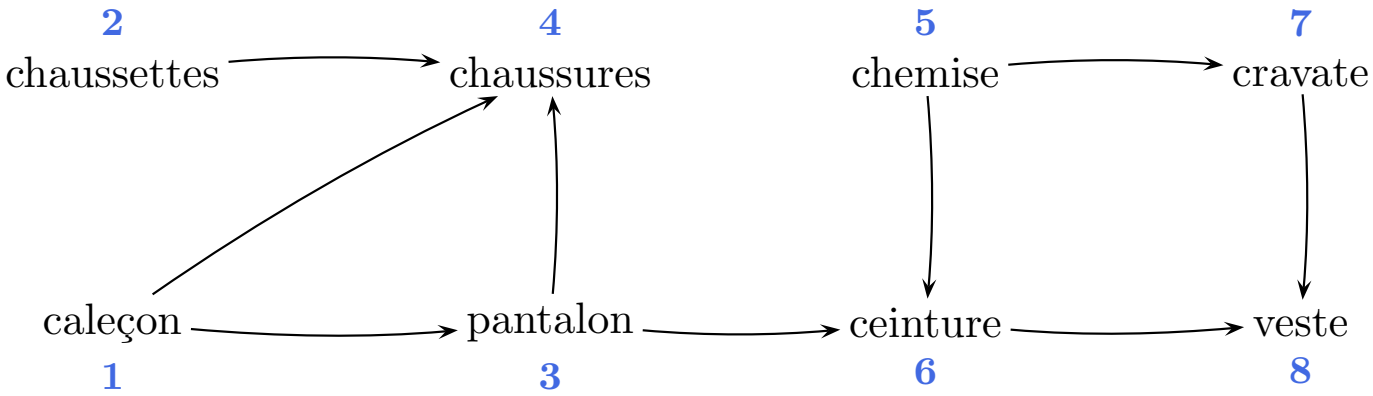
Un projet est décomposé en tâches soumises à des contraintes d'antériorité de la forme : une ou plusieurs tâches doivent être terminées avant qu'une autre tâche ne commence. **Il s'agit de déterminer un ordre d'exécution des tâches ou ordonnancement des tâches qui minimise la durée totale d'exécution du projet**.

Première modélisation (méthode MPM). On appelle **graphe potentiel-tâches** le DAG construit comme suit : les sommets représentent les tâches et les arcs représentent les contraintes. Un ordre topologique pour ce DAG est alors un ordre d'exécution des tâches qui respecte les contraintes de précédence.

Seconde modélisation (méthode PERT). On appelle **graphe PERT** le DAG construit comme suit : les arcs représentent les tâches et les sommets représentent des étapes (des dates) dans l'exécution du projet. Les arcs sortant d'un sommet s représentent des tâches qui ne peuvent commencer que quand toutes les tâches de but s sont terminées.

16 / 35

Exemple trivial de graphe potentiel-tâches



Un ordre topologique possible est : (caleçon, chaussettes, pantalon, chaussures, chemise, ceinture, cravate, veste).

17 / 35

Tri topologique

Un algorithme de **tri topologique** est un algorithme qui détermine un ordre topologique.

L'algorithme de base (cf. diapo 6) supprime les sommets d'un DAG dans un ordre topologique.

Soit G un DAG d'ordre n et soit $L = (s_1, s_2, \ldots, s_n)$ la liste ordonnée des sommets de G supprimés par l'algorithme de base. Le sommet s_1 est une source de G : $\varphi(s_1) \leftarrow 1$. Le k^{e} sommet supprimé s_k, $1 < k \leq n$, est tel que $V^-(s_k) \subseteq \{s_1, \ldots, s_{k-1}\}$. Tout arc de la forme (x, s_k) avec $x \in V^-(s_k)$ vérifie $\varphi(x) < \varphi(s_k)$.

Pour obtenir un ordre topologique, **il suffit de compléter l'algorithme de base en numérotant les sommets dans l'ordre de suppression** : on attribue le numéro topologique k au k^{e} sommet supprimé.

18 / 35

Implémentation

Comment identifier, mémoriser et supprimer (numéroter) un candidat ?

Pour mémoriser les candidats on utilise un ensemble noté C.

Pour le reste on utilise un tableau nommé ***marque*** qui à chaque sommet s associe le nombre de prédécesseurs non supprimés de s :

- **Le nombre *marque*[*s*] est décrémenté à chaque fois qu'un prédécesseur du sommet *s* est supprimé.**
- **Un sommet *s* devient candidat quand le nombre *marque*[*s*] devient nul.**

Remarque : on utilise habituellement une structure de file (FIFO : First In First Out) pour mémoriser les candidats. Chaque nouveau candidat est enfilé (push, put) et on récupère un candidat en défilant (pop, get).

19 / 35

Implémentation : initialisations

Soit $G = (S, A)$ un graphe orienté. **Initialement, la marque d'un sommet est égale à son degré intérieur**, c.-à-d. au nombre de ses prédécesseurs. Un sommet de marque 0 est une source de G et il est ajouté à l'ensemble des candidats C.

$C \leftarrow \emptyset$ ..... ✎ l'ensemble des candidats
$k \leftarrow 0$ ..... ✎ k : compteur de sommets supprimés
pour tout $s \in S$ **faire**
  $marque[s] \leftarrow \delta^-(s)$ ..... ✎ $\delta^-(s)$ est le degré intérieur de s
  si $marque[s] = 0$ **alors** $C \leftarrow C \cup \{s\}$ ..... ✎ s est une source
fin pour

20 / 35

Implémentation : itérations

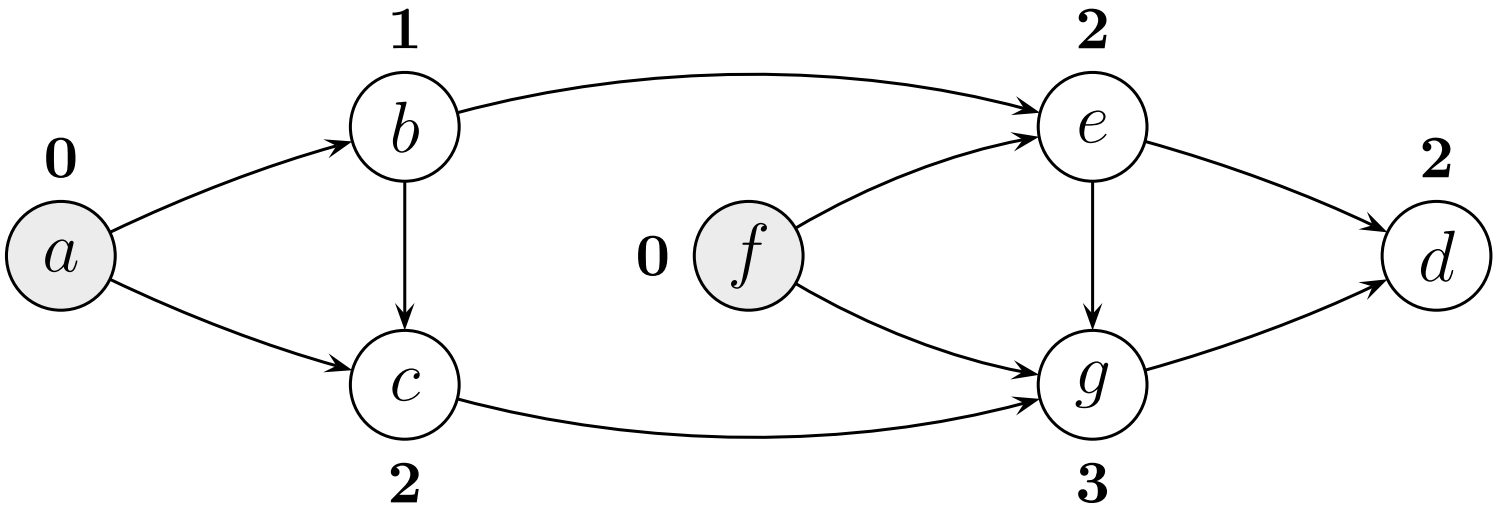
À chaque itération, un candidat s est choisi et supprimé : on décrémente la marque des successeurs de s et tout successeur dont la marque s'annule devient candidat.

tant que $C \neq \emptyset$ **faire**
  $s \leftarrow$ choisir(C) . . . . . ✎ choisir un candidat et l'extraire de C
  $\varphi(s) \leftarrow k \leftarrow k+1$ . . . . . ✎ numérotation du sommet s
  pour tout $x \in V^+(s)$ **faire**
    $marque[x] \leftarrow marque[x] - 1$
    ✎ test d'identification d'un candidat
    si $marque[x] = 0$ **alors** $C \leftarrow C \cup \{x\}$
  fin pour
fin tq
si $k = |S|$ **alors** ecrire ("ordre topologique :", φ)
sinon ecrire ("le graphe n'est pas un DAG")

21 / 35

Trace de l'algorithme

Itération 0 : initialisation des marques et de C



C	a, f
s	
$\varphi(s)$	

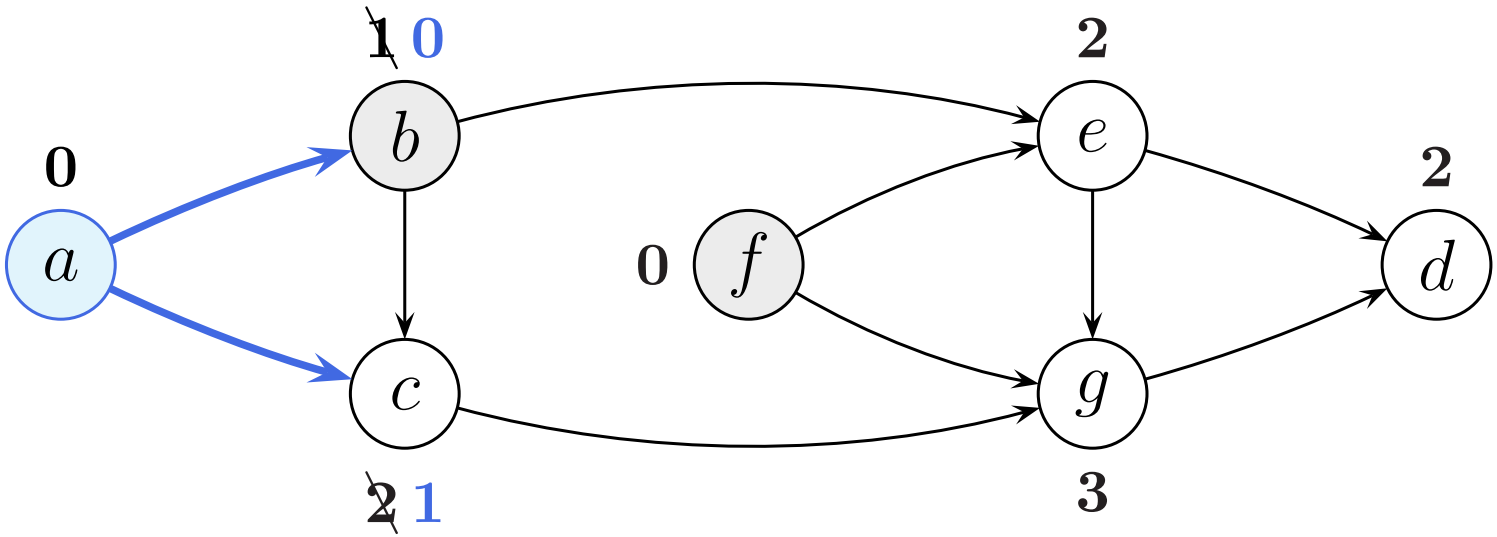
candidat

supprimé

22 / 35

Trace de l'algorithme

Itération 1 : suppression de a, maj des marques et de C



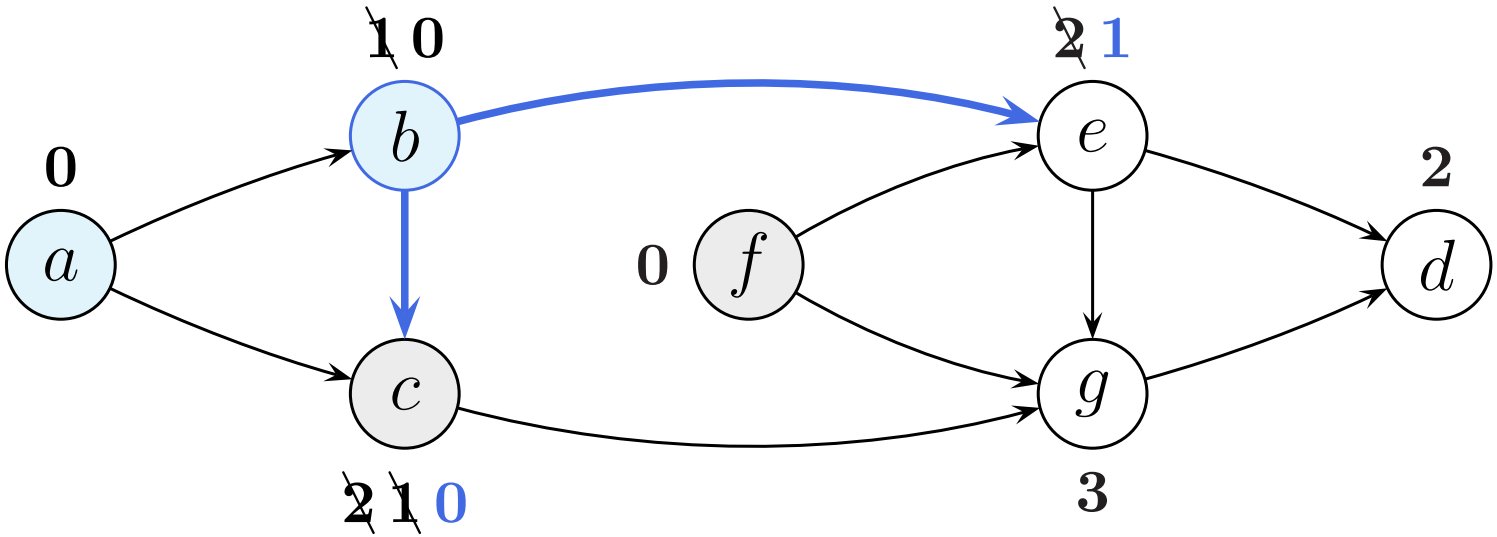
C	a, f	b, f
s	a	
$\varphi(s)$	1	

candidat

supprimé

23 / 35

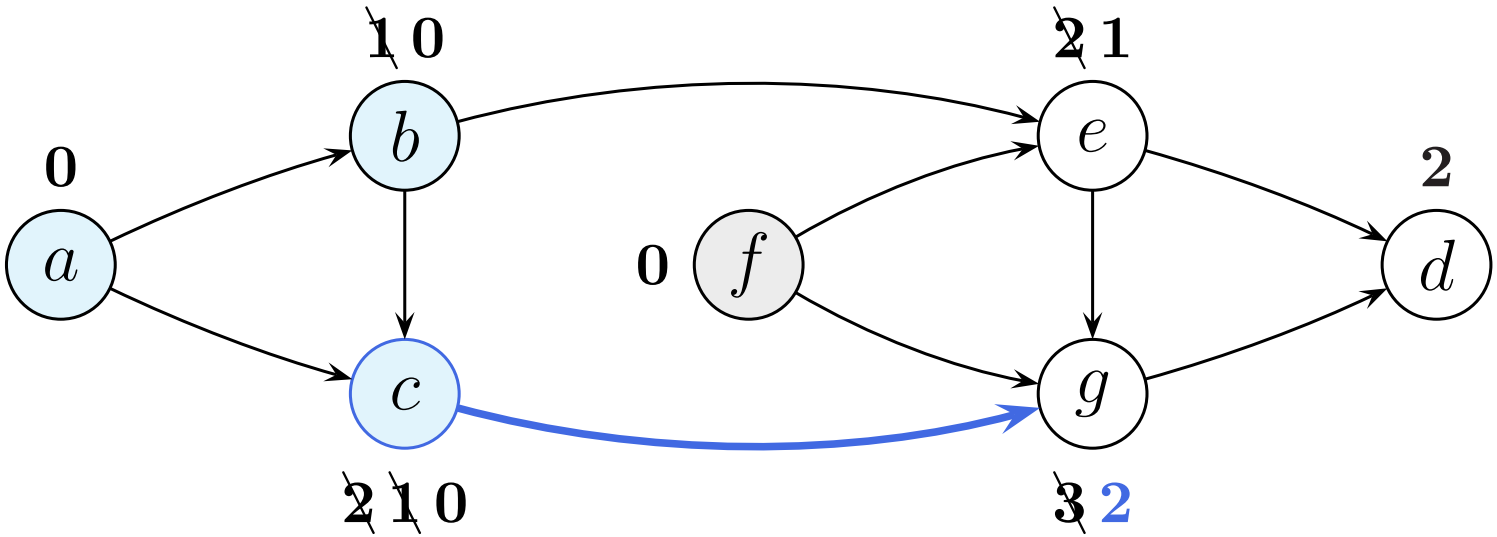
Itération 2 : suppression de b, maj des marques et de C



C	a, f	b, f	c, f
s	a	b	
$\varphi(s)$	1	2	

24 / 35

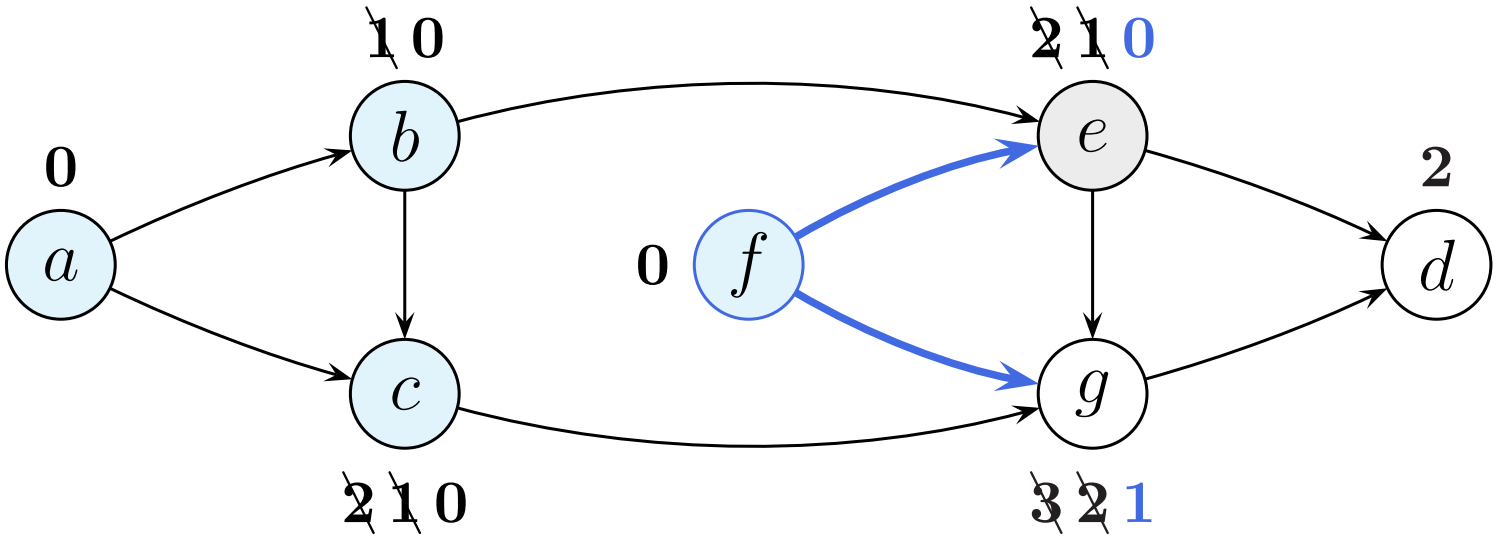
Itération 3 : suppression de c, maj des marques et de C



C	a, f	b, f	c, f	f
s	a	b	c	
$\varphi(s)$	1	2	3	

25 / 35

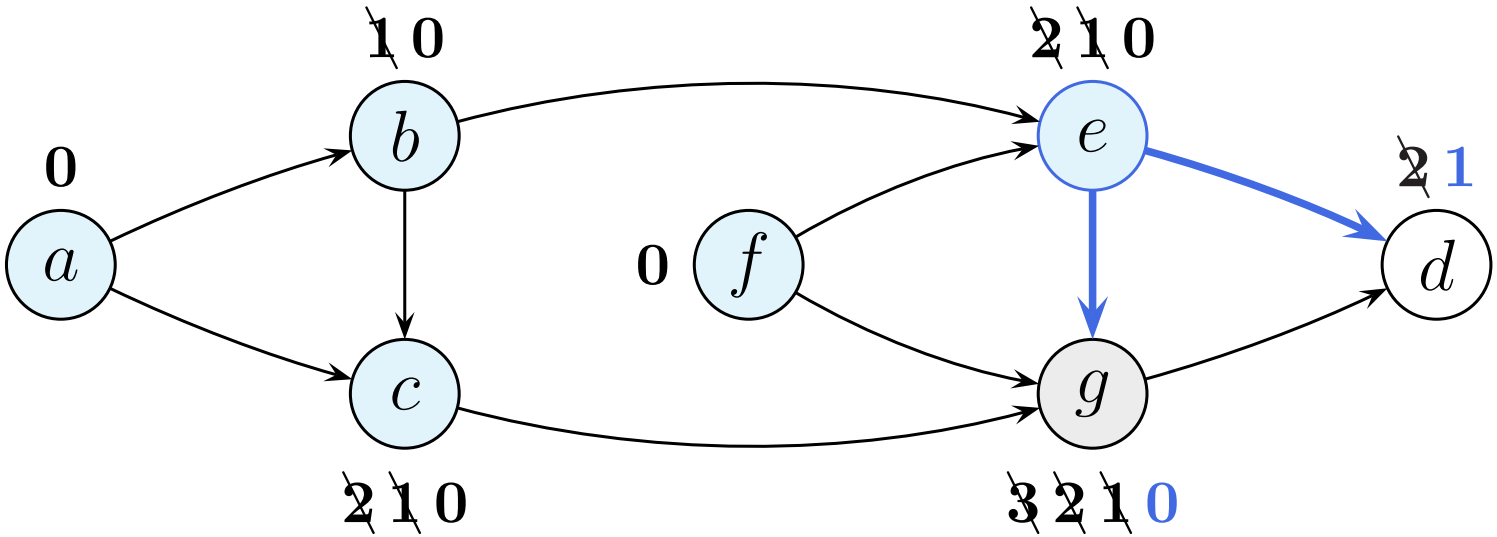
Itération 4 : suppression de f, maj des marques et de C



C	a, f	b, f	c, f	f	e
s	a	b	c	f	
$\varphi(s)$	1	2	3	4	

26 / 35

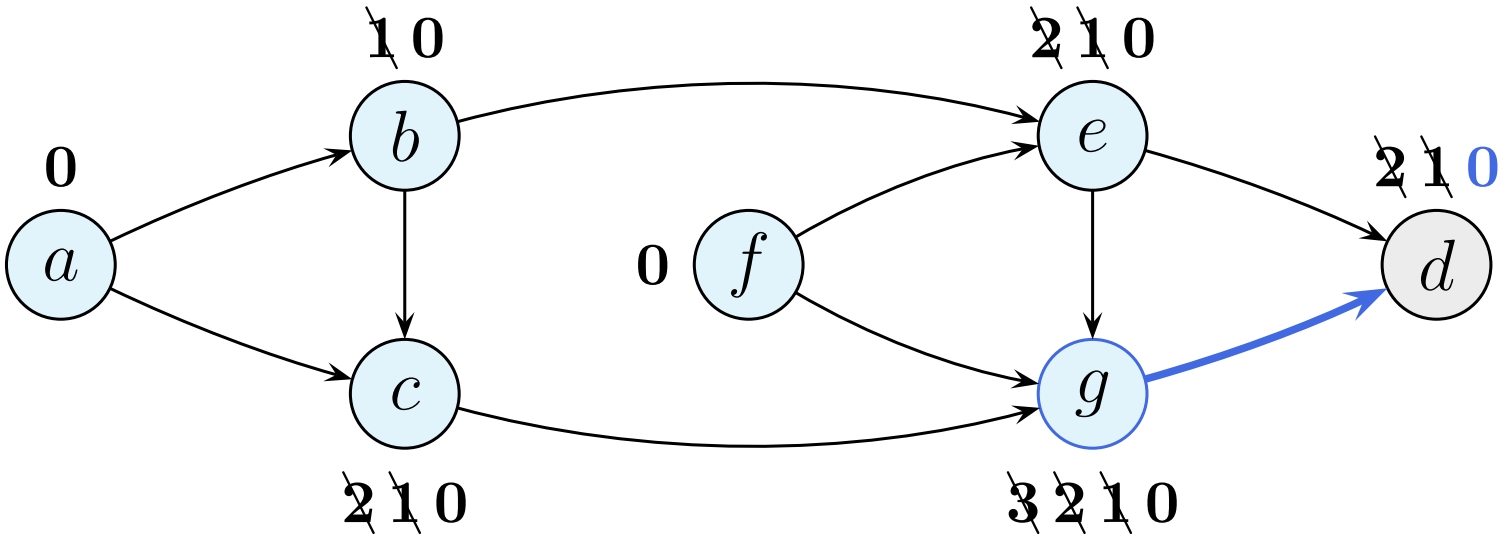
Itération 5 : suppression de e, maj des marques et de C



C	a, f	b, f	c, f	f	e	g
s	a	b	c	f	e	
$\varphi(s)$	1	2	3	4	5	

27 / 35

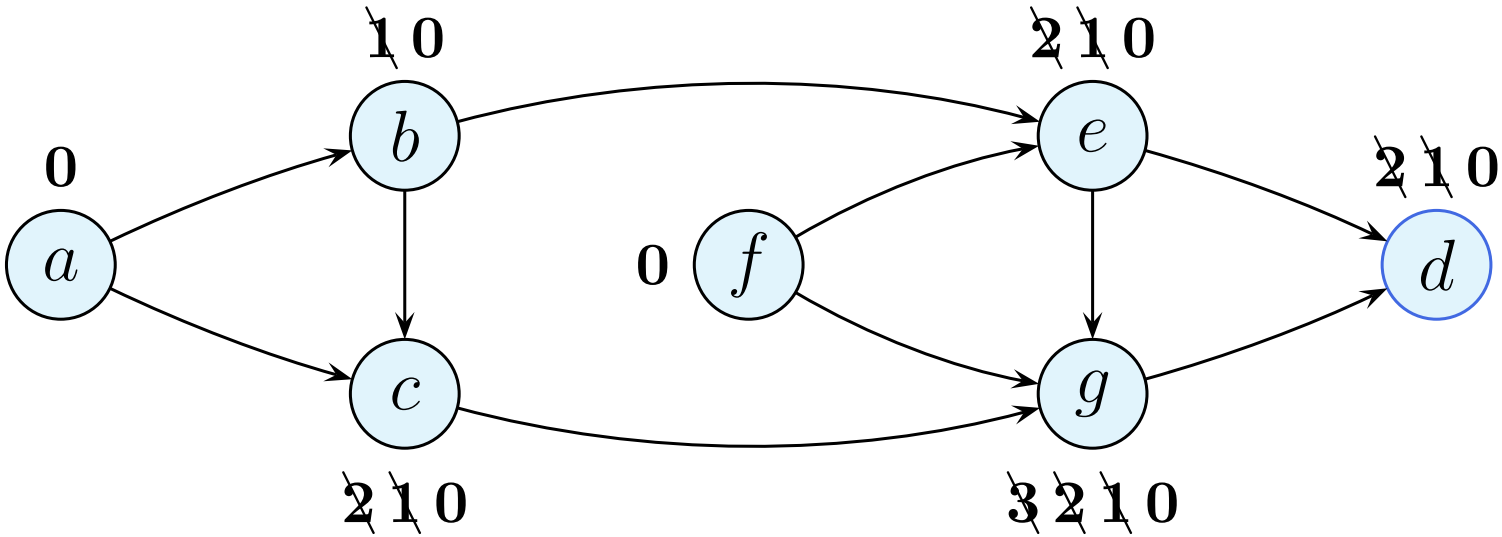
Itération 6 : suppression de g, maj des marques et de C



C	a, f	b, f	c, f	f	e	g	d
s	a	b	c	f	e	g	
$\varphi(s)$	1	2	3	4	5	6	

28 / 35

Itération 7 : suppression de d, maj des marques et de C



C	a, f	b, f	c, f	f	e	g	d
s	a	b	c	f	e	g	d
$\varphi(s)$	1	2	3	4	5	6	7

C est vide $\Rightarrow$ fin de l'algorithme

29 / 35

Remarques

Soit L un ordre topologique pour un DAG. Les prédécesseurs (et tous les antécédents) d'un sommet donné x se trouvent à gauche de x dans L et les successeurs (et tous les descendants) de x se trouvent à droite de x dans L.

Si un algorithme examine les sommets d'un DAG dans un ordre topologique alors tout sommet sera examiné **après ses prédécesseurs** (et avant ses successeurs). Au moment d'examiner un sommet, les résultats obtenus lors du traitement de ses prédécesseurs (et de tous ses antécédents) seront connus. Notons qu'il est aussi possible d'examiner les sommets dans un ordre topologique inverse.

Par exemple, la méthode PERT (ou la méthode MPM) examine les sommets dans un ordre topologique pour déterminer les dates de début au plus tôt des tâches. Elle examine ensuite les sommets dans un ordre topologique inverse pour déterminer les dates de début au plus tard.

30 / 35

Complément : parcours en profondeur d'abord

Un graphe est généralement représenté par une liste d'adjacence qui associe à chaque sommet la liste de ses successeurs.

L'algorithme de tri topologique présenté précédemment nécessite le calcul du degré intérieur de chaque sommet. Pour ce faire il faut donc parcourir une première fois la liste d'adjacence. Elle sera ensuite parcourue une seconde fois pour déterminer un ordre topologique.

Il est possible de déterminé un ordre topologique en ne parcourant qu'une seule fois la liste d'adjacence. On utilise pour cela l'algorithme de **parcours en profondeur d'abord** d'un graphe.

L'algorithme de parcours en profondeur d'abord d'un graphe est un algorithme qui permet d'**explorer systématiquement un graphe** : il s'agit de cheminer dans le graphe dans le but de **visiter** (atteindre) chaque sommet exactement une fois.

31 / 35

Complément : parcours en profondeur d'abord

Partant d'un sommet donné, on chemine dans le graphe aussi profondément que possible en veillant à ne pas visiter deux fois le même sommet. Lorsque le parcours est bloqué (impasse ou impossible d'avancer vers un sommet non visité), on revient sur ses pas et dès que possible, on avance vers un sommet non visité. Il s'agit d'une stratégie récursive dite de retour sur trace (backtracking).

Si r est le sommet de départ, l'algorithme de parcours ne pourra visiter que les descendants de r. Les chemins suivis pour atteindre les descendants de r forment alors une r-arborescence.

Si tous les sommets du graphe n'ont pas été visités lors d'un premier parcours, on effectue d'autres parcours (en partant à chaque fois d'un sommet non visité) afin de visiter tous les sommets du graphe.

32 / 35

Parcours en profondeur d'abord et tri topologique

Un **horodatage** (timestamp) est mis en place lors du parcours en profondeur d'abord. Il est utilisé pour mémoriser les dates de réalisation de deux événements particuliers :

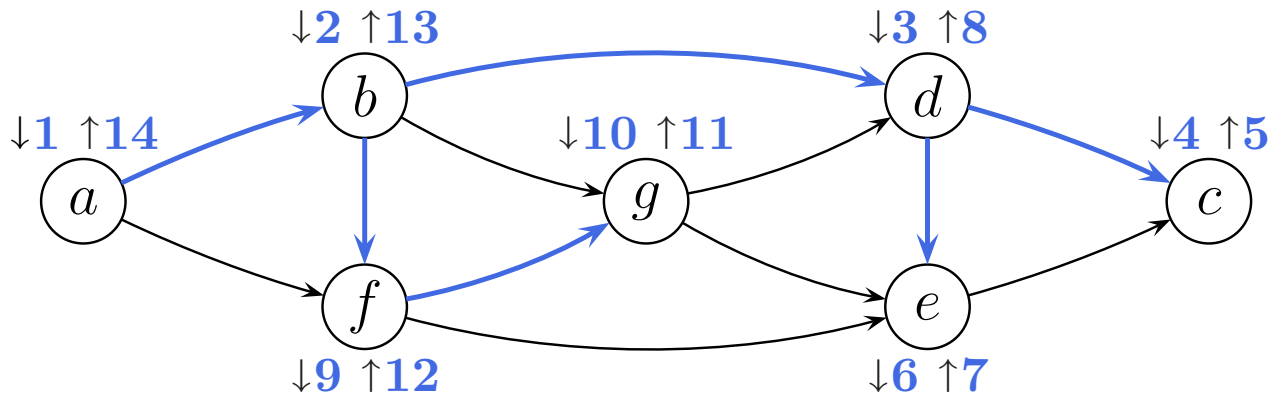
- ☞ À chaque fois qu'un nouveau sommet est visité, on attribue à ce sommet une **date de début visite** ou **date d'entrée**.
- ☞ À chaque fois que le parcours est bloqué sur un sommet, on attribue à ce sommet une **date de fin de visite** (avant d'effectuer un retour sur trace) ou **date de sortie**.

L'horodatage est réalisé à l'aide d'un compteur qui est incrémenté à chaque fois qu'un événement se produit.

Le sommet de départ est choisi arbitrairement. Si nécessaire, on effectue plusieurs parcours afin de visiter tous les sommets du graphe, mais sans interrompre l'horodatage.

33 / 35

Parcours en profondeur d'abord et tri topologique



légende : ↓**date d'entrée** ↑**date de sortie**

Soit $D_{in} = (a, b, d, c, e, f, g)$ la liste des sommets rangés dans l'ordre chronologique des entrées et soit $D_{out} = (c, e, d, g, f, b, a)$ la liste des sommets rangés dans l'ordre chronologique des sortiesx.

La liste D_{out} **est un ordre topologique inverse** pour le graphe : c est un puits, si on supprime c alors e devient un puits, etc.

On en déduit que **la liste D_{out} inversée égale à (a, b, f, g, d, e, c) est un ordre topologique** pour le graph : a est une source, si on supprime a alors b devient une source, etc.

34 / 35

Parcours en profondeur d'abord d'un graphe orienté

✎ programme principal

$D_{in} \leftarrow \text{list}()$; $D_{out} \leftarrow \text{list}()$

pour tout $s \in S$ **faire**

  si non D_{in}.includes(s) **alors**

    dfs(G, s, D_{in}, D_{out}) ..... ✎ depth-first search (dfs)

  fin si

fin pour

D_{in}.print() ; D_{out}.print()

✎ procédure de parcours en profondeur d'abord

proc dfs(G, s, D_{in}, D_{out})

  D_{in}.push(s) ..... ✎ début de visite du sommet s

  pour tout $x \in V^+(s)$ **faire**

    si non D_{in}.includes(x) **alors** dfs(G, x, D_{in}, D_{out})

  fin pour

  D_{out}.push(s) ..... ✎ fin de visite du sommet s

fin proc

35 / 35