

Exploration d'un labyrinthe

Pour explorer un labyrinthe, il suffit d'une pelote de ficelle et d'un morceau de craie :

- marquer les carrefours que vous avez déjà visités avec la craie pour empêcher de boucler
- utiliser une ficelle pour pouvoir revenir au point de départ.

On peut utiliser le même principe pour explorer un graphe.

Parcours en profondeur (DFS) pour les graphes connexes

Entrées : graphe $G = (V, E)$ et sommet $r \in V$

début

créer pile(S)

pour tous les $u \in V$ **faire**

└ marqué[u] \leftarrow False

empiler(S, r)

tant que $S \neq \emptyset$ **faire**

└ $u \leftarrow$ dépiler(S)

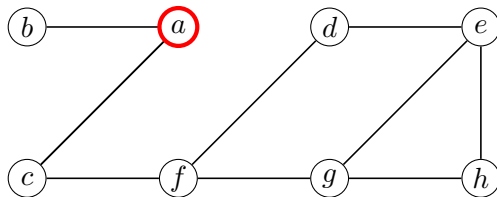
└ **si** marqué[u] = Faux **alors**

└└ marqué[u] \leftarrow Vrai

└└ **pour tous les** $uv \in E$ **faire**

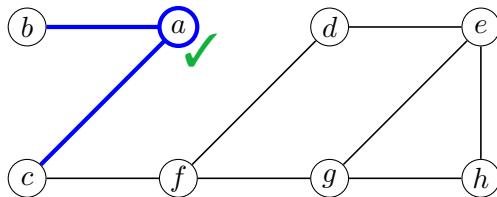
└└└ empiler(S, v)

Illustration du parcours en profondeur



$$S = [a]$$

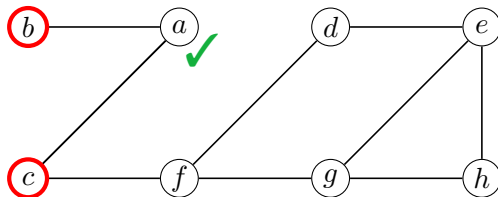
Illustration du parcours en profondeur



$$S = []$$

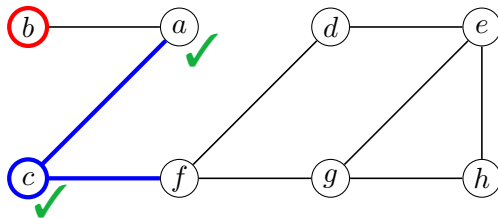
$$u = a$$

Illustration du parcours en profondeur



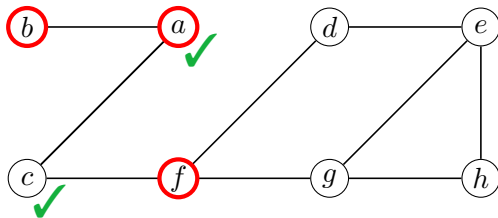
$$S = [b, c]$$

Illustration du parcours en profondeur



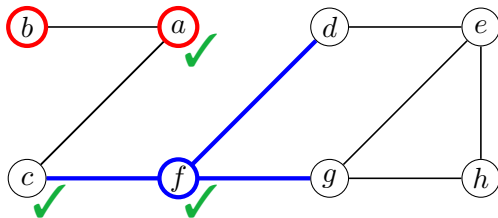
$$S = [b]$$
$$u = c$$

Illustration du parcours en profondeur



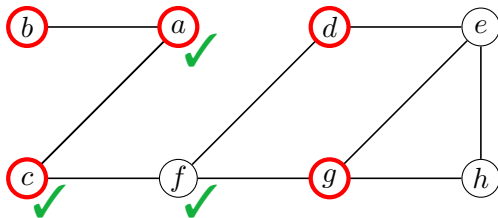
$$S = [b, a, f]$$

Illustration du parcours en profondeur



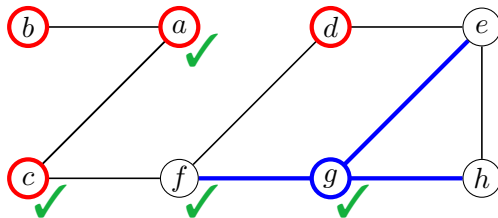
$$S = [b, a]$$
$$u = f$$

Illustration du parcours en profondeur



$$S = [b, a, c, d, g]$$

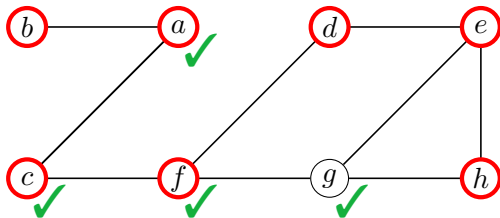
Illustration du parcours en profondeur



$$S = [b, a, c, d]$$

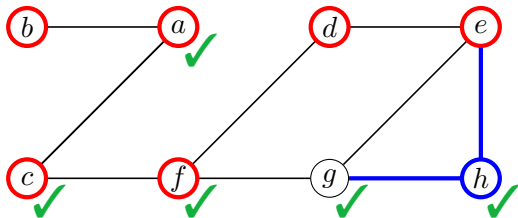
$u = g$

Illustration du parcours en profondeur



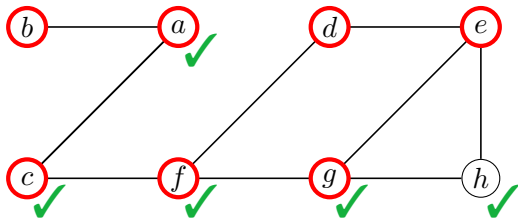
$$S = [b, a, c, d, e, f, h]$$

Illustration du parcours en profondeur



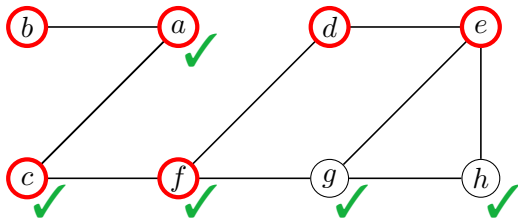
$$S = [b, a, c, d, e, f]$$
$$u = h$$

Illustration du parcours en profondeur



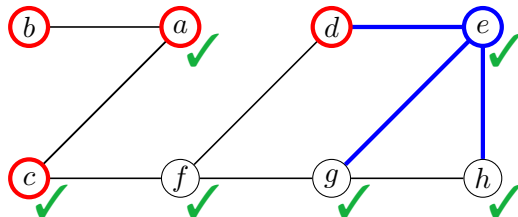
$$S = [b, a, c, d, e, f, e, g]$$

Illustration du parcours en profondeur



$$S = [b, a, c, d, e]$$

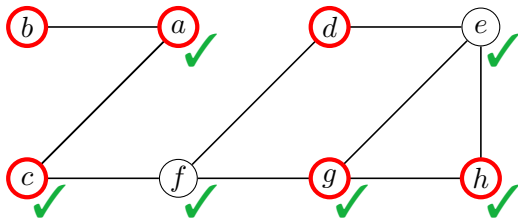
Illustration du parcours en profondeur



$$S = [b, a, c, d]$$

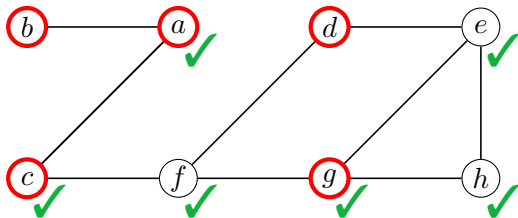
$u = e$

Illustration du parcours en profondeur



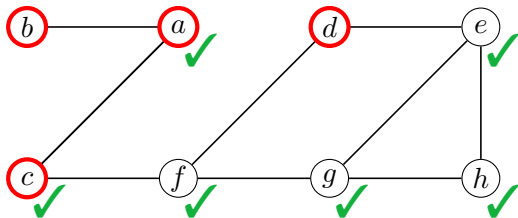
$S = [b, a, c, d, d, g, h]$

Illustration du parcours en profondeur



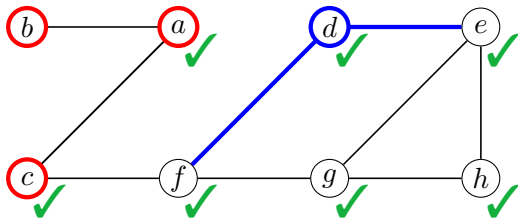
$$S = [b, a, c, d, d, g]$$

Illustration du parcours en profondeur



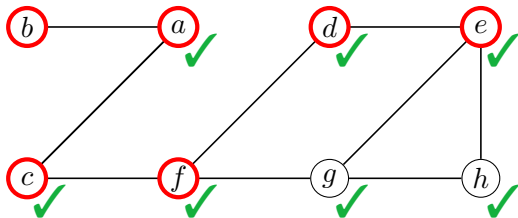
$$S = [b, a, c, d, d]$$

Illustration du parcours en profondeur



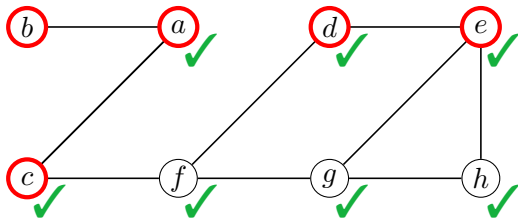
$$S = [b, a, c, d, d]$$
$$u = d$$

Illustration du parcours en profondeur



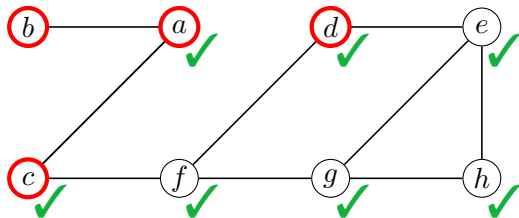
$$S = [b, a, c, d, e, f]$$

Illustration du parcours en profondeur



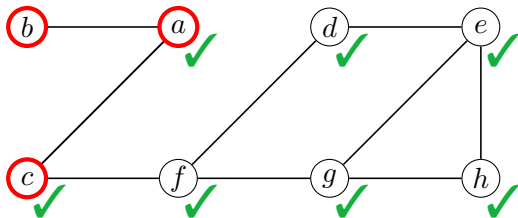
$$S = [b, a, c, d, e]$$

Illustration du parcours en profondeur



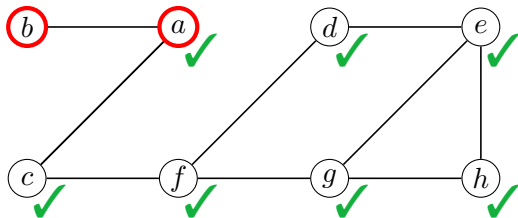
$$S = [b, a, c, d]$$

Illustration du parcours en profondeur



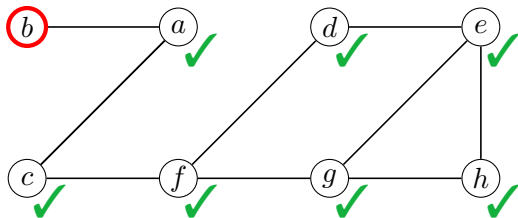
$$S = [b, a, c]$$

Illustration du parcours en profondeur



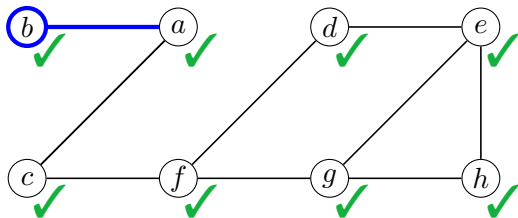
$$S = [b, a]$$

Illustration du parcours en profondeur



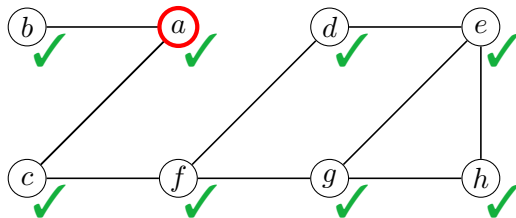
$$S = [b]$$

Illustration du parcours en profondeur



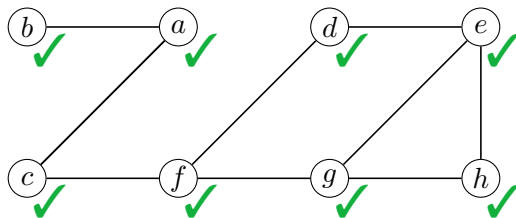
$S = []$
 $u = b$

Illustration du parcours en profondeur



$$S = [a]$$

Illustration du parcours en profondeur



$S = []$

Version recursive de DFS pour les graphes connexes

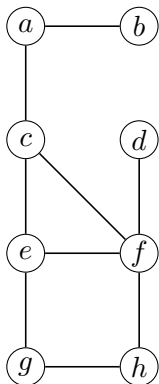
Procédure `explorer(G, u)` :

- | `marqué[u] \leftarrow Vrai`
- | **pour tous les** $(u, v) \in E(G)$ **faire**
 - | | **si** `marqué[v] = Faux` **alors**
 - | | | `explorer(G, v)`

Correction de la procédure `explorer(u)`

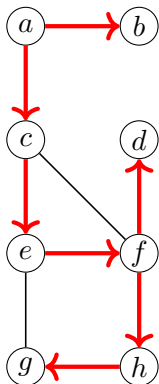
- Il faut montrer que la procédure `explorer(u)` visite tous les sommets atteignables à partir de u .
- Supposons par l'absurde que, à la fin d'exécution de `explorer(u)`, il existe un sommet v non marqué.
- Soit P une chaîne de u à v .
- Soit w le dernier sommet sur P (le plus lointain de u) qui est marqué.
- Soit x le successeur de w dans P .
- Contradiction : la procédure `explorer(w)` aurait marqué le sommet x .

Classification des arêtes



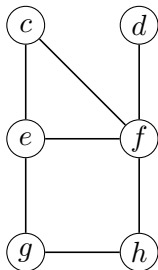
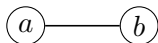
- Voici le résultat de l'exécution d'explorer sur un graphe, en commençant par le sommet a (et parcourant les arêtes par ordre alphabétique).
- Chaque fois qu'un nouveau sommet v est marqué, soit u le voisin de v
- Il y a une flèche rouge de u vers v si v a été marqué lors d'un appel de `explorer(v)` a été appelé quand l'algorithme traitait le sommet u .
- Ces arêtes forment un arbre.
- Les autres arêtes sont appelés les arêtes *retour*.

Classification des arêtes



- Voici le résultat de l'exécution d'explorer sur un graphe, en commençant par le sommet *a* (et parcourant les arêtes par ordre alphabétique).
- Chaque fois qu'un nouveau sommet *v* est marqué, soit *u* le voisin de *v*
- Il y a une flèche rouge de *u* vers *v* si *v* a été marqué lors d'un appel de `explorer(v)` a été appelé quand l'algorithme traitait le sommet *u*.
- Ces arêtes forment un arbre.
- Les autres arêtes sont appelés les arêtes *retour*.

... et si le graphe n'est pas connexe ?



Procédure $\text{explorer}(G, u)$:

marqué[u] \leftarrow Vrai

pour tous les $(u, v) \in E(G)$ **faire**

si marqué[v] = Faux **alors**

$\text{explorer}(G, v)$

Procédure $\text{DFS}(G)$:

pour tous les $u \in V(G)$ **faire**

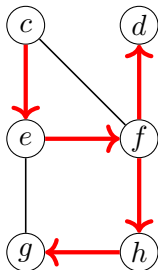
 marqué[u] \leftarrow Faux

pour tous les $u \in V(G)$ **faire**

si marqué[u] = Faux **alors**

$\text{explorer}(G, u)$

... et si le graphe n'est pas connexe ?



Procédure $\text{explorer}(G, u)$:

marqué[u] \leftarrow Vrai

pour tous les $(u, v) \in E(G)$ **faire**

si marqué[v] = Faux **alors**

$\text{explorer}(G, v)$

Procédure $\text{DFS}(G)$:

pour tous les $u \in V(G)$ **faire**

 marqué[u] \leftarrow Faux

pour tous les $u \in V(G)$ **faire**

si marqué[u] = Faux **alors**

$\text{explorer}(G, u)$

Composantes connexes d'un graphe

- On peut utiliser le parcours en profondeur pour identifier les composantes connexes d'un graphe.

Procédure prévisite(u) :

└ ccnum[u] = cc

Procédure explorer(G, u) :

└ marqué[u] ← Vrai

└ prévisite(u)

└ **pour tous les** $(u, v) \in E(G)$

└ **faire**

└ └ **si** marqué[v] = Faux **alors**

└ └ └ explorer(G, v)

Procédure DFS(G) :

└ cc ← 0

└ **pour tous les** $u \in V(G)$ **faire**

└ └ marqué[u] ← Faux

└ **pour tous les** $u \in V(G)$ **faire**

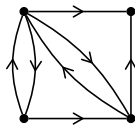
└ └ **si** marqué[u] = Faux **alors**

└ └ └ cc ← cc + 1

└ └ └ explorer(G, u)

Graphes orientés

- Un *graphe orienté* est un couple $G = (V, E)$ formé par un ensemble fini V et un sous-ensemble E de V^2 .
- Comme pour les graphes non orientés, V est l'ensemble des sommets de G .
- E est l'ensemble d'arcs (arêtes orientés).
- On représente les arcs par des flèches.
- Si $(u, v) \in E$, alors on met une flèche de u vers v ; u est la tête et v la queue de l'arc (u, v) .



Chemins (chaînes orientées)

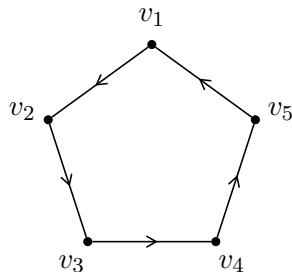


Définition

Un *chemin* dans un graphe orienté $G = (V, E)$ est une suite de la forme $(v_0, e_1, v_1, \dots, e_k, v_k)$ où

- $v_i \in V$
- $e_i \in E$
- $e_{i+1} = (v_i, v_{i+1})$ pour $i = 0, \dots, k - 1$.
- L'entier k est la *longueur* du chemin.

Circuits (cycles orientées)



Définition

Un *circuit* dans un graphe orienté $G = (V, E)$ est une suite de la forme $(v_0, e_1, v_1, \dots, e_k, v_0)$ où

- $v_i \in V$
- $e_i \in E$
- $e_{i+1} = (v_i, v_{i+1})$ pour $i = 0, \dots, k-1$.
- L'entier k est la *longueur* du chemin.

Parcours en profondeur dans les graphes orientés

Entrées : graphe $G = (V, E)$ et sommet $r \in V$

début

créer pile(S)

pour tous les $u \in V$ **faire**

└ marqué[u] \leftarrow False

empiler(S, r)

tant que $S \neq \emptyset$ **faire**

└ $u \leftarrow$ dépiler(S)

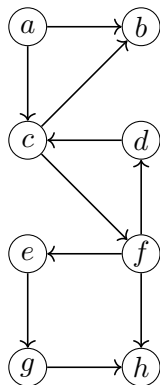
└ **si** marqué[u] = Faux **alors**

└└ marqué[u] \leftarrow Vrai

└└ **pour tous les** $(u, v) \in E$ **faire**

└└└ empiler(S, v)

Parcours en profondeur dans les graphes orientés (version récursive)



Procédure $\text{explorer}(G, u)$:

marqué[u] \leftarrow Vrai

pour tous les $(u, v) \in E(G)$ **faire**

si marqué[v] = Faux **alors**

$\text{explorer}(G, v)$

Procédure $\text{DFS}(G)$:

pour tous les $u \in V(G)$ **faire**

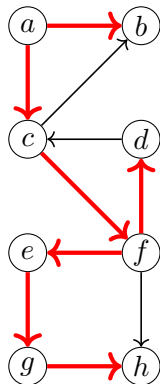
 marqué[u] \leftarrow Faux

pour tous les $u \in V(G)$ **faire**

si marqué[u] = Faux **alors**

$\text{explorer}(G, u)$

Parcours en profondeur dans les graphes orientés (version récursive)



Procédure $\text{explorer}(G, u)$:

marqué[u] \leftarrow Vrai

pour tous les $(u, v) \in E(G)$ **faire**

si marqué[v] = Faux **alors**

$\text{explorer}(G, v)$

Procédure $\text{DFS}(G)$:

pour tous les $u \in V(G)$ **faire**

 marqué[u] \leftarrow Faux

pour tous les $u \in V(G)$ **faire**

si marqué[u] = Faux **alors**

$\text{explorer}(G, u)$

Parcours en profondeur : pré- et post-visites

Procédure prévisite(u) :

```
┌   pre[s]  $\leftarrow t$   
└   t  $\leftarrow t + 1$ 
```

Procédure postvisite(u) :

```
┌   post[s]  $\leftarrow t$   
└   t  $\leftarrow t + 1$ 
```

Procédure explorer(G, u) :

```
┌   marqué[u]  $\leftarrow$  Vrai  
┌   prévisite( $u$ )  
┌   pour tous les  $(u, v) \in E(G)$   
┌       faire  
┌           si  $v$  non marqué alors  
┌               ┌ explorer( $G, v$ )  
└   postvisite( $u$ )
```

Procédure DFS(G) :

```
┌   t = 1  
┌   pour tous les  $u \in V(G)$   
┌       faire  
┌           ┌ marqué[u]  $\leftarrow$  Faux  
┌       pour tous les  $u \in V(G)$   
┌           faire  
┌               si marqué[u] = Faux  
┌                   alors  
┌                       ┌ explorer( $G, u$ )
```

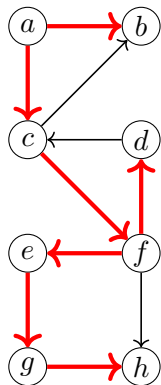
Intervalles imbriqués

Observation (Théorème des parenthèses)

Pour tout sommet u et v , les deux intervalles $[\text{pre}(u), \text{post}(u)]$ et $[\text{pre}(v), \text{post}(v)]$ sont soit disjoints, soit l'un est contenu dans l'autre.

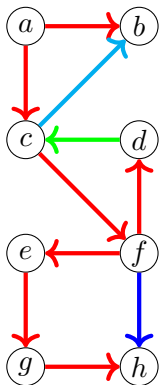
- L'intervalle $[\text{pre}(u), \text{post}(u)]$ représente le temps pendant lequel le sommet u était sur la pile S .
- Si $[\text{pre}(u), \text{post}(u)] \cap [\text{pre}(v), \text{post}(v)] \neq \emptyset$, alors il existe un temps t auquel u et v étaient dans la pile S .
- Si u a été empilé avant v , alors u sera dépilé après v , et donc $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$.
- De même, si v a été empilé avant u , alors $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$.

Terminologie pour l'analyse du BFS



- a est la *racine* de l'arbre.
- Les autres sommets sont des *descendants* de a .
- De même, f a des *descendants* e , g et h .
- Inversement, f est un *ancêtre* de e , g et h .
- Les ancêtres immédiats sont les *parents*, et les descendants immédiats sont les *enfants* : c est le parent de f , et f est l'enfant de c .

Classification des arcs (1/3)



Un parcours en profondeur dans un graphe orienté G donne lieu à 4 types d'arcs de G .

On dit que l'arc (u, v) est :

1. un arc *de l'arbre* si u est un parent de v .
2. *avant* si u est un ancêtre (non parent) de v
3. *retour* si v est un ancêtre de u
4. *transverse* dans les autres cas

Classification des arcs (2/3)

- u est un ancêtre de v ssi u est marqué en premier et v est marqué pendant $\text{explore}(u)$ ssi $[\text{pre}(u), \text{post}(u)] \supset [\text{pre}(v), \text{post}(v)]$.
- Puisque u est un descendant de v ssi v est un ancêtre de u , (u, v) est un arc retour ssi $[\text{pre}(u), \text{post}(u)] \subset [\text{pre}(v), \text{post}(v)]$.
- Finalement, (u, v) est transverse ssi $[\text{pre}(u), \text{post}(u)] \cap [\text{pre}(v), \text{post}(v)] = \emptyset$.

Classification des arcs (3/3)

- Notons par $[_u \]_u$ l'intervalle $[\text{pre}[u], \text{post}[u]]$.
- Voici un résumé des différentes possibilités pour un arc (u, v) :

$[_u \]_v \]_v \]_u$ arcs de l'arbre, avant

$]_v \]_u \]_u \]_v$ arcs retour

$]_v \]_v \]_u \]_u$ arcs trasverses

Remarque

Soit (u, v) un arc. Si $\text{post}(u) < \text{post}(v)$, alors (u, v) est un arc retour.

Complexité du parcours en profondeur

- Chaque sommet n'est exploré qu'une seule fois, grâce au marquage.
- Pendant l'exploration d'un sommet, il y a les étapes suivantes :
 1. marquer le sommet (et éventuellement la pré- et la post-visite).
 2. parcourir les arêtes incidentes à u pour voir si elles mènent à un sommet non marqué.
- Cette boucle prend un temps différent pour chaque sommet ; considérons donc tous les sommets ensemble.
- Le temps total de l'étape 1 est alors $O(n)$.
- Dans l'étape 2, chaque arête $uv \in E$ est examinée exactement deux fois — une fois pendant $\text{explorer}(u)$ et une fois pendant $\text{explorer}(v)$.
- On conclut que la complexité du parcours en profondeur est de $O(m + n)$ (égale à celle du parcours en largeur).

Graphes orientés acycliques

Définition

Un graphe orienté sans circuits est dit *acyclique*.

Observation

Un graphe orienté contient un circuit ssi le parcours en profondeur trouve une arête retour.

Démonstration (1/2)

- Soit G un graphe orienté et soit T l'arbre DFS, avec racine r .
- Supposons que (u, v) est un arc retour.
- v est donc un ancêtre de u ; il existe un chemin P de v à u dans T .
- P et l'arc (u, v) forment un circuit.

Graphes orientés acycliques (DAG¹)

Démonstration (2/2)

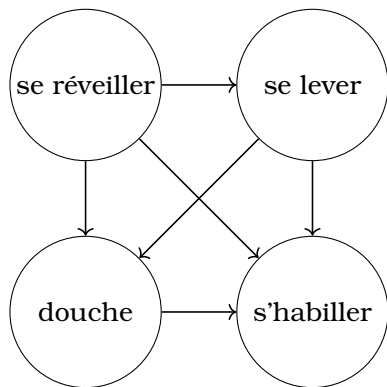
- Inversement, si le graphe possède un cycle $C = (v_1, v_2, \dots, v_k, v_1)$, soit v_i le premier sommet visité de C .
- Tous les autres sommets v_j de C sont atteignables à partir de v_i et seront donc ses descendants dans T .
- En particulier, l'arc (v_{i-1}, v_i) (ou (v_k, v_1) au cas où $i = 1$) est un arc retour.

1. Pour *directed acyclic graph*

À quoi ça sert... ?

- Les DAG permettent de modéliser des relations telles que :
 - les causalités
 - les hiérarchies
 - les dépendances temporelles
- Par exemple, supposons que vous deviez effectuer de nombreuses tâches, mais que certaines d'entre elles ne puissent pas commencer avant que d'autres ne soient terminées.
- La question qui se pose alors est de savoir quel est l'ordre valide dans lequel les tâches doivent être accomplies.

Exemple



- Vous devez vous réveiller avant de se lever.
- Vous devez être levé, mais pas encore habillé, pour prendre une douche.

L'existence d'un bon ordre

- De telles contraintes sont commodément représentées par un graphe orienté dans lequel chaque tâche est un sommet, et il existe un arc de u à v si u est une *précondition* pour v .
- En d'autres termes, avant d'exécuter une tâche, toutes les tâches qui y sont liées doivent être achevées.
- Si ce graphe orienté comporte un circuit, il n'y a pas de solution.
- Si par contre le graphe est un DAG, on aimerait ordonner les sommets de sorte que chaque arête aille d'un sommet antérieur à un sommet postérieur, afin que toutes les contraintes de précédence soient satisfaites.

Dans cet exemple, il existe (heureusement !) un bon ordre

