# Chapitre 3 : Algorithmes élémentaires pour les graphes

Y. DIENG, Département Informatique
UFR ST, Université Assane Seck de Ziguinchor

30. august 2021

## 1 Representation de graphe

L'objectif de cette section est de présenter quelques méthodes de représentation de graphe.

Pour décrire un graphe, un certain nombre de représentations peuvent etre utilisées. En pratique, elles ne sont generalement pas equivalentes du point de vue de l'efficacité des algorithmes. Nous présentons dans ce chapitre, deux des représentations de graphe les plus courantes : `les listes d'adjacences et les matrices d'adjacences.` La représentation par listes d'adjancence est souvent préférée, car elle fournit un moyen peu encombrant de representer un graphe peu dense.

### 1.1 Liste d'adjacence

Dans une liste d'ajacence, les sommets sont désignés par des entiers entre 1 et $n$ ( $n$ étant le nombre de sommets). On a :

1. Une implémentation par un `Tableau` : les sommets de la liste d'adjacence sont rangés consécutivement dans un tableau (avantage pour les langages sans pointeur)

   - Un tableau `Succ` est rempli dans l'ordre avec les listes de successeurs des sommets 1, 2, ..., $n$ (tableau des listes de successeurs)
   - Succ a donc autant d'élément qu'il y en a d'arcs dans le graphe ($m$ note le nombre d'arcs du graphe)
   - Pour délimiter les listes, un tableau `Head` à $n$ éléments donne pour tout sommet l'indice dans `Succ` où commencent ses successeurs. (tableau des têtes de liste)
   - les successeur d'un sommet $y$ sont rangés dans `Succ` entre l'indice $Head[y]$ à $Head[y+1]-1$ inclus.
   - si un sommet $y$ n'a pas de successeurs, on a $Head[y] = Head[y+1]$.
   - Pour éviter les test pour le cas particulier $y = n$, on pose par convention $Head[n+1] = m+1$.

Avec ces conventions, la boucle suivante énumère tous les successeurs d'un sommet $y$ même si $y = n$.

```
Writeln ('Liste des successeurs du sommet , y ,':');
For k := Head[y] to Head[y + 1] − 1
    Writeln(Succ[k]);
```

- Un graphe simple est codé comme un graphe orienté symétrique
  - Si $(x, y)$ est une arête, alors $x$ apparait dans les successeurs de $y$ et $y$ dans ceux de $x$.
- Pour un graphe valué, on crée un tableau $W$ en regard de $Succ$.
  - Si $x$ a un succeseur de $y$ à l'indice $k$ de Succ, le poids de $(x, y)$ sera donc dans $W[k]$

2. Implémentation par des `Listes chainées` : les sommets sont représentés par un tableau $Adj$ de $\mid V \mid$ listes chainées, une liste pour chauque sommet de $V$. Pour chaque sommet $u \in V$, la liste $Adj[u]$ est une liste de sommet de $V$ tel qu'il existe un arc $(u, v) \in E$. Ainsi, $Adj[u]$ est constitué de tous les sommets adjacents a $u$ dans $G$. L'ordre du chainage des sommets de chaque liste est généralement arbitraire.

   Si $G$ est un graphe orienté, la somme des longueurs de toutes les listes d'adjancence vaut $\mid E \mid$. En effet, l'existance d'un arc $(u, v)$ se traduit par la présence de $v$ dans $Adj[u]$. Dans le cas d'un graphe non orienté, la somme des longueurs de toutes les listes d'adjancence vaut $2 \mid E \mid$. Qu'un graphe soit orienté ou non, la représentation par liste d'adjacence à la propriété avantageuse de ne pas demander une quantité de memoire en $O(max(V, E))$.

## Avantages

1. Capacité:
   - Le codage d'un graphe $G$ utilise $n + 1 + m$ mots-memoire et non $n^2$ mots pour une matrice si le graphe est dense.
2. Meilleurs complexité pour les algorithmes utilisant les listes:
   - Les successeurs d'un sommet sont regroupés ($d^+(x) = Head[x+1] - Head[x]$ ). Ce qui est une instruction à la place de compter le nombre de 1 dans une matrice d'adjacence.

## Les inconvénients

1. Existance d'un arc : test non immédiat
   - On a besoins de balayer $Succ$ en $O(m)$ alors que dans une martice d'adjacence on a besoins de balayer juste une ligne en $O(n)$.

## 1.2 Matrice d'adjacence

Soit un graphe orienté $G = (V, E)$ avec $V$ l'ensemble des sommets et $E$ l'ensemble des arêtes. Soit $n = |V| etm = |E|$

- Une matrice `d'adjacence` est une matrice $A$ de taille $n \times n$ à élément bouléens ou dans $\{0, 1\}$.
  - L'élément $A_{i,j}$ vaut 1 (ou True) si et seulement si l'arc $(i, j)$ existe dans le graphe.
  - La taille minimale théorique d'une telle matrice est de $n^2$.
- Quand un graphe est `valué`, il est représenté par une matrice $W$ de taille $n \times n$ donnant les coûts de chaque arc et faisant fonction de matrice d'adjancence.
  - Il faut cependant choisir une valeur spéciale, non présente dans les coûts, pour indiquer l'abssence d'arcs: ce n'est pas forcément 0.

### Exemple : Un Graphe sa matrice d'adjacence

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 1 | 0 | 1 | 0 | 0 | 1 |
| 3 | 0 | 1 | 0 | 1 | 1 | 0 |
| 4 | 1 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | 1 | 0 | 0 | 1 |
| 6 | 1 | 1 | 0 | 0 | 1 | 0 |

### Les avantages

- La simplicité
- La facilité d'adaptation aux graphes valués
- La facilité de détection des boucles (1 sur la diagonale pricipale)
- La symetrie
- La facilité de vérification de l'existance d'un arc entre deux sommets $i$ et $j$ (vérifier la valeur de $A[j][j]$).
- La facilité de detection des successeurs d'un sommet $i$ (un balayage de la ligne $i$).
- La facilité de detection des prédécésseurs d'un sommet $i$ (un balayage de la colone $i$).

### Les inconvénients

- Consommation de trop de mémoire lorsque le graphe est peu danse.
- Exemple : Dans un graphe modélisant une carte routière, le degré moyen est proche de 3
  - Avec un tel graphe ayant 1 000 sommets, on aura besoin d'une matrice de 1 Go alors que dans la matrice on aura 3 000 éléments significatifs (égaux à 1).
- Les algorithmes de graphe basés sur des matrices d'adjacance ne sont pas très éfficaces : car le balayage des successeurs de d'un sommet $i$ perd du temps à tester des $A[i][j]$ nuls.

## 2 Parcours de graphe

L'objectif de cette section est de présenter quelques méthodes de parcours de graphe. Le parcours d'un graphe consiste à emprunter de facon systématique les arcs du graphe dans l'objectif de visiter les sommets du graphe. Un algorithme de parcours de graphe permet de mettre en évidence plusieurs caractéristiques de la structure du graphe. Ainsi, pour obtenir les caractéristiques structurelles des graphes, les algorithmes utilisent souvent le résultat d'un parcours. D'autres sont simplement des adaptations des algorithmes élémentaires de parcours de graphe. C'est ce qui explique le fait que les techniques de parcours de graphe soient au coeur des algorithmes de graphe.

Dans ce chapitre, nous présentons les deux algorithmes de base de parcours de graphes. Le premier algorithme présente est `le parcours en largeur`. Une application de cet algorithme, présentée dans ce chapitre, est comment créer une arborescence de parcours en largeur. Le deuxième algorithme présente est `le parcours en profondeur` . Nous présenterons aussi quelques résultats fondamentaux sur l'ordre de visite des sommets. Comme application de ce parcours, le prétexte qui sera utilisé est le tri topologique d'un graphe oriente sans circuit.

### 2.1 Definitions sur les parcours

Un `chemin` $\mu$ de `longueur` $q$ est une séquence de $q$ arcs $(u_1, u_2, \ldots, u_q)$. L'extrémité initiale de $u_i$ il doit être l'extrémité terminal de $u_{i+1}$ si $i \geq 1$, et l'extrémité terminal de $u_i$ doit être l'extrémité initial de $u_{i+1}$ si $i \leq q$. Un chemin fermé est un `circuit`. Un arc peut être vu comme un chemin de longueur 1, et une boucle comme un circuit de longueur 1.

Un sommet $y$ `descendant` de $x$ s'il est situé sur un chemin d'origine $x$. On dit aussi que $x$ est `ascendant`, ou `ancêtres` des $y$. Notez que, contrairement au monde vivant, un sommet peut être à la fois ascendant et descendant d'un autre, s'ils sont situés sur un meme circuit.

Une `chaine` de `longueur` $q$ est une séquence de $q$ arêtes $[e_1, e_2, \ldots, e_q]$ $e_i$ a une extrémité commune avec $e_{i-1}$ si $i \geq 1$ et une autre avec $e_{i+1}$ si $i \leq q$. Une chaîne fermee est un `cycle`. Une arête est une chaine de longueur 1. On peut

considérer des chaînes sur un Graphe orienté en parlant du graphe non orienté associés.

Le terme de `parcours` regroupe les chemins, les circuits, les chaines et les cycles. Une parcours est `élémentaire` s'il emprunte qu'une fois ses sommets. Dans un graphe simple, on peut définir un parcours par une suite de sommets, en répétant le premier sommet à la fin pour un cycle ou un circuit : ainsi $(x, y)$ désigne un chemin réduit à l'arc $(x, y)$, tandis que $(x, y, z)$ indique un circuit passant sur deux arcs $(x, y)$ et $(y, x)$. Par convention, un simple sommet sera considéré comme un parcours de longueur 0.

## 2.2 Pacours en largeur

L'algorithme du parcours en largeur est l'un des algorithmes de parcours les plus simple, et il est à la base de nombreux d'algorithmes importants sur les graphes. L'algorithme de `Dijkstra` pour calculer les plus courts chemins à origine unique et l'algorithme de `Prim` pour trouver l'arbre couvrant minimum utilisent des idees similaires à celles qui sont appliquees l'ors d'un parcours en largeur.

Étant donné un graphe $G = (V, E)$ et un sommet origine $s$, le parcours en largeur emprunte systématiquement les arc de $G$ pour découvrir tous les sommets accessibles depuis $s$. Il calcule la distance entre $s$ et tous les autres sommets accessibles. Il construit également une arborescence de parcours en largeur de racine $s$, qui contient tous les sommets accessibles depuis $s$.

Pour tout sommet $v$ accessible depuis $s$, le chemin reliant $s$ à $v$ dans l'arborescence de parcours en largeur correspond à un plus court chemin de $s$ vers $v$ dans $G$, autrement dit un chemin contenant le plus petit nombre d'arc. L'algorithme fonctionne aussi bien sur les graphes orientes que sur les graphes non orientes.

L'algorithme de parcours en largeur tient son nom au fait qu'il decouvre d'abord tous les sommets situes a une distance k de s avant de decouvrir tout sommet situe a la distance k + 1.

Pour un meilleur suivi sur la progression du pqrcours, l'algorithme du parcours en largeur colorie chaque sommet en blanc, gris, ou noir. Au départ, tous les sommets sont considérés comme blancs, et peuvent devenir gris, puis noirs. Un sommet est découvert la premiere fois qu'il est rencontré au cours de la recherche et il perd alors sa couleur blanche pour devenir gris. Les sommets gris ou noirs ont donc été découverts, et la distinction entre gris et noir permet de garantir que la recherche se poursuit en largeur d'abord. Si $(u, v) \in E$ et si le sommet $u$ est noir, alors le sommet $v$ est gris ou noir ; autrement dit, tout sommet adjacent à un sommet noir a déja été découvert. Les sommets gris peuvent etre adjacents à des sommets blancs. Les sommets gris représentent donc la frontiere entre les sommets découverts et les sommets non découverts.

Le parcours en largeur construit une arborescence de parcours en largeur, qui ne contient initialement que sa propre racine, représentant le sommet origine $s$. A chaque découverte d'un sommet $v$ blanc pendant le balayage de la liste d'adjacences d'un sommet $u$ déja découvert, le sommet $v$ et l'arc $(u, v)$ sont ajoutés a l'arborescence. On dit alors que $u$ est le prédécesseur ou parent de $v$ dans l'arborescence de parcours en largeur. Comme un sommet est découvert au plus une fois, il possede au plus un parent. Les relations ancêtre/descendant

dans l'arborescence de parcours en largeur sont définies relativement a la racine $s$ de la maniere habituelle : si $u$ se trouve sur un chemin de l'arbre entre la racine $s$ et le sommet $v$, alors $u$ est un ancêtre de $v$ et $v$ est un descendant de $u$.

## L'algorithme du parcours en largeur

Soit $G = (V, E)$ représenté par une liste d'adjancence et un sommet $s$ choisi pour etre la racine. Pour chaque sommet $u \in G$, on a les structures de données suivantes:

- $couleur[u]$ qui stocke sa couleur
- $parent[u]$ qui stocke l'dentifiant de son parent; si $u$ n'a pas de parent, alors $parent[u] = NIL$
- $d[u]$ qui est la distance calculee par l'algorithme entre le sommet $u$ et le sommet origine $s$
- $F$ une file pour gerer l'ensemble des sommets gris

## Procedure parcoursEnLargeur (G, s)

1. pour chaque sommet $u \in G - s$, faire
   - $couleur[u] \leftarrow BLANC$
   - $d[u] \leftarrow \infty$
   - $parent[u] \leftarrow NIL$
2. $couleur[s] \leftarrow GRIS$
3. $d[s] \leftarrow 0$
4. $parent[u] \leftarrow NIL$
5. $F \leftarrow \{s\}$
6. Tant que $F \neq \emptyset$ Faire
   - $u \leftarrow tete(F)$
   - Pour chaque sommet $v \in Adj[u]$ faire
     - Si $couleur[v] \leftarrow BLANC$ alors
       - $couleur[v] \leftarrow GRIS$
       - $d[v] \leftarrow d[v] + 1$
       - $parent[v] \leftarrow u$
       - ENFILER $(F, v)$
       - DEFILER $(F)$
   - $couleur[u] \leftarrow NOIR$

**Théorème 1.** *Le temps d'exécution de l'algorithme sur un graphe $G = (V, E)$ est $O(V + E)$.*

*Bevis.* Soit $G = (V, E)$ un graphe donné à notre algorithme en entrée. Après l'initialisation, aucun sommet n'est coloré en blanc et le test $Si$ de la boucle garantit que chaque sommet est enfilé au plus une fois et, par temps défilé au plus une fois. Comme les opération d'enfilement et de défilement sont en $O(1)$, le temps total des opérations de la file est $O(V)$. Vu que la liste d'adjacence de chaque sommet n'est balayé que lorsque le sommet est défilé, alors, la liste d'ajacence de chaque sommet est parcourue au plus une fois. Comme la somme de longueurs de toutes les listes d'adjacence est $\theta(E)$, alors, le temps d'exécution total est $O(V + E)$. Le parcours en largeur s'exécute en un temps qui est linéaire par rapport à la taille de la représentation par liste d'adjacence de $G$.

Dans la suite de cette partie, il sera question de montrer que, comme dans un parcours en largeur, notre algorithme trouve la distance entre le sommet $s$ et tout autre sommet du graphe. Soit $\delta(s, v)$ le nombre minimal d'arcs d'un chemin reliant $s$ à $v$ qui est aussi, la distance dans $G$ entre $s$ et $v$. S'il n'existe pas de chemin entre $s$ et $t$, on $\delta(s, v) = \infty$. On dit qu'un chemin de longueur $\delta(s, v)$ de $s$ à $t$ est un plus court chemin e $s$ à $t$. Le lemme suivant sera utile pour la suite.

**Lemme 1.** *Soit $G$ un graphe orienté ou non, et soit $s \in E$ un sommet arbitraire. Alors, pour tout arc $(u, v) \in E$ on a : $\delta(s, v) \leq \delta(s, u) + 1$.*

*Bevis.* Si $u$ est accessible depuis $s$, alors $v$ l'est aussi. Dans ce cas, le plus cour chemin de $s$ à $v$ ne peut pas être plus long que le plus cours chemin de $s$ à $u$. prolongé par l'arc $(u, v)$; l'inégalité est donc valide. Si $u$ ne peut pas être atteint depuis $s$, alors $\delta(s, v) = \infty$, et l'inégalité est véroifiée.

On cherche à montrer que pour n'importe quel sommet $v \in V$ $d[v] = \delta(s, v)$. Pour ce faire, nous allons d'abord montrer que $d[v]$ est un majorant de $\delta(s, v)$.

**Lemme 2.** *Soit $G$ un graphe tel que notre algorithme est exécuté sur $G$ à partir d'un sommet $s$. Alors quant l'algorithme termine , on a pour tout sommet $v$ de $G$, $d[v] \geq \delta(s, v)$.*

*Bevis.* A faire, ...

Pour démontrer que $d[v] = \delta(s, v)$, nous avons besoins de bien caractérirer le comportement de la file $F$ au cours de l'exécution. Nous avons besoin du lemme suivant:

**Lemme 3.** *Supposons que pendant l'exécution de l'algorithme sur un graphe $G = (V, E)$, la file $F$ contient $\{v_1, v_2, \ldots, v_r\}$, avec $v_1$ la tête de la file $F$ et $v_r$ la queue de la file $F$.*

*Alors $d[v_r] \leq d[v_1] - 1$ et $d[v_i] \leq d[v_{i+1}] - 1$ pour $i = 1, 2, 3, \ldots, r - 1$.*

*Bevis.* A faire, ...

Comme les éléments de la file sont ajoutés successivement en partant de $v_1$ jusqu'à $v_r$. Alors, on a le corrolaire suivant:

**Corollaire 1.** *Si deux sommets $v_i$ et $v_j$ sont enfilés et que $v_i$ est enfilé avant $v_j$ durant l'exécution. Alors $d[v_i] \leq d[v_j]$ au moment où $v_j$ est enfilé.*

On est maintenant à même d'énoncer le théorème suivant:

**Théorème 2.** *Soit $G = (V, E)$ un graphe sur lequel, nous exécutons notre algorithme à partir d'un sommet $s$ donné. Alors, pendant l'exécution, l'algorithme découvre chaque sommet $v$ de $V$ accessible à partir de $s$ et à la fin on a $d[v] = \delta(s, v)$.*

### Arborescence de pacours en largeur

Pendant le parcours du graphe, la procédure construit l'arborescence du parcours en largeur. Cet arborescence est représenté dans le champ *Parent*. Plus formellement, pour un graphe $G = (V, E)$ d'origine $s$ on définit le sous graphe prédécésseur de $G$ par $G_p = (V_p, E_p)$ où :

$V_p = \{v \in V : parent[v] \neq NIL\} \cup \{s\}$

et

$E_p = \{(parent[v], v) \in E_p : v \in V_p - \{s\}\}$

Le sous graphe prédécésseur $G_p = (V_p, E_p)$ est une arborescence de parcours en largeur si $V_p$ est constitué des sommets accessibles depuis $s$. Tous chemin élémentaire de $s$ à $v$ dans $G_p$ est un plus court chemin de $s$ à $v$ dans $G$.

En executant l'algorithme sur un graphe $G$ et en supposant que le tableau *parant* est produit à la fin de cet exécution. L'algorithme ci-après permet d'imprimer les sommets d'un plus court chemin de $s$ à $v$.

```
 IMPRIMER_CHEMIN(G, s, v )
1  Si v = s alors
2       Imprimer (s)
3  Sinon si parent[v] = NIL alors
4                        imprimer ("Il n'existe pas de chemin entre s et v")
5            Sinon IMPRIMER_CHEMIN(G, s, parent[v] )
6                        imprimer  (v)
```

Le temps d'exécution de cet algorithme est linéaire par rapport au nombre de sommets du chemin.

## 2.3 Parcours en profondeur

La stratégie suivie par un parcours en profondeur est, comme son nom l'indique, de descendre plus `profondément` dans le graphe chaque fois que c'est possible. Lors d'un parcours en profondeur, les arcs sont explorés à partir du sommet $v$ découvert le plus récemment et dont on n'a pas encore exploré tous les arcs qui en partent. Lorsque tous les arcs de $v$ ont été explorés, l'algorithme `revient en arrière` pour explorer les arcs qui partent du sommet à partir duquel $v$ a été découvert. Ce processus se répète jusqu'à ce que tous les sommets accessibles à partir du sommet origine initial aient été découverts. S'il reste des sommets non découverts, on en choisit un qui servira de nouvelle origine, et le parcours reprend à partir de cette origine. Le processus complet est répété jusqu'à ce que tous les sommets aient été découverts. Comme dans le parcours en largeur, chaque fois qu'un sommet $v$ est découvert pendant le balayage d'une liste d'adjacences d'un sommet $u$, le parcours en profondeur enregistre cet événement en donnant la valeur $u$ à $p[v]$, parent de $v$. Contrairement au parcours

en largeur, pour lequel le sous-graphe prédécesseur forme une arborescence, le sous-graphe prédécesseur obtenu par un parcours en profondeur peut être composé de plusieurs arborescences, car le parcours peut être répété à partir de plusieurs origines. Le sous-graphe prédécesseur d'un parcours en profondeur est donc défini un peu différemment de celui d'un parcours en largeur : on pose $G_p = (V, E_p)$, où

$E_p = \{(parent[v], v) : v \in V_p \text{ et } parent[v] \neq NIL\}$

Le sous-graphe prédécesseur d'un parcours en profondeur forme une forêt de parcours en profondeur composée de plusieurs arborescences de parcours en profondeur. Les arcs de $E_p$ sont des arcs de liaison.

Comme dans le parcours en largeur, les sommets sont coloriés pendant le parcours pour indiquer leur état. Chaque sommet est initialement blanc, puis gris quand il est découvert pendant le parcours, et enfin noirci en fin de traitement, c'est à dire quand sa liste d'adjacences a été complètement examinée. Cette technique assure que chaque sommet appartient à une arborescence de parcours en profondeur et un seul, de sorte que ces arborescences sont disjointes.

En plus de créer une forêt de parcours en profondeur, le parcours en profondeur date chaque sommet.

Chaque sommet $v$ porte deux dates : la première, $d[v]$, marque le moment où $v$ a été découvert pour la première fois (et colorié en gris), et la seconde, $f[v]$, enregistre le moment ou le parcours a fini d'examiner la liste d'adjacences de $v$ (et le colorie en noir). Ces dates sont utilisées dans de nombreux algorithmes de graphes et sont utiles pour analyser le comportement du parcours en profondeur.

La procédure ci-après enregistre le moment où elle découvre le sommet $u$ dans la variable $d[u]$, et le moment où elle termine le traitement du sommet $u$ dans la variable $f[u]$. Ces dates sont des entiers compris entre 1 et $2|S|$, puisque découverte et fin de traitement se produisent une fois et une seule pour chacun des $|S|$ sommets. Pour tout sommet $u$,

$d[u] \lneqq f[u]$,

Le sommet $u$ est BLANC avant l'instant $d[u]$, GRIS entre $d[u]$ et $f[u]$, et NOIR après. Le pseudo code suivant correspond à l'algorithme de base de parcours en profondeur. Le graphe d'entrée $G$ peut être orienté ou non. La variable date est une variable globale qui sert à la datation.

```
PP(G)
1 Pour chaque sommet u de G Faire
2      couleur [u] = BLANC
3      parent[u] = NIL
4 date = 0
5 Pour chaque sommet u de G Faire
6        Si couleur [u]  == BLANC alors
7                      visiter-PP(u)

 visiter-PP(u)
 1    couleur [u] =  GRIS
 2   date = date + 1
 3   d[u] = date
 4   Pour chaque sommet v adjacent à u faire
 5          Si couleur[v] == BLANC alors
 6                       parent[v] = u
```

```
7                       visiter-PP(v)
8   couleur[v] = NOIR
9   f[u] = date
10 date = date +1
```

Notez que le résultat du parcours en profondeur peut dépendre de l'ordre dans lequel les sommets sont examinés en ligne 5 de PP, et de l'ordre dans lequel les voisins d'un sommet sont visités en ligne 4 de VISITER-PP. Ces ordres de visite différents ne posent généralement pas problème dans la pratique, dans la mesure où chaque résultat d'une recherche en profondeur fournit généralement des sorties équivalentes.

## Propriétés du parcours en profondeur

Le parcours en profondeur révèle beaucoup d'informations sur la structure d'un graphe. La propriété la plus fondamentale du parcours en profondeur est sans doute que le sous-graphe prédécesseur $G_p$ forme en fait une forêt, puisque la structure des arborescences de parcours en profondeur reflète exactement la structure des appels récursifs à VISITER-PP. Autrement dit, $u = p[v]$ si et seulement si VISITER-PP(v) a été appelé pendant le parcours de la liste d'adjacences de $u$. En outre, le sommet $v$ est un descendant du sommet $u$ dans la forêt de parcours en profondeur si et seulement si $v$ est découvert pendant que $u$ est gris.

Une autre propriété importante du parcours en profondeur tient au fait que les dates de découverte et de fin de traitement ont une structure parenthésée. Si l'on représente la découverte d'un sommet $u$ par une parenthèse gauche « (u » et la fin de son traitement par une parenthèse droite « u) », alors la succession des découvertes et des fins de traitement crée une expression bien formée, au sens où les parenthèses sont correctement imbriquées. Une autre manière d'énoncer la condition de structure parenthésée est donnée par le théorème suivant.

**Théorème 3.** *(Théorème des parenthèses) Dans un parcours en profondeur d'un graphe $G = (V, E)$ (orienté ou non), pour deux sommets quelconques $u$ et $v$, une et une seule des trois conditions suivantes est vérifiée :*

- *les intervalles $[d[u], f[u]]$ et $[d[v], f[v]]$ sont complètements disjoints, et ni $u$ ni $v$ n'est un descendant de l'autre dans la forêt de parcours en profondeur,*
- *l'intervalle $[d[u], f[u]]$ est entièrement inclus dans l'intervalle $[d[v], f[v]]$, et $u$ est un descendant de $v$ dans une arborescence de parcours en profondeur, ou*
- *l'intervalle $[d[v], f[v]]$ est entièrement inclus dans l'intervalle $[d[u], f[u]]$, et $v$ est un descendant de $u$ dans une arborescence de parcours en profondeur.*

**Corollaire 2.** *(Imbrications des intervalles des descendants) Le sommet $v$ est un descendant propre du sommet $u$ dans la forêt de parcours en profondeur d'un graphe $G$ (orienté ou non) si et seulement si $d[u] < d[v] < f[v] < f[u]$.*

Le théorème suivant donne une autre caractérisation importante des descendants d'un sommet dans une forêt de parcours en profondeur.

**Théorème 4.** *(Théorème du chemin blanc) Dans une forêt de parcours en profondeur d'un graphe $G = (V, E)$ (orienté ou non), un sommet $v$ est un descendant d'un sommet $u$ si et seulement si, au moment $d[u]$ où le parcours découvre $u$, il existe un chemin de $u$ à $v$ composé uniquement de sommets blancs.*

### classification des arcs

Une autre propriété intéressante du parcours en profondeur est que le parcours peut servir à classer les arcs du graphe d'entrée $G = (V, E)$. Cette classification des arcs peut servir à glaner d'importantes informations concernant le graphe. Par exemple, dans la section suivante, nous verrons qu'un graphe orienté est acyclique si et seulement si un parcours en profondeur n'engendre aucun arc « arrière ».

Il est possible de définir quatre types d'arcs en fonction de la forêt de parcours en profondeur $G_p$ obtenue par un parcours en profondeur sur $G$.

- Les arcs de liaison sont les arcs de la forêt de parcours en profondeur $G_p$. L'arc $(u, v)$ est un arc de liaison si $v$ a été découvert la première fois pendant le parcours de l'arc $(u, v)$.
- Les arcs arrières sont les arcs $(u, v)$ reliant un sommet $u$ à un ancêtre $v$ dans une arborescence de parcours en profondeur. Les boucles (graphes orientés uniquement) sont considérées comme des arcs arrières.
- Les arcs avants sont les arcs $(u, v)$ qui ne sont pas des arcs de liaison, et qui relient un sommet $u$ à un descendant $v$ dans une arborescence de parcours en profondeur.
- Les arcs transverses sont tous les autres arcs. Ils peuvent relier deux sommets d'un même arbre de parcours en profondeur, du moment que l'un des sommets n'est pas un ancêtre de l'autre ; ils peuvent aussi relier deux sommets appartenant à des arborescences de parcours en profondeur différentes.

## 2.4 Trie topologique

Cette section montre comment utiliser le parcours en profondeur pour effectuer un tri topologique d'un graphe orienté sans circuit. Le tri topologique d'un graphe orienté sans circuit $G = (V, E)$ consiste à ordonner linéairement tous ses sommets de sorte que, si $G$ contient un arc $(u, v)$, $u$ apparaisse avant $v$ dans le tri. (Si le graphe n'est pas sans circuit, aucun ordre linéaire n'est possible.) Le tri topologique d'un graphe peut être vu comme un alignement de ses sommets le long d'une ligne horizontale de manière à ce que tous les arcs soient orientés de gauche à droite.

Les graphes orientés sans circuit sont utilisés dans de nombreuses applications pour représenter des précédences entre événements.

L'algorithme simple suivant permet d'effectuer le tri topologique d'un graphe orienté sans circuit.

```
TRI-TOPOLOGIQUE(G)
1      appeler PP(G) pour calculer les dates de fin de traitement f [v] pour chaque som
2      chaque fois que le traitement d'un sommet se termine, insérer le sommet début d'
3      retourner la liste chaînée des sommets
```

On peut effectuer un tri topologique en $\theta(|V| + |E|)$, car le parcours en profondeur prend $\theta(|V|+|E|)$ et l'insertion de chacun des $|V|$ sommets au début de la liste chaînée nécessite $O(1)$. On démontre la validité de cet algorithme à l'aide du lemme fondamental suivant, qui caractérise les graphes orientés sans circuit.

**Lemme 4.** *Un graphe orienté $G$ est sans circuit si et seulement si un parcours en profondeur de $G$ ne génère aucun arc arrière.*

*Bevis.* $\Rightarrow$ On suppose qu'il existe un arc arrière $(u, v)$. Alors, le sommet $v$ est un ancêtre du sommet $u$ dans la forêt de parcours en profondeur. Il existe donc un chemin de $v$ à $u$ dans $G$, et l'arc arrière $(u, v)$ complète un circuit.

$\Leftarrow$: Supposons que $G$ contienne un circuit $c$. On va montrer qu'un parcours en pro- fondeur de $G$ génère un arc arrière. Soit $v$ le premier sommet découvert dans $c$, et soit $(u, v)$ l'arc précédent dans $c$. A l'instant $d[v]$, les sommets de $c$ forment un chemin entre $v$ et $u$ composé de sommets blancs. D'après le théorème du chemin blanc, le sommet $u$ devient un descendant de $v$ dans la forêt de parcours en profondeur. $(u, v)$ est donc un arc arrière.

**Théorème 5.** *TRI-TOPOLOGIQUE(G) effectue le tri topologique d'un graphe orienté sans circuit $G$.*

*Bevis.* Supposons que PP soit exécutée sur un graphe orienté sans circuit $G = (S, A)$ donné pour déterminer les dates de fin de traitement de ses sommets. Il suffit de montrer que, pour toute paire de sommets distincts $u, v$ de $S$, s'il existe un arc dans $G$ menant de $u$ à $v$, alors $f[v] < f[u]$. On considère un arc $(u, v)$ quelconque exploré par PP(G). Lorsque cet arc est exploré, $v$ ne peut pas être gris, car alors $v$ serait un ancêtre de $u$, et $(u, v)$ serait un arc arrière, ce qui contredit le lemme précédent. $v$ est donc soit blanc, soit noir. Si $v$ est blanc, il devient un descendant de $u$, et donc $f[v] < f[u]$. Si $v$ est noir, c'est que son traitement est achevé, et donc que $f[v]$ a déjà été initialisé. Comme on est encore en train d'explorer à partir de $u$, il reste encore à dater $f[u]$, et une fois que ce sera fait, on aura derechef $f[v] < f[u]$. Donc, pour un arc $(u, v)$ quelconque du graphe orienté sans circuit, on a $f[v] < f[u]$, ce qui démontre le théorème.

## 2.5 Composante fortement connexe

Nous allons à présent considérer une application classique du parcours en profondeur : la décomposition d'un graphe orienté en ses composantes fortement connexes. Cette section montre comment effectuer cette décomposition à l'aide de deux parcours en profondeur. Beaucoup d'algorithmes de graphe orientés commencent par cette décomposition ; cette approche permet souvent de diviser le problème initial en sous-problèmes, un par composante fortement connexe. La combinaison des solutions des sous-problèmes se fait en suivant la structure des connexions entre les composantes.

Une composante fortement connexe d'un graphe orienté $G = (V, E)$ est un ensemble maximal de sommets $R \subseteq V$ tel que, pour chaque paire de sommets $u$ et $v$ de $R$, on ait à la fois $u \rightsquigarrow v$ et $v \rightsquigarrow u$ ; autrement dit, les sommets $u$ et $v$ sont mutuellement accessibles.

Notre algorithme de recherche des composantes fortement connexes d'un graphe $G = (V, E)$ utilise le transposé de $G$, notée par $T_G = (V, {}^T E)$, où ${}^T E = \{(u, v) : (v, u) \textit{ de } E\}$. Autrement dit, ${}^T E$ est constitué des arcs de $G$ dont le sens a été inversé. Étant donné une représentation par listes d'adjacences de $G$, la création de $T_G$ demande un temps en $O(V + E)$. Il est intéressant d'observer que $G$ et $T_G$ ont exactement les mêmes composantes fortement connexes : $u$ et $v$ sont accessibles dans $G$ l'un à partir de l'autre si et seulement si ils le sont aussi dans $T_G$.

L'algorithme à temps linéaire (c'est-à-dire en $Q(S + A)$) que voici calcule les composantes fortement connexes d'un graphe orienté $G = (S, A)$ à l'aide de deux parcours en profondeur, l'un sur $G$ et l'autre sur $T_G$.

```
COMPOSANTES-FORTEMENT-CONNEXES(G)
1     appeler PP(G) pour calculer les dates de fin
         de traitement $f [u]$ pour chaque
         sommet $u$
2      calculer $T_G$
3      appeler PP(TG), mais dans la boucle principale de PP,
         traiter les sommets par ordre de $f [u]$
         (calculés en ligne 1) décroissants
4      imprimer les sommets de chaque arborescence de la forêt
          obtenue en ligne 3 en tant que composante fortement
          connexe distincte
```

L'idée sous-jacente à cet algorithme vient d'une propriété majeure du graphe des composantes $G^C FC = (V^{CFC}, E^{CFC})$, que nous définissons comme suit. Supposons que $G$ ait des composantes fortement connexes $C_1, C_2, \ldots, C_k$. L'ensemble des sommets $S^{CFC}$ est $\{v_1, v_2, \ldots, v_k\}$, et il contient un sommet $v_i$ pour chaque composante forte- ment connexe $C_i$ de $G$. Il y a un arc $(v_i, v_j)$ in $E^{CFC}$ si $G$ contient un arc $(x, y)$ pour un certain $x \in C_i$ et un certain $y \in C_j$. Autrement dit, en contractant tous les arcs dont les sommets incidents sont à l'intérieur de la même composante fortement connexe de $G$, le graphe résultant est $G^{CFC}$.

La propriété majeure est que le graphe des composantes est un graphe orienté sans circuit, ce qu'implique le lemme suivant.

**Lemme 5.** *Soient $C$ et $C'$ des composantes fortement connexes distinctes du graphe orienté $G = (S, A)$, soit $u, v \in C$, soit $u', v' \in C'$, et supposons qu'il y ait un chemin $u \rightsquigarrow u'$ dans $G$. Alors, il ne peut pas y avoir aussi un chemin $v' \rightsquigarrow v$ dans $G$.*

*Bevis.* S'il y a un chemin $v' \rightsquigarrow v$ dans $G$, alors il y a des chemins $u \rightsquigarrow u' \rightsquigarrow v'$ et $v' \rightsquigarrow v \rightsquigarrow u$ dans $G$. Donc, $u$ et $v'$ sont accessibles mutuellement, ce qui contredit l'hypothèse selon laquelle $C$ et $C'$ sont des composantes fortement connexes distinctes.

Nous allons voir que le fait de traiter, dans le second parcours en profondeur, les sommets dans l'ordre décroissant des dates de fin de traitement calculées lors du premier parcours en profondeur, revient fondamentalement à visiter les sommets du graphe des composantes (dont chacun correspond à une composante fortement connexe de $G$) dans un ordre topologique.

Comme COMPOSANTES-FORTEMENT-CONNEXES effectue deux parcours en profondeur, il risque d'y avoir ambiguïté quand on étudie $d[u]$ ou $f[u]$. Dans cette section, ces valeurs désignent systématiquement les dates de découverte et de fin de traitement calculées par le premier appel à $PP$ en ligne 1.

Nous allons étendre la notation des dates de découverte et de fin de traitement à des ensembles de sommets. Si $U \subseteq V$, on définit

$d(U) = min_{u \in U}\{d[u]\}$ et $f(U) = max_{u \in U}\{f[u]\}$

En d'autres termes, $d(U)$ et $f(U)$ désignent respectivement la date de découverte la plus précoce et la date de fin de traitement la plus tardive d'un sommet de $U$.

Le lemme suivant et son corollaire donnent une propriété fondamentale qui relie composantes fortement connexes et dates de fin de traitement dans le premier parcours en profondeur.

**Lemme 6.** *Soient $C$ et $C'$ des composantes fortement connexes distinctes d'un graphe orienté $G = (V, E)$. On suppose qu'il y a un arc $(u, v) \in E$, tel que $u \in C$ et $v \in C'$. Alors, $f(C) > f(C')$.*

*Bevis.* A fair, ...

Le corollaire suivant dit que chaque arc de ${}^{T}G$ qui relie des composantes fortement connexes différentes part d'une composante ayant une date de fin de traitement plus précoce (dans la première recherche en profondeur) et arrive à une composante ayant une date de fin de traitement plus tardive.

**Corollaire 3.** *Soient $C$ et $C'$ des composantes fortement connexes distinctes d'un graphe orienté $G = (V, E)$. Supposons qu'il y ait un arc $(u, v) \in^{T} E$, tel que $u \in C$ et $v \in C'$. Alors, $f(C) < f(C')$.*

*Bevis.* Comme $(u, v) \in^{T} E$, on a $(v, u) \in E$. Comme les composantes fortement connexes de $G$ et $T_G$ sont les mêmes, le lemme 6 précédent entraîne que $f(C) < f(C')$.

Le corollaire 3 permet de comprendre le fonctionnement de COMPOSANTES-FORTEMENT-CONNEXES. Examinons ce qui se passe quand on fait le second parcours en profondeur, qui concerne $T_G$. On commence par la composante fortement connexe $C$ dont la date de fin de traitement $f(C)$ est maximale. Le parcours part d'un certain sommet $x \in C$, et il visite tous les sommets de $C$. D'après le corollaire 3, il n'y a pas d'arc dans $T_G$ qui relie $C$ à une autre composante fortement connexe, et donc la recherche issue de $x$ ne visitera pas les sommets d'une quelconque autre composante. Par conséquent, l'arborescence de racine $x$ contient les sommets de $C$, ni plus ni moins. Ayant fini de visiter tous les sommets de $C$, la recherche en ligne 3 sélectionne comme racine un sommet d'une certaine autre composante fortement connexe $C'$ dont la date de fin de traitement $f(C')$ est maximale sur l'ensemble de toutes les composantes autres

que $C$. Ici aussi, la recherche visitera tous les sommets de $C'$, mais d'après le corollaire 3, les seuls arcs de $T_G$ qui relient $C'$ à une autre composante doivent aller vers $C$, que l'on a déjà visité. Plus généralement, quand le parcours en profondeur de $T_G$ en ligne 3 visite une composante fortement connexe, tous les arcs partant de cette composante mènent forcément vers des composantes qui ont déjà été visitées. Chaque arborescence de parcours en profondeur, par conséquent, est une seule composante fortement connexe. Le théorème suivant formalise cette démonstration.

**Théorème 6.** *COMPOSANTES-FORTEMENT-CONNEXES(G) calcule effectivement les composantes fortement connexes d'un graphe orienté G.*

*Bevis.* Montrons, par récurrence sur le nombre d'arborescences de parcours en profondeur trouvées lors du parcours en profondeur de $T_G$ en ligne 3, que les sommets de chaque arborescence forment une composante fortement connexe. L'hypothèse de récurrence est que les $k$ premières arborescences produites en ligne 3 sont des composantes fortement connexes.

La base de la récurrence, quand $k = 0$, est triviale.

Dans l'étape inductive, on suppose que chacun des $k$ premières arborescences de parcours en profondeur produites en ligne 3 est une composante fortement connexe.

Considérons la (k + 1)ème arborescence produite. Supposons que la racine de cette arborescence soit le sommet $u$, et que $u$ soit dans la composante fortement connexe $C$. Compte tenu de la facon dont on a choisi les racines dans le parcours en profondeur en ligne 3, $f[u] = f(C) > f(C')$ pour toute composante fortement connexe $C'$ autre que $C$ qui reste à visiter. D'après l'hypothèse de récurrence, à l'instant où la recherche visite $u$, tous les autres sommets de $C$ sont blancs. D'après le théorème du chemin blanc, par conséquent, tous les autres sommets de $C$ sont des descendants de $u$ dans son arborescence de parcours en profondeur.

En outre, d'après l'hypothèse de récurrence et le corollaire 3, tous les arcs de $T_G$ qui partent de $C$ mènent forcément vers des composantes fortement connexes ayant été déjà visitées. Donc, aucun sommet d'une composante fortement connexe autre que $C$ ne sera un descendant de $u$ lors du parcours en profondeur de $T_G$. Donc, les sommets de l'arborescence de parcours en profondeur de $T_G$ qui a pour racine $u$ forment une et une seule composante fortement connexe, ce qui termine l'étape inductive et la démonstration.