# Chapitre IV

# Graphes

## 1 Généralités

### 1.1 Définitions

**Définition 1. Graphe orienté**

Un **graphe orienté** $G$ est un couple $(S, A)$ où $S$ est un ensemble fini et $A$ une relation sur $S$, c'est-à-dire un ensemble de couples $(u, v)$ d'éléments de $S$.
$S$ et appelé ensemble des sommets (ou nœuds) de $G$ et $A$ ensemble des arcs.

**Définition 2. Graphe non orienté**

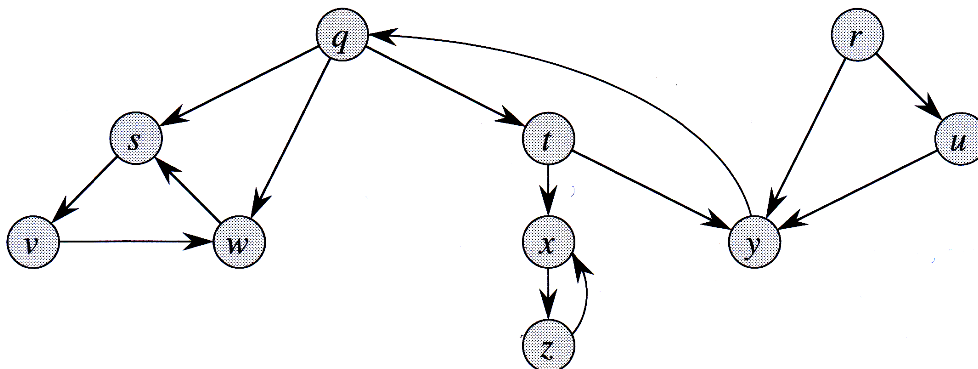
Un **graphe non orienté** $G$ est un couple $(S, A)$ où $S$ est un ensemble fini et $A$ une relation symétrique sur $S$.
$A$ est alors appelé ensemble des arêtes de $G$.

*Remarque* : on peut aussi voir $A$ comme un ensemble de couples non ordonnés, c.a.d. tel que tout arc $(u, v)$ est identifié à $(v, u)$.

**Définition 3. Graphe pondéré**

Un **graphe pondéré** $G = (A, S, w)$ est un graphe munie d'une fonction de poids $w : A \to \mathbb{R}$.
On peut distinguer les graphes pondérés positivement. On a alors $w : A \to \mathbb{R}^+$.

**Remarque** : Un graphe non pondéré est un graphe pondéré muni d'une fonction de poids constante égale à 1.




Exemple de graphe orienté non pondéré

**Définition 4. Vocabulaire à propos des graphes**

Soit $G = (S, A)$ un graphe.

- **Sous-graphe** : $G' = (S', A')$ est un sous-graphe de $G = (S, A)$ si $S' \subset S$ et $A' \subset A$.
- **Graphe engendré** : le sous-graphe de $G = (S, A)$ engendré par le sous-ensemble de sommet $S' \subset S$ est le graphe $G' = (S', A')$ tel que $A' = \{(s_1, s_2) \in A | s_1, s_2 \in S'\}$, c.a.d. $G'$ contient toutes les arêtes (arcs) de $G$ qui joignent deux sommets de $G'$.
- Arc **incident** à un sommet $u$ :
  - si $G$ est orienté, on dit que l'arc $a$ est incident intérieurement au sommet $u$ si $a = (v, u)$, i.e. $a$ arrive à $u$. On dit que $a$ est incident extérieurement à $u$ si $a = (u, v)$, i.e. $a$ part de $u$.
  - si $G$ est non orienté, on dit que l'arête $a$ est incidente au sommet $u$ si $a$ s'écrit $(u, v)$ ou $(v, u)$.
- **Adjacence** : si $(u, v)$ est un arc de $G$ (orienté ou non orienté), on dit que $v$ est adjacent à $u$. La relation définie par $A$ est appelée relation d'adjacence. Cette relation est symétrique si $G$ est non orienté.

- **Degré d'un sommet** :
  - Le **degré entrant** d'un sommet $s$ est le nombre d'arcs qui arrivent en $s$.
  - Le **degré sortant** d'un sommet $s$ est le nombre d'arcs qui partent de $s$.
  - Le **degré** d'un sommet $s$ est le nombre d'arcs qui partent ou arrivent de $s$. C'est la somme du degré entrant et du degré sortant.

  Dans un **graphe non orienté**, seul la notion de degré existe.

- **Chemin** : on appelle chemin une suite de sommets adjacents. On note $\langle s_1, s_2, \dots, s_k \rangle$ le chemin qui suit les sommets adjacents $s_1, s_2, \dots, s_k$.
- **Longueur d'un chemin**
  - dans un **graphe non pondéré** la longueur d'un chemin est le nombre d'arcs du chemin.
  - dans un **graphe pondéré** la longueur d'un chemin est la somme des poids des arcs du chemin.
- **Sommet accessible** : on dit que le sommet $s'$ est accessible depuis le sommet $s$ si il existe un chemin de $s$ à $s'$.

- **Circuit** : dans un graphe **orienté**, un chemin $\langle s_1, s_2, \dots, s_k \rangle$ est un circuit si $s_1 = s_k$.
- **Cycle** : c'est la même notion que celle de circuit pour les graphes **non orientés**.
- **Circuit (cycle) élémentaire** : un circuit (cycle) est élémentaire si tous ses sommets sont distincts deux à deux.
- **Boucle** : un arc ou arête $(u, u)$ est appelé boucle. On n'accepte pas en général les boucles pour les graphes non orientés. Les boucles sont des cas particulier de circuit ou cycle.
- **Graphe acyclique** : un graphe sans circuit ou cycle est dit acyclique.

- **Connexité** :
  - **Graphe non orienté** : un graphe non orienté est **connexe** si chaque paire de sommets est relié par un chemin.
    Dans un graphe non orienté, la relation "est accessible depuis" est une relation d'équivalence. On appelle **composante connexe** de $G$ toute classe d'équivalence de cette relation. C'est donc une partie $C$ maximale de $S$ (au sens de l'inclusion) possédant la propriété suivante :
    Pour tout $(u, v) \in C^2$, $v$ est accessible depuis $u$.
  - **Graphe orienté** : un graphe orienté est **fortement connexe** si pour tout couple de sommets $(u, v)$, $v$ est accessible depuis $u$.
    **Composante fortement connexe** : c'est une partie $C$ maximale de $S$ (au sens de l'inclusion) possédant la propriété suivante : pour tout $(u, v) \in C^2$, $v$ est accessible depuis $u$.
- **Graphe complet** : un graphe non-orienté est dit complet si les sommets sont tous adjacents deux à deux.

- **Densité** :
  - La densité d'un graphe est définie par le quotient du nombre d'arêtes (ou d'arcs) du graphe divisé par le nombre d'arêtes (ou d'arcs) du graphe complet sur le même ensemble de sommets. On ne compte pas les boucles $(u, u)$ d'un sommet vers lui-même.
  - La densité maximale est 1 et correspond à un graphe complet. La densité minimale est 0 et correspond à un graphe où tous les sommets sont isolés.
  - Un graphe est **dense** si sa densité et proche de 1. Un graphe est **peu dense ou creux** si sa densité est proche de 0.
- **Arbre** : un arbre est un graphe orienté sans circuit où chaque sommet sauf un (la racine) a un degré entrant égal à 1. La racine a un degré entrant nul.
- **Forêt** : un graphe composé de plusieurs arbres est appelé forêt.

## 1.2 Implémentation d'un graphe

### 1.2.1 Matrice d'adjacence

**Définition 5. Matrice d'adjacence**

Soit $G = (S, A)$ un graphe, on note $n = \#S$, puis on numérote les sommets de $S$, on a $S = \{s_1, \ldots, s_n\}$.
On peut alors définir une matrice carrée $M = (a_{i,j})$ d'ordre $n$ de la manière suivante :

- Si le graphe est non pondéré, alors $a_{i,j} = 1$ si $(s_i, s_j) \in A$ et $a_{i,j} = 0$ si $(s_i, s_j) \notin A$.
- Si le graphe est pondéré, alors $a_{i,j} = w(i, j)$ si $(s_i, s_j) \in A$ et $a_{i,j} = nil$ si $(s_i, s_j) \notin A$. On peut aussi choisir $\infty$ (parfois aussi 0) pour le poids d'un arc absent de $A$.

La matrice $M$ est alors appelée **matrice d'adjacence** du graphe $G$.
Si le graphe est **non orienté**, la matrice d'adjacence est **symétrique**.
Pour un **graphe orienté**, la matrice n'est pas symétrique et sa **transposée** est la matrice du graphe $G'$ dont les arcs sont obtenus à partir de ceux de $G$ par changement de sens.

L'implémentation d'un graphe par stockage de la matrice d'adjacence a des avantages :

- Simplicité de la structure de donnée et donc de l'écriture des algorithmes.
- Pour un graphe non pondéré, un seul bit suffit au stockage d'un arc.

L'implémentation d'un graphe par stockage de la matrice d'adjacence a des inconvénients :

- Complexité spatiale en $O(n^2)$ à comparer avec $\#A$.
- Complexité temporelle du parcours de l'ensemble des arcs en $O(n^2)$ à comparer avec $\#A$.

Pour un graphe de taille limitée, les problèmes de complexité sont peu importants donc une matrice d'adjacence est un choix judicieux.

Pour un graphe dense, $\#A$ est proche de $n^2$ donc là aussi, les avantages d'une matrice d'adjacence peuvent l'emporter.

### 1.2.2 Listes d'adjacence

**Définition 6. Listes d'adjacence**

Soit $G = (S, A)$ un graphe. On peut définir pour chaque $u \in S$ une liste $Adj(u)$ de la manière suivante :

- Si le graphe est non pondéré, alors $Adj(u)$ contient la liste des sommets $v$ tels que $(u, v) \in A$.
- Si le graphe est pondéré, alors $Adj(u)$ contient la liste des couples $(v\ ,\ w(u, v))$ tels que $(u, v) \in A$.

Donc, pour tout $u \in S$, $Adj(u)$ contient tous les sommets adjacents à $u$, on l'appelle **liste d'adjacence** de $u$.
Si le graphe est non orienté, l'arc $(u, v)$ apparaît deux fois, par la présence de $v$ dans $Adj(u)$ et par la présence de $u$ dans $Adj(v)$.

Pour l'implémentation, on numérote les sommets de $S$, on peut alors écrire $S = \{s_1, \ldots, s_n\}$.

Puis on stocke dans un tableau de longueur $n$, la référence de chaque liste $Adj(s_i)$.

Les listes d'adjacence elle-mêmes peuvent être stockées sous forme de tableaux ou de listes chaînées.

L'implémentation d'un graphe par stockage des listes d'adjacence a des avantages :

- Parcours de l'ensemble des arcs avec une complexité linéaire (en $O(A)$). Or ce parcours est nécessaire dans de nombreux algorithmes.
- Pour un graphe peu dense, faible complexité spatiale (peu de données à stocker).

L'implémentation d'un graphe par stockage des listes d'adjacence a des inconvénients :

- Implémentation des listes d'adjacence.
- Complexité spatiale et temporelle des listes d'adjacence pour un graphe dense.

# 2 Parcours en largeur

## 2.1 Introduction

**Définition 7. Vocabulaire**

Lors d'un algorithme de parcours :

- On dit qu'un sommet est **découvert** lorsque l'algorithme voit ce sommet pour la première fois.
- On dit qu'un sommet est **visité** lorsque l'algorithme a terminé le traitement de ce sommet (par exemple affichage).

**Problème du parcours en largeur**

Soit $G = (S, A)$ un graphe et $s$ un sommet de $G$. Le parcours en largeur consiste à visiter les sommets de $G$ par ordre croissant du nombre d'arcs (arêtes) parcourus depuis $s$.
Plus précisément, on suit le principe suivant :

- On visite $s$.
- Puis les sommets accessibles depuis $s$ par le parcours d'une arête (arc).
- Puis les sommets accessibles depuis $s$ par le parcours de deux arêtes (arcs).
- etc.

L'algorithme qui résout ce problème est à la base de nombreuses idées sur les graphes, par exemple pour les algorithmes de Prim et Dijkstra vus par la suite de ce cours.

Il fonctionne aussi bien sur les graphes orientés ou non orientés.

Il permet d'établir si un graphe non orienté est **connexe**, sinon de donner ses **composantes connexes**.

Il calcule **les plus court chemin depuis une origine** $s$ dans un **graphe non-pondéré**.

*Remarque* : le nom anglais du parcours en largeur est **breadth first search**.

## 2.2 Algorithme de parcours en largeur

L'algorithme reprend le principe du parcours en largeur sur les arbres avec une **file** F :

A chaque itération, on défile un sommet $u$, on enfile tous les sommets adjacents à $u$ et on visite $u$.
On initialise la file par $F = < s >$, et on itère tant que la file $F$ n'est pas vide.

Mais contrairement à un arbre, dans un graphe, un sommet peut-être atteint par **plusieurs chemins**. Il peut donc être découvert plusieurs fois. Pourtant, on ne veut le visiter qu'une fois.

Pour empêcher des visites redondantes d'un même sommet le plus en amont possible, il faut empêcher d'enfiler des sommets déjà découverts.

Une idée simple s'impose : **marquer tout sommet au moment où il est découvert et enfilé** et ne pas enfiler tout sommet déjà marqué.

Dans l'algorithme présenté ici, on a utilisé trois couleurs :

- **blanc** pour les sommets non découverts,
- **gris** pour les sommets découverts mais non visités,
- **noir** pour les sommets visités.

Il est possible de se limiter à deux couleurs, c'est-à-dire à des booléens.

**Structures de données de l'algorithme**

On note $n = \#S$ puis $S = \{s_1, \ldots, s_n\}$.
Comme l'algorithme permet de calculer les plus courts chemins depuis l'origine $s$, on utilise au total trois tableaux de longueur $n$, seul le premier étant indispensable :

- Un tableau `couleur` selon le principe précédent.
- Un tableau `d` pour stocker les plus courtes distances depuis $s$. Un distance infinie est synonyme d'inaccessibilité depuis $s$.
- Un tableau $\pi$ pour stocker en $\pi[i]$ le sommet précédent $s_i$ sur un plus court chemin depuis $s$.

**Algorithme 1. Parcours en largeur**

```
fonction breadth first search(G,s)
1   créer une file F et trois tableaux π, d et couleur de longueur #S(G)
2   pour chaque sommet u ∈ S(G)\{s} faire     # INITIALISATION
3     couleur[u] = blanc
4     d[u] = ∞
5     π[u] = nil
6   d[s]= 0
7   π[s]= nil
8   enfiler(F,s)
9   couleur[s]=gris
10  tant que F ≠ ∅ faire             # BOUCLE DE L'ALGORITHME
11    u = defiler(F)
12    pour chaque sommet v ∈ Adj[u] faire
13      si couleur[v]==blanc alors
14        couleur[v]=gris
15        d[v] = d[u] +1
16        π[v] = u
17        enfiler(F,v)
18    couleur[u]=noir
19  retourner d, π, couleur
```

**Propriété 1. Complexité**

La complexité du parcours en largeur d'abord est $O(\#A + \#S)$ si le graphe est implémenté par des listes d'adjacence.

*Démonstration*
Les lignes 2 à 5 sont à temps constant et exécutées $\#S$ fois. Donc leur complexité est $O(\#S)$.
Les lignes 6 à 9 et 19 sont a temps contant et exécutées 1 fois. Donc leur complexité est $O(1)$.
Les lignes 10, 11 et 18 sont exécutées autant de fois que le nombre de sommets qui passent dans la file. Or chaque sommet passe au plus une fois dans la file (0 fois si il est inaccessible). Donc leur complexité est $O(\#S)$.
Les ligne 12 et 13 sont exécutées une fois par arc sortant d'un sommet accessible, donc au plus $\#A$ fois,

leur complexité est $O(\#A)$ pour une implémentation par listes d'adjacence.
Les lignes 14 à 17 sont exécutées seulement pour les sommets blancs accessibles qui sont immédiatement coloriés, donc elles sont exécutées au plus $\#S$ fois. Leur complexité est $O(\#S)$.

Au total, on obtient une complexité $O(\#S + \#A)$

## 2.3 Propriétés du parcours en largeur

On considère dans cette section un graphe $G = (A, S)$ pondéré ou non, mais on fait abstraction de la pondération, c.a.d. on utilise une pondération constante. Donc **la notion de plus court chemin ou plus courte distance correspond uniquement au nombre d'arêtes ou d'arcs**.

**Théorème 2. Plus courts chemins**

Soit $G = (A, S)$ un graphe, on suppose que le parcours en largeur est exécuté sur $G$ à partir d'un sommet origine $s$.
Alors, pendant l'exécution du parcours, l'algorithme découvre chaque sommet $u \in S$ accessible à partir de $s$ et à la fin $d[u]$ est la longueur d'un plus court chemin de $s$ à $u$ (pour une pondération constante).
De plus, pour tout sommet $u \neq s$ accessible depuis $s$, l'un des plus court chemin (PCC) est calculable récursivement par : $PCC(s, u) = PCC(s, \pi(u)) + \langle u \rangle$ où $+$ est la concaténation.

**Remarque** :
Pour tout sommet $v$ accessible depuis $s$, la relation récursive permet de parcourir un plus court chemin de $s$ à $v$.

*Démonstration du théorème*
Pour tout sommet $u \in S$, on note $\delta(s, u)$ la plus courte distance de $s$ à $u$ (pour une pondération constante) et $PCC(s, u)$ un plus court chemin de $s$ à $u$. On choisit $\delta(u) = \infty$ si $u$ n'est pas accessible depuis $s$.

**Lemme PL1**

Pour tout arc $(u, v)$ du graphe, $\delta(s, v) \leqslant \delta(s, u) + 1$. (I)

*Démonstration du lemme PL1* :

- Si $\delta(s, v)$ est infini alors $v$ n'est accessible depuis $s$, mais $u$ est également inaccessible car sinon l'arc $(u, v)$ assurerait l'accessibilité de $v$. Il suit que les deux membres de l'inégalité (I) valent $\infty$ donc (I) est vraie.
- Si $\delta(s, v)$ est fini alors $v$ est accessible.
  - Soit $u$ est inaccessible et le membre de gauche de (I) est fini alors que le membre de droite est infini, donc (I) est vraie.
  - Soit $u$ est également accessible. On note $PCC(s, u)$ un plus court chemin de $s$ à $u$.
    Alors $PCC(s, u) + \langle v \rangle$ est un chemin de $s$ à $v$ de longueur $\delta(s, u) + 1$. Comme $\delta(s, v)$ est un minimum sur tous les chemins de $s$ à $v$, l'inégalité (I) s'en déduit.

**Lemme PL2**

A la fin du parcours en largeur, pour tout sommet $v$, $d(v) \geqslant \delta(s, v)$.

*Démonstration du lemme PL2* : remarquons d'abord qu'avec le système de coloriage, pour tout sommet $v$, $d(v)$ est affecté au plus une fois par la ligne 15 (bien entendu sans compter l'initialisation).

Par récurrence sur les opérations enfiler, on montre d'abord que pour tout sommet $v$ entré dans la file, $d(v) \geqslant \delta(s, v)$.

Initialisation : lorsqu'on enfile $s$, la propriété est vraie car $d[0] = 0 = \delta(s, s)$.

Hérédité : Supposons la propriété vraie pour tout sommet passé dans la file.

Lorsqu'on enfile $v$ à la ligne 17, on a écrit ligne 15 : `d[v]=d[u]+1`.

Or $u$ vient d'être défilé, donc $u$ vérifie l'hypothèse de récurrence et $d(u) \geqslant \delta(s,u)$.
Il suit que $d(v) = d(u) + 1 \geqslant \delta(s,u) + 1 \geqslant \delta(s,v)$ d'après le lemme PL1.
Donc la propriété est vraie pour $v$. (CQFD)

Il reste à montrer que $d(v) \geqslant \delta(s,v)$ est vraie si $v$ n'est pas passé dans la file. Dans ce cas, $d(v)$ n'a jamais été affecté par la ligne 15 donc il a gardé sa valeur initiale $\infty$ donc l'inégalité est encore vraie.

**Lemme PL3**

A tout moment, les sommets dans la file sont dans l'ordre croissant de la valeur $d$ et l'écart de la valeur de $d$ entre la tête et la queue est au maximum 1.

Cette propriété contient le principe du parcours en largeur, à partir de $s$, on a enfilé tous les nœuds tels que $d = 1$ puis on défile ces nœuds et pour chacun d'eux, on enfile des nœuds tels que d=2, et ainsi de suite.
On ne commence à enfiler les nœuds tels que d=3 que lorsqu'on commence à défiler les nœuds tels que d=2.
Mais à ce moment, tous les nœuds tels que d=1 sont sortis de la file, et ainsi de suite.

*Démonstration du lemme PL3* :

On fait une récurrence en montrant que toutes les opérations sur la file maintiennent cette propriété.

Initialisation : $F =< s >$ donc la propriété est vraie.

Héridité pour les opération défiler : la propriété reste évidemment vraie.

Héridité pour les opération enfiler : lorsqu'on enfile $v$ à la ligne 17, $d(v) = d(u) + 1$ où $u$ est le dernier élément défilé.
Avant de défiler $u$, $u$ était en tête et tout les sommets $x$ dans la file vérifiaient $d(u) \leqslant d(x) \leqslant d(u) + 1$ par hypothèse de récurrence.

Depuis que $u$ a été défilé, on a enfilé éventuellement des sommets $y$, adjacents à $u$ pour lesquels $d(y) = d(u) + 1$.

Lorsqu'on enfile $v$, on a aussi $d(v) = d(u) + 1$ donc la file est toujours par ordre croissant de $d$, et la tête $h$ vérifie $d(u) \leqslant d(h) \leqslant d(u) + 1 = d(v)$.

Il suit que l'écart entre la tête et la queue est toujours au maximum 1.

**Conséquence immédiate PL4**

Soient $u$ et $v$ des sommets tels que $u$ a été enfilé avant $v$ alors $d(u) \leqslant d(v)$. Autrement dit, l'ordre des opérations enfiler respecte l'ordre croissant sur $d$.

*Démonstration* : Comme la file à tout moment respecte un ordre croissant sur $d$ (lemme PL3) et qu'elle n'est jamais vide avant la fin de l'algorithme (d'après le test d'entrée de boucle), la transitivité de la relation d'ordre impose que les sommets sont enfilés dans un ordre qui respecte l'ordre croissant sur $d$.

### Retour à la démonstration du théorème

Raisonnons par l'absurde et supposons que l'ensemble $E$ des sommets $x$ vérifiant à la fin de l'algorithme $d(x) \neq \delta(s,x)$, ne soit pas vide. On note $v$ le sommet de $E$ tel que $\delta(s,v) = min\,(\{\delta(s,x) \,|\, x \in E\})$.

D'après le lemme PL2, $d(v) \geqslant \delta(s,v)$, donc $d(v) > \delta(s,v)$ car $v \in E$.

$v$ est accessible depuis $s$ sinon on aurait $\delta(s,v) = \infty$ donc l'inégalité stricte précédente ne serait pas possible.
De plus $v$ est distincte de $s$ car l'égalité $d(s) = 0 = \delta(s,s)$ assure que $s \notin E$.

Considérons un plus court chemin de $s$ à $v$, $v \neq s$ donc il existe un sommet $u$ qui précède $v$ sur ce chemin.

Alors le sous-chemin de $s$ à $u$ est de longueur minimal pour aller de $s$ à $u$, sinon on pourrait améliorer le chemin de $s$ à $v$.

Il suit que $\delta(s,v) = \delta(s,u) + 1$.

Par conséquent $\delta(s,v) > \delta(s,u)$ et d'après le choix de $v$, $u \notin E$ donc $d(u) = \delta(s,u)$.

Il suit que $d(v) > \delta(s,v) = \delta(s,u) + 1 = d(u) + 1$. (1)

Considérons le moment où $u$ est défiler, alors à ce moment, il y a 3 cas :

- Soit $v$ est encore blanc, et dans ce cas, il est découvert lors du parcours des sommets adjacents à $u$, il reçoit $d(v) = d(u) + 1$ contradictoire avec (1) puisque cette valeur sera inchangée par la suite.
- Soit $v$ est gris, donc il est dans la file. Alors d'après le lemme PL3, $d(v) \leqslant d(u) + 1$ contradictoire avec (1).
- Soit $v$ est noir, donc il déjà est sorti de la file, ce qui implique qu'il a été enfilé avant $u$ donc $d(v) \leqslant d(u)$ d'après le dernier corollaire. Mais $d(v) \leqslant d(u)$ est encore contradictoire avec (1).

Donc l'hypothèse $E$ non vide est fausse.

**Il suit que pour tout $u \in G$, $d(u) = \delta(s, u)$.**

Ceci qui implique que pour tout sommet $v$ accessible depuis $s$, $d(v) = \delta(s, v) < \infty$ donc la ligne 15 a été exécutée sur $v$, autrement dit $v$ a été visité.
**Donc le parcours en largeur visite tout sommet $v$ accessible depuis $s$.**

Enfin, pour tout sommet $v$ accessible depuis $s$, $d(v)$ compte le nombre d'arêtes parcourues depuis $s$ par le parcours en largeur. Il suit, d'après $d(v) = \delta(s, v)$, que **le chemin suivi de $s$ à $v$ par le parcours en largeur est un plus court chemin**.
Lorsque la valeur de $d(v)$ est affectée, c'est qu'on a accédé à $v$ par $u$, donc la ligne 16 "$\pi[v] = u$" affecte bien à $\pi[v]$ le sommet précédent $v$ sur un plus court chemin de $s$ à $v$.
Il suit l'égalité $PCC(s, u) = PCC(s, \pi(u)) + (u)$.

**Propriété 3. Corollaire**

Soit $G$ est un graphe non orienté et $s$ un sommet de $G$, à la fin du parcours en largeur de $G$ à partir de $s$, la composante connexe à laquelle appartient $s$ est l'ensemble des sommets noirs.

*Démonstration* : immédiate à partir du théorème 2.

# 3 Parcours en profondeur

## 3.1 Introduction

**Problème du parcours en profondeur**

Soit $G = (S, A)$ un graphe et $s$ un sommet de $G$. Le parcours en profondeur consiste à visiter les sommets de $G$ en s'éloignant prioritairement de $s$.
Plus précisément, on suit le principe récursif suivant : dès qu'on découvre un sommet, on le visite et on découvre tous ses sommets adjacents.

L'algorithme qui résout ce problème fonctionne aussi bien sur les graphes orientés ou non orientés. Il permet d'établir des propriétés intéressantes sur le graphe, comme l'existence d'un circuit et la construction d'un tri topologique si le graphe est acyclique.

*Remarque* : le nom anglais du parcours en profondeur est **depth first search**.

## 3.2 Algorithme de parcours en profondeur

L'algorithme récursif de parcours en profondeur reprend la propriété récursive qui définit ce parcours : lorsqu'on exécute `visit-DFS(u)`, on lance un appel récursif à `visit-DFS` pour tous les sommets adjacents à $u$.

Pour empêcher les visites redondantes d'un même sommet, on marque les sommets sur lesquels un appel récursif a **commencé à s'exécuter**.

Dans l'algorithme présenté ici, on a utilisé trois couleurs :

- **blanc** pour les sommets non découverts,

- **gris** pour les sommets découverts mais dont la visite n'est pas terminée (donc l'appel récursif sur ce sommet n'est pas terminé),
- **noir** pour les sommets dont la visite est terminée.

Il est possible de se limiter à deux couleurs, c'est-à-dire à des booléens. Mais l'utilisation de trois couleurs permet de rechercher certaines propriétés comme l'existence de circuits.

**Structures de données de l'algorithme**

On note $n = \#S$ puis $S = \{s_1, \ldots, s_n\}$.
On utilise au total quatre tableaux de longueur $n$, seul le premier étant indispensable :

- Un tableau `couleur` selon le principe précédent.
- Un tableau $\pi$ pour stocker en $\pi[i]$ le sommet précédent $s_i$ sur le chemin du parcours en profondeur.
- Deux tableaux `d` et `f` pour stocker les dates de début et de fin de visite de chaque sommet. D'après le principe du parcours en profondeur, `d[u]` est à la fois la date découverte et de début de visite de $u$ ("dès qu'on découvre un sommet, on le visite").

**Algorithme 2. Parcours en profondeur**

```
créer un variable globale date
fonction depth first search(G)
1  créer 4 tableaux π, d, f et couleur de longueur #S(G)
2   pour chaque sommet u ∈ S(G) faire
3      couleur[u] = blanc
4      π[u] = nil
5   date = 0
6   pour chaque sommet u ∈ S(G) faire
7      si couleur[u] == blanc alors
8         visit-DFS(G,u,π,d,f,couleur)
9   retourner π, d, f

fonction visit-DFS(G,u,π,d,f,couleur)
10  couleur[u]=gris  # u vient d'être découvert
11   date = date+1
12   d[u] = date
13   pour chaque sommet v ∈ Adj[u] faire
14      si couleur[v] == blanc alors
15         π[v]= u
16         visit-DFS(G,v,π,d,f,couleur)
17   couleur[u]=noir
18   date=date+1
19   f[u]=date
```

**Propriété 4. Complexité**

La complexité du parcours en profondeur d'abord est $O(\#A + \#S)$ si le graphe est implémenté par des listes d'adjacence.

*Démonstration*
Les lignes 2 - 4 et 6 sont à temps constant et exécutées $\#S$ fois. Donc leur complexité est $O(\#S)$.
La ligne 7 est exécutée au plus $O(\#S)$ fois, sa complexité est $O(\#S)$.

Il faut maintenant calculer la complexité des appels à visit-DFS. La ligne 10-12 et 17-19 sont exécutées $\#S$ car visit-DFS est appelé pour tout sommet blanc et seulement pour eux. Comme la première instruction de visit-DFS est de colorier le sommet en gris, visit-DFS est appelée une et une seule fois par sommet. Leur complexité est $O(\#S)$.

Les lignes 13-14 sont exécutées une fois par arc donc au total $\#A$ fois, leur complexité est $O(\#A)$ pour une implémentation par listes d'adjacence.
Les lignes 15-16 sont exécutées seulement lorsque un sommet est blanc et il est alors colorié immédiatement donc la complexité de la ligne 14 est $O(\#S)$.

Au total, on obtient une complexité $O(\#S + \#A)$

### 3.3 Propriété du parcours en profondeur

**Propriété 5. Forêt du parcours en profondeur**

Soit $G = (S, A)$ un graphe sur lequel on exécute un parcours en profondeur.
On note $A_\pi = \{(\pi(u), u) \mid u \in S \text{ et } \pi(u) \neq nil\}$, c.a.d. $A_\pi$ contient les arcs définis par $\pi$.
Alors le graphe $G_\pi = (S, A_\pi)$ est une forêt qui couvre le graphe $G$.
On l'appelle **forêt du parcours en profondeur**.

*Démonstration* : $G_\pi = (S, A_\pi)$ contient tous les sommets de $G$ donc il couvre $G$. De plus, dans $G_\pi$, tout sommet $u$ a au plus un arc entrant, c'est $(\pi(u), u)$.

Enfin on montre par récurrence qu'à chaque moment où la ligne 15 s'exécute : $G_\pi$ ne contient pas de circuit et tous les sommets blancs sont isolés dans $G_\pi$.

Initialisation : à la première exécution, $G_\pi$ n'a pas d'arc donc la propriété est vraie.

Hérédité : supposons que $G_\pi$ ne contient pas de circuit et que tous les sommets blancs sont isolés dans $G_\pi$.

En exécutant la ligne 15, on ajoute un arc $(\pi(v), v)$ avec $v$ blanc et $\pi(v)$ gris. Il suit que $\pi(v) \neq v$ donc $(\pi(v), v)$ n'est pas une boucle. De plus $v$ est isolé dans $G_\pi$ car blanc, donc l'arc $(\pi(v), v)$ ne peut pas créer un circuit dans $G_\pi$. Enfin $v$ n'est plus isolé dans $G_\pi$ mais il est immédiatement colorié en gris par l'appel récursif à visit-DFS. Tous les autres sommets blancs restent isolés dans $G_\pi$. Donc la propriété de récurrence reste vraie. (CQFD)

**Conclusion** : à la fin du parcours en profondeur, $G_\pi$ est sans circuit et tous ses sommets n'ont au plus qu'un arc entrant, c'est donc une forêt.

**Propriété 6.**

La foret $G_\pi$ du parcours en profondeur est identique à la forêt des appels récursifs à la fonction `visit-DFS`.

*Démonstration*
Soient $u$ et $v$ des sommets de $G$.
Si $u$ est le parent de $v$ dans $G_\pi$, c'est que la ligne 15 a été exécutée avec $u$ et $v$. Mais alors, par la ligne 16, l'appel `visit-DFS(v)` est un noeud enfant de l'appel `visit-DFS(u)` dans l'arbre des appels récursifs.

Réciproquement, si l'appel `visit-DFS(v)` est un nœud enfant de l'appel `visit-DFS(u)` dans l'arbre des appels récursifs, alors la ligne 15 a établi $u$ comme parent de $v$ dans $G_\pi$.

**Lemme PP1**

Soient $u$, $v$ des sommets de $G$, $v$ est un descendant de $u$ dans $G_\pi$ si et seulement si $[d[v], f[v]] \subset [d[u], f[u]]$.

*Démonstration*
Soient $u$ et $v$ des sommets de $G$.
Supposons que $v$ est un descendant de $u$ dans $G_\pi$, comme l'arbre des appels récursifs est identique à $G_\pi$, cela signifie que le nœud `visit-DFS(v)` est un descendant du nœud `visit-DFS(u)` dans l'arbre des appels récursifs donc $[d[v], f[v]] \subset [d[u], f[u]]$.

Réciproquement, supposons $[d[v], f[v]] \subset [d[u], f[u]]$.
Cela implique que le nœud `visit-DFS(v)` est un descendant du nœud `visit-DFS(u)` dans l'arbre des appels récursifs, donc le sommet $v$ est un descendant de $u$ dans $G_\pi$.

**Lemme PP2**

Un sommet $u$ est **blanc** avant la date $d(u)$, **gris** entre $d(u)$ et $f(u)$, **noir** après $f(u)$.

*Démonstration* : cette propriété est immédiate à la lecture de l'algorithme.

**Propriété 7.**

Soient $u$, $v$ des sommets de $G$, $v$ est un descendant de $u$ dans $G_\pi$ si et seulement si $v$ est découvert pendant que $u$ est gris.

*Démonstration*
Soient $u$ et $v$ des sommets de $G$.
Supposons que $v$ est un descendant de $u$ dans $G_\pi$, d'après le lemme PP1, $[d[v], f[v]] \subset [d[u], f[u]]$ et comme $d(v)$ est aussi la date de découverte de $v$, $v$ a été découvert pendant que $u$ était gris.

Réciproquement, supposons que $v$ est découvert pendant que $u$ est gris. Comme un sommet est visité dès qu'il est découvert, cela implique que `visit-DFS(v)` est lancée pendant que l'exécution de `visit-DFS(u)` est en cours. Donc dans l'arbre des appels récursifs, le nœud `visit-DFS(v)` est un descendant du nœud `visit-DFS(u)`. Comme la forêt des appels récursifs de `visit-DFS` est identique à $G_\pi$, le sommet $v$ est un descendant de $u$ dans $G_\pi$.

**Théorème 8. Emboitement**

Soients $u$, $v$ des sommets de $G$, une seule des trois conditions suivantes est vérifiée :

1) $[d[u], f[u]] \cap [d[v], f[v]] = \emptyset$ et ni $u$, ni $v$ n'est un descendant de l'autre dans $G_\pi$ .

2) $[d[u], f[u]] \subset [d[v], f[v]]$ et $u$ est un descendant de $v$ dans $G_\pi$ .

3) $[d[v], f[v]] \subset [d[u], f[u]]$ et $v$ est un descendant de $u$ dans $G_\pi$ .

*Remarque* : $[d[u], f[u]]$ et $[d[v], f[v]]$ ne peuvent pas se chevaucher.

*Démonstration*
Remarquons que la date $d$ étant incrémentée à chaque début et fin de visite d'un sommet, on ne peut pas avoir égalité de deux dates pour des sommets distincts.

Soit $u$ et $v$ deux sommets tels que $d(u) < d(v)$.

- Supposons $d(v) < f(u)$. Comme $u$ est gris dans l'intervalle $[d[u], f[u]]$, il suit que $v$ est découvert pendant que $u$ est gris. D'après la propriété 7 et le lemme PP1, $v$ est un descendant de $u$ dans $G_\pi$ et $[d[v], f[v]] \subset [d[u], f[u]]$.
- Supposons $d(v) > f(u)$ alors $[d[u], f[u]] \cap [d[v], f[v]] = \emptyset$. D'après le lemme PP1, ni $u$, ni $v$ n'est un descendant de l'autre dans $G_\pi$.

**Théorème 9. Chemin blanc**

Soients $u$, $v$ des sommets de $G$, $v$ est un descendant de $u$ dans $G_\pi$ si et seulement si au moment où le parcours découvre $u$, il existe dans $G$ un chemin de $u$ à $v$ composé uniquement de sommets blancs.

*Démonstration*
Soient $u$, $v$ des sommets de $G$.
Supposons que $v$ est un descendant de $u$ dans $G_\pi$. A la découverte de $u$, $u$ lui-même est encore blanc. Considérons maintenant un sommet $w \neq u$ sur le chemin de $u$ à $v$ dans $G_\pi$. $w$ est un descendant de $u$, donc d'après le lemme PP1, $d[u] < d[w]$ et $w$ est encore blanc à la découverte de $u$. (CQFD)

Réciproquement, supposons qu'au moment où le parcours découvre $u$, il existe dans $G$ un chemin $p$ de $u$ à $v$ composé uniquement de sommets blancs.
Montrons par l'absurde que $v$ est à la fin du parcours un descendant de $u$. Si tel n'est pas le cas, notons $w$ le sommet le plus proche de $u$ sur le chemin $p$ dans l'ensemble $N = \{s \in p \mid s \neq u \text{ et } s \text{ n'est pas un descendant de } u\}$.

$N$ contient au moins $v$ donc $w$ existe.
$w$ est distinct de $u$ par définition de $N$ donc $w$ a un prédécesseur sur le chemin $p$ que l'on note $w'$. $w' \in p$ et $w' \notin N$ donc $w'$ est un descendant de $u$.

$u$ n'est pas un descendant de $w$ car sinon $w$ serait gris à la découvert de $u$ or par hypothèse $w$ est blanc à cet instant.
$w$ n'est pas non plus un descendant de $u$ par définition de $w$ donc, d'après le théorème 8, $[d[w], f[w]]$ et $[d[u], f[u]]$ sont disjoints. Comme $w$ est blanc à la découverte de $u$, il suit que l'intervalle $[d[w], f[w]]$ est postérieure à l'intervalle $[d[u], f[u]]$ donc $f(v) < d(w)$. (1)

$w'$ est un descendant de $u$ donc $[d[w'], f[w']] \subset [d[u], f[u]]$. D'après (1), il suit que $f[w'] < d(w)$ ce qui implique qu'au moment de l'exploration de la liste d'adjacence de $w'$ ligne 13, $w$ est blanc, donc le test de la ligne 14 est vrai et la ligne 15 établit $w$ comme enfant de $w'$, ce qui fait de $w$ un descendant de $u$, en contradiction avec $w \in N$. (CQFD)

**Définition 8. Classification des arcs**

Soit $G$ un graphe **orienté**. On note $G_\pi$ la forêt du parcours en profondeur.
On peut classer les arcs du graphe $G$ en quatre catégories :

1) Les arcs de liaison sont les arcs de $G_\pi$.
2) Les arcs avant sont les arcs absents de $G_\pi$ qui relient un sommet à un de ses descendants dans $G_\pi$.
3) Les arcs arrière sont les arcs absents de $G_\pi$ qui relient un sommet à un de ses ascendants dans $G_\pi$. Les boucles d'un graphe orienté sont considérées comme des arcs arrière.
4) Les arcs transverses sont tous les autres arcs.

**Propriété 10.**

Soit $G$ un graphe **orienté** et $(u, v)$ un arc de $G$.
On peut classer l'arc $(u, v)$ en fonction de la couleur de $v$ au moment où $(u, v)$ est exploré à la ligne 13 :

1) si $v$ est blanc alors $(u, v)$ est un arc de liaison.
2) si $v$ est gris alors $(u, v)$ est un arc arrière.
3) si $v$ est noir alors $(u, v)$ est un arc avant ou transverse.

*Démonstration*
On remarque que l'arc $(u, v)$ est exploré pendant l'exécution de `visit-DFS(u)`.

1) si $v$ est blanc alors le test 14 est vrai et la ligne 15 fait de $u$ le parent de $v$. (CQFD)
2) si $v$ est gris alors $[d[u], f[u]] \cap [d[v], f[v]] \neq \emptyset$ donc on est dans le cas où $u$ et $v$ sont descendants l'un de l'autre.
   Supposons que $v$ soit un descendant de $u$, d'après le théorème 9 des chemins blancs, $v$ était blanc à découverte de $u$, c.a.d. au début de l'exécution de `visit-DFS(u)`.
   Si $v$ n'est plus blanc à la ligne 13 de l'exécution de `visit-DFS(u)`, c'est qu'il a été découvert pendant l'exploration d'un arc $(u, w)$ situé avant l'arc $(u, v)$ dans la liste $Adj(u)$. Il suit que $v$ est un descendant de $w$, donc $[d[v], f[v]] \subset [d[w], f[w]]$ et $v$ est noir à la fin de l'exécution de `visit-DFS(w)`.
   Au moment où $(u, v)$ est exploré ligne 13, comme l'exécution de `visit-DFS(w)` est terminée (nécessaire pour passer à l'arc suivant dans $Adj(u)$), $v$ est noir, en contradiction avec l'hypothèse $v$ gris.
   La seule possibilité est donc que $u$ est un descendant de $v$, c.a.d. que $(u, v)$ est un arc arrière. (CQFD)
3) si $v$ est noir, c'est que l'exécution de l'appel `visit-DFS(v)` est terminée alors que l'exécution de `visit-DFS(u)` est cours, donc $f(v) < f(u)$.
   Ou bien $f(v) < d(u)$ donc $[d[u], f[u]] \cap [d[v], f[v]] = \emptyset$. D'après le théorème d'emboîtement 8, il suit que ni $u$ ni $v$ n'est descendant l'un de l'autre donc $(u, v)$ est un arc transverse.
   Ou bien $f(v) > d(u)$, donc $[d[v], f[v]] \subset [d[u], f[u]]$ car ces intervalles ne peuvent pas se chevaucher d'après le théorème d'emboîtement 8. Il suit que $v$ est un descendant de $u$ et que $(u, v)$ est un arc avant.

**Propriété 11. Circuit et arc arrière du parcours en profondeur**

Soit $G$ un graphe **orienté**.
$G$ est acyclique si et seulement si un parcours en profondeur de $G$ ne génère aucun arc arrière.

*Démonstration*
On montre par contraposition que $G$ est cyclique si et seulement si un parcours en profondeur de $G$ génère un arc arrière.

Supposons que $G$ soit cyclique. Alors il existe un circuit $c$ dans $G$. Si c'est une boucle, alors on a un arc arrière (cf. définition des arcs arrière).
Sinon, on peut supposer que $c$ est sans boucle, sinon on les élimine et on conserve un circuit avec au moins deux sommets puisque $c$ n'est pas réduit à une boucle.
Notons $u$ le sommet de $c$ avec la date de fin minimale. Notons $v$ le successeur de $u$ selon le circuit $c$, il existe car $c$ a au moins deux sommets. Comme $c$ est sans boucle, $u$ et $v$ sont distincts et $f(u) < f(v)$ par définition de $u$.
Lors de l'exécution de `visit-PP(u)`, au moment où l'arc $(u, v)$ a été exploré ligne 13, $v$ n'était pas noir car $f(v) > f(u)$. Si $v$ était blanc alors par la ligne 15, $v$ est devenu un enfant de $u$ et d'après le théorème d'emboîtement 8, $f(v) < f(u)$ (contradiction).
Il suit que $v$ était gris donc $(u, v)$ est un arc arrière d'après la propriété 10.

Réciproquement, supposons qu'un parcours en profondeur génère un arc arrière noté $(v, u)$. Si $v = u$ alors la boucle $(u, u)$ est un circuit.
Sinon, par définition d'un arc arrière, $u$ est un ascendant de $v$ dans $G_\pi$ donc il existe un chemin $\langle u, \ldots, v\rangle$ dans $G_\pi$ donc dans $G$. Il suit que $\langle u, \ldots, v, u\rangle$ est un circuit de $G$.

# 4 Tri topologique sur un graphe orienté

## 4.1 Introduction

**Définition 9.**

Soit $G = (S, A)$ un graphe **orienté**. On appelle tri topologique de $G$, un ordre total $\prec$ sur $S$ tel que,
$\forall u, v \in S,\ (u, v) \in A \ \Rightarrow\ u \prec v$.

*Remarques*
Autrement dit, tous les arcs de $G$ respectent l'ordre $\prec$.
Si on a un ordre topologique, on peut alors "aplatir" le graphe et faire figurer sur une droite tous les sommets selon cet ordre topologique. Alors, tous les arcs de $G$ vont dans le mêmes sens.

*Exemple*
Supposons que le graphe $G$ représente une relation de précédence entre des opérations à effectuer (par exemple opérations de montage d'un objet). Un tri topologique de $G$ fournit un ordre pour effectuer ces taches en respectant la relation de précédence.

**Propriété 12.**

Soit $G = (S, A)$ un graphe **orienté** muni d'un tri topologique $\prec$.
Pour tous sommets $u, v \in S$, s'il existe un chemin de $u$ à $v$ alors $u \prec v$.

*Démonstration* : elle découle immédiatement de la transitivité de la relation d'ordre $\prec$.

**Propriété 13.**

Soit $G = (S, A)$ un graphe **orienté sans boucle**.
Si $G$ possède au moins un circuit, il n'existe pas d'ordre topologique sur $G$.

`Remarque` : si $G$ possède des boucles, on peut appliquer ce théorème en les ignorants.

*Démonstration*
On suppose que $G$ possède un circuit et l'on note $u$ et $v$ deux sommets distincts de ce circuit. Il en existe car ce n'est pas une boucle.

On suppose l'existence d'un tri topologique $\prec$ sur $G$. Par définition d'un circuit, il existe un chemin de $u$ à $v$ et un chemin de $v$ à $u$, et d'après la propriété 12, il suit que $u \prec v$ et $v \prec u$.

Par anti-symétrie de la relation d'ordre $\prec$, on en conclut que $u = v$. (Contradiction)

**Propriété 14.**

Soit $G = (S, A)$ un graphe **orienté acyclique**.
Un parcours en profondeur de $G$ fournit un tri topologique selon l'ordre :
$\forall u, v \in S,\ u \prec v \ \Leftrightarrow\ f(u) \geqslant f(v)$.

*Remarque* : si le graphe contient des boucles, le parcours en profondeur les ignore car le sommet de tête de la boucle n'est jamais blanc à la ligne 14, donc le parcours en profondeur fournit un tri topologique.

*Démonstration*
L'ordre ainsi défini est un tri topologique si et seulement si $\forall u, v \in S,\ (u, v) \in A \ \Rightarrow\ f(u) \geqslant f(v)$.

Soit $(u, v)$ un arc quelconque de $G$. Considérons les quatre possibilités pour le type de $(u, v)$ :

1) arc arrière : comme $G$ est acyclique, $(u, v)$ n'est pas un arc arrière d'après la proposition 11.
2) arc de liaison ou arc avant : alors $v$ est un descendant de $u$ et d'après le théorème d'emboîtement 8, $f(u) \geqslant f(v)$.
3) arc transverse : ni $u$ et ni $v$ n'est descendant l'un de l'autre. D'après le théorème d'emboîtement 8, $[d[u], f[u]]$ et $[d[v], f[v]]$ sont disjoints. Il y a donc deux cas : $f(u) < d(v)$ ou bien $d(u) > f(v)$.
   On montre par l'absurde que le premier cas est impossible. Supposons que $f(u) < d(v)$, alors au moment de l'exploration de l'arc $(u, v)$ à la ligne 13 de l'algorithme de parcours en profondeur, $v$ était blanc, donc le test ligne 14 était vrai et $v$ est devenu un enfant de $u$ dans $G_\pi$ en contradiction avec $(u, v)$ transverse.
   Il suit que nécessairement $d(u) > f(v)$ et par suite $f(u) \geqslant f(v)$.

## 4.2 Algorithme de tri topologique

La propriété 14 montre que l'algorithme de parcours en profondeur fournit tous les outils pour effectuer le tri topologique d'un graphe orienté sans circuit.

On peut construire, pendant le parcours en profondeur, une liste des sommets triés dans l'ordre topologique. Pour cela il faut utiliser une structure de liste telle que **l'insertion en tête s'effectue à temps constant** (cas d'une liste chaînée).

**Algorithme 3. Algorithme de tri topologique**

On modifie l'algorithme du parcours en profondeur de la manière suivante :

- Ajouter ligne 1 de la fonction `depth first search` la création d'une liste chaînée **T** initialisée à vide.
- Ajouter la liste `T` aux paramètres de la fonction `visit-DFS`.
- Ajouter ligne 20 de `visit-DFS` l'instruction :
  `insérer u en tête de la liste T`
- Ajouter ligne 9 de la fonction `depth first search`, la liste `T` aux valeurs retournées. On peut supprimer $\pi$, `d` et `couleur` des valeurs de retour.

**Propriété 15.**

L'algorithme tri topologique effectue le tri topologique d'un graphe orienté sans circuit.

*Démonstration* : c'est une conséquence immédiate de la propriété 14 puisque on empile chaque sommet dans la liste `T` à sa date de fin. Donc l'ordre de $T$ est l'ordre décroissant des dates de fin.

**Théorème 16.**

Soit $G = (S, A)$ un graphe **orienté sans boucle**.
$G$ est acyclique si et seulement si il existe un ordre topologique sur $G$.

*Démonstration* : conséquence immédiate des deux propriétés 13 et 15.

**Propriété 17. Complexité**

La complexité du tri topologique est $O(\#A + \#S)$.

*Démonstration* : la complexité est la même que celle du parcours en profondeur si on effectue l'insertion dans la liste `T` à temps constant.