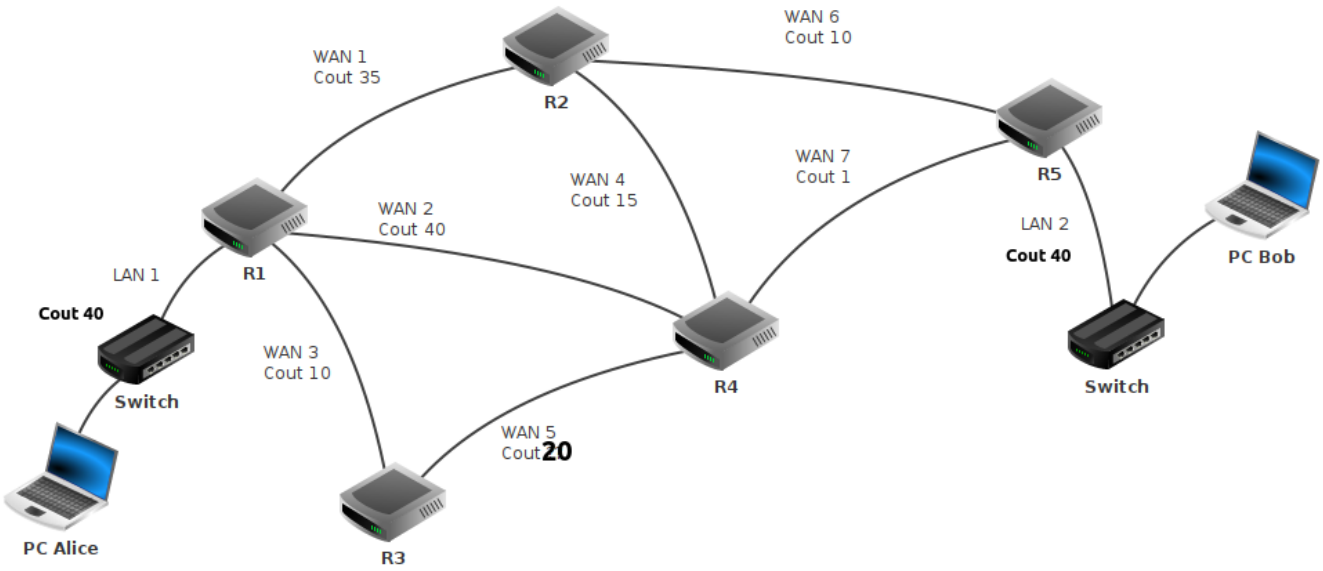
Version pdf

Algorithmes de graphes (Bac 🎯)

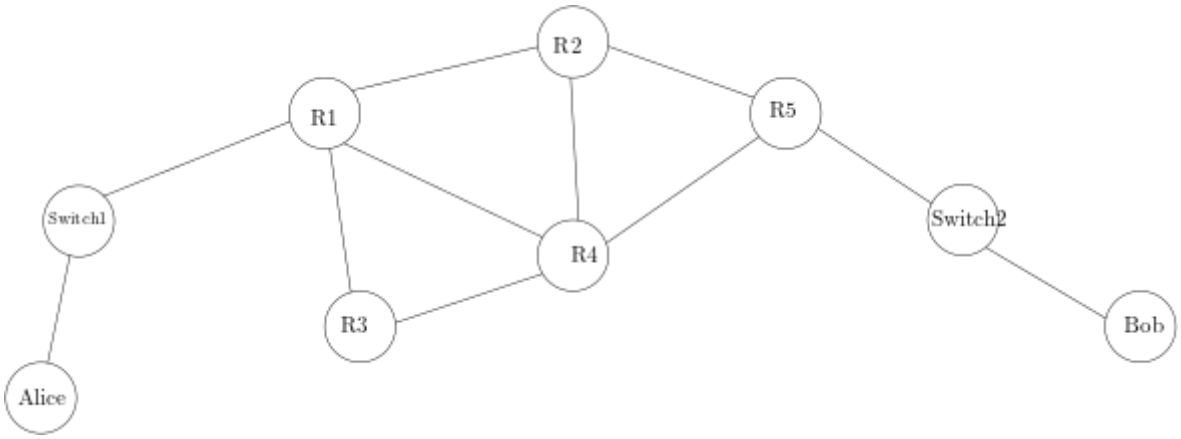
Quelques exemples

"Routage dans un réseau informatique et plus court chemin dans un graphe"



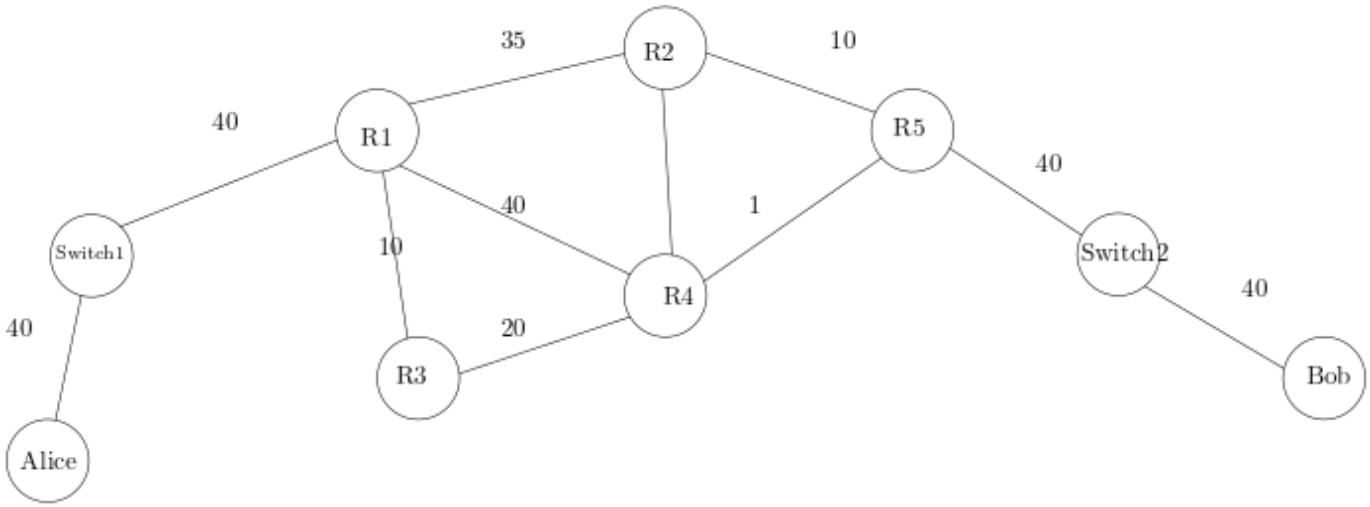
Le réseau informatique ci-dessus peut être modélisé par un *graphe non orienté* dont les sommets sont les routeurs, switchs ou ordinateurs, et les arcs sont les liaisons.

Problème dans la situation modélisée	Problème sur le graphe	Algorithme de graphe
Rechercher la route la plus courte pour acheminer un paquet de Alice vers Bob (protocole de routage RIP)	Recherche de plus court chemin en nombre d'arcs du sommet "Alice" vers le sommet "Bob"	Algorithme de parcours en largeur



Si on tient compte de la bande passante de chaque liaison on peut modéliser le réseau par un *graphe non orienté pondéré* dont les sommets sont les routeurs, switchs ou ordinateurs, et les arcs sont les liaisons, étiquetés par un poids inversement proportionnel à la bande passante.

Problème dans la situation modélisée	**Problème sur le graphe**	**Algorithme de graphe**
Rechercher la route la moins coûteuse pour acheminer un paquet de Alice vers Bob (protocole de routage OSPF)	Recherche de plus court chemin en coût total du chemin du sommet "Alice" vers le sommet "Bob"	Algorithme de Dijkstra

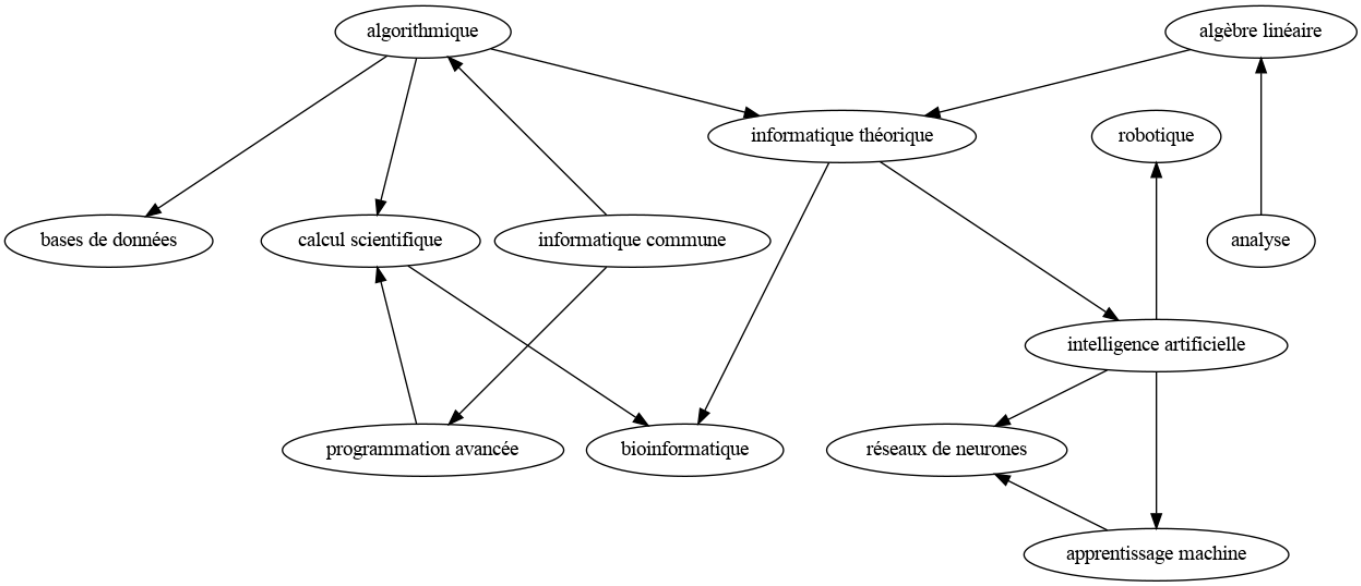


"Planification de tâches et ordre topologique des sommets d'un graphe"

Un professeur d'informatique doit construire une progression à partir de modules d'enseignement dont un graphe orienté de précédence est donné ci-dessous.

Chaque sommet est un module d'enseignement et un arc relie le sommet étiqueté "module A" au sommet étiqueté "module B" si le module A doit être traité avant le module B.

Le problème qui se pose au professeur est donc un problème d'ordonnancement / tri : dans quel ordre peut-il traiter les modules pour respecter les contraintes ? Un ordonnancement des étiquettes de sommets respectant les contraintes de précédence est un ordre topologique du graphe orienté.

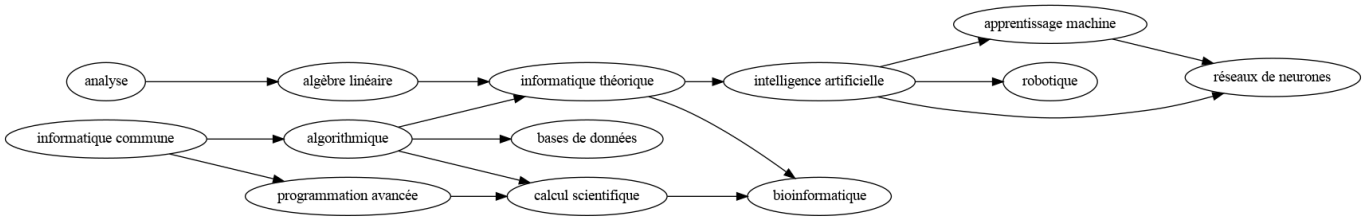


Une solution existe si le graphe est sans cycle et elle est donnée par un *parcours en profondeur du graphe (DFS)* permettant d'énumérer les sommets dans un *ordre topologique* : chaque sommet est nommé avant tous les sommets atteignables depuis lui.

Problème dans la situation modélisée	**Problème sur le graphe**	**Algorithme de graphe**
Ordonner des modules d'enseignement	Recherche d'un ordre tolpologique sur les étiquettes de sommets	Parcours en profondeur DFS

Sur le graphe précédent une solution possible est donnée ci-dessous en plaçant les sommets dans l'ordre topolgique de gauche à droite.

```
analyse -> algèbre linéaire -> informatique commune -> programmation avancée -> inf
```



Parcours de graphes

Parcours générique de graphe

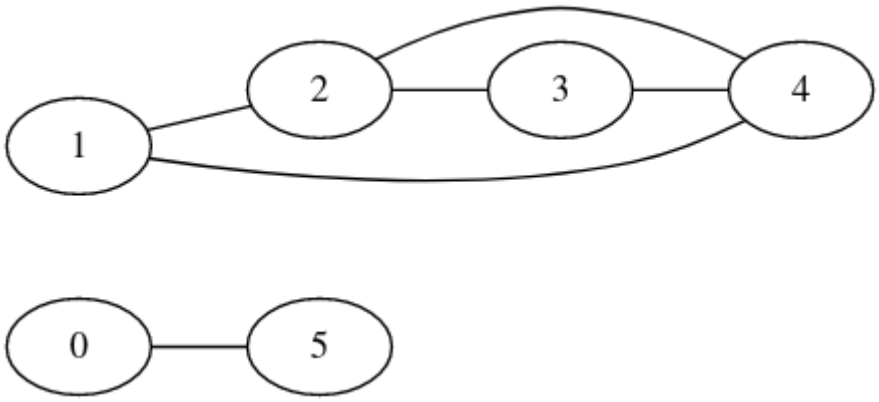
"Point de cours 1 : parcours générique de graphe"

On considère un graphe orienté ou non et un sommet `s` du graphe.

"Définition"

On dit qu'un sommet `w` est **atteignable** depuis `s` s'il existe un chemin dans le graphe d'origine `s` et d'extrémité `w` .

Naturellement on souhaiterait répondre à la question : quels sont les sommets atteignables depuis `s` par un chemin dans le graphe ?



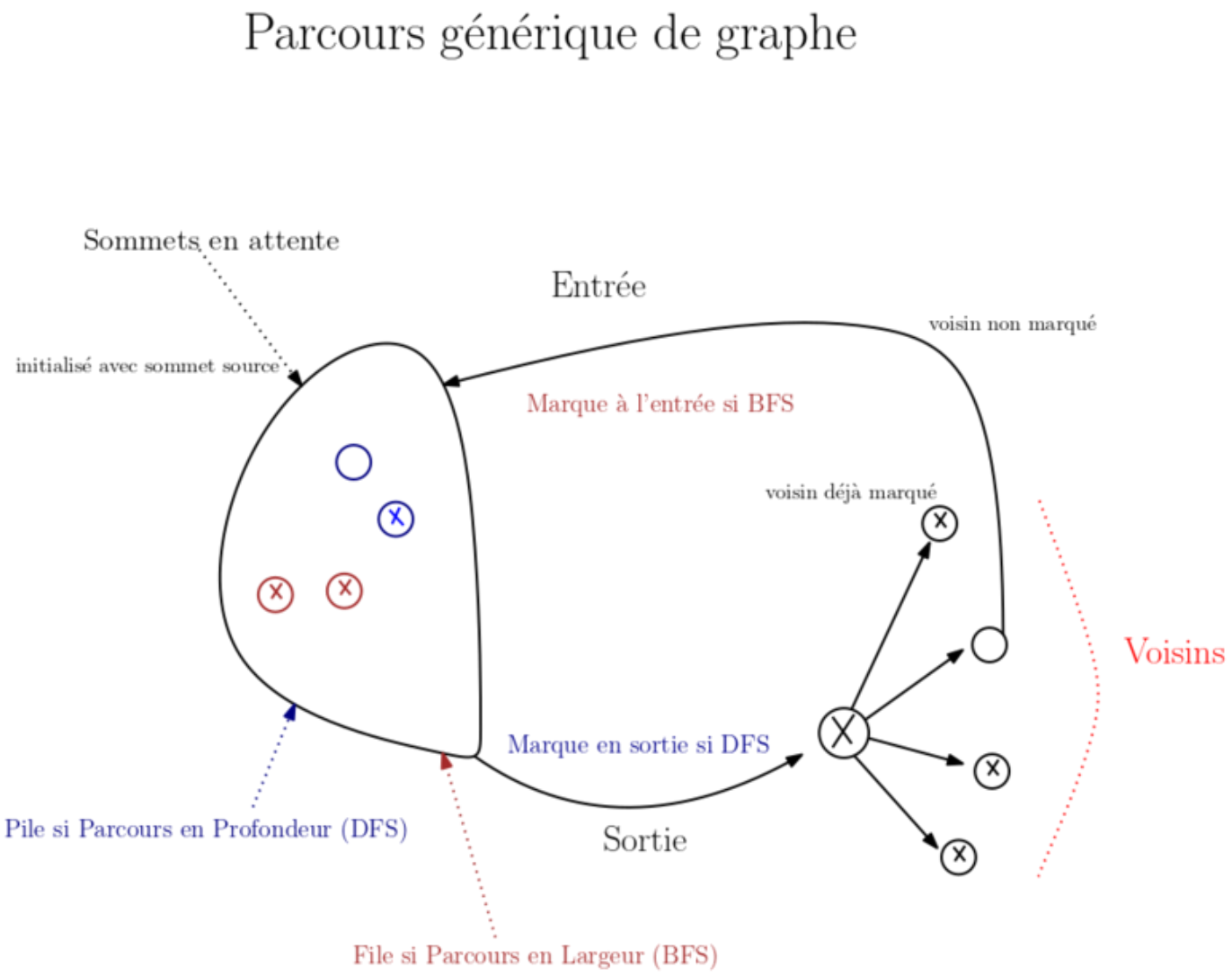
Par exemple dans le graphe non orienté ci-dessus, les sommets d'étiquettes 2, 3 et 4 sont atteignables depuis le sommet d'étiquette 1 mais pas les sommets étiquetés 0 et 5.

Un algorithme permettant de répondre à ce problème est un **parcours de graphe**.

Si on divise les sommets en deux catégories : ceux qui ont déjà été *découverts* par le parcours et les autres , alors on peut décrire simplement un *algorithme générique de parcours de graphe* :

```
On marque le sommet `s` comme découvert
Tant qu'il existe un arc  v -> w tel que v est marqué comme découvert et w comme
    On choisit un tel arc
    On marque son extrémité w comme sommet découvert
```

A chaque itération de la boucle, un sommet passe de l'ensemble des sommets non découverts à l'ensemble des sommets découverts. À la fin de l'algorithme les sommets marqués comme découverts sont exactement ceux atteignables depuis `s`.



Parcours en largeur d'un graphe

"Point de cours 2 : parcours en largeur d'un graphe (BFS)"

On considère un graphe orienté ou non orienté et un sommet `s` du graphe.

Le **parcours en largeur de graphe** ou *Breadth First Search (BFS)* en anglais est une version de l'algorithme générique de parcours de graphe où les sommets en attente sont stockés dans une *file*. On marque un sommet comme découvert lorsqu'on l'insère dans la file d'attente.

L'algorithme découvre les sommets atteignables depuis `s` par couches successives :

- d'abord le sommet `s` en couche 0
- puis les sommets voisins de `s` en couche 1
- puis les voisins de voisins de `s` qui n'ont pas encore été découverts en couche 2
- ...
- puis les voisins des sommets de la couche `i-1` qui n'ont pas encore découverts en couche `i`
- etc ... jusqu'à ce que tous les sommets atteignables depuis `s` soient découverts

Voici une implémentation du **parcours en largeur (BFS)** sous la forme d'une fonction Python qui prend en paramètres le sommet source `s` et le graphe qui est un objet de la classe `Graphe` .

```
def bfs(sommet, graphe):
    """Parcours en largeur d'un graphe instance de la classe Graphe
    depuis un sommet source"""
    decouvert = {s: False for s in graphe.sommets()}
    en_attente = File()
    decouvert[sommet] = True
    en_attente.enfiler(sommet)
    while not en_attente.file_vide():
        s = en_attente.defiler()
        for v in graphe.voisins(s):
            if not decouvert[v]:
                decouvert[v] = True
                en_attente.enfiler(v)
```

"Point de cours 3 : complexité du parcours en largeur d'un graphe"

On considère un graphe orienté ou non orienté et un sommet `s` du graphe.

L'algorithme de **parcours en largeur** est une déclinaison du parcours générique dont on a prouvé qu'il découvrait exactement les sommets atteignables depuis le sommet source `s` .

De plus, le **parcours en largeur** a une complexité en $O(n_s + m_s)$ où n_s et m_s sont respectivement le nombre de sommets et le nombre d'arcs atteignables depuis le sommet source `s` .

"Point de cours 4 : parcours en largeur et calcul de distance"

On considère un graphe orienté ou non orienté, non pondéré, et un sommet `s` du graphe.

"Définition"

On dit qu'un sommet `w` est à distance `d` du sommet source `s` si le plus court chemin d'origine `s` et d'extrémité `w` a pour longueur `d` , en nombre d'arcs.

On a vu dans le point 2 de cours, que **le parcours en largeur** découvre les sommets par couches de plus en éloignées du sommet source `s` . Les sommets découverts dans la couche `d` étant des voisins des sommets découverts dans la couche `d - 1` , on peut démontrer par récurrence que les sommets découverts dans la couche `d` sont exactement ceux à distance `d` du sommet source `s` .

On peut alors augmenter l'algorithme de **parcours en largeur** avec un dictionnaire `distance` permettant de mémoriser la distance à la source des sommets découverts. On initialise ce dictionnaire avec des distances infinies pour tous les sommets sauf le sommet source de distance nulle. On peut ainsi calculer les distances à la source de tous les sommets atteignables.

```python
def bfs_distance(sommet, graphe):
    decouvert = {s: False for s in graphe.sommets()}
    distance = {s:float('inf') for s in graphe.sommets()}
    en_attente = File()
    decouvert[sommet] = True
    distance[sommet] = 0
    en_attente.enfiler(sommet)
    while not en_attente.file_vide():
        s = en_attente.defiler()
        for v in graphe.voisins(s):
            if not decouvert[v]:
                decouvert[v] = True
                distance[v] = distance[s] + 1
                en_attente.enfiler(v)
    return distance
```

Parcours en profondeur d'un graphe

"Point de cours 5 : parcours en profondeur d'un graphe (DFS)"

On considère un graphe orienté ou non orienté et un sommet `s` du graphe.

Le **parcours en profondeur de graphe** ou *Depth First Search (DFS)* en anglais est une version de l'algorithme générique de parcours de graphe où les sommets en attente sont stockés dans une *pile*. On marque un sommet comme découvert lorsqu'on l'extrait de la pile.

"Attention"

On a choisi de garder le terme *découvert* introduit dans le *parcours générique* mais dans le cas du *parcours en profondeur* on devrait parler plutôr de *visité*. La structure de données où sont stockés les sommets en attente est une pile *Last In First Out* donc l'ordre de visite (sortie de la structure) est l'inverse de celui de découverte (entrée dans la pile). Dans un *parcours en largeur*, la structure est *First In First Out* et l'ordre de découverte et de visite sont les mêmes.

L'algorithme découvre les sommets atteignables depuis `s` en s'éloignant toujours plus de la source tant que c'est possible ou en revenant en arrière sinon :

- d'abord le sommet `s`
- puis un des sommets voisins de `s`, jusque là c'est comme pour le parcours en largeur
- mais ensuite au lieu de découvrir un des autres voisins de `s`, le parcours en profondeur va chercher à découvrir l'un des voisins encore non découverts du dernier sommet découvert `v` (le sommet de la pile). S'il n'y en a pas, il revient sur ses pas jusqu'au prédécesseur de `v` pour explorer d'autres voisins non découverts de ce sommet ou encore revenir en arrière ... jusqu'à ce que tous les sommets atteignables soient découverts.

Voici une implémentation du **parcours en profondeur (DFS)** sous la forme d'une fonction Python qui prend en paramètres le sommet source `s` et le graphe qui est un objet de la classe `Graphe`.

```
def dfs(sommet, graphe):
    """Parcours en profondeur d'un graphe instance de la classe Graphe
    depuis un sommet source s"""
    decouvert = {s: False for s in graphe.sommets()}
    en_attente = Pile()
    en_attente.empiler(sommet)
    while not en_attente.pile_vide():
        s = en_attente.depiler()
        if not decouvert[s]:
            decouvert[s] = True
            for v in graphe.voisins(s):
                if not decouvert[v]:
                    en_attente.empiler(v)
```

"Point de cours 6 : complexité du parcours en profondeur d'un graphe"

On considère un graphe orienté ou non orienté et un sommet `s` du graphe.

L'algorithme de **parcours en profondeur** est une déclinaison du parcours générique dont on a prouvé qu'il découvrait exactement les sommets atteignables depuis le sommet source `s`.

De plus, le **parcours en profondeur** a la même complexité que le parcours en largeur, en $O(n_s + m_s)$ où n_s et m_s sont respectivement le nombre de sommets et le nombre d'arcs atteignables depuis le sommet source `s`.

"Point de cours 7 : parcours en profondeur récursif"

On considère un graphe orienté ou non orienté et un sommet `s` du graphe.

La *pile* des sommets en attente d'un parcours en profondeur peut être simulée par la pile des appels imbriqués d'une fonction *récursive*.

On peut alors donner une version récursive élégante du **parcours en profondeur** :

```
def dfs_rec(sommet, graphe, decouvert):
    """Parcours en profondeur d'un graphe instance de la classe Graphe
    depuis un sommet source.
    Decouvert est un dictionnaire associant à chaque sommet sa marque de visite"
    decouvert[sommet] = True
    for v in graphe.voisins(sommet):
        if not decouvert[v]:
            dfs_rec(v, graphe, decouvert)
```