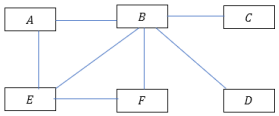
Chapitre 16 : Parcours d'un graphe

I- Introduction

Nous utiliserons le graphe non pondéré et non orienté représenté ci-dessous pour illustrer ce chapitre :




```
g={"A":["B","E"],
   "B":["A","C","D","E","F"],
   "C":["B"],
   "D":["B"],
   "E":["A","B","F"],
   "F":["B","E"]}
```

L'idée est de se doter de méthodes permettant de parcourir un graphe à partir d'un sommet *s* permettant ainsi de :

- Savoir s'il est connexe (ou la classe de connexité de *s*)
- Savoir si le graphe présente un cycle
- Déterminer le plus court chemin reliant deux sommets

II- Parcours en profondeur d'un graphe

a) Principe

A partir du sommet de départ, on passe à un de ses voisins (le dernier dans la liste des voisins) et ainsi de suite. S'il n'y a pas de voisin (ou de nouveau voisin), on revient au sommet précédent et on passe à un autre voisin. On utilise une liste *visite* pour stocker les éléments visités et une liste *attente* pour stocker les voisins encore non visités d'un sommet rencontré. Si on obtient une liste *visite* contenant tous les sommets c'est que le graphe est connexe. Sinon, on a l'ensemble des sommets que l'on peut joindre depuis le sommet de départ *s* c'est-à-dire la classe de connexité de *s*.

b) Programme itératif :

```
def pp_iteratif(g,sommet):
    """g est le graphe représenté par un dictionnaire des lites d'adjacence
    sommet est initialement
    le sommet de départ pour lequel on va lister tous les sommets accessibles
    c-a-d sa classe de connexité
    Principe du parcours en profondeur:
    on prend un sommet et on va visiter un de ses voisins
    et ainsi de suite. Si plus de voisin ou de nouveau voisin
    on revient au sommet précédent et on teste ses autres voisins"""
    visite=[]#liste des sommets visités
    attente=deque([sommet])
    while len(attente)!=0:
        sommet=attente.pop()#le prochain sommet est le dernier de la pile
        if sommet not in visite :
            visite.append(sommet)
            for s in g[sommet]:
                if s not in visite:
                    attente.append(s)
    return visite
```

c) Programme récursif

```
def pp_recursif(g,attente,visite):
    """g est le dictionnaire des listes d'adjacence
    attente est la pile des sommets en attente (initialisé sur le sommet de départ)
    visite est la liste, initialement vide, des sommets rencontrés"""
    if len(attente)==0:
        return visite
    else :
        sommet=attente.pop()
        if sommet not in visite:
            visite.append(sommet)
            for s in g[sommet]:
                if s not in visite :
                    attente.append(s)
        return pp_recursif(g,attente,visite)
```

III- Parcours en largeur

a) Principe

A partir d'un sommet, on explore tous ses voisins immédiats. Puis à partir d'un voisin, on explore tous ses voisins immédiats sauf ceux déjà explorés et ainsi de suite.

b) Programme itératif et récursif

Ici, on utilise une file :

```
def pl_itératif(g,sommet):
    visite=[]
    attente=deque([sommet])#file
    while len(attente)!=0:
        sommet=attente.popleft()
        if sommet not in visite:
            visite.append(sommet)
            for s in g[sommet]:
                if s not in visite:
                    attente.append(s)
    return visite
```

```
def pl_recursif(g,attente,visite):
    if len(attente)==0:
        return visite
    else :
        sommet=attente.popleft()
        if sommet not in visite:
            visite.append(sommet)
            for s in g[sommet]:
                if s not in visite:
                    attente.append(s)
        return pl_recursif(g,attente,visite)
```

Ce type de parcours est très utilisé pour étudier des graphes orientés et pondérés.

Commenté [AM1]: Les applications en logistique sont évidentes

Commenté [AM3]: Cette progression qui impose d'abord de visiter les voisins les plus proches s'obtient avec une structure de type file

Commenté [AM4]: Comme avec le parcours en profondeur, pour un graphe connexe, tous les sommets sont visités et on peut obtenir un chemin reliant deux sommets quelconques

Commenté [AM2]: C'est l'algorithme le plus efficace d'un point de vue complexité temporelle car il évite de tester tous les sommets à chaque étape alors qu'ils ont été visités. Le parcours en profondeur est donc efficace pour vérifier si un graphe est connexe

Commenté [AM5]: Car ce type de parcours test vraiment toutes les combinaisons