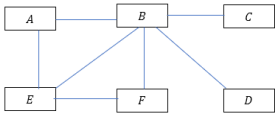
Parcours d'un graphe

I- Introduction

Nous utiliserons le graphe représenté ci-dessous pour ce chapitre :




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

II- Parcours en profondeur d'un graphe

a) Principe

A partir du sommet de départ, on passe à un de ses voisins (le 1e dans la liste d'adjacence), puis à un voisin de ce voisin et ainsi de suite. S'il n'y a pas de voisin, on revient au sommet précédent et on passe à un autre de ses voisins. On utilise une liste *visite* pour stocker les éléments visités et une liste *attente* pour stocker les voisins encore non visités d'un sommet. 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 *S* c'est-à-dire la classe de connexité de *S*.

b) Programme :

```
def iteratif_pp(g,sommet):
    """g est le dictionnaire des listes d'adjacence
    sommet est le sommet à visiter (initialisé sur le sommet
de départ)"""
    visite=[]
    attente=deque()#pile vide
    attente.append(sommet)
    while len(attente)>0:
        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 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

Ici, on utilise une file :

```
def parcours_largeur(g,sommet):
    """g est le dictionnaire des listes d'adjacence
    sommet est le sommet à visiter (initialisé sur le sommet
    de départ) en prenant une file, on assure la parcours en
    largeur"""
    visite=[]
    attente=deque()#file vide
    attente.append(sommet)
    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
```

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

Commenté [AM1]: Les applications en logistique sont évidentes

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.

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