TD: algorithmes sur les graphe non orientés:

Exercice 1: représentation d'un graphe et fonction affichage:
Un graphe est représenté en Python par la liste de ses successeurs et est implémenté de la façon suivante:

```
#graphe represente par la liste de ses successeurs
g = {
      'A':['B','E'],
      'B':['A','C','E'],
      'C':['B'],
      'D':[],
      'E':['A','B']
      }
```

affiche.py

1. Quel est le type de g?
2. Représenter g à l'aide d'un schéma.

On écrit ensuite une fonction affiche qui aura pour but d'afficher un parcours dans un graphe:

```
def affiche(chemin):
    ch = ""
    if len(chemin) == 0:
        return ch
    ch = str(chemin[0])
    for i in range(1,len(chemin)):
        ch = ch + "->" + chemin[i]
    return ch
```

affiche.py

3. Quel est le type de la variable chemin? Et celui de la variable ch?
4. Expliquer le fonctionnement de la fonction *affiche*. La tester.

Exercice 2: parcours en profondeur d'un graphe (version récursive):
On s'interesse au programme suivant où g est le graphe de l'exercice 1:

```
def parcours_dfs_rec(graphe, sommet, visite = []):
    if sommet not in visite:
        visite.append(???)
    nonvisite = [s for s in graphe[sommet] if s not in visite]
    for s in nonvisite:
        parcours_dfs_rec(???, ???, ???)
    return visite

parcours = parcours_dfs_rec(g, 'A')
print("parcours:", affiche(parcours))
```

profondeur_recursif.py

1. En quoi cette fonction est-elle récursive?
2. A l'aide de l'algorithme correspondant du cours, complétez les ???.
3. Tester alors cette fonction.

Exercice 3: parcours en profondeur d'un graphe (version itérative):
On s'interesse au programme suivant où g est le graphe de l'exercice 1. La bibliothèque collections doit être préalablement importée.

```
def parcours_dfs_it(graphe, sommet):
    visite = []
    pile = deque([])
    pile.append(sommet)
    while len(pile) > ???:
        sommet = pile.pop()
        if sommet not in visite:
            ???
            for s in graphe[sommet]:
                if s not in visite:
                    ???
    return(visite)

parcours = parcours_dfs_it(g, 'A')
print("parcours:", affiche(parcours))
```

profondeur_iteratif.py

1. Cette fonction est-elle récursive?
2. Quelle structure de donnée particulière est utilisée dans cette fonction? L'utilisation de la bibliothèque collections est-elle indispensable ici?
3. A l'aide de l'algorithme correspondant du cours, complétez les ???.
4. Tester alors cette fonction.

Exercice 4: parcours en profondeur d'un graphe (version objet):

On s'interesse au programme suivant:

```
class graph:
    def __init__(self):
        self.sommets = []
        self.aretes = {}

    def ajoute_sommet(self, s):
        if s in self.sommets:
            raise ValueError("sommet deja present")
        else:
            self.sommets.append(s)
            self.aretes[s] = []

    def ajoute_arete(self, s1, s2):
        if ((s1 not in self.sommets) or (s2 not in self.sommets)):
            raise ValueError("sommet(s) absent(s)")
        else:
            self.aretes[s1].append(s2)
            self.aretes[s2].append(s1)

    def lien_avec(self, s):
        return self.aretes[s]

    def __str__(self):
        res = ""
        for k in self.aretes:
            for d in self.aretes[k]:
                res = res + str(k) + "-" + str(d) + "\n"
        return res
```

profondeur_objet_graphe.py

1. Expliquer en détail le fonctionnement des différentes méthodes.
2. Implémenter le graphe de l'exercice 1.
3. Tester la méthode spéciale d'affichage.
4. Ecrire une fonction récursive qui permette le parcours en profondeur d'un arbre. On utilisera la fonction *affiche* de l'exercice 1 ainsi que la fonction récursive vue dans les exercices précédents.
5. Ecrire une fonction itérative qui permette le parcours en profondeur d'un arbre. On utilisera la fonction *affiche* de l'exercice 1 ainsi que la fonction itérative vue dans les exercices précédents.

Exercice 5: recherche d'un plus court chemin (parcours en profondeur):
On cherche le plus court chemin entre deux sommets d'un graphe. Cette recherche peut s'effectuer avec un parcours en profondeur d'abord. On utilise la fonction suivante, adaptée à l'implémentation objet d'un graphe:

```
def pcc_dfs(graphe, debut, fin, chemin = [], pluscourt = None):
    chemin = chemin + [debut]
    #print("chemin courant:", affiche(chemin))   #si besoin d'afficher la progression
    if debut == fin:
        return chemin
    for s in graphe.lien_avec(debut):
        if s not in chemin:
            if pluscourt == None or len(chemin) < len(pluscourt):
                nouveau = pcc_dfs(graphe, s, fin, chemin, pluscourt)
                if nouveau != None:
                    pluscourt = nouveau
    return pluscourt
```

profondeur_objet_graphe.py

1. Expliquer le fonctionnement de cette fonction et la tester.
2. Adapter cette fonction à un graphe implémenté sous la forme d'un dictionnaire.

Exercice 6: parcours en largeur:

```
from collections import *

def largeur_bfs(graphe,sommet):
    visite = []
    file = deque([])
    file.append(sommet)
    while len(file) > ???:
        sommet = ???
        if sommet not in visite:
            visite.append(???)
            for s in graphe[sommet]:
                if s not in visite:
                    file.append(???)
    return visite
```

largeur.py

1. Cette fonction est-elle récursive?
2. Quelle structure de donnée particulière est utilisée dans cette fonction? L'utilisation de la bibliothèque collections est-elle indispensable ici?
3. A l'aide de l'algorithme correspondant du cours, complétez les ???.
4. Tester alors cette fonction.

Exercice 7: recherche d'un plus court chemin (parcours en largeur):

On s'interesse à la fonction suivante qui permet l'affichage des chemins entre deux sommets d'un graphe:

```
def chemin_bfs(graphe, dep, arr):
    file = deque([])
    file.append((dep,[dep]))
    while len(file) > 0:
        sommet, chemin = file.popleft()
        for s in graphe[sommet]:
            if s not in chemin:
                if s == arr:
                    yield chemin + [s] #yield comme return mais pour un generateur
                else:
                    file.append((s, chemin + [s]))
    return chemin

chemins = chemin_bfs(g, 'A', 'E')
for chem in chemins:
    print(affiche(chem))
```

largeur.py

1. Ecrire cette fonction et la tester sur le graphe g.
2. Ecrire une nouvelle fonction *plus_court_chemin_bfs* qui permet d'afficher le plus court chemin entre deux sommets d'un graphe.

Exercice 8: recherche d'un cycle:

On s'interesse à la fonction suivante qui permet de dire si un cycle existe à partir d'un sommet:

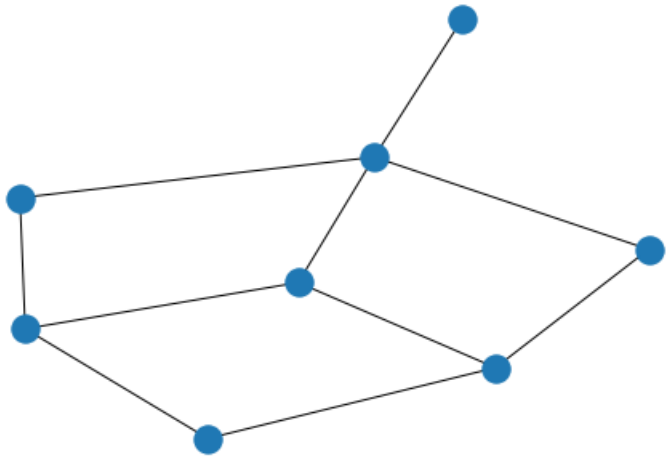
```
def cycle_bfs(graphe,sommet):
        niveaux = {s : None for s in graphe}
        niveaux[sommet] = 0
        file = deque([])
        file.append(sommet)
        while len(file) > 0:
            sommet = file.popleft()
            for s in graphe[sommet]:
                if niveaux[s] is None:
                    niveaux[s] = niveaux[sommet] + 1
                    file.append(s)
                elif niveaux[s] >= niveaux[sommet]:
                    return True
        return False
```

largeur.py

1. Ecrire cette fonction et la tester sur le graphe g.
2. Expliquer son fonctionnement. En quoi utilise t-elle un parcours en largeur.
3. L'algorithme du cours utilise lui un parcours en profondeur. Ecrire une fonction *cycle_dfs(graphe, sommet)* reprenant l'algorithme du cours et permettant de dire si un cycle existe à partir d'un sommet.

Exercice 9: parcours 'à la main':

On s'interesse au graphe suivant. Choisir un sommet et représenter:



1. un parcours en largeur et l'arbre associé. L'arbre est-il couvrant? Le graphe est-il connexe?
2. un parcours en profondeur et l'arbre associé. L'arbre est-il couvrant? Le graphe est-il connexe?
3. quel parcours permet de déterminer les plus courts trajets au sommet choisi?