Feuille de TP n° 4 : graphes

Ce TP s'écrit en 70 lignes de code environ.

Exercice 1. ***Parcours en largeur***

a) Récupérer les modules `Queue.py`, `List.py` et `Graph.py` sur Moodle et étudier les classes définies.

Le module `Queue.py` implémente une file par une liste doublement chaînée, le module `List.py` implémente un liste simplement chainée.

Dans le module `Graph.py`, on a importé les deux modules précédents avec l'instruction `import` pour pouvoir utiliser les classes qui y sont définies.

De ce fait, pour désigner la classe `Queue` du module `Queue.py`, il faut écrirer `import Queue` dans `Graph.py` puis : `Queue.Queue`. Le premier nom est pour le nom du module, le deuxième pour le nom de la classe ou de la fonction que l'on désigne.

De même, si vous voulez créer une instance de `Linked_list` du module `List.py` dans le module `Graph.py`, il faut utiliser `import List` puis : `List.Linked_list()`.

Dans `Graph.py`, on a créé un classe `Adj_list` dérivée de `List.Linked_list`. Le but est de surcharger la méthode `print` qui existait dans `List.Linked_list`.

Comme maintenant, l'attribut `data` est un objet, la méthode `print` héritée aurait affiché les références (des adresses) des objets. On veut que la méthode `print` affiche maintenant les noms des nœuds adjacents.

b) Utiliser la méthode d'initialisation puis la méthode `add_edge` de la classe `Graph` pour créer le graphe $G = (S, A)$ tel que :
$S = \{a, b, c, d, e, f, g, h\}$ et
$A = \{(a, b), (a, g), (b, a), (b, c), (b, e), (c, a), (c, f), (e, c), (e, g), (f, d), (g, b), (g, h), (h, c), (h, d), (h, f)\}$.

On créera donc le graphe par l'appel :
`G=Graph(nodes)` avec : `nodes=['a','b','c','d','e','f','g','h']`.
Puis on pourra utiliser la liste suivantes pour créer les arcs :
`A=[[0,1],[0,6],[1,0],[1,2],[1,4],[2,0],[2,5],[4,2],[4,6],[5,3],[6,1],[6,7],[7,2],[7,3],[7,5]]`
Puis vous pourrez itérer la méthode `add_edge()` pour chaque couple de la liste `edges`.

Vérifier que votre graphe est créé correctement en utilisant la méthode `print_adj_list` déjà écrite dans la classe `Graph`.

c) Ecrire une fonction de parcours en largeur à partir d'un sommet s. On rappelle l'algorithme en page suivante.

d) Utiliser le parcours en largeur pour afficher la liste des sommets accessibles depuis 'c', les plus courtes distances depuis le sommet 'c' ainsi qu'un plus court chemin de 'c' à 'g'.

```
fonction BFS(G,s)
|   créer une file F et trois tableaux π, d et couleur de longueur #S(G)
|   pour tout sommet u ∈ S(G) \{s} faire
|   |   couleur[u] = blanc
|   |   d[u] = ∞
|   |   π[u] = nil
|   d[s]= 0
|   π[s]= nil
|   enfiler(F,s)
|   couleur[s]=gris
|   tant que F ≠ ∅ faire
|   |   u = defiler(F)
|   |   Pour tout sommet v adjacent à u faire
|   |   |   si couleur[v]==blanc alors
|   |   |   |   couleur[v]=gris
|   |   |   |   d[v] = d[u] +1
|   |   |   |   π[v] = u
|   |   |   |   enfiler(F,v)
|   |   couleur[u]=noir
|   retourner d, π, couleur
```

Exercice 2. ***Parcours en profondeur***

a) En ajoutant un système de coloriage à l'algorithme récursif du parcours en profondeur sur les arbres, écrire une fonction `visit_DFS(G,u,couleur)` qui effectue un parcours en profondeur récursif depuis un sommet u.
   NB : Les dates de début et de fin ne sont pas nécessaires. Si vous voulez les utiliser, il faut ajouter les deux tableaux `d` et `f` aux paramètres de la fonction et il faut déclarer la variable `date` comme une variable globale.
   Pour cela, il faut écrire dans toutes les fonctions où la variable est utilisée (`visit_DFS` et `DFS`), la ligne :
   `global date`
   avant la première utilisation de la variable dans la fonction.

b) Ecrire une fonction `DFS(G)` qui initialise le coloriage et effectue de manière itérative des appels à `visit_DFS` pour tous les sommets de G s'ils n'ont pas déjà été visités.

c) Créer une liste Python `parent` dans la fonction `DFS(G)` puis modifier la fonction `visit_DFS` pour stocker le parent de chaque nœud dans la forêt du parcours en profondeur.

d) Ecrire une fonction `acyclique(G)` qui détecte si un graphe est cyclique ou non. On rappelle qu'une condition nécessaire et suffisante pour l'absence de circuit est l'absence d'**arc arrière** généré par le parcours en profondeur. Une méthode efficace est d'utiliser une couleur grise pour les nœuds dont la visite est commencée mais pas terminée.

e) Tester si le graphe $G = (S, A)$ suivant est acyclique, sinon supprimer des arcs pour qu'il le devienne puis afficher un tri topologique à l'aide d'une fonction Python `topologic_sort(G)`.
   $S = \{a, b, c, d, e, f, g, h\}$ et
   $A = \{(a, b), (a, g), (b, a), (b, c), (b, e), (c, a), (c, f), (e, c), (e, g), (f, d), (g, b), (g, h), (h, c), (h, d), (h, f)\}$.
   La liste des arcs est donc :
   `A=[[0,1],[0,6],[1,4],[2,0],[2,5],[4,3],[4,6],[5,3],[5,4],[5,7],[6,7],[7,2],[7,3],[7,5]]`

   On rappelle qu'un algorithme de tri topologique est :
   ```
   Créer une liste L initialisée à vide.
   Appeler DFS(G) modifiée de sorte que lorsque la visite d'un sommet se termine,
   il est inséré au début de L
   ```