ITC – TD n°12-3

Autour des arbres

Il est recommandé de garder les fonctions que vous écrirez dans chaque TD sous la main : elles pourront servir de brique élémentaires / fonctions utilitaires pour d'autres TD.

On fournit un module proposant plusieurs graphes pré-codés ainsi que des méthodes d'affichage. On pourra les charger avec l'instruction

```
import module_graphes as gr
```

Représentation des données

Voir TD précédents

On écrit un certain nombre de fonctions sur les graphes non orientés, non pondérés. On peut récupérer et visualiser des graphes de test avec les instructions

```
g1 = gr.recuperer_graphe('g4')
g2 = gr.recuperer_graphe('g6')
gr.afficher_graphes_multiples([g1, g2])
```

En cas de besoin, on peut obtenir une file et une pile avec les instructions suivantes

```
from queue import Queue      # file
from queue import LifoQueue  # pile
```

qui ont le comportement suivant (les fonctions sont les mêmes pour les piles)

```
file = Queue()      # création d'une file
pile = LifoQueue()  # création d'une pile
file.empty()  # teste si vide
file.put(x)   # ajoute x à la file
file.get()    # renvoie et enlève le premier élément
```

I. Détecter les cycles

On s'intéresse aux arbres, qui sont des graphes **simples**, **non orientés** et **acycliques**. Ainsi, si on sait qu'un graphe non orienté est connexe et ne contient pas de boucle, il ne reste qu'à tester la présence de cycles. Pour ce faire, on s'appuie sur les algorithmes classiques de parcours : si un sommet en cours de traitement a un voisin déjà trouvé (bleu ou rouge pour le parcours en largeur, bleu pour le parcours en profondeur) alors il y a un cycle.

1 – Écrire une fonction `init(g)` qui prend en paramètre un graphe et renvoie trois dictionnaires : `état` qui contient l'état de chaque sommet (initialement 'V'), `préc` qui contient le sommet précédent au cours du parcours (initialement ''), et dist qui contient la distance au sommet de départ (initialement ∞).

2 – Écrire une fonction `présence_cycle(g)` qui effectue un parcours en pronfondeur du graphe g et qui renvoie `True` si le parcours détecte un cycle, et `False` sinon. On commencera l'exploration sur le sommet `'k'`. On pourra se doter d'une fonction de visite des sommets.

II. Arbres de parcours

Puisque les algorithmes de parcours construisent un arbre lors de leur exploration du graphe, on peut les utiliser pour construire un arbre qui soit un sous-graphe de G contenant tous les sommets de S.

3 – Écrire une fonction `extraire_arbre(préc)` qui prend en paramètre le dictionnaire des précédents lors du parcours et créé à partir de lui un graphe représentant l'arbre de parcours : si u est le précédent de v, alors cet arbre de parcours est constitué de S et des arêtes (u, v). **On veillera à ce que l'arbre renvoyé soit non orienté**.

4 – Écrire une fonction `arbre_p(g)` qui prend en paramètre un graphe et renvoie l'arbre de parcours en profondeur, commençant sur le sommet 'k'.

5 – Écrire une fonction `arbre_l(g)` qui prend en paramètre un graphe et renvoie l'arbre de parcours en largeur, commençant sur le sommet 'k'.

6 – Afficher ces deux arbres avec

```
g1_p = arbre_p(g1)
g1_l = arbre_l(g1)
gr.afficher_graphes_multiples([g1, g1_p, g1_l])
```

Lequel semble avoir la plus petite hauteur (si on enracine en 'k') ? Quelle caractéristique obtenue lors du parcours donne la hauteur de l'arbre ?

III. Hauteur minimale

On cherche à construire l'arbre de parcours de hauteur minimale ; on s'appuie sur le parcours en largeur.

7 –Écrire une fonction `parcours_arbre_min(g, u0)` qui effectue un parcours en largeur du graphe fourni depuis le sommet `u0`, puis qui renvoie la plus petite distance du parcours depuis u_0 ainsi que le dictionnaire des précédents.

8 – Écrire une fonction `arbre_min(g)` qui détermine le sommet u_0 permettant l'arbre de parcours le plus bas, et qui renvoie l'arbre en question. On pourra tester à l'aide de

```
g1_l = arbre_l(g1)
g1_min = arbre_min(g1)
gr.afficher_graphes_multiples([g1, g1_l, g1_min])
```