ITC – TD n°12-1
Prise en main des graphes non orientés

Le présent TD a pour but de prendre en main les définitions et les outils que nous utiliserons tout au long des chapitres sur les graphes.

> 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

Dans tous nos travaux, les graphes sont représentés par un dictionnaire de dictionnaires d'adjacence, les sommets ayant pour type str ; tous les sommets u sont des clés valides du graphe g, et v est une clé valide de $g[u]$ uniquement si g contient l'arc (u, v) ; dans ce cas $g[u][v]$ est le poids de l'arc (u, v), noté $w_{(u,v)}$ dans le sujet ; si le graphe n'est pas pondéré, $g[u][v] = 1$ $\forall (u, v)$. Lorsqu'une fonction « prend en paramètre un graphe », elle prend donc en paramètre un dictionnaire `{str: {str: int}}` .

Si $g[u][v] = g[v][u]$ $\forall (u, v)$, alors le graphe est non orienté ; sinon il est orienté.

Les informations supplémentaires sont stockées dans des dictionnaires ayant pour clés les sommets.

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

```
g1 = gr.recuperer_graphe('g1')
gr.afficher_graphe(g1)
```

Si on souhaite récupérer un graphe pondéré, on peut écrire

```
g1 = gr.recuperer_graphe('g1', ponderation=True)
gr.afficher_graphe(g1)
```

I. Échauffement

1 – Écrire une fonction `degré_non_orienté(g, u)` qui prend en paramètres un graphe g et un sommet u, et renvoie le degré du sommet.

2 – Écrire une fonction `degré_moyen(g)` qui calcule et renvoie le degré moyen des sommets du graphe g.

3 – Écrire une fonction `liste_boucles(g)` qui prend en paramètres un graphe, et renvoie une liste contenant tous les sommets qui apartiennent à une boucle, c'est-à-dire les sommets u tels que l'arête $\{u, u\}$ appartienne au graphe.

4 – Un graphe (non orienté) ayant n_S sommets, il a au plus $n_{\max=}n_S(n_S + 1)/2$ arêtes. On définit alors la densité du graphe comme $d = n_A/n_{\max}$. Écrire une fonction `densité(g)` qui calcule la densité du graphe fourni. Quelle est la complexité de cette fonction ?

II. Pondération

5 – Écrire une fonction `est_pondéré(g)` qui renvoie `True` si le graphe g est pondéré et `False` sinon.

6 – Écrire une procédure `ajouter_poids(g)` qui modifie g en ajoutant des poids aléatoires (compris entre 0 et 10) à chaque arête. On fera attention à laisser le graphe non orienté. On peut importer la fonction `randint` du module `random` pour tirer un nombre au hasard avec `randint(1, 10)`.

7 –Écrire une procédure `retirer_poids(g)` qui modifie g en enlevant les poids à chaque arête.

III. Créer des graphes

8 – Écrire une fonction `sous_graphe_engendré(g, liste_sommets)` qui prend en paramètres un graphe (S, A) et une liste de sommets $S' \subset S$, et renvoie le sous-graphe induit par G sur S'. On pourra vérifier si la fonction agit comme attendu en affichant côte à côte le graphe de départ et le sous-graphe extrait :

```
g1 = gr.recuperer_graphe('g1')
g2 = sous_graphe_engendré(g1, ['a', 'k', 'h', 'o', 'n'])
gr.afficher_graphes_multiples([g1, g2])
```

9 – Écrire une fonction `est_un_sous_graphe(g1, g2)` qui renvoie `True` si g_1 est un sous-graphe de g_2, et `False` sinon. On pourra tester en récupérant deux graphes candidats et en les affichant à côté de g_1 pour vérifier à l'œil lequel est un sous-graphe de g_1 :

```
g1 = gr.recuperer_graphe('g1')
g2 = gr.recuperer_graphe('sg1-1')
g3 = gr.recuperer_graphe('sg1-2')
gr.afficher_graphes_multiples([g1, g2, g3])
```

10 – Le carré G^2 d'un graphe $G = (S, A)$ est le graphe (S, A^2) avec A^2 l'ensemble des arêtes (u, w) telles que $(u, v) \in A$ et (v, w) inA ; autrement dit, il existe un chemin de longueur exactement deux dans G pour aller de u à w. Écrire une fonction `carré(g)` qui calcule et renvoie le graphe carré de g. Quelle est la complexité de cette fonction ? Pourrait-on faire mieux avec une représentation par matrice d'adjacence ?