CCTP

ACM Kruskal

TP à réaliser seul sur un notebook python bien organisé et commenté et à déposer sur Moodle avant le 14 décembre 23h55.
N'oubliez pas d'indiquer sur le notebook : numéro d'étudiant, nom et prénom.

Construction d'un arbre couvrant minimum avec l'algorithme de Kruskal

Dans ce TP, vous devez reprendre le notebook dans lequel vous avez codé les exercices du TP3 (construction d'un arbre couvrant de poids minimum avec l'algorithme de PRIM).

L'objectif est de compléter votre code afin de réaliser la construction d'un arbre couvrant de poids minimum avec l'algorithme de Kruskal.

Pour un graphe non orienté qui possède n sommets, l'algorithme de Kruskal consiste à trier les arêtes du graphe en ordre de poids croissant et à construire l'arbre couvrant en sélectionnant parmi les arêtes triées, les $n-1$ premières arêtes qui ne forment pas de cycle.
Il est donc nécessaire pour le mettre en oeuvre d'ajouter des fonctionnalités à la classe `GrapheNOPLS`

Exercice 1

A l'issu du TP3 votre classe `GrapheNOPLS` devrait fournir (au minimum) les méthodes suivantes.
- `sommets`, qui retourne la liste de tous les sommets du graphe
- `ajouteArete`, qui permet d'ajouter au graphe une arête pondérée entre deux sommets du graphe
- `ajouteLesAretes`, qui permet d'ajouter au graphe une liste d'arêtes pondérées
- `poids` qui retourne le poids d'une arête dont les extrémités sont données en paramètres
- `voisins`, qui retourne la liste des sommets voisins d'un sommet donné en paramètre
- `degre`, qui retourne le degré d'un sommet donné en paramètre
- `affiche`, qui produit un affichage du graphe

Si ça n'est pas le cas, vous devez déjà coder ce qui manque.

Exercice 2

Et afin de mettre en oeuvre l'algorithme de Kruskal vous devez compléter cette classe en ajoutent les méthodes :
- `lesAretes`, qui retourne la liste des arêtes du graphe

- `supprimeArete`, qui permet de supprimer du graphe l'arête dont les extrémités et le poids sont donnés en paramètre.
  Cette méthode est nécessaire pour la construction de l'arbre couvrant : pour savoir si un arête doit être dans l'arbre, on l'y ajoute et si le graphe obtenu contient un cycle, on l'en retire !

Détection d'un cycle par un parcours en largeur

Lors du parcours en largeur d'un graphe tel que décrit dans le cours, chaque sommet passe par trois couleurs :
- d'abord Bleu lorsque le sommet n'a pas encore été visité ;
- Vert lorsqu'il est en cours de visite, c'est à dire lorsque tous ses voisins n'ont pas encore été visités ;
- enfin Rouge lorsque que tous ses voisins ont été visités.

Dans ce parcours, le coloriage des sommets en vert permet de ne pas « tourner en rond » dans un éventuel cycle en évitant de recommencer la visite d'une noeud déjà en cours de visite (c'est à dire recommencer la visite d'un noeud vert).

Autrement dit, si lors du parcours en largeur d'un graphe, un sommet en cours de visite a parmi ses voisins un sommet vert c'est qu'il y a un cycle.
Mais pour détecter cela il faut légèrement modifier l'algorithme de parcours : en effet, pour un sommet en cours de visite l'algorithme ne considère que ses voisins bleus (justement pour ne pas boucler) mais en procédant de la sorte on ne peut pas détecter s'il a un voisin vert...

Exercice 3

Ajoutez à la classe `GrapheNOPLS` une méthode `existeCycle(x)` qui teste l'existence d'un cycle dans le graphe en effectuant un parcours en largeur à partir d'un sommet donné x.

L'algorithme de Kruskal

Exercice 4

La classe `GrapheNOPLS` fournit maintenant toutes les méthodes nécessaires au codage de l'algorithme de Kruskal.
Écrivez **une fonction** `kruskal(G)` qui prend en entrée un `GrapheNOPLS` G et qui retourne un `GrapheNOPLS` qui est un arbre couvrant de poids minimum de G.

Exercice 5

Écrivez **une fonction** `aleaG(nbsom, nbar)` qui construit un graphe aléatoire formé de `nbsom` sommets et `nbar` arêtes, tel que toutes les arêtes ont des poids différents (pour être certain de l'unicité de l'arbre couvrant de poids minimum).

Exercice 6

Vérifier la construction d'un arbre couvrant :

- Construisez un graphe aléatoire G de 20 sommets et 50 arêtes
- Construisez A1, un arbre couvrant de G avec l'algorithme de Kruskal, affichez son poids et la liste de ses arêtes.
- Si vous avez déjà codé les fonctions `cocycle(G, L)`, `minCocyle(cc)` et `ACM(G)` du TP3, construisez A2, un arbre couvrant de poids minimum de G avec l'algorithme de PRIM, affichez son poids, affichez la liste de ses arêtes.

A1 et A2 doivent avoir le même poids et les mêmes arêtes (Si vous n'avez pas codé l'algorithme de PRIM du TP3 et bien ... tant pis, vous ne pouvez pas vérifier.)

A propos du tri d'une liste

La méthode `sort` des listes Phyton permet de trier une liste :

```
>>> l = [7, 3, 9, 1, 9]
>>> l.sort()
>>> print(l)
[1, 3, 7, 9, 9]

>>> l = ['chou', 'genou', 'caillou', 'hibou']
>>> l.sort()
>>> print(l)
['caillou', 'chou', 'genou', 'hibou']
```

Pour une liste dont les éléments ne sont pas des valeurs simples (par exemple des listes, tuples, des dictionnaires), on peut préciser quelle partie de la valeur prendre en compte pour le tri :

```
>>> l = [('Louis', 23), ('Ana', 18), ('Leo', 20), ('Marie', 14)]
>>> l.sort(key = lambda c : c[1])   # tri sur le 2e élément des couples
>>> print(ll)
[('Marie', 14), ('Ana', 18), ('Leo', 20), ('Louis', 23)]

>>> l=[{'a':12, 'b':22, 'c':45}, {'a':2, 'b':9,'c':18}, {'a':88, 'b':15, 'c':33}]
>>> l.sort(key = lambda d : d['b'])   # tri sur la valeur associée à la clé 'b'
>>> print(l)
[{'a': 2, 'b': 9, 'c': 18}, {'a': 88, 'b': 15, 'c': 33}, {'a': 12, 'b': 22, 'c': 45}]
```