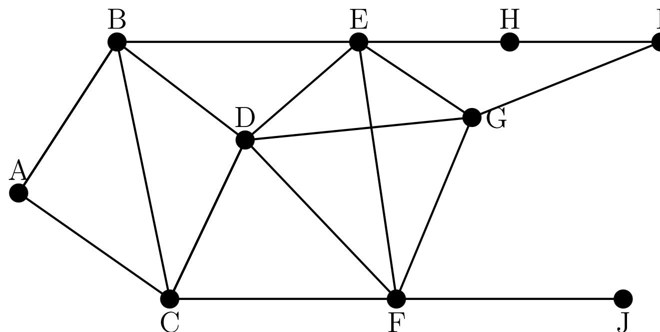
TP6 : PARCOURS DE GRAPHES

Introduction

Les graphes interviennent dans de nombreux problèmes, en particulier ils servent à modéliser les réseaux. Le problème fondateur de cette théorie est celui des pont de Königsberg, posé en 1736 par Euler. Parmi les autres problèmes connus, citons ceux de colorations d'un graphe, modélisant par exemple le problème suivant : combien au minimum faut-il de couleurs pour colorier une carte de sorte que deux pays frontalier n'aient pas la même couleur. Cette question a été résolue par un théorème célèbre, celui des quatre couleurs.
Nous nous intéresserons plus spécifiquement dans ce TP aux questions de parcours d'un graphe et au problème du plus cours chemin dans un graphe pondéré. Ces points ont été abordés dans votre cours de MPSI ; on ne fera que des rappels rapides sur ce cours.

1 Représentation

La façon la plus élémentaire de représenter un graphe non pondéré est une matrice d'adjacence : un tableau à double entrée indexé par les sommets (le sommet A est représenté par 0, B par 1 ...). Ce tableau sera implémenté par exemple par une liste de listes en python. On mettra 1 dans la case (i, j) si les sommets i et j sont reliés, et 0 sinon.
Si le graphe est pondéré, on met ce poids dans la case (i, j) et on prend une convention pour le cas où deux sommets ne sont pas reliés.
On peut aussi représenter un graphe (non pondéré) par une liste d'adjacence : une liste de listes `L` telle que `L[s]` soit la liste des sommets accessibles à partir du sommet `s` (les sommets sont alors représentés par des entiers) ; ou par un dictionnaire, les clefs étant les sommets et la valeur associée à chaque clef la liste des sommets accessible à partir de la clef (on peut ainsi représenter les sommets par leurs noms et non par des nombres).
On testera nos algorithmes sur l'exemple suivant (non pondéré dans un premier temps).



Dans un premier temps :

1. Écrire en python la matrice d'adjacence du graphe ci-dessus (on testera les fonctions suivantes sur cette matrice).
2. Écrire une fonction `matriceVersListe(M : [[int]] ) -> [[int]]` qui renvoie la liste d'adjacence associée au graphe représenté par la matrice d'adjacence `M`.
3. Écrire la fonction `listeVersMatrice(L : [[int]] ) -> [[int]]`.
4. Écrire une fonction `matriceVersDico(M : [[int]] ) -> {str:[str]}` qui renvoie la liste d'adjacence associée au graphe représenté par la matrice d'adjacence `M` sous la forme d'un dictionnaire (dont les clefs et les valeurs sont des lettres).
5. Écrire la fonction `degre(M : [[int]] , i : int ) -> int` qui renvoie le degré du sommet `i` dans le graphe représenté par la matrice d'adjacence `M`.

2 Parcours d'un graphe

2.1 Piles, files

Nous verrons que les deux types de parcours sont associés à l'usage d'une structure de pile ou de file. Nous utiliserons des implémentations naïves de ces structures par des listes python. Une bibliothèque spécifique contient un objet `deque` qui permet de les implémenter plus efficacement.
Pour une pile, l'action d'empiler une valeur `a` sur une pile (liste) `p` se fera par la commande `p.append(a)` ; le dépilage par la commande `p.pop()`.
Pour une file, l'action d'adjoindre une valeur `a` sur une file (liste) `f` se fera par la commande `f.append(a)` ; le fait d'enlever la valeur de tête par la commande `f.pop(0)`.

2.2 Parcours en profondeur

Le principe est le suivant :

- On crée une pile contenant le sommet de départ (pile contenant les prochains sommets à visiter).
- Tant que la pile est non vide, on marque (et on dépile) le sommet figurant en haut de celle-ci (sommet visité) et on empile tous ses voisins (ceux qui ne sont pas encore dans la pile).

Ce principe général peut donner lieu à des fonctions donnant des résultats divers : la composante connexe d'un sommet par exemple. Notre but étant de visualiser la différence entre les deux types de parcours, nous renverrons une liste (ordonnées) des sommets visités. Nous pourrons de plus renvoyer une liste contenant l'antécédent (père) de chaque sommet visité, ceci afin de pouvoir reconstruire un chemin.

1. Mettre en œuvre cet algorithme à la main sur le graphe correspondant à la matrice d'adjacence `M1` données dans le fichier `parcours_Acompleter.py`.
2. Compléter la fonction `parcours_prof(M,sommet)` dans le fichier `parcours_Acompleter.py`.
3. Écrire dans un second temps une fonction `chemin(M,sommet1,sommet2)` qui permet de récupérer le chemin entre deux sommets.

2.3 Parcours en largeur

Le principe est le suivant :

- On crée une file contenant le sommet de départ (pile contenant les prochains sommets à visiter).
- Tant que la file est non vide, on marque (et on sort de la file) le sommet figurant en première position dans celle-ci (sommet visité) et on adjoint à la file tous ses voisins (ceux qui ne sont pas encore dans la file).

1. Mettre en œuvre cet algorithme à la main sur le graphe correspondant à la matrice d'adjacence `M1` données dans le fichier `parcours_Acompleter.py`.
2. Écrire une fonction `parcours_larg(M,sommet)` sur le modèle de la précédente (on ne créera pas la liste des antécédents).
3. Le parcours en largeur permet d'obtenir la distance entre le sommet de départ et un sommet quelconque car on parcourt le graphe couche par couche. Adapter la fonction précédente en une fonction `parcours_larg_dist(M,sommet)` afin qu'elle renvoie une liste contenant la distance de chaque sommet au sommet de départ.

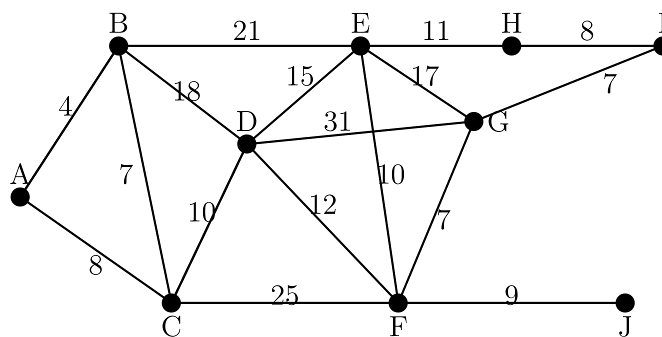
2.4 Versions récursives du parcours en profondeur

Le parcours en profondeur se prête facilement à une implémentation par une fonction récursive. L'idée est la suivante :

- La fonction récursive `parcours_prof_rec(M,sommet)` va s'appuyer sur une fonction auxiliaire `aux(sommet)` qui sera appelée de manière récursive.
- Les listes `sommets_visites` et `liste_peres` sont définies dans la fonction globale et actualisées à chaque appel récursif.
- Le processus des appels récursifs est le suivant : dès que la fonction `aux` est appelée sur un sommet, ce sommet est marqué et la fonction est appelée sur chacun de ses voisins non encore marqués.

3 Algorithme de Dijkstraa

On reprend le graphe du paragraphe précédent, avec des pondérations :



Supposons que celui-ci représente le temps de trajet entre des villes. On souhaite connaître le trajet le plus court entre les villes A et G. L'algorithme de Dijkstraa va nous permettre de répondre efficacement à cette question.
Avant tout on définit la distance $d(x, y)$ entre deux sommets x et y comme le minimum des poids des chemins joignant x à y. Remarquons que ce minimum est atteint parmi les chemins élémentaires.

3.1 Description de l'algorithme

Données :
Un graphe simple pondéré X fini, fortement connexe , poids ≥ 0, sans boucle. On note $p(x, y)$ le poids de l'arrête joignant x à y, avec la convention $p(x, y) = +\infty$ si il n'y a pas d'arrête joignant x à y.

Résultat attendu :
Étant donné un sommet A (départ) et un sommet Z (arrivée), trouver $d(A, Z)$ ainsi que le chemin qui réalise cette distance. En fait l'algorithme va trouver la distance entre A et n'importe quel autre sommet.
Principe :

- Variables :
  - Une liste S de sommets (ceux dont on connaît la distance à A)
  - Une liste S' de sommets (ceux qui, sans être dans S, ont un antécédent dans S)
  - Pour chaque sommet x, un réel $d_A(x)$ (appelé à être égal à $d(A, x)$)
  - Pour chaque élément $x \in S \cup S' \setminus \{A\}$, un élément de S : $\pi(x)$ (élément qui précède x sur le plus court chemin dont tous les éléments sont dans S (sauf éventuellement x) joignant A à x - on convient d'appeler un tel chemin un S-chemin).
- Initialisation :

- $S = \{A\}$
- $S' = \{x \in X,\ p(A,x) < +\infty\}$
- $d_A(A) = 0$, pour $x \in S'$, $d_A(x) = p(A,x)$, pour les autres $d_A(x) = +\infty$.
- Comme on a vu, $\pi(A)$ n'est pas défini ; pour $x \in S'$, $\pi(x) = A$.

- Progression :

  P1 Déterminer $x_0 \in X \setminus \{S\}$ tel que $d_A(x_0)$ est minimal (par conséquent $x_0 \in S'$)

  P2 On adjoint x_0 à S

  P3 Pour tous les $x \notin S$ tels que $p(x_0,x) < +\infty$, on adjoint x à S'. Pour chacun de ces x, on compare $d_A(x)$ et $d_A(x_0) + p(x_0,x)$ et si ce dernier est inférieur, on remplace.

- Condition d'arrêt : $S = X$

- Résultat : pour tout x, $d_A(x) = d(A,x)$, $\pi(x)$ le prédécesseur de x sur un plus court chemin de A à x.

Établir la trace de cet algorithme afin de déterminer le plus court chemin de A à J sur notre exemple. On présentera cela dans un tableau contenant à chaque étape, pour chaque sommet x, le couple $(d_A(x), \pi(x))$ dès qu'il est défini.

3.2 Implémentation

Un graphe pondéré d'ordre n est donné.
On suppose donné un tableau (liste de listes) `Table_poids` contenant les données relatives au graphe ayant servi d'exemple : en position `[i,j]` le poids de l'arrête reliant le sommet `i` au sommet `j` (`inf` si il n'y a pas d'arrête).
On souhaite calculer les distances d'un sommet fixé à tout autre sommet.
Pour cela, on va faire évoluer 3 listes :

- `listeS` , liste de 0 et de 1 signalant les sommets qui sont dans S ;

- `listeDA` , contenant pour chaque sommet x la valeur de $d_A(x)$ (remarquons que cette liste permet également de savoir si un élément est dans $S \cup S'$) ;

- `listeAnte` , liste donnant, pour chaque éléments de S ou S', son antécédant sur le plus cours chemin.

Compléter le programme fourni dans le fichier à compléter.
On complétera le programme également par des instructions permettant de récupérer le plus court chemin de A à J.