
TRAVAUX PRATIQUES n° 16

Graphes

Dans ce TP, on représente des graphes en Python (listes d'adjacence, matrices, dictionnaires), on convertit ces représentations, puis on calcule degrés et distances via un parcours en largeur.

On distinguera :

- ★ les graphes **orientés** (arcs) et **non orientés** (arêtes) ;
- ★ trois représentations : listes d'adjacence, matrice d'adjacence, dictionnaire d'adjacence.

I. Introduction

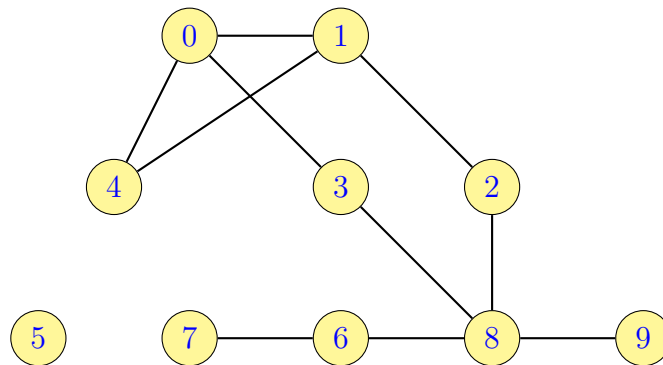
Exercice 1 (*Introduction*) :

- 1) Tracer, sur une feuille, le graphe **non orienté** dont la liste d'adjacence est

```
L = [[1], [0, 1, 3, 4], [1, 3], [2], [0, 3]]
```

(*Remarque* : le sommet 1 possède une boucle.)

- 2) Définir la liste d'adjacence et la matrice d'adjacence du graphe **non orienté** suivant :

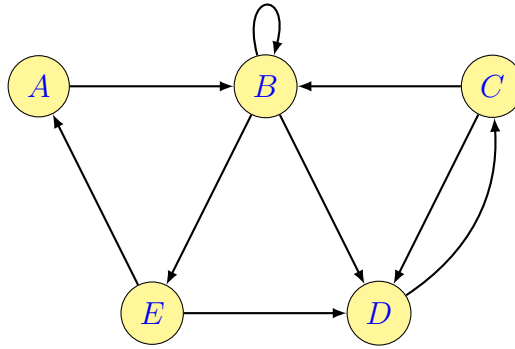


Comment récupérer, à partir de la matrice d'adjacence, le nombre de sommets ?

- 3) Tracer, sur une feuille, le graphe **orienté** dont la matrice d'adjacence est

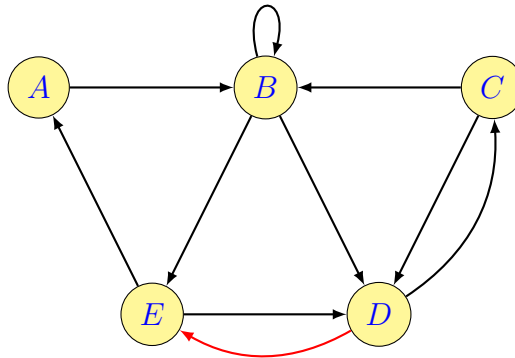
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

- 4) Définir le dictionnaire d'adjacence du graphe **orienté** suivant :



Comment récupérer, à partir du dictionnaire d'adjacence, la liste des voisins (successeurs) de B ? la liste des sommets du graphe ?

Modifier le dictionnaire d'adjacence (sans le réécrire en entier) pour ajouter un arc de D vers E :



Exercice 2 (*Nombre d'arcs*) : Écrire une fonction `nombre_d_arcs(M)` qui prend en argument la matrice d'adjacence `M` d'un graphe orienté et qui renvoie le nombre d'arcs du graphe.

Exercice 3 (*Modes de représentation*) : Dans cet exercice, on suppose que les sommets des graphes sont toujours numérotés par des entiers (en partant de zéro).

- 1) Écrire une fonction `liste_vers_matrice(L)` qui prend en argument la liste des listes d'adjacence d'un graphe orienté et qui renvoie la matrice d'adjacence du graphe.
- 2) Écrire une fonction `matrice_vers_liste(M)` qui prend en argument la matrice d'adjacence d'un graphe orienté et qui renvoie la liste des listes d'adjacence du graphe.

Exercice 4 (*Degré, successeurs et prédécesseurs*) : Dans un graphe orienté, le **degré sortant** d'un sommet `s` est le nombre d'arcs qui partent de `s`. De même, le **degré entrant** d'un sommet `s` est le nombre d'arcs qui arrivent en `s`.

- 1) Écrire une fonction qui prend en arguments un graphe orienté et un sommet `s` du graphe, et qui renvoie le degré sortant du sommet `s` :
 - a) `degre_sortant_liste(L, s)` dans le cas où le graphe est donné par listes d'adjacence ;
 - b) `degre_sortant_matrice(M, s)` dans le cas où le graphe est décrit par sa matrice d'adjacence.
- 2) Écrire une fonction `successeurs(M, s)` qui prend en arguments la matrice d'adjacence `M` d'un graphe orienté et un sommet `s`, et qui renvoie la liste des successeurs (voisins) du sommet `s`.
- 3) Écrire une fonction `degre_entrant(M, s)` qui prend en arguments la matrice d'adjacence `M` d'un graphe orienté et un sommet `s`, et qui renvoie le degré entrant du sommet `s`.
- 4) Écrire une fonction `predcesseurs(M, s)` qui prend en arguments la matrice d'adjacence `M` d'un graphe orienté et un sommet `s`, et qui renvoie la liste des prédécesseurs du sommet `s`, c'est-à-dire la liste des sommets dont `s` est un successeur.

Exercice 5 (*Parcours en largeur*) : Soit $\mathcal{G}$ un graphe orienté et soient s et t deux sommets du graphe $\mathcal{G}$.

- ★ S'il existe un chemin allant du sommet s vers le sommet t , la longueur du plus court chemin allant de s à t est appelée la distance du sommet s au sommet t .
- ★ Sinon, on dit que la distance du sommet s au sommet t est infinie.

Algorithme de parcours en largeur :

- ★ Données : M matrice d'adjacence et s un sommet du graphe
 - ★ Initialiser deux listes :
 - une liste `a_visiter` qui ne contient que le sommet s
 - une liste `distances`, de taille égale au nombre de sommets du graphe et telle que pour tout sommet u : `distances[u] = np.inf` si $u \neq s$ et `distances[s] = 0`
 - ★ Tant que la liste `a_visiter` n'est pas vide :
 - on enlève le premier élément de la liste `a_visiter` : `sommet = a_visiter.pop(0)`
 - on détermine la liste des voisins de ce sommet : `liste_voisins = successeurs(M, sommet)` (fonction de l'exercice 4)
 - pour chacun de ses voisins v , si v n'est pas déjà dans la liste `a_visiter` et si `distances[v] = np.inf` :
 - ▶ on ajoute v à la fin de la liste `a_visiter`
 - ▶ on fixe sa distance au sommet s : `distances[v] = distances[sommet] + 1`
 - ★ Renvoyer la liste `distances`
- 1) Écrire la fonction `parcours_en_largeur(M, s)` et la tester sur les matrices définies dans l'exercice 1.
 - 2) Écrire une fonction `accessibles(M, s)` qui prend en arguments un graphe donné par sa matrice d'adjacence et un sommet s , et qui renvoie la liste de tous les sommets accessibles par un chemin depuis s .
 - 3) Écrire une fonction `existe_chemin(M, s, t)` qui prend en arguments un graphe donné par sa matrice d'adjacence et deux sommets s et t , et qui détermine s'il existe un chemin de s à t .
 - 4) Écrire une fonction `est_connexe(M)` qui prend en argument la matrice d'adjacence d'un graphe **non orienté** (matrice symétrique) et qui détermine si ce graphe est connexe. On pourra vérifier que tous les sommets sont accessibles depuis le sommet 0.