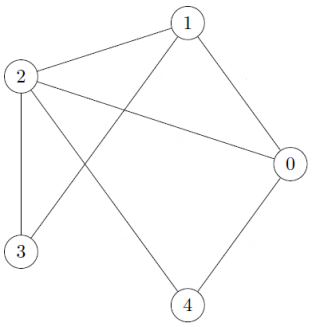
TP 12 – Introduction aux graphes.

Consignes

- Créer un dossier TP12_graphe où sauvegarder votre fichier python que vous nommerez TP12_graphe ;
- Chaque fonction sera implémentée avec sa signature ;
- Tester chacune des fonctions et copier le test dans le script ;
- Commenter vos fonctions.

Exercice 1 - Implémentation des graphes par une liste d'adjacence

On considère le graphe `G` suivant,



Pour implémenter le graphe, on utilise une liste `G` qui a pour taille le nombre de sommets. Chaque élément `G[i]` est la liste des voisins de `i`.
Dans ce cas, `G[0]=[1,2,4]` car Les sommets 1, 2 et 4 sont des voisins de 0.

Question 1 *Construire la liste d'adjacence `G` en utilisant la méthode énoncée ci-dessus.*

Question 2 *Écrire une fonction `voisins_l(G:list, i:int) -> list`, d'argument la liste d'adjacence `G` et un sommet `i`, renvoyant la liste des voisins du sommet `i`.*

Question 3 *Écrire une fonction `aretes_l(G:list) -> list`, renvoyant la liste des arêtes. Les arêtes seront constitués de couples de sommets (l'arête entre les sommets 0 et 1 sera donnée par `(0,1)`). Tester votre fonction pour le graphe proposé.*

Les instructions suivantes permettent de tracer un graphe.

```
import networkx as nx
import matplotlib.pyplot as plt

def plot_graphe_l(G):
    Gx = nx.Graph()
    edges = aretes_l(G)
    Gx.add_edges_from(edges)
    nx.draw(Gx,with_labels = True)
    plt.show()
```

Question 4 *Écrire et tester la fonction `plot_graphe_l(G)`.*

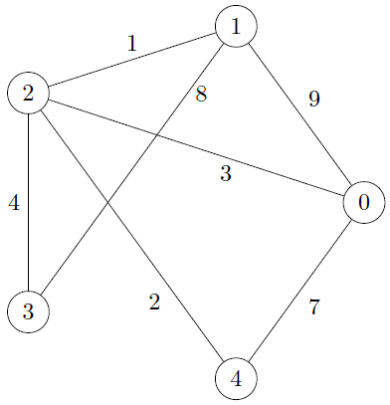
Question 5 *Écrire une fonction* `degre_l(G:list, i:int) -> int`*, d'argument un sommet* i*, renvoyant le nombre des voisins du sommet* i*, c'est-à-dire le nombre d'arêtes issues de* i*.*

Question 6 *Écrire la fonction* `ajout_sommet_l(G:list, L:list) -> None` *permettant d'ajouter un sommet au graphe.* `L` *désigne la liste des sommets auxquels le nouveau sommet est relié.* `ajout_sommet_l` *agit avec effet de bord sur* `G`*. Tester votre fonction et tracer le nouveau graphe.*

Question 7 *Écrire la fonction* `supprime_sommet_l(G:list, i: int) -> None` *permettant de supprimer le sommet* i *du graphe. Tester votre fonction avec le sommet* 1 *et tracer le nouveau graphe. (Le sommet 1 est supprimé sur le graphe et la numérotation des autres sommets reste inchangée.) La méthode de liste* `remove()` *peut être avantageusement utilisée.*

Exercice 2 - Implémentation par une matrice d'adjacence d'un graphe pondéré

On considère le graphe `G` suivant, où le nombre situé sur l'arête joignant deux sommets est leur distance, supposée entière.



Question 8 *Construire la matrice* $(M_{ij})_{0\leq i,j\leq 4}$*, matrice de distances du graphe* `G`*, définie par : « pour tous les indices* i*,* j*,* M_{ij} *représente la distance entre les sommets* i *et* j*, ou encore la longueur de l'arête reliant les sommets* i *et* j *». Cette matrice sera implémentée sous forme d'une liste de listes. Chaque « sous-liste » représentant une ligne de la matrice d'adjacence.*
On convient que, lorsque les sommets ne sont pas reliés, cette distance vaut -1*. La distance du sommet* i *à lui-même est égale à 0.*

Question 9 *Écrire une fonction* `voisins(M:list, i:int) -> list`*, d'arguments la matrice d'adjacence* `M` *et un sommet* `i`*, renvoyant la liste des voisins du sommet* `i`*.*

Question 10 *Écrire une fonction* `aretes(M:list) -> list`*, renvoyant la liste des arêtes du graphe. Les arêtes seront constitués de couples de sommets (l'arête entre les sommets 0 et 1 sera donnée par* `(0,1)`*).*

A partir des instructions vues précédemment permettant de tracer un graphe,
Question 11 *Écrire et tester la fonction* `plot_graphe(M)`*. Les pondérations n'apparaissent pas sur le tracé du graphe.*

Question 12 *Écrire une fonction* `degre(M:list, i:int) -> int`*, d'arguments la matrice d'adjacence* `M` *du graphe et un sommet* `i`*, renvoyant le nombre des voisins du sommet* `i`*, c'est-à-dire le nombre d'arêtes issues de* `i`*.*

Question 13 *Écrire une fonction* `longueur(M:list,L:list) -> int`*, d'arguments la matrice d'adjacence* `M` *du graphe et une liste* `L` *de sommets de* `G` *à parcourir dans le sens proposé par* `L`*, renvoyant la longueur du trajet décrit par* `L`*, c'est-à-dire la somme des longueurs des arêtes empruntées. Si le trajet n'est pas possible, la*

fonction renverra -1.

Question 14 *Écrire la fonction* `ajout_sommet(M:list, L:list, poids:list) -> None` *permettant d'ajouter un sommet au graphe.* `L` *désigne la liste des sommets auxquels le nouveau sommet est relié,* `poids` *la liste des poids respectifs.* `ajout_sommet` *agit avec effet de bord sur* `M`.

Question 15 *Tester la fonction* `ajout_sommet(M:list, L:list, poids:list) -> None` *avec un sommet* 5 *voisin de* `L=[1,4]` *et poids=*`[6,5]`. *Tracer le nouveau graphe.*

Question 16 *Écrire la fonction* `supprime_sommet(M:list, i:int) -> None` *permettant de supprimer le sommet* `i` *du graphe. Tester votre fonction pour le sommet* 0 *et tracer le nouveau graphe.*

Question 17 *Écrire la fonction* `from_list_to_matrix(G:list) -> list` *permettant de convertir un graphe non pondéré implémenté sous forme de liste d'adjacence en matrice d'adjacence contenant des 0 et des 1.*

Question 18 *Écrire la fonction* `from_matrix_to_listmatrix(M:list) -> list` *permettant de convertir un graphe non pondéré implémenté sous forme de matrice d'adjacence en liste d'adjacence.*

Exercice 3 - Déplacement d'un cavalier sur un échiquier

Un cavalier se déplace, lorsque c'est possible, de 2 cases dans une direction verticale ou horizontale, et de 1 case dans l'autre direction (le trajet dessine une figure en L).

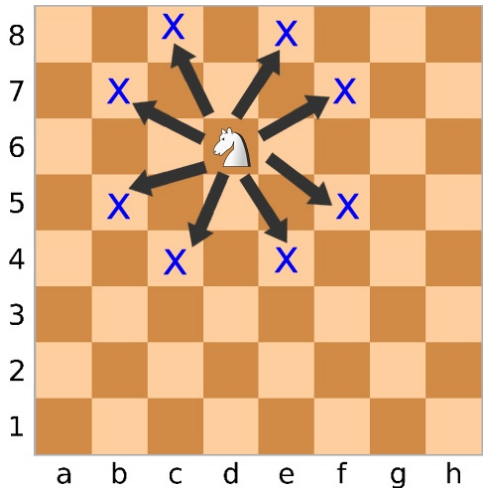


FIGURE 1 – Illustration du mouvement d'un cavalier sur un échiquier

Dans un premier temps, les cases de l'échiquier sont représentées par des tuples : le couple `(i,j)` désigne la case d'abscisse `i` et d'ordonnée `j`. Un échiquier possède 8 colonnes et 8 lignes, donc `i` et `j` seront compris entre `0` et `7`.

Question 19 *Écrire une fonction* `estDansEch(i:int, j:int)->bool` *qui renvoie* `True` *si* `(i, j)` *correspond à une case valide de l'échiquier et* `False` *sinon.*

Question 20 *Écrire une fonction* `mvtsPossibles(i:int, j:int)->list` *qui renvoie la liste des cases où le cavalier peut se déplacer à partir de la case* `(i,j)` *à l'ordre près.*

Question 21 *Vérifier que :*

— `mvtsPossibles(0,0)` renvoie `[(1, 2), (2, 1)]`,

— `mvtsPossibles(3,5)` renvoie bien `[(1, 4), (1, 6), (2, 3), (2, 7), (4, 3), (4, 7), (5, 4), (5, 6)]`,

— `mvtsPossibles(7,7)` renvoie bien `[(5, 6), (6, 5)]`.

Tous ces résultats sont à l'ordre près.

Question 22 *Créer un graphe `G` sous la forme d'un dictionnaire d'adjacence avec pour sommets les différentes cases de l'échiquier et les arêtes qui correspondent à un mouvement possible du cavalier.*

Question 23 *Vérifiez que vous avez un graphe avec 64 sommets et 168 arêtes.*

Le codage des cases d'échiquier se fait classiquement par une chaine de caractères comprenant : une lettre minuscule pour l'abscisse (de `a` à `h`) et un chiffre pour l'ordonnée (de `1` à `8`). La case en bas à gauche de l'échiquier est donc de code `'a1'`.

Remarque

`ord(c)` retourne le codage Unicode correspondant au caractère `c` et `chr(n)` renvoie le caractère dont le codage Unicode est `n`.

Question 24 *Écrire une fonction `codage(i:int, j:int)->str` qui renvoie le code correspondant à la case `(i, j)`.*

Question 25 *Créer un dictionnaire d'adjacence `G2` comme défini précédemment avec les cases maintenant nommées d'après leur code.*

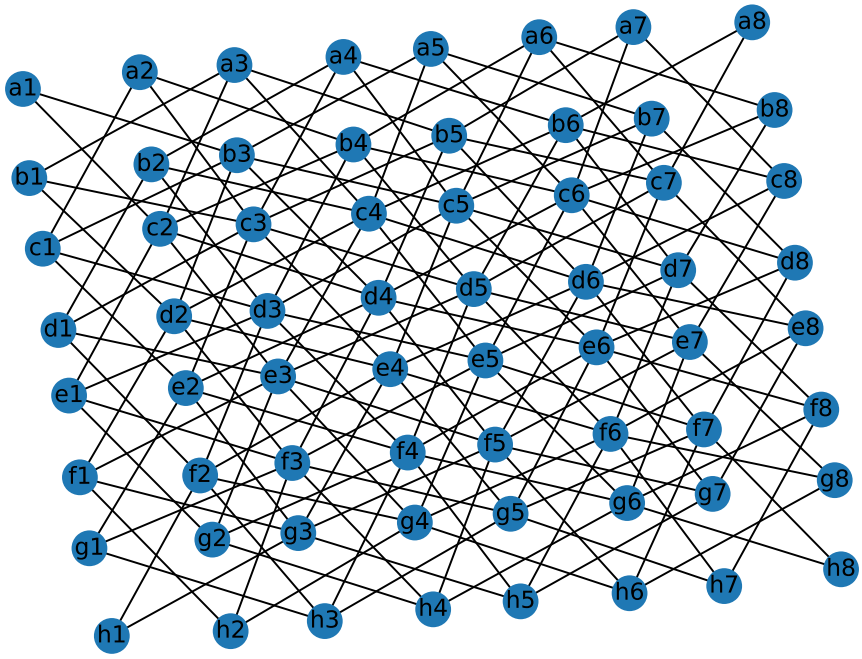


FIGURE 2 – Représentation graphique du graphe `G2`

Question 26 *Vérifier que :*

— *`G2['a1']` renvoie `['b3', 'c2']`,*

— *`G2['d6']` renvoie bien `['b5', 'b7', 'c4', 'c8', 'e4', 'e8', 'f5', 'f7']`,*

— *`G2['h8']` renvoie bien `['f7', 'g6']`.*

Tous ces résultats sont à l'ordre près.