Graphes et représentations

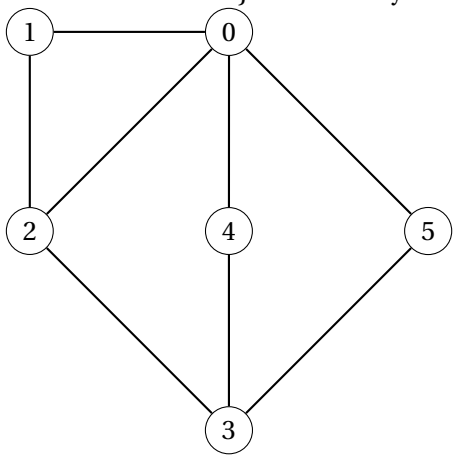
INFORMATIQUE COMMUNE - TP n° 2.3 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ représenter un graphe en machine par une liste d'adjacence ou une matrice d'adjacence
- ☞ transformer une représentation en une autre
- ☞ calculer les degrés des sommets d'un graphe
- ☞ transposer un graphe
- ☞ caractériser un graphe d'après sa séquence de degrés ou son spectre

A Représentation et transformation d'un graphe

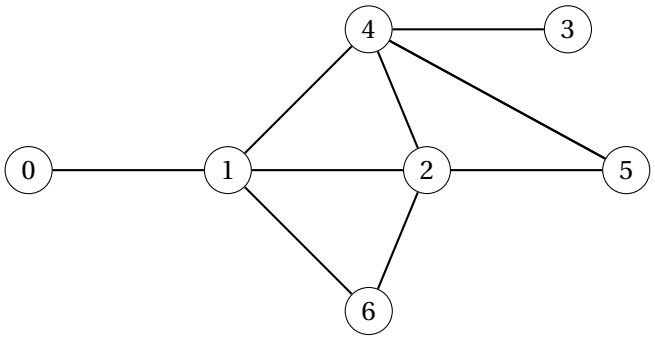
A1. Créer une liste d'adjacence en Python qui représente le graphe suivant :



Solution :

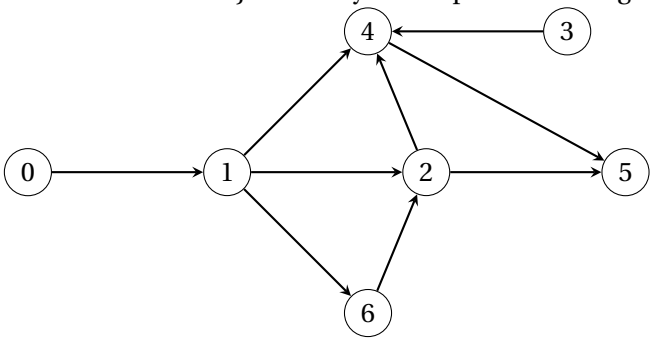
```
ga1 = [[1, 2, 4, 5], [0, 2], [0, 1, 3], [2, 4, 5], [0, 3], [0, 3]]
```

A2. Dessiner le graphe correspondant à la liste d'adjacence :
`[[1],[0,2,4,6],[1,4,5,6],[4],[1,2,3,5],[2,4],[1,2]]`



Solution :

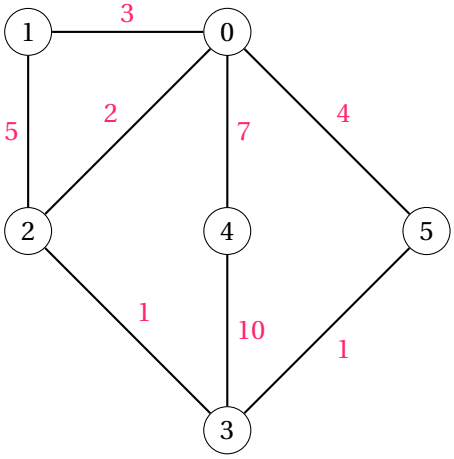
A3. Créer une liste d'adjacence Python représentant le graphe orienté suivant :



Solution :

```
go = [[1], [2, 4, 6], [4, 5], [4], [5], [], [2]]
```

A4. Créer une liste d'adjacence Python représentant le graphe pondéré suivant :



Solution :

```
gp = [[(1,3), (2,2), (4,7), (5,4)], [(0,3), (2,5)], [(0,2), (1,5), (3,1)],
    [(2,1), (4,10), (5,1)], [(0,7), (3,10)], [(0,4), (3,1)]]
```

A5. Modéliser les graphes des questions précédentes en utilisant le concept de matrice d'adjacence.

Solution :

```
import numpy as np

ma1 = np.array([[0,1,1,0,1,1],[1,0,1,0,0,0],[1,1,0,1,0,0],
                [0,0,1,0,1,1],[1,0,0,1,0,0],[1,0,0,1,0,0]])
print(ma1)
# [[0 1 1 0 1 1]
#  [1 0 1 0 0 0]
#  [1 1 0 1 0 0]
#  [0 0 1 0 1 1]
#  [1 0 0 1 0 0]
#  [1 0 0 1 0 0]]
ma2 = np.array
    ([[0,1,0,0,0,0,0],[1,0,1,0,1,0,1],[0,1,0,0,1,1,1],[0,0,0,0,1,0,0],
                [0,1,1,1,0,1,0],[0,0,1,0,1,0,0],[0,1,1,0,0,0,0]])
print(ma2)
# [[0 1 0 0 0 0 0]
#  [1 0 1 0 1 0 1]
#  [0 1 0 0 1 1 1]
#  [0 0 0 0 1 0 0]
#  [0 1 1 1 0 1 0]
#  [0 0 1 0 1 0 0]
#  [0 1 1 0 0 0 0]]

mo = np.array
    ([[0,1,0,0,0,0,0],[0,0,1,0,1,0,1],[0,0,0,0,1,1,0],[0,0,0,0,1,0,0],
                [0,0,0,0,0,1,0],[0,0,0,0,0,0,0],[0,0,1,0,0,0,0]])
print(mo)
# [[0 1 0 0 0 0 0]
#  [0 0 1 0 1 0 1]
#  [0 0 0 0 1 1 0]
#  [0 0 0 0 1 0 0]
#  [0 0 0 0 0 1 0]
#  [0 0 0 0 0 0 0]
#  [0 0 1 0 0 0 0]]
```

A6. Écrire une fonction de prototype `ladj_to_madj(g)` qui prend comme paramètre un graphe (non pondéré) sous la forme d'une liste d'adjacence et renvoie le même graphe sous la forme d'une matrice d'adjacence. Le type retourné est une liste de listes Python.

Solution :

```
def ladj_to_madj(g):
    n = len(g)
    m = [[0 for _ in range(n)] for _ in range(n)]
    # on peut créer m avec une double boucle !
    for i in range(n):
        for v in g[i]:
            m[i][v] = 1
    return m
```

A7. Écrire une fonction de prototype `ladj_to_madj_n(g)` qui prend comme paramètre un graphe (non pondéré) sous la forme d'une liste d'adjacence et renvoie le même graphe sous la forme d'une matrice d'adjacence. Le type retourné est un tableau Numpy.

Solution :

```
def ladj_to_madj_n(g):
    n = len(g)
    m = np.zeros((n,n))
    for i in range(n):
        for v in g[i]:
            m[i,v] = 1
    return m
```

A8. Écrire une fonction de prototype `madj_to_ladj(g)-> list[list[int]]` qui prend comme paramètre un graphe (non pondéré) sous la forme d'une matrice d'adjacence et renvoie le même graphe sous la forme d'une liste d'adjacence. La matrice d'entrée pourra indifféremment être donnée sous la forme d'un tableau Numpy ou d'une liste de liste. La matrice de sortie sera une liste de listes.

Solution :

```
# version qui marche pour un tab numpy et une liste Python
def madj_to_ladj(g):
    n = len(g)
    L = [[] for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if g[i][j] != 0:
                L[i].append(j)
    return L
```

A9. Écrire une fonction de prototype `transpose_m(g)` qui prend en paramètre un graphe orienté sous la forme d'une matrice d'adjacence et qui renvoie le graphe transposé correspondant, c'est-à-dire le graphe dont les arcs sont dirigés dans le sens opposé. Quelle est la complexité de cette fonction ?

Solution : La complexité de cette fonction est en $O(n^2)$, si n est l'ordre du graphe.

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} c = O(n^2)$$

```
def transpose_m(g):
    n = len(g)
    tg = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(n):
            tg[j][i] = g[i][j]
    return tg
```

A10. Écrire une fonction de prototype `transpose(g)` qui prend en paramètre un graphe orienté sous la forme d'une liste d'adjacence et qui renvoie le graphe transposé correspondant, c'est-à-dire le graphe dont les arcs sont dirigés dans le sens opposé. Quelle est la complexité de cette fonction ?

Solution :

```
def transpose(g):
    n = len(g)
    tg = [[] for _ in range(n)]
    for i in range(n):
        voisins = g[i]
        for v in voisins:
            tg[v].append(i)
    return tg
```

La complexité de cette fonction est en $O(n+m)$, si n est l'ordre du graphe et m le nombre d'arc du graphe. La boucle extérieure effectue autant d'itération que de sommets dans le graphe (n). La boucle intérieure effectue autant d'itérations que d'arêtes incidente au sommet i. De plus, le reste des opérations est de complexité constante. Ceci s'écrit :

$$C(n) = \sum_{i=0}^{n-1}\left(c + \sum_{v\in g[i]} c\right) = \sum_{i=0}^{n-1} c + \sum_{i=0}^{n-1}\sum_{v\in g[i]} c = nc + \sum_{a\in A} c = nc + mc = O(n+m)$$

où A est l'ensemble des arcs du graphe $G = (S, A)$ de cardinal $|A| = m$. Dans la deuxième double somme, (i, v) est un arc du graphe G. Cela signifie qu'on parcourt tous les sommets.

A11. Écrire une fonction de prototype `degrees(g)` qui renvoie la liste des degrés des sommets d'un graphe non orienté. Le paramètre est donné sous la forme d'une liste d'adjacence. Par exemple pour le graphe de la première question, la fonction renvoie : `[4, 2, 3, 3, 2, 2]`.

Solution :

```
def degrees(g):
    return [len(L) for L in g]
```

A12. Écrire une fonction de prototype `in_degrees(g)` qui renvoie la liste des degrés entrants des sommets d'un graphe orienté. Le paramètre est donné sous la forme d'une liste d'adjacence. Par exemple, pour le graphe orienté de la question 3, la fonction renvoie `[0, 1, 2, 0, 3, 2, 1]`.

Solution :

```
def in_degrees(g):
    return [len(L) for L in transpose(g)]
```

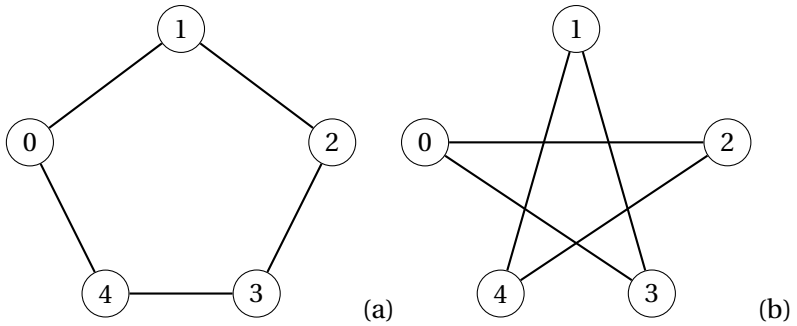


FIGURE 1 – Deux exemples de graphe : le pentagone (a), l'étoile (b)

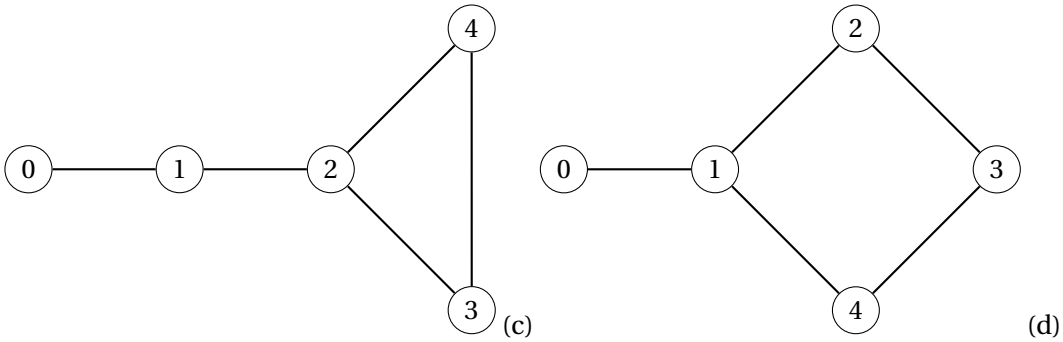


FIGURE 2 – Deux exemples de graphe : queue de poisson(c), cerf-volant (d)

B Deux graphes sont-ils isomorphes ?

Cette section s'attache à déterminer si deux graphes sont isomorphes. Deux approches successives sont proposées :

1. utiliser la séquence des degrés,
2. utiliser le spectre de la matrice d'adjacence.

B13. Donner les listes d'adjacence des deux graphes de la figure 1. Sont-ils isomorphes ?

Solution :

```
# Graphe 1: cycle à 5 sommets
pentagone = [[4,1],[0,2],[1,3],[2,4],[3,0]]
# Graphe 2: étoile à 5 sommets
etoile = [[2,3],[3,4],[4,0],[1,0],[1,2]]
```

Ces deux graphes sont isomorphes.

B14. Écrire une fonction de signature `seq_degres(g: list[list[int]])-> list[int]` qui renvoie la séquence des degrés d'un graphe triée dans l'ordre croissant.

Solution :

```
def seq_degres(g: list[list[int]]) -> list[int]:
    S = []
    for voisins in g:
        S.append(len(voisins))
    return sorted(S)
```

B15. En déduire une fonction `isomorphique_par_seq_deg` qui teste si deux graphes sont isomorphes d'après leur séquence de degrés.

Solution :

```
def isomorphique_par_seq_deg(g1: list[list[int]], g2: list[list[int]]) ->
    bool:
    S1 = seq_degres(g1)
    S2 = seq_degres(g2)
    assert len(S1) == len(S2)
    n = len(S1)
    for i in range(n):
        if S1[i] != S[j]:
        return False
    return True
```

B16. Appliquer cette fonction aux deux graphes de la figure 1 et 2. Que constatez-vous ?

Solution : Si on considère la suite des degrés, celle-ci est identique pour (a) et (b) et ils sont effectivement isomorphes. Dans le cas de la queue de poisson et du cerf volant, on pourrait en conclure que les deux graphes sont isomorphes alors que ce n'est pas le cas.

B17. Donner les matrices d'adjacence des deux graphes de la figure 1.

Solution :

```
[[0, 1, 0, 0, 1], # Pentagone
 [1, 0, 1, 0, 0],
 [0, 1, 0, 1, 0],
 [0, 0, 1, 0, 1],
 [1, 0, 0, 1, 0]]

[[0, 0, 1, 1, 0],  # Étoile
 [0, 0, 0, 1, 1],
 [1, 0, 0, 0, 1],
 [1, 1, 0, 0, 0],
 [0, 1, 1, 0, 0]]
```

B18. En utlisant la bibliothèque Numpy et notamment la fonction `eigvalsh` du module `numpy.linalg`, écrire une fonction de signature `spectre(matrice)` qui renvoie le spectre trié dans l'ordre croissant des valeurs propres de la matrice d'adjacence d'un graphe.

Solution :

```
import numpy as np
def spectre(matrice):
    return sorted(np.linalg.eigvals(matrice))
```

B19. En déduire une fonction `isomorphique_par_spectre(m1, m2)` qui teste si deux graphes représentés sous forme de matrice d'adjacence sont isomorphes d'après leur spectre. On pourra utiliser la fonction `allclose` pour comparer deux vecteurs de flottants.

Solution :

```
def isomorphique_par_spectre(m1, m2):
    S1 = spectre(m1)
    S2 = spectre(m2)
    return np.allclose(S1,S2)
```

B20. Appliquer cette fonction aux deux graphes (c) et (d) de la figure 2. Que constatez-vous ?

Solution : D'après leur spectre, les deux graphes ne sont pas isomorphes alors qu'ils le sont d'après la séquence de leurs degrés !

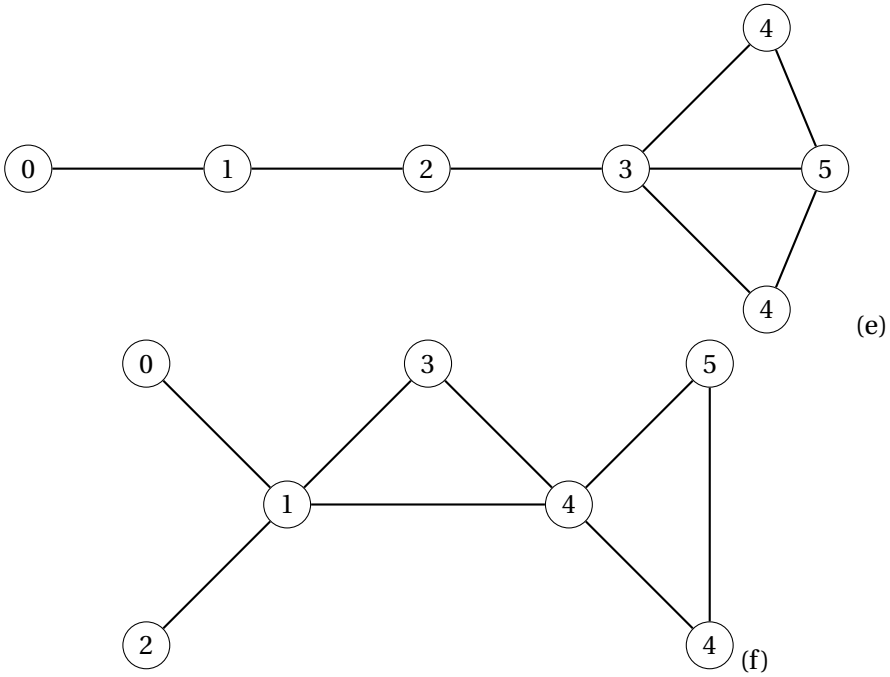


FIGURE 3 – Deux exemples de graphe : grand cerf-volant(e), passe-partout (f)

B21. Tester si les deux graphes de la figure 3 sont isomorphes d'après leur degré et/ou d'après leur spectre. Que pouvez-vous en conclure ?

Solution : Le passe-partour et le grand cerf-volant ne sont pas isomorphes d'après leur degré mais le sont d'après leur spectre ! Il fautdrait donc trouver un autre moyen de caracériser les isomorphismes de graphe !