Corrigé – Les graphes

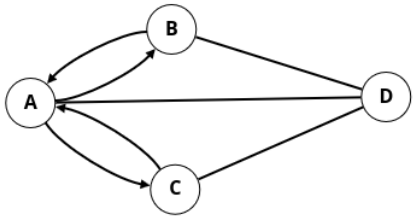
Exercice 1

1. Voisins du sommet A : C, F et B. Voisins du sommet B : A, F, G et H.
2. Un n-uplet (`tuple` en Python) permettrait de représenter les arêtes.
3. (a) min : 3 pour A, C et E. max : 5 pour F.

   (b) Degré moyen de ce graphe : $(3+4+3+4+3+5+4+4)/8 = 3,75$
4. Nombre de chemins de longueur 3 reliant le sommet A au sommet E : 3 (ABHE, AFDE, AFCE).

Exercice 2

1.

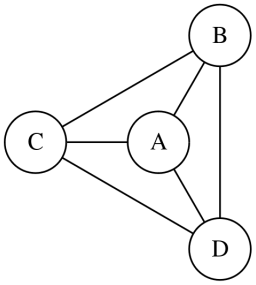


2. « Une telle promenade n'existe pas, et c'est Euler qui donna la solution de ce problème en caractérisant les graphes que l'on appelle aujourd'hui « eulériens » en référence à l'illustre mathématicien, à l'aide d'un théorème dont la démonstration rigoureuse ne fut en fait publiée qu'en 1873, par Carl Hierholzer.

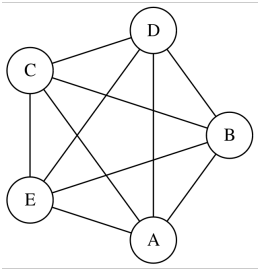
   Ce problème n'a sous cette forme non généralisée qu'un intérêt historique, car pour ce cas, il est assez intuitif de démontrer que la promenade demandée n'existe pas. Sans aller jusqu'à utiliser la théorie des graphes il suffit pour cela de remarquer que pour parcourir tous les ponts une fois et une seule, il faudra entrer et sortir à nouveau de chaque île (ou rive). Ainsi sur chaque île durant le parcours entier, il y aura autant de ponts d'entrée que de ponts de sortie. Chaque île (ou rive) doit donc avoir un nombre pair de ponts qui la relient aux autres îles ou rives. Comme ce n'est manifestement pas le cas, la promenade demandée n'existe pas. » *in* https://fr.wikipedia.org/wiki/Problème_des_sept_ponts_de_Königsberg

Exercice 3

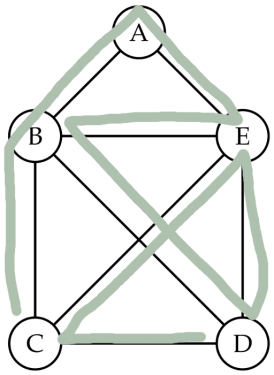
1. Le graphe complet à 4 sommets est planaire :



2. Le graphe complet à 5 sommets n'est pas planaire :



Exercice 4



Exercice 5

On peut représenter par des graphes un réseau routier, un réseau informatique, un réseau social, la navigation web, une molécule, un labyrinthe, etc.

Exercice 6

```
from modules.gmanager import display, display_from_adjacency_list

liste_adjacence = {"A" : ["B", "E", "F", "G", "H"],
                   "B" : ["A", "E", "F", "G", "C", "D"],
                   "C" : ["B", "D", "G", "H"],
                   "D" : ["B", "C", "H"],
                   "E" : ["A", "B", "F"],
                   "F" : ["A", "B", "E", "G"],
                   "G" : ["A", "B", "C", "F", "H"],
                   "H" : ["A", "C", "D", "G"]}

display_from_adjacency_list(liste_adjacence)
```

Exercice 7 @TODO !

```
pass
```

Exercice 8

1.

$$\begin{bmatrix} & A & B & C & D & E & F & G & H \\ A & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ B & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ C & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ D & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ F & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ G & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ H & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

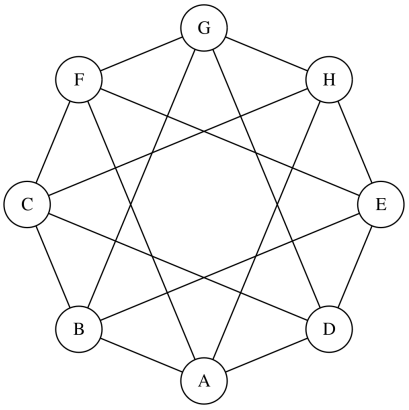
2. Pour calculer le degré d'un sommet à l'aide de la matrice de l'adjacence, on compte le nombre de 1 par ligne ou par colonne.

Exercice 9

La représentation de graphe par liste d'adjacence est plus adaptée à un graphe peu dense, c'est-à- dire avec peu de lien avec d'autres sommets (un réseau routier par exemple).

La représentation avec une matrice sera plus adaptée à un graphe dense car il y a énormément de liens entre les sommets (donc beaucoup de 1 dans la matrice).

Exercice 10



Exercice 11

La matrice d'adjacence d'un graphe complet à n sommets ne comporte que des 1 sauf la diagonale qui aura seulement des zéros.

Exemple avec 4 sommets :

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

Exercice 12 @TODO !

```
1
```

Exercice 13

1.
```python
def get_adjacency_matrix(adj_list):
    # tableau des étiquettes triées
    labels = [label for label in adj_list.keys()]
    labels.sort()

    # initialisation de la matrice
    n = len(labels)
    matrix = [[0 for _ in range(n)] for _ in range(n)]

    # remplissage
    for i in range(n):
        label = labels[i]
        neighbors = adj_list[label]
        for j in range(n):
            if labels[j] in neighbors:
                matrix[i][j] = 1

    return matrix
```

2.
```python
def get_adjacency_list(adj_matrix):
    # tableau des étiquettes des sommets
    n = len(adj_matrix)
    labels = [chr(65 + i) for i in range(n)]

    # initialisation de la liste (un dictionnaire)
    dico = {}

    # remplissage
    for i in range(n):
        dico[labels[i]] = []
        neighbours = adj_matrix[i]
        print(neighbours)
        for j in range(len(neighbours)):
            if neighbours[j] == 1:
                dico[labels[i]].append(labels[j])

    return dico
```

(TSVP)

Exercice 14

1. (a) A, B, A, B, A, *etc.* sans jamais s'arrêter.

   (b) Ordre de visite des sommets lors du parcours en profondeur d'abord, en partant :

   - du sommet A : A, B, G, C, D, I, E, J, H, F
   - du sommet E : E, I, D, C, G, A, B, F, H, J
   - du sommet H : H, G, A, B, F, C, D, I, E, J

2. Ordre de visite des sommets lors du parcours en profondeur d'abord, en partant :

   - du sommet A : A, G, H, I, J, E, D, C, F, B
   - du sommet E : E, J, I, H, G, F, A, B, D, C
   - du sommet H : H, I, J, E, D, G, F, A, B, C

Exercice 15

```
class GraphAL:
    # ...
    def parcours_prof(self, sommet, deja_vus=[]):
        if sommet in deja_vus:
            return []

        voisins = self.adj_lst[sommet]
        deja_vus.append(sommet)
        lst_parcours = [sommet]

        for voisin in voisins:
            lst_parcours += self.parcours_prof(voisin, deja_vus)

        return lst_parcours
```

Exercice 16

1. Ordre de dépilage des sommets lors du parcours en profondeur d'abord itératif, en partant :

   - du sommet A : A, G, H, I, J, E, D, C, F, B
   - du sommet E : E, J, I, H, G, F, C, B, A, D
   - du sommet H : H, I, J, E, D, C, G, F, B, A

(TSVP)

2.
```python
class GraphAL:
    # ...
    def parcours_prof_it(self, sommet, deja_vus=[]):
        if sommet in deja_vus:
            return []

        lst_parcours = []
        deja_vus = [sommet]
        pile_a_traiter = [sommet]

        while len(pile_a_traiter) != 0:
            sommet_courant = pile_a_traiter.pop()
            lst_parcours.append(sommet_courant)
            for voisin in self.adj_lst[sommet_courant]:
                if not voisin in deja_vus:
                    deja_vus.append(voisin)
                    pile_a_traiter.append(voisin)

        return lst_parcours
```

Exercice 17 @TODO !

Exercice 18

1. Ordre de visite des sommets lors du parcours en largeur d'abord, en partant :
   - du sommet A : A, B, F, G, C, D, H, I, E, J
   - du sommet E : E, I, J, D, H, C, G, A, B, F
   - du sommet H : H, G, I, A, B, C, D, F, E, J

(TSVP)

2.

```
class GraphAL:
    # ...
    def parcours_largeur(self, sommet, deja_vus=[]):
        if sommet in deja_vus:
            return []

        lst_parcours = []
        deja_vus = [sommet]
        file_a_traiter = [sommet]

        while len(file_a_traiter) != 0:
            sommet_courant = file_a_traiter.pop(0) # seule différence ici, on
            ↪  utilise une "file"
            lst_parcours.append(sommet_courant)
            for voisin in self.adj_lst[sommet_courant]:
                if not voisin in deja_vus:
                    deja_vus.append(voisin)
                    file_a_traiter.append(voisin)

        return lst_parcours
```

3.

```
assert graphe.parcours_largeur("A", []) == ['A', 'B', 'F', 'G', 'C', 'D', 'H',
↪  'I', 'E', 'J']
assert graphe.parcours_largeur("E", []) == ['E', 'I', 'J', 'D', 'H', 'C', 'G',
↪  'A', 'B', 'F']
assert graphe.parcours_largeur("H", []) == ['H', 'G', 'I', 'A', 'B', 'C', 'D',
↪  'F', 'E', 'J']
```