Graphes et représentations

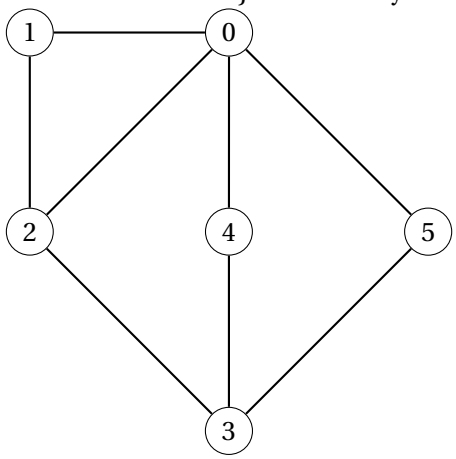
INFORMATIQUE COMMUNE - TP n° 2.3 - Olivier Reynet

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

A Représentation et transformation d'un graphe

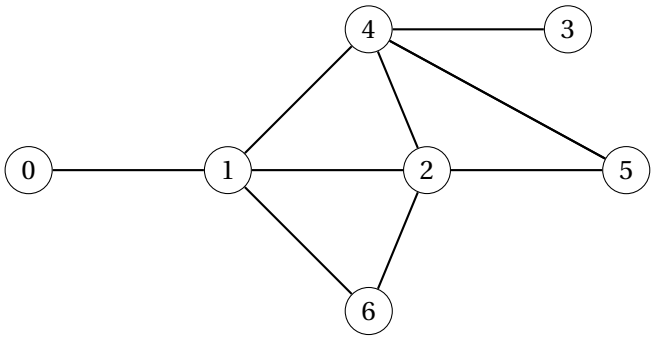
A1. Créer une liste d'adjacence en Python qui représente le graphe suivant :



Solution :

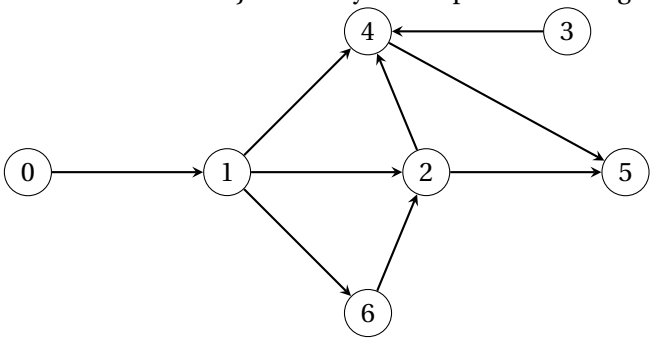
```
1 ga1 = [[1, 2, 4, 5], [0, 2], [0, 1, 3], [2, 4, 5], [0, 3], [0, 3]]
```

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 :

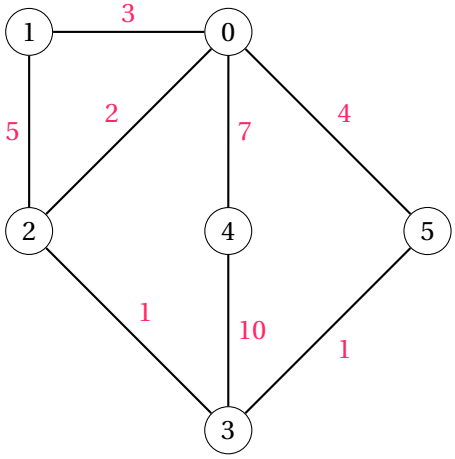
A3. Créer une liste d'adjacence Python représentant le graphe orienté suivant :



Solution :

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

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



Solution :

```
1 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 un tableau Numpy.

Solution :

```
def ladj_to_madj(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
```

A7. Écrire une fonction de prototype `ladj_to_madj_l(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 imbriquée Python.

Solution :

```
def ladj_to_madj_l(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
```

A8. Écrire une fonction de prototype `madj_to_ladj(g)` qui prend comme paramètre un graphe (non pondéré) sous la forme d'une matrice d'adjacence de type tableau Numpy et renvoie le même graphe sous la forme d'une liste d'adjacence.

Solution :

```
def madj_to_ladj(g):
    n = g.shape[0]
    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(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 : 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 création de la liste est de complexité linéaire en $O(n)$ et la double boucle effectue autant d'itérations qu'il y a de sommets et d'arêtes. De plus, l'opération effectuée dans cette double boucle est de complexité constante.

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

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

```
    return tg
```

A10. 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]
```

A11. 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)]
```

B Création d'un graphe à partir de données dans un fichier

On dispose d'un fichier qui contient des distances entre des villes de l'ouest de la France (le télécharger sur le site!). On souhaite construire un graphe dont les sommets sont les villes, les arêtes les liaisons routières entre les villes et le poids des arêtes la distance en km entre les villes.

B1. Importer les données présentes dans le fichier dans une liste dont les éléments sont des tuples ("ville de départ", "ville d'arrivée", distance en km).

Solution : Au besoin, si ce code est difficile à comprendre, ne pas hésiter à faire des `print` à chaque étape pour visualiser le contenu des variables.

```
def import_csv(filename):
    # ouverture du fichier en lecture seule
    file = open(filename, 'r')
    # lecture de toutes les lignes ->  une liste de chaînes de caractères.
    all_lines = file.readlines()
    # fermeture du fichier (on en a plus besoin)
    file.close()
    head = all_lines[0].split(',')
    data = []
    for i in range(1, len(all_lines)): # pour chaque ligne
        # on récupère la liste des champs de la ligne courante
        words = all_lines[i].split(',')
        #print(words)
```

```
        # on ajoute les données en enlevant les espaces
        data.append((words[0].strip(), words[1].strip(), int(words[2])))
    return data

def import_csv_bis(filename):
    # ouverture du fichier en lecture seule
    file = open(filename, "r")
    # récupération des entêtes
    head = file.readline().split(',')
    data = []
    for line in file:  # énumère les lignes suivantes du fichier
        # on récupère la liste des champs de la ligne courante
        words = line.split(",")
        # on ajoute les données en enlevant les espaces
        data.append((words[0].strip(), words[1].strip(), int(words[2])))
    # fermeture du fichier (on en a plus besoin)
    file.close()
    return data
```

B2. Afin de construire un graphe, on souhaite utiliser une correspondance arbitraire entre le nom des villes et un numéro. Un dictionnaire Python est une structure de données qui associe une clef à une valeur et permet de réaliser cette correspondance. Écrire une fonction de prototype `create_mapping(data)` dont le paramètre est la liste `data` des données importées et qui renvoie le dictionnaire suivant :

```
mapping = {"Paris": 0, "Limoges": 1, "Toulouse": 2, "Tours": 3, "Poitiers": 4, "
    Bordeaux": 5, "Bayonne": 6, "Pau": 7, "Nantes": 8, "Vannes": 9, "Lorient":
    10, "Quimper": 11, "Brest": 12, "Rennes": 13, "Le Mans": 14}
```

Solution :

```
def create_mapping(data):
    mapping = {} # un dictionnaire vide
    numero = 0 # le numéro de la première ville
    for v1, v2, _ in data:
        if v1 not in mapping:
            mapping[v1] = numero
            numero += 1
        if v2 not in mapping:
            mapping[v2] = numero
            numero += 1
    return mapping
```

B3. En utilisant le dictionnaire `mapping`, écrire une fonction de protoype `create_graph(data, mapping)` qui renvoie le graphe correspondant aux données importées sous la forme d'une liste d'adjacence.

Solution :

```python
def create_graph(data, mapping):
    n = len(mapping)
    g = [[] for i in range(n)]
    for t1, t2, d in data:
        g[mapping[t1]].append((mapping[t2], d))
    return g
```

B4. En utilisant le dictionnaire `mapping`, écrire une fonction de protoype `create_matrix_graph(data, mapping)` renvoie le graphe correspondant aux données importées sous la forme d'une matrice d'adjacence de type tableau Numpy.

Solution :

```python
def create_matrix_graph(data, mapping):
    n = len(mapping)
    m = np.zeros((n,n))
    for t1, t2, d in data:
        m[mapping[t1], mapping[t2]] = d
    return m
```

B5. En utilisant la fonction de Numpy `count_nonzero` (cf. documentation en ligne) et la représentation matricielle du graphe, construire la liste des degrés de chaque sommet du graphe. Quel sens pouvez-vous donner à cette information ?

Solution : Il s'agit du nombre de destinations directes possibles (voisines) à partir d'une ville. Plus le nombre est grand, moins une ville est isolée.

```python
def degrees(m):
    return np.count_nonzero(m, axis=1)
```

B6. En utilisant les fonction de Numpy (notamment `nonzero`, cf. documentation en ligne) et la représentation matricielle du graphe, calculer les distances minimales, maximales, moyennes ainsi que l'écart-type des distances entre les villes.

Solution :

```python
def stats(m):
    return np.min(m[np.nonzero(m)]), np.max(m), np.mean(m[np.nonzero(m)]),
        np.std(m[np.nonzero(m)])
```