La théorie des graphes

1. Introduction

Un graphe est une modélisation d'un ensemble d'objets reliés : les objets sont appelés *nœuds*, et les liens sont appelés *arêtes*.

2. Notion de graphe

a. Graphe non orienté :

Définition :

Un graphe non orienté est un couple (S,A)

- **S** : l'ensemble des **sommets** de l'arbre
- **X** : l'ensemble des **arêtes** joignant deux sommets tel que **X⊆*A***
- $\{x_1,x_2\}$: l'arête d'extrémité $x_1$et x_2

Un graphe non orienté est dit **simple** s'il n'a ni boucle ni arête multiple

b. Graphe orienté :

Définition

Un graphe orienté est un couple (S , A)

- S : l'ensemble des sommets de l'arbre e
- X : l'ensemble des arcs joignant deux sommets tel que X⊆*A*
- $\{x_1,x_2\}$:un **arc** ou $x_1$est un prédécesseur de x_2(ou réciproquement)

Un graphe orienté est dit **simple** s'il n'a ni boucle ni arc multiple

c. Graphe pondéré

Un graphe pondéré est un graphe où les arcs ou les arêtes sont munis de poids

3. Matrice d'adjacence

Définition

Une matrice d'adjacence A associée au graphe G=(S , A) est une matrice à valeur a_{ij}=n (n est le nombre d'arêtes ou arc entre a_i et a_j) telle que :

a. si le graphe est simple les éléments de la diagonale sont nuls et $\mathbf{a_{ij}} \in \boldsymbol{Z/2Z}$
b. si le graphe est non orienté, la matrice d'adjacence est symétrique

4. Représentation d'un graphe

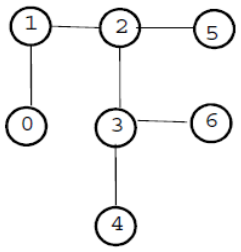
Il existe différentes manières de représenter un graphe en machine. A l'aide :

- d'une liste de listes
- d'une matrice
- d'un dictionnaire

5. Exemple d'une représentation matricielle

Soit le graphe suivant avec sa représentation matricielle :

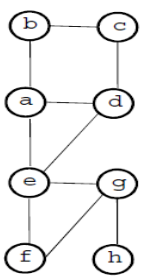
```
M=array([[0,1,0,0,0,0,0],
         [1,0,1,0,0,0,0],
         [0,1,0,1,0,1,0],
         [0,0,1,0,1,0,1],
         [0,0,0,1,0,0,0],
         [0,0,1,0,0,0,0],
         [0,0,0,1,0,0,0]])
```



$$M = \begin{pmatrix} 0&1&0&0&0&0&0 \\ 1&0&1&0&0&0,&0 \\ 0&1&0&1&0&1&0 \\ 0&0&1&0&1&0&1 \\ 0&0&0&1&0&0&0 \\ 0&0&1&0&0&0&0 \\ 0&0&0&1&0&0&0 \end{pmatrix}$$

6. Exemple2 représentation d'un graphe sous forme d'un dictionnaire

```
G={   'a':['b','d','e'],
      'b':['a','c'],
      'c':['b','d'],
     'd' :['a','c','e'],
     'e' :['a','d','f','g'],
     'f' :['e','g'],
     'g' :['e','f','h'],
     'h' :['g']
}
```



7. **Terminologies**
   - ***Ordre du Graphe :*** le nombre de sommets du Graphe
   - ***Degré d'un sommet :*** nombre d'arêtes reliées à ce sommet
   - ***Adjacences:*** Deux **arcs** sont dits adjacents s'ils ont une extrémité en commun. Et deux **sommets** sont dits adjacents si un arc les relie.
   - ***Boucle :*** est un arc qui part d'un sommet vers le même sommet
   - ***Chaîne :*** Une chaine de longueur **n** est une suite de **n** arêtes permettant de relier un sommet **i** à un autre **j** ou à lui-même.
   - ***Cycle :*** Un cycle est une chaine qui permet de partir d'un sommet et revenir à ce sommet en parcourant ***une et une seule fois*** les autres sommets.
   - ***Distance*** entre deux sommets **i** et **j** est la longueur de la chaine la plus courte qui les relie
   - ***Chemin :*** c'est une chaine bien orientée
   - ***Circuit :*** est un cycle "bien orienté", à la fois cycle et chemin.
   - ***Chaine eulérienne:*** est une chaine comportant exactement ***une fois*** toutes les arêtes du graphe.
   - ***Cycle eulérien :*** si le sommet de ***départ*** d'une chaine eulérienne est celui ***d'arrivé***
   - ***Graphe eulérien :*** Un graphe admettant une chaine eulérienne est dit Graphe eulérien
   - ***Cycle hamiltonien :*** c'est un cycle passant une seule fois par tous les **sommets** d'un graphe et revenant au sommet de départ**.**
   - ***Graphe connexe:*** c'est un graphe dont tout couple de sommet peut être relié par une chaine de longueur n>=1.

     Théorème d'Euler :

     Un graphe connexe admet une chaine eulérienne si deux sommets ou 0 sommet exactement sont de degré impair.

     Un graphe comporte un Cycle eulérien s'il est connexe et n'admet aucun sommet de degré impair

     Un graphe comporte une chaine entre deux sommets i et j s'il est connexe et i et j sont deux sommets de degré impair

     ➔ *Conséquence: tout graphe connexe comporte plus de deux sommets de degré impair n'est pas eulérien*

8. **Parcours de graphe**

8.1 Le parcours en largeur (Breadh First Search): on procède par niveau en considérant d'abord tous les sommets à une distance donnée, avant de traiter ceux du niveau suivant.

Nous utiliserons les éléments suivants : • Un graphe **G** défini à l'aide d'un dictionnaire • Un dictionnaire **P** pour définir les sommets visités en précisant pour chaque sommet le père du nœud. • Une file **Q** de type **FIFO.**	*def BFS(G,s) :* *P,Q={s :None},[s]* *while Q :* *u=Q.pop(0)* *for v in G[u] :* *if v not in P :* *P[v]=u* *Q.append(v)* *return P*

8.2 Le parcours en profondeur DFS **(Depht First Search)** : on va aussi loin que possible en faisant des choix lors des branchements, et ensuite on remonte aussi près que possible pour faire les choix restants ;

Nous utiliserons les éléments suivants : • Un graphe **G** défini à l'aide d'un dictionnaire • Un dictionnaire **P** pour définir les sommets visités en précisant pour chaque sommet le père du nœud. • Une file **Q** de type **LIFO.**	*From random import choice* *def DFS(G,s) :* *P,Q={s :None},[s]* *while Q :* *u=Q[-1]* *R=[v for v in G[u] if v not in P]* *if R :* *v= choice(R)* *P[v]=u* *Q.append(v)* *else :* *Q.pop()* *return P*

9. Recherche du plus court chemin (Dijkstra)

L'algorithme dû à Dijkstra est basé sur le principe suivant : Si le plus court chemin reliant **S** à **D** passe par les sommets **s1, s2, …, sk** alors, les différentes étapes sont aussi les plus courts chemins reliant **E** aux différents sommets **s1, s2, …, sk**.

On construit de proche en proche le chemin cherché en choisissant à chaque itération de l'algorithme, un sommet s_i du graphe parmi ceux qui n'ont pas encore été traités, tel que la longueur connue provisoirement du plus court chemin allant de **S** à s_i soit la plus courte possible.

Initialisation de l'algorithme :

Étape 1 : On affecte le poids 0 au sommet origine (**s**) et on attribue provisoirement un poids ∞ aux autres sommets.

Répéter les opérations suivantes tant que le sommet de sortie (s) n'est pas affecté d'un poids définitif

Étape 2 : Parmi les sommets dont le poids n'est pas définitivement fixé choisir le sommet **X** de poids ***p*** minimum. Marquer définitivement ce sommet **X** affecté du poids **p(X)**.

Étape 3 : Pour tous les sommets Y qui ne sont pas définitivement marqués, adjacents au dernier sommet fixé **X** :

- Calculer la somme ***s*** du poids de X et du poids de l'arête reliant **X** à **Y**.
- Si la somme ***s*** est inférieure au poids provisoirement affecté au sommet Y, affecter provisoirement à Y le nouveau poids ***s*** et indiquer entre parenthèses le sommet X pour se souvenir de sa provenance.

Quand le sommet s est définitivement marqué

Le plus court chemin de **S** à **D** s'obtient en écrivant de gauche à droite le parcours en partant de la fin **D**.

```
def dijkstra(M,s):
    # M : matrice, s: sommet de départ
    infini=M[0,0] #valeur des cases non définies
    n= len(M)  # le nombre de sommets
    A=[]
    C=list(range(0,n))
    distance=ones(n)* infini
    distance[s]=0 # on somme les poids
    pred=ones(n)*s
    while C!=[]:
        C.sort(key=lambda i:distance[i])
        a=C[0]
    for c in C:
        if distance[a]+ M[a,c] < distance[c]:
            distance[c]=distance[a]+M[a,c]
            pred[c]=a
    A.append(a)
    C.remove(a)
    return s,pred,distance,infini
```

Exercice01 :

1. **Dessiner les Graphes suivants G1 ,G2 et G3 (remarque :99 signifié pas d'arc directe entre les deux nœuds)**

```
G1={    'a':['b','d','e'],
        'b':['a','c'],
        'c':['b','d'],
        'd' :['a','c','e'],
        'e' :['a','d','f','g'],
        'f' :['e','g'],
        'g' :['e','f','h'],
        'h' :['g']}
```

G2=

```
[[99 ,1 ,1, 99, 99, 99,99,99],
[ 99 ,99 ,99, 5 ,99 ,99,99,99],
[99 ,99, 99, 99 ,3, 1,99,99],
[ 99 ,99 ,99 ,99 ,99,99,99 ,10],
[ 99 ,99 ,99 ,99 ,99,2,99 ,99],
[ 99 ,99 ,99 ,99 ,99,99,1 ,99],
[ 99 ,99 ,99 ,99 ,99,99,99 ,2],
[ 99 ,99 ,99 ,99 ,99,99,99 ,99]]
```

G3=

```
[[99 ,1 ,2, 99, 99, 99],
[ 99 ,99 ,99, 99 ,1 ,99],
[99 ,3, 99, 3 ,99, 99],
[ 99 ,99 ,99 ,99 ,99 ,2],
[99 ,99 ,99 ,2 ,99 ,5],
[ 99 ,99 ,99 ,99, 99 ,99]]
```

Exercice02 :

Définir les fonctions suivantes :

1. **def BFS(G,s):** qui permet de parcourir un Graphe en **largeur**
2. **def DFS(G,s):** qui permet de parcourir un Graphe en **profondeur**
3. **def chercher(G,x):** qui retourne True si **x** existe dans G ,False sinon
4. **def *OrdreDuGraphe* (G):** qui retourne le nombre de sommet du graphe
5. **def DegreSommet*(G,s) :qui retourne le*** nombre d'arêtes reliées à ce sommet
6. ***def Adjacences(G,s1,s2):qui retourne True si s1 et s2*** deux sommets adjacents **False** sinon.
7. **comporte*Boucle*(G):** ***qui retourne True si G admet une boucle*** **False** sinon.
8. **def grapheSimple**(M) : qui retourne True si un graphe M est simple ,False sinon
9. **def admetChaineEulerienne (G):** ***qui retourne True si G admet une chaine eulérien*** **False** sinon.
10. **def is*Eulerien(G) : qui retourne True si G est eulérien*** **False** sinon.
11. **def comporteCycleEulerien(G):** ***qui retourne True si G comporte un cycle eulérien*** **False** sinon.

12. **def dijkstra(M,s):**qui permet d'afficher Le plus court chemin entre deux sommets .
13. **def grapheNonOriente**(M) : qui retourne True si le graphe est non orienté, False sinon

Remarque :G :le dictionnaire d'un graphe
M :la matrice d'adjacence d'un graphe
s,s1,s2 :sont des sommets