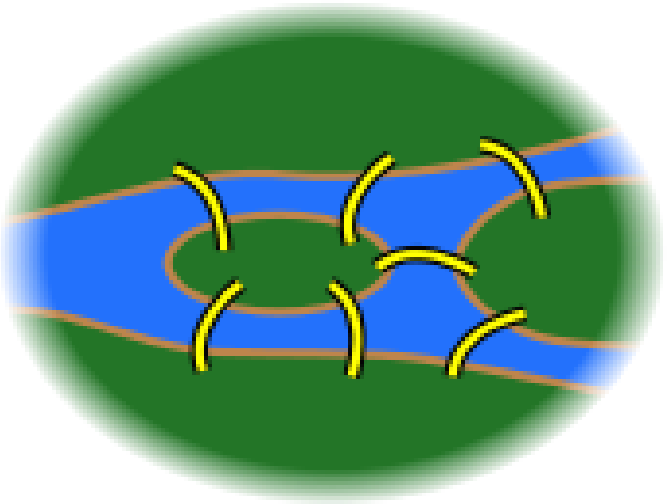
Introduction à la théorie des graphes

1) Origines

Un des plus anciens problèmes combinatoires, la détermination d'un itinéraire à travers la ville de konigsberg(aujourd'hui kaliningrad) en n'utilisant qu'une fois et une seule chacun des sept ponts qui enjambaient les bras de la pregel ou conduisaient à l'île de kneiphof, dont Euler montra en 1735 l'impossibilité, semble constituer le premier témoignage de l'emploi des graphes finis.



2) Définitions

Graphe :

Un graphe **G** est défini par **G=(V,U)**, où **V** est un ensemble de sommets et **U** l'ensemble d'arcs(ou arêtes) ; Un arc (ou arête) est un couple de sommets, donc, un élément du produit cartésien **VxV**

Graphe orienté et Graphe non orienté :

Si les arêtes ne sont pas orientées, on parle d'un graphe non orienté. Dans le cas contraire on parle d'un graphe orienté et les arêtes sont appelées aussi les arcs.

Graphe Pondéré :

Un graphe pondéré est défini par le triplet **(V,U,C)** où : **V** est l'ensemble des sommets**, U** est l'ensemble des arrêtes (ou arcs), et **C** est la fonction de coût de **U** dans **IR**. Par convention **Cu** représente le coût ou le poids de l'arc (ou de l'arête) **u**.

Graphe Connexe :

Un graphe connexe est un graphe dont tout couple de sommets peut être relie par une chaine de longueur n>=1.

3) Terminologies

- **Ordre du Graphe :** le nombre de sommet 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 qui relient 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 a ce sommet en parcourant une et une seule fois les autres sommets.
- **Distance** entre deux sommet**s** 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**: une chaine est dite eulérienne est une chaine comportant exactement une fois toutes les arêtes du graphe.
- **Cycle eulérien** : si le sommet de départ d'une chaine eulérienne et celui d'arrivé on parle de cycle eulérienne
- **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.

4) Théorèmes d'Euler

Théorème 1:

Un graphe connexe **G** admet un cycle eulérien si et seulement si tous ses sommets sont de degré pair.

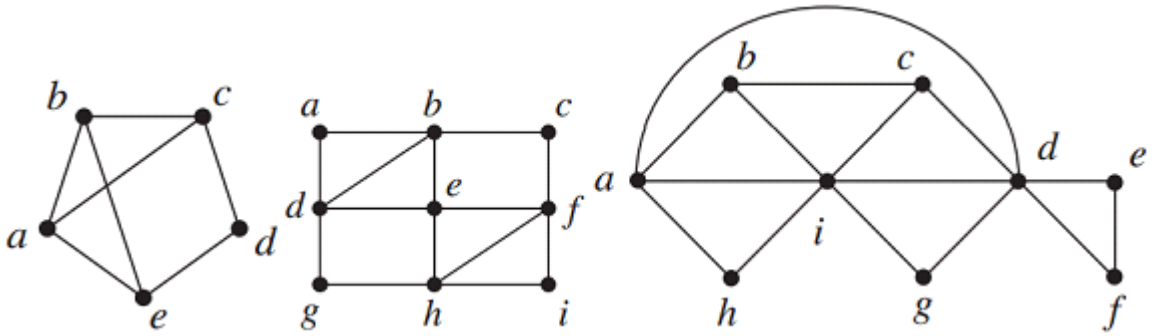
Théorème 2:

Un graphe connexe **G** admet une chaîne eulérienne distincte d'un cycle si et seulement si le nombre de sommets de **G** de degré impair est égal à 2.

Dans ce cas, si **A** et **B** sont les deux sommets de **G** de degré impair, alors le graphe **G** admet une chaîne eulérienne d'extrémités **A** et **B**.

Exercice :

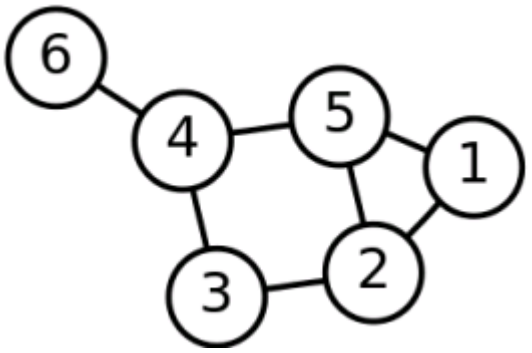
Vérifier si les graphes suivant sont eulériens ou admettent des chaines eulériens.



5) Représentation d'un Graphe

Un graphe peut être implémenté de différentes manières selon le langage utilisé. En **Python** en peut représenter un graphe à l'aide d'un **dictionnaire** ou à l'aide d'une **matrice d'adjacence**.

Exemple :

	**Graphe={** **1 : [2,5] ,** **2 : [1,3,5] ,** **3 : [2,4] ,** **4 : [3,5,6] ,** **5 : [1,2,4] ,** **6 : [4]** **}**	**Graphe=[** **[0, 1, 0, 0, 1, 0],** **[1, 0, 1, 0, 1, 0],** **[0, 1, 0, 1, 0, 0],** **[0, 0, 1, 0, 1, 1],** **[1, 1, 0, 1, 0, 0],** **[0, 0, 0, 1, 0, 0]** **]**
Graphe	**Dictionnaire**	**Matrice d'adjacence**

Propriétés de la matrice d'adjacence

- La matrice est symétrique si le graphe n'est pas orienté.
- La somme des nombres d'une même ligne (ou d'une même colonne) donne le degré du sommet correspondant.
- La diagonale ne contient que des zéros.
- Les termes $\mathbf{a_{ij}}$ de la matrice $\mathbf{A^n}$ donnent le nombre de chaînes de longueur **n** reliant **i** à **j.**

6) Parcours d'un graphe

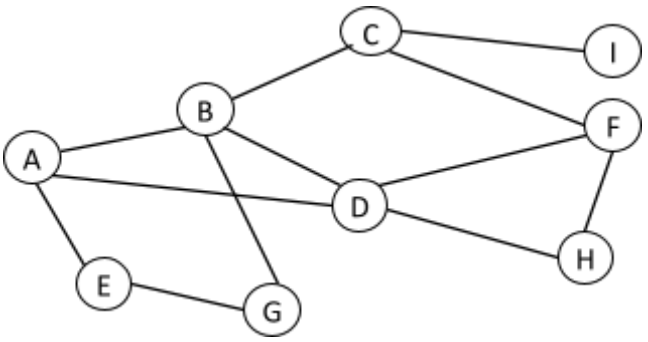
Par la suite nous utilisons la représentation en utilisant la **matrice d'adjacence** d'un graphe.

a- Le parcours en profondeur d'abord 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 ;

Algorithme :

- Initialement tous les nœuds sont marqués " non visités".
- Choisir un nœud **v** de départ et le marquer " visité".
- Chaque nœud adjacent à **v**, non visité, est à son tour visité en utilisant **DFS** récursivement.
- Une fois tous les nœuds accessibles à partir de **v** ont été visités, la recherche de **v** (**DFS(v)**) est complète.
- Si certains nœuds du graphe restent "non visités", sélectionner un comme nouveau nœud de départ et répéter le processus jusqu'à ce que tous les nœuds soient visités.

Exemple : Soit le graphe suivant :



Le parcours en profondeur de ce graphe : A, B, C, F, D, H, I, G, E

Implémentation du parcours en profondeur en **Python**.

```
def parcoursProfondeur(G) :
      T=[0]*len(G)
      for i in range (len(G)) :
            if T[i]==0 :
               parcoursDFS(G,i,T)
```

```
def succNonvisite(G,s,T) :
      n=len(G[s])
      L=[ ]
      for i in range(n) :
            If G[s][i]==1 and
T[i]==0 :
                  L.append(i)
      return L
```

```
def parcoursDFS(G,s,T) :
      P=[s]
      while P !=[] :
            noeud=P.pop()
            if T[noeud]==0 :
                  print(nœud,end='  ')
                  T[noeud]=1
                  succ= succNonvisite(G, nœud,T)
                  succ.reverse()
                  if succ !=[] :
                        for x in succ :
                              P.append(x)
```

La fonction parcoursDFS peur être récursive :

```
def DFSRecursif(G,s,T) :
     T[s]= 1
     print(s ,end=' ')
     for x in succNonvisite(G, s ,T) :
          if T[x]== 0:
               DFSRecursif(G,x,T)
```

parcoursProfondeur(G) : fonction qui lance le parcours, elle peut faire appel soit à la fonction **parcoursDFS** soit à la fonction **DFSRecursif** selon le parcours souhaité

succNonVisite(g,s,test) : retourne la liste des successeurs non visités de **s**

b- Le parcours en largeur d'abord (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.

Exemple :

Le parcours en largeur du graphe précèdent visite les sommets dans l'ordre suivant : **A, B, D, E, C, G, F, H, I**

En python :

```
def BFS(G,s,T) :
      F=[s]
      while F !=[ ] :
```

```
def ParcoursEnlargeur(G) :
      T=[0]*len(G)
      for i in range(len(G)) :
```

```
            x=F.pop(0)
            if T[x]==0 :
                  T[x]=1
                  print(x, end='   ')
                  succX=
succNonvisite(G,x,T)
                  if succX !=[] :
                        F+= succX
```

```
      if T[i]==0 :
            BFS(G,i,T)
```

7) Le Plus court chemin

Il existe de nombreux algorithmes déterminant un ou le plus court chemin dans un graphe connexe pondéré.

Par exemple: **Warshall, Floyd, Dijkstra, Branch and Bound, Bellman-Ford**

On se limitera à la recherche d'un plus court chemin entre deux sommets du graphe pondéré avec des poids positifs en utilisant **la matrice d'adjacence** pour représenter le graphe.

❖ Algorithme de dijkstra

L'algorithme dû à **Dijkstra** est basé sur le principe suivant :

Si le plus court chemin reliant **E** à **S** passe par les sommets s_1, s_2, …, s_k alors, les différentes étapes sont aussi les plus courts chemins reliant **E** aux différents sommets s_1, s_2, …, s_k.

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 **E** à s_i soit la plus courte possible.

Initialisation de l'algorithme :

Étape 1 :

On affecte le poids 0 au sommet origine (E) 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** minimal.

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

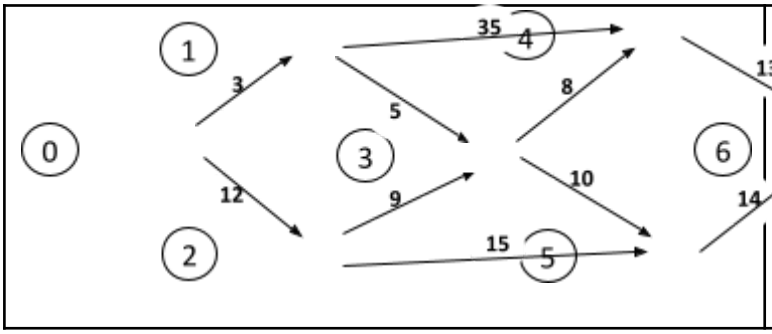
Le plus court chemin de **E** à **S** s'obtient en écrivant de gauche à droite le parcours en partant de la fin **S.**

```
def dijkstra(G,s) :
       infini = G[0][0]                    # infini est la valeur des cases non définies
       n=len(G)                            # le nombre de sommets
       D=[infini for i in range (n)]       # initialisation des couts du plus court chemin entre i et s.
       D[s]=0
       P =[s for i in range(n)]            # P[i] : dernier sommet visité avant i.
       C =[i for i in range(n)]            # Sommets à visiter
       while C !=[] :                      # tant qu'il y a des sommets à visiter
          x=minD(D,C)                      # le prochain sommet le plus proche
          for k in C :
             if D[x]+G[x][k]<D[k] :        #si la distance vers k est minimale en passant par x
                  D[k]=D[x]+G[x][k]        #Mise à jour de la distance de k
                  P[k]=x                   #mise à jour du sommet visité k
          C.remove(x)                      #ne plus revenir
       return P,D
```

Sachant que la fonction **minD(D,C)** retourne l'indice du sommet de **C** ayant la petite valeur dans **D**. c.à.d. le sommet prochain ayant une distance minimale avec le sommet du départ.

```
def minD(LD,LC) :
    imin=LC[0]
    for i in LC :
     if LD[i]<LD[imin] :
           imin=i
    return imin
```

Exemple d'exécution :




```
Gr=[
    [99, 3, 12, 99, 99, 99, 99],
    [99, 99, 99, 5, 35, 99, 99],
    [99, 99, 99, 9, 99, 15, 99],
    [99, 99, 99, 99, 8, 10, 99],
    [99, 99, 99, 99, 99, 99, 13],
    [99, 99, 99, 99, 99, 99, 14],
    [99, 99, 99, 99, 99, 99, 99]
    ]
```

>>>dijkstra(Gr,0)

([0, 0, 0, 1, 3, 3, 4] , [0, 3, 12, 8, 16, 18, 29])