Les graphes

1. Généralités sur les graphes :

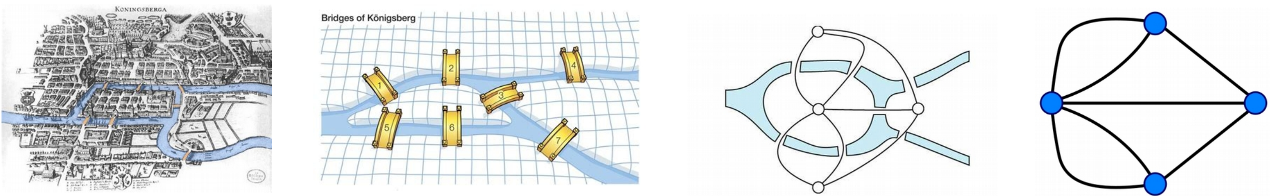
L'histoire de la théorie des graphes débuterait avec les travaux du mathématicien suisse Leonhard Euler au cours du 18$^{\text{ème}}$ siècle et trouve son origine dans l'étude de certains problèmes, tels que celui des ponts de Königsberg, le problème du coloriage de cartes et du plus court trajet entre deux points.

La théorie des graphes est la discipline mathématique et informatique qui étudie les graphes, du fait de l'importance que revêt l'aspect algorithmique ; elle s'utilise pour représenter des données qui ont naturellement une structure de graphe :

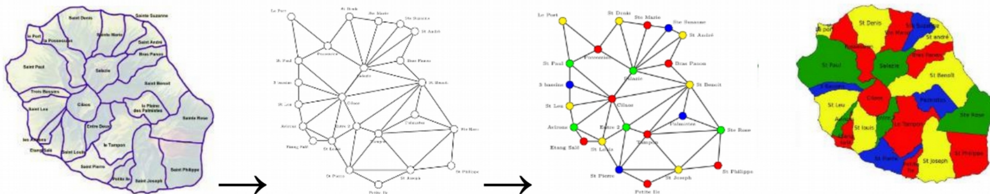
- *les réseaux de transport (routes, rail, métro, …) ;*
- *les réseaux sociaux ;*
- *les jeux (échecs, …).réseaux routiers ou les circuits électriques…)*

Un graphe permet de représenter des objets ainsi que les relations entre ses éléments.

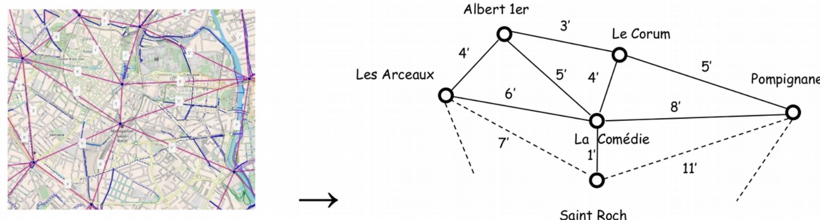
Exemples :
Les 7 ponts de Königsberg
La ville de Königsberg est construite autour de deux îles situées sur la rivière Pregel.
Les deux îles sont reliées entre elles par un pont. Six autres ponts relient les rives de la rivière à l'une ou l'autre des deux îles.
Le problème consiste à déterminer s'il existe une promenade permettant, à partir d'un point de départ au choix, de passer une et une seule fois par chaque pont et de revenir à son point de départ, étant entendu qu'on ne peut traverser la rivière qu'en passant sur les ponts.



Coloriage des cartes
Comment colorier une carte en associant une couleur par région, de telle sorte que deux régions voisines aient deux couleurs différentes ?
Exemple : Carte des 24 communes de la Réunion



Calcul du plus court chemin
Comment se rendre en vélo à l'hôtel de Région (Pompignane) à partir du quartier des Arceaux de Montpellier ? Quel chemin doit-on prendre pour y aller la plus vite possible ?
Quel chemin doit-on prendre pour parcourir la distance la plus courte ?

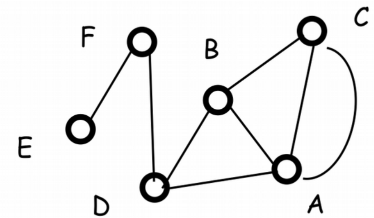


2. Quelques définitions :

2.1. Graphes non orientés :

Un graphe est un ensemble de points et de lignes reliant ces points.

- Les points sont appelés ***sommets*** ou ***nœuds*** du graphe (***vertex*** en anglais).
- Les lignes reliant deux sommets sont appelées ***arêtes*** ou liens (***edge*** en anglais).



Ce graphe comporte 6 sommets et 8 arêtes.

Un ***sommet*** est dit ***isolé*** s'il n'a pas de liaison avec les autres sommets du graphe.
Deux ***sommets*** sont dits ***adjacents*** (ou ***voisins***) s'ils sont reliés par une arête.

Une ***boucle*** est une arête reliant un sommet à lui-même.

Une ***chaîne*** est une ***suite d'arêtes*** permettant de se rendre d'un sommet à un autre.
Les sommets de départ et d'arrivée sont appelés ***extrémités***.

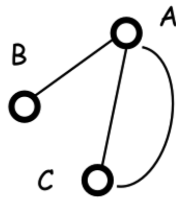
Un ***cycle*** est un chemin permettant de revenir au sommet d'origine en passant une seule fois par d'autres sommets.

Un graphe est dit :

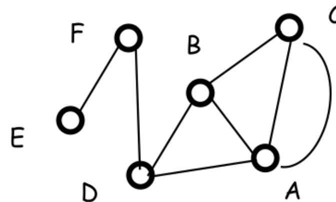
- ***Simple*** si il y a au plus une arrête entre deux sommets.
- ***Orienté*** si ses arêtes sont orientées. Les arêtes sont alors appelées ***arcs*** et sont représentées avec des ***flèches*** indiquant le ***sens*** de la liaison.
- ***Connexe*** lorsque, il existe un chemin entre deux sommets (le graphe est d'un seul tenant).
- ***Complet*** si chacune des sommets est relié directement à tous les autres sommets.
- ***Pondéré*** si chaque arête est affectée d'un poids (nombre positif).

Graphe :				Graphe :				Graphe :			
avec boucle		simple		avec boucle		simple		avec boucle		simple	
avec cycle		orienté		avec cycle		orienté		avec cycle		orienté	
connexe		complet		connexe		complet		connexe		complet	
Graphe (E est isolé) :				Graphe :				Graphe :			
avec boucle		simple		avec boucle		simple		avec boucle		simple	
avec cycle		orienté		avec cycle		orienté		avec cycle		orienté	
connexe		complet		connexe		complet		connexe		complet	

Le nombre de ***sommets*** d'un graphe détermine son ***ordre***.
Le ***degré*** d'un ***sommet*** est le nombre d'arêtes dont il est une extrémité.



Ordre : 3
Degré de A : $d°(A)=3$



Ordre : 6
Degré de A : $d°(A)=4$

2.2. Graphes orientés :

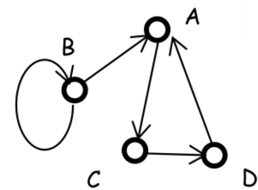


fig 1

pour un graphe orienté le vocabulaire change quelque peu :
Les arêtes sont appelées **arcs.**
Un arc possède une **extrémité initiale** et une **extrémité finale** et un sommet un **degré entrant** $d°_-(A)=2$ et un **degré sortant** $d°_+(A)=1$.
Le degré du point A est : $d°(A)=d°_-(A)+d°_+(A)$
On ne parle pas de chaîne mais de **chemin** ainsi que cycle devient **circuit**.
Dans un graphe orienté, il peut exister un chemin menant du sommet A au sommet D, alors que l'inverse n'est pas possible.

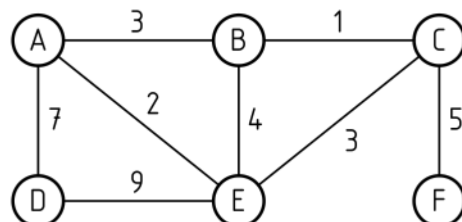
Exemple d'un chemin et d'un circuit (fig 1) :
Chemin : BACD circtuit : ACDA
remarque le raisonnement est le même pour un graphe non orienté (sans prise en compte du sens de parcours).

Bilan	Graphe non orienté	Graphe orienté
Lien entre sommets	Arêtes	Arcs
une suite d'arêtes/arcs	Chaîne	Chemin
chemin/chaine permettant de revenir au sommet d'origine	Cycle	Circuit

On définit :
On définit un graphe G par un couple $G=(S,A)$ avec :

- S un ensemble de sommets : $S=\{s_1,s_2,\cdots,s_n\}$
- A un ensemble d'arêtes : $A=\{a_1,a_2,\cdots,a_m\}$

2.3. Graphes pondéré :



On s'intéresse maintenant à des graphes dont les arêtes/arcs représentent une valeur utile au problème étudié (distances ou temps entre deux points d'un réseau de transport par exemple).

Ainsi, dans l'exemple proposé :
Sur le graphe non orienté, le trajet de A à B prendrait 3 minutes.

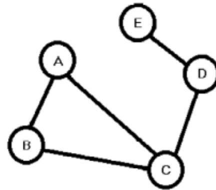
3. Représentation informatique des graphes :

3.1. Matrice adjacente :

Définition :
A tout graphe G d'ordre n, de sommets notés s_i, on peut associer une matrice carrée (n, n) : $M=(m_{ij})$ où m_{ij} est le nombre d'arcs/arêtes reliant les sommets s_i à s_j. Cette matrice est appelée matrice d'adjacence associée à G.

Graphe non orienté :

$$M=(m_{ij})=\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$



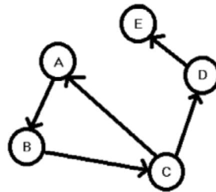
$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Remarque : la matrice d'adjacence d'un graphe non orienté est symétrique.

en rouge PROVENANCE
en bleu DESTINATION

Graphe orienté :

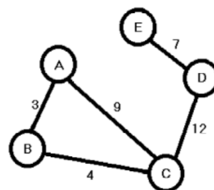
$$M=(m_{ij})=\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Graphe pondéré :

$$M=(m_{ij})=\begin{pmatrix} 0 & 3 & 9 & 0 & 0 \\ 3 & 0 & 4 & 0 & 0 \\ 9 & 4 & 0 & 12 & 0 \\ 0 & 0 & 12 & 0 & 7 \\ 0 & 0 & 0 & 7 & 0 \end{pmatrix}$$



$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 3 & 9 & 0 & 0 \\ 3 & 0 & 4 & 0 & 0 \\ 9 & 4 & 0 & 12 & 0 \\ 0 & 0 & 12 & 0 & 7 \\ 0 & 0 & 0 & 7 & 0 \end{pmatrix} \end{array}$$

Code en python du graphe non orienté, la représentation utilise les listes de listes :

```
graphe_matrice=[[0,1,1,1,0,0],[1,0,0,1,0,0],[1,0,0,1,1,0],
[1,1,1,0,1,1],[0,0,1,1,0,1],[0,0,0,1,1,0]]
```

3.2. Liste d'adjacence :

Une liste d'adjacence est :

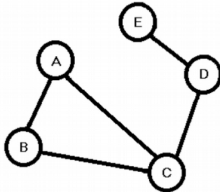
- Pour un graphe non orienté : la liste des voisins de chaque sommet
- Pour un graphe orienté : liste des sommets que l'on peut atteindre (en bout des arcs)

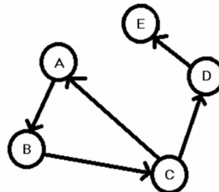
Sous Python, on pourra stocker la liste d'adjacence sous la forme d'un dictionnaire.

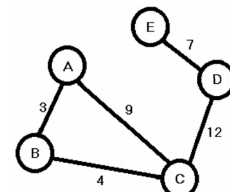
Exemple :

Graphe non orienté :	Graphe orienté :	Graphe pondéré :

G={
'A':['B','C'],
'B':['A','C'],
'C':['A','B','D'],
'D':['C','E'],
'E':['D']
}
	G={
	'A':['B'],
	'B':['C'],
	'C':['A','D'],
	'D':['E'],
	'E':[]
	}
		G={
		'A':[[3,'B'],[9,'C']],
		'B':[[3,'A'],[4,'C']],
		'C':[[9,'A'],[4,'B'],[12,'D']],
		'D':[[12,'C'],[7,'E']],
		'E':[[7,'D']]
		}

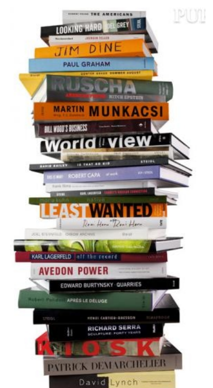






4. Outils de parcours des graphes (les files et piles) :

4.1. Les Piles :



Une pile (**stack** en anglais) est un ensemble de données dans lequel on ajoute ou l'on supprime un élément à partir de la même extrémité appelée sommet de la pile (**top** en anglais).
On effectue donc les opérations suivantes :

- Empiler (**push** ou `append`) : Ajouter un élément sur le sommet de la pile
- Dépiler ou désempiler (`pop`) : Supprimer un élément sur le sommet de la pile

Le principe des opérations repose sur la règle « dernier arrivé, premier sorti » ou LIFO en anglais : « last in, first out ».

Soit une pile composée de $n+1$ objets : $pile=\begin{pmatrix} objet_n \\ \vdots \\ objet_1 \\ objet_0 \end{pmatrix}$

La fonction `empiler()` :

Soit `newObjet` un objet à empiler dans `pile`.
Soit une fonction `empiler(pile,newObjet)`, qui va inclure l'objet `newObjet` dans la pile `pile` et retourner la nouvelle pile `newPile`.

Point de départ	→ $newPile=empiler(x,newObjet)$ →	Résultat
$pile=\begin{pmatrix} objet_n \\ \vdots \\ objet_1 \\ objet_0 \end{pmatrix}$ Objet à empiler : $x=newObjet$		$newPile=\begin{pmatrix} newObjet \\ objet_n \\ \vdots \\ objet_1 \\ objet_0 \end{pmatrix}$
En python :		

```
L = []            # Création d'une liste vide
L.append(x)       # Ajoute l'objet x à la liste L
```

La fonction `dépiler()` :

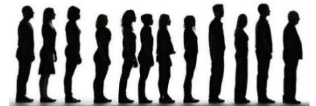
Nous souhaitons dépiler la `pile` d'un élément, soit le dernier inséré `objet_n`. Nous créons donc une fonction `depiler(pile)` :

Point de départ	$\rightarrow newPile, objet = depiler(pile) \rightarrow$	Résultat
$pile = \begin{pmatrix} objet_n \\ \vdots \\ objet_1 \\ objet_0 \end{pmatrix}$		$newPile = \begin{pmatrix} objet_n\text{-}1 \\ \vdots \\ objet_1 \\ objet_0 \end{pmatrix}$ Objet dépilé : $objet = objet_n$

En python :
`L = [1, 2, 3]    # Création d'une liste vide`
`x = L.pop()      # Enlève le dernier élément de la liste L et l'affecte à x`		

Cette implémentation est intéressante car les fonctions append(x) et pop() sont de complexité O(1) quelle que soit la taille de la liste.

4.2. Les Files :



Une file (**queue** en anglais) est un ensemble de données dans lequel

- On ajoute les éléments toujours du même côté appelé fin, arrière ou queue (**back** ou **tail** en anglais)
- On supprime les éléments toujours du même côté appelé début, avant ou tête (**front** ou **head** en anglais), côté opposé à l'arrière

On effectue donc les opérations suivantes :

- Enfiler (**enqueue**) : Ajouter un élément à la queue de la file
- Défiler (**dequeue**) : Supprimer un élément à la tête de la file

Le principe des opérations repose sur la règle « premier arrivé, premier sorti » ou FIFO en anglais : « first in, first out ».

Soit une file composée de $n+1$ objets : $file = \begin{pmatrix} objet_0 \\ objet_1 \\ \vdots \\ objet_n \end{pmatrix}$

La fonction `Enfiler()` :

Soit `newObjet` un objet. Nous souhaitons enfiler `newObjet` dans `file`. Nous créons donc une fonction `enfiler(file, newObjet)` qui va inclure l'objet `newObjet` dans la file `file`

Point de départ	$\rightarrow newFile = enfiler(file, newObjet) \rightarrow$	Résultat
$file = \begin{pmatrix} objet_0 \\ objet_1 \\ \vdots \\ objet_n \end{pmatrix}$ Objet à enfiler : $x = New_o bjet$		$newFile = \begin{pmatrix} objet_0 \\ objet_1 \\ \vdots \\ objet_n \\ newObjet \end{pmatrix}$

En python :
`L = []          # Création d'une liste vide`
`L.append(x)     # Ajoute l'objet x à la liste L`		

La fonction `Défiler()` **:**

Nous souhaitons défiler la file d'un élément, soit le premier inséré `objet_0`. Nous créons donc une fonction `defiler(file)` :

Point de départ	$\rightarrow newFile, objet = defiler(file) \rightarrow$	Résultat
$file = \begin{pmatrix} objet_0 \\ objet_1 \\ \vdots \\ objet_n \end{pmatrix}$		$newFile = \begin{pmatrix} objet_1 \\ objet_2 \\ \vdots \\ objet_n \end{pmatrix}$ Objet défilé : $objet = objet_0$

En python :
 `L = [1, 2, 3]     # Création d'une liste vide`
 `x = L.pop(0)      # Enlève le premier élément de la liste L, l'affecte à x`		

Toutefois, si la liste L possède n termes, **`pop(0)`** est de complexité **O(n)**.

Il est utile d'utiliser le librairie « collections » et son sous module « deque ». Ce module implémente des types de données de conteneurs spécialisés permettant un ajout/retrait rapide d'éléments de chaque côté de l'objet (pile ou file ici) utilisé.

Autre code python à préférer :

```
from collections import deque

f = deque()         # Création une « dèque » vide
f.append(x)         # Ajoute l'objet x à droite
f.appendleft(x)     # Ajoute l'objet x à gauche
x = f.pop()         # Enlève l'élément à droite
x = f.popleft()     # Enlève l'élément à gauche
t = len(f)          # Nombre d'éléments de f
```

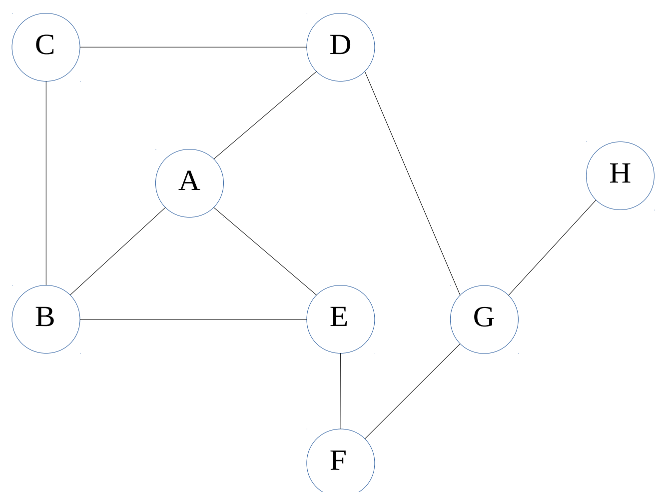
Toutes ces fonctions sont de complexité O(1).

5. Les algorithmes de parcours :

Les algorithmes seront illustrés avec le graphes suivant :

On définit ce graphe à l'aide d'un dictionnaire des listes d'adjacence :

$$G = \{ \\ 'A':['B','D','E'], \\ 'B':['A','C','E'], \\ 'C':['B','D'], \\ 'D':['A','C','G'], \\ 'E':['A','B','F'], \\ 'F':['E','G'], \\ 'G':['D','F','H'], \\ 'H':['G'] \\ \}$$



5.1. Le parcours en largeur :

Le parcours en largeur d’un graphe consiste par explorer un nœud départ, puis ses successeurs, puis les successeurs non explorés des successeurs, etc. Ainsi, à partir d’un nœud départ dep, on liste d’abord les nœuds voisins de dep pour ensuite les explorer un par un et ainsi de suite. Mais une illustration sera sans doute plus représentative :

L’algorithme se déroule donc ainsi :

- Mettre le nœud de départ dans une file et le marquer comme visité
- Tant que la file n’est pas vide :
    - Retirer le premier sommet de la file pour le traiter
    - Mettre tous ses voisins non visités dans la file (à la fin)
    - Marquer le sommet comme visité

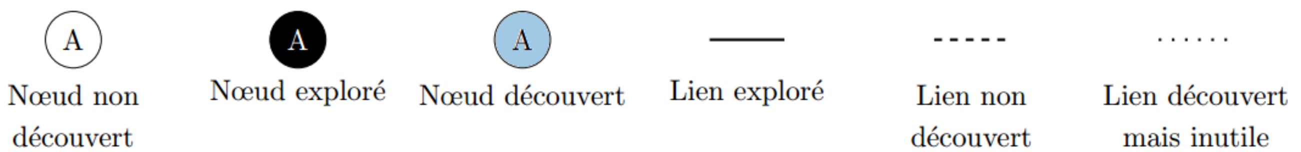


Illustration	État d’avancement	Étapes - commentaires
	File= [A] ***Liste des sommets visités: [ ]***	***On commence le parcours à partir du sommet A.*** ***On enfile A.***
	File= [A,B,D,E] ***File = [B,D,E]*** ***Liste des sommets visités: [A]***	*Exploration de A* *On enfile les voisins de A* ***On défile : on enlève A*** ***On met A dans la liste des sommets visités***
	File= [B,D,E,C] ***File = [D,E,C]*** ***Liste des sommets visités: [A,B]***	*Exploration de B* *On enfile les voisins de B: seul C doit être ajouté car A et E ont déjà été découverts* ***On défile: on enlève B*** ***On met B dans la liste des sommets visités***

	File= [D,E,C,G] **File = [E,C,G]** ***Liste des sommets visités: [A,B,D]***	*Exploration de D* *On enfile les voisins de D: seul G doit être ajouté car C a déjà été découvert* ***On défile: on enlève D*** ***On met D dans la liste des sommets visités***
	File= [E,C,G,F] ***File = [C,G,F]*** ***Liste des sommets visités: [A,B,D,E]***	*Exploration de E* *On enfile les voisins de E: seul F doit être ajouté car A et B ont déjà été découverts* ***On défile: on enlève E*** ***On met E dans la liste des sommets visités***
	File= [C,G,F] ***File = [G,F]*** ***Liste des sommets visités: [A,B,D,E,C]***	*Exploration de C* *Tous les voisins de C ont déjà été découverts* ***On défile: on enlève C*** ***On met C dans la liste des sommets visités***
	File= [G,F] ***File = [F,H]*** ***Liste des sommets visités: [A,B,D,E,C,G]***	*Exploration de G* *On enfile les voisins de G: seul I doit être ajouté car F a déjà été découvert* ***On défile: on enlève G*** ***On met G dans la liste des sommets visités***
	File= [F,H] ***File = [H]*** ***Liste des sommets visités: [A,B,D,E,C,G,F]***	*Exploration de F* *Tous les voisins de F ont déjà été découverts* ***On défile: on enlève F*** ***On met F dans la liste des sommets visités***

	File= [H] ***File = []*** ***Liste des sommets visités: [A,B,D,E,C,G,F,H]***	*<u>Exploration de H</u>* *Tous les voisins de F ont déjà été découverts* ***On défile: on enlève H*** ***On met H dans la liste des sommets visités***
FIN	***On obtient: [A,B,D,E,C,G,F,H]***	

Voici la traduction en pseudo-code du parcours en largeur d'un graphe décrit ci-dessus.

```
Initialisation :
Initialiser la liste file avec le sommet de départ
Initialiser la liste sommets_visites

tant que file n'est pas vide faire :
    S ← premier élément de file
    pour chacun des voisins de S qui ne sont pas déjà dans file ou
    sommets_visites faire :
        Enfiler le voisin à la liste file
    Défiler file et ajouter S à la liste sommets_visites
retourner sommets_visites
```

On propose le code d'une fonction Python nommée `parcours_en_largeur` qui prend en paramètre un graphe et un sommet, qui, à partir de ce sommet, parcourt en largeur un graphe dont la liste d'adjacence est représentée par un dictionnaire. Cette fonction retourne la liste des sommets visités.

```
def parcours_en_largeur(graphe, sommet):
    file = [sommet]
    sommets_visites = []
    while len(file) != 0:
        S = file[0]
        for voisin in graphe[S]:
            if not(voisin in sommets_visites)and not(voisin in file):
                file.append(voisin)
        sommets_visites.append(file.pop(0))
    return sommets_visites
```

Lors de ce parcours, les sommets sont explorés par distance croissante au sommet source.

5.2. Le parcours en Profondeur :

A partir d'un sommet donné, on passe à un des voisins puis à un voisin de ce voisin et ainsi de suite. En l'absence de voisin, on revient au sommet précédent et on passe à un autre de ses voisins.

<u>Parcours en profondeur du graphe exemple à partir du sommet A :</u>

L'algorithme se déroule donc ainsi :

- Mettre le nœud de départ dans une pile et le marquer comme visité
- Tant que la pile n'est pas vide :
    - Retirer le dernier sommet de la pile pour le traiter
    - Mettre tous ses voisins non visités dans la pile (à la fin)
    - Marquer le sommet comme visité

	Pile= [A] ***Liste des sommets visités: [ ]***	***On commence le parcours à partir du sommet A.*** ***On empile A.***
	Pile= [] ***Liste des sommets visités: [A]*** ***Pile = [B,D,E]***	*On dépile A* ***On met A dans la liste des sommets visités*** ***On empile les voisins de A***
	Pile= [B,D] ***Liste des sommets visités: [A,E]*** ***Pile = [B,D,F]***	*On dépile E* ***On met E dans la liste des sommets visités*** ***On empile les voisins de E***
	Pile= [B,D] ***Liste des sommets visités: [A,E,F]*** ***Pile = [B,D,G]***	*On dépile F* ***On met F dans la liste des sommets visités*** ***On empile les voisins de F***
	Pile= [B,D] ***Liste des sommets visités: [A,E,F,G]*** ***Pile = [B,D,H]***	*On dépile G* ***On met G dans la liste des sommets visités*** ***On empile les voisins de G***

	Pile= [B,D] ***Liste des sommets visités: [A,E,F,G,H]*** ***Pile = [B,D]***	*On dépile H* ***On met H dans la liste des sommets visités*** ***Tous les voisins de H sont déjà découverts***
	Pile= [B] ***Liste des sommets visités: [A,E,F,G,H,D]*** ***Pile = [B,C]***	*On dépile D* ***On met D dans la liste des sommets visités*** ***On empile les voisins de D***
	Pile= [B] ***Liste des sommets visités: [A,E,F,G,H,D,C]*** ***Pile = [B]***	*On dépile C* ***On met C dans la liste des sommets visités*** ***Tous les voisins de C sont déjà découverts***
	Pile= [] ***Liste des sommets visités: [A,E,F,G,H,D,C,B]*** ***Pile = []***	*On dépile B* ***On met B dans la liste des sommets visités*** ***L'algorithme s'arrête car la liste des nœuds est vide.***
FIN	***On obtient: [A,E,F,G,H,D,C,B]***	

Voici la traduction en pseudo-code du parcours en profondeur d'un graphe décrit ci-dessus.

```
Initialisation :
Initialiser la liste pile avec le sommet de départ
Initialiser la liste sommets_visites

tant que pile n'est pas vide faire :
    S ← dernier élément de pile
    Dépiler pile et ajouter S à la liste sommets_visites
    pour chacun des voisins de S qui ne sont pas déjà dans pile ou
    sommets_visites faire :
        Ajouter le voisin à la liste pile
retourner sommets_visites
```

On propose le code d'une fonction Python nommée `parcours_en_profondeur` qui prend en argument un graphe et un sommet, qui parcourt en profondeur le graphe à partir de ce sommet, dont la liste d'adjacence graphe est représentée par un dictionnaire. Cette fonction retourne la liste des sommets visités.

```
def parcours_en_profondeur(graphe, sommet):
    pile = [sommet]
    sommets_visites = []
    while len(pile) != 0:
        S = pile.pop()
        sommets_visites.append(S)
        for voisin in graphe[S]:
            if not(voisin in sommets_visites)and not(voisin in pile):
                pile.append(voisin)
    return sommets_visites
```

6. Annexe :

Utilisation de la bibliothèque `networkx` pour afficher les graphes.

Comme toute bibliothèque il convient de l'importer :

```
import networkx as nx
```

Pour l'affichage et le traitement des matrices imorter les bibliothèque suivantes

```
import matplotlib.pyplot as plt
import numpy as np
```

La déclaration d'une matrice d'adjacence avec array de numpy :

```
matrice_adj = np.array([[0,1,1],[1,0,1],[1,1,0]])
```

La déclaration d'un dictionnaire d'adjacence se fait de la facon suivante :

```
dictionnaire_adj = {'A': {'B': {'weight': 1}, 'C': {'weight': 2}}, 'B':
{'A': {'weight': 1}, 'C': {'weight': 1}}, 'C': {'A': {'weight': 2}, 'B':
{'weight': 1}}}
```

Le script suivant permet l'affichage d'un graphe, la variable X est une matrice ou un dictionnaire d'adjacence.

```
graphe = nx.Graph(X)

nouveaux_noms = {0:'A', 1:'B', 2:'C'}
graphe = nx.relabel_nodes(graphe, nouveaux_noms) # utile pour la matrice d'adjacence

position = nx.circular_layout(graphe)

nx.draw(graphe,with_labels=True)#, pos=position)
plt.show()
```

Si le graphe est orienté et/ou pondéré la méthode permettra de le visualiser.

```
graphe = nx.DiGraph(X)
```