Info 6 - Graphes Partie I
Définitions, exemples et construction

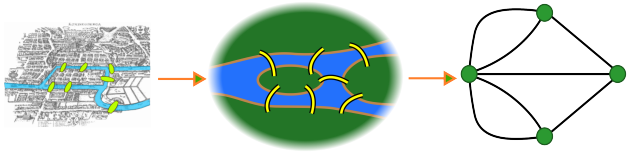
Contents

1 Description "simple" de graphes

1.1 Un premier exemple historique

Les graphes sont des objets permettant de représenter et caractériser des données liées entre elles, telles que peuvent l'être des données issues de réseaux. Les graphes sont avant tout des objets de la théorie mathématique des graphes, initialement développée par Euler en 1735. A cette date, il a effectivement résolu le problème des 7 ponts de Königsberg (aujourd'hui Kaliningrad).

Voici un énoncé du problème : *La ville de Königsberg est construite autour de deux îles situées sur le Pregel et reliées entre elles par un pont. Six autres ponts relient les rives de la rivière à l'une ou l'autre des deux îles, comme représentés sur le plan ci-dessous. Le problème consiste à déterminer s'il existe ou non une promenade dans les rues de Königsberg 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 le Pregel qu'en passant sur les ponts.* La réponse est qu'une telle promenade n'existe pas…



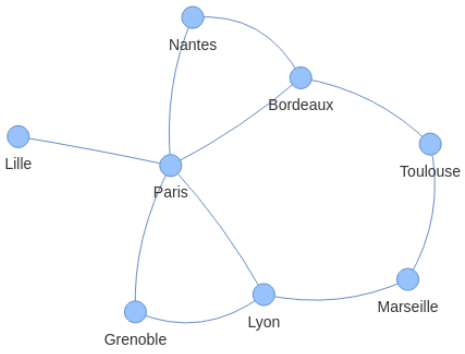
La solution à ce problème est bien plus facile à intuiter en construisant le graphe ci-dessus, dans lequel les *disques*, nommés **sommets**, représentent les terres; et les *traits*, nommés **arêtes**, représentent les ponts. L'abscence de solution au problème est lié au fait que chaque sommet possède un nombre impair d'arêtes.

On peut lire l'article original d'Euler à l'adresse suivante : http://eulerarchive.maa.org/docs/originals/E053.pdf

1.2 Création d'un premier graphe : introduction du vocabulaire et des notations

Un **graphe** est un ensemble de **sommets** (ou **noeuds**) et d'**arêtes**, les arrêtes reliant deux sommets. Deux sommets reliés par une arêtes sont **adjacents**. L'**ordre** d'un graphe est son nombre de sommets.

Visualisons un exemple d'un tel graphe :

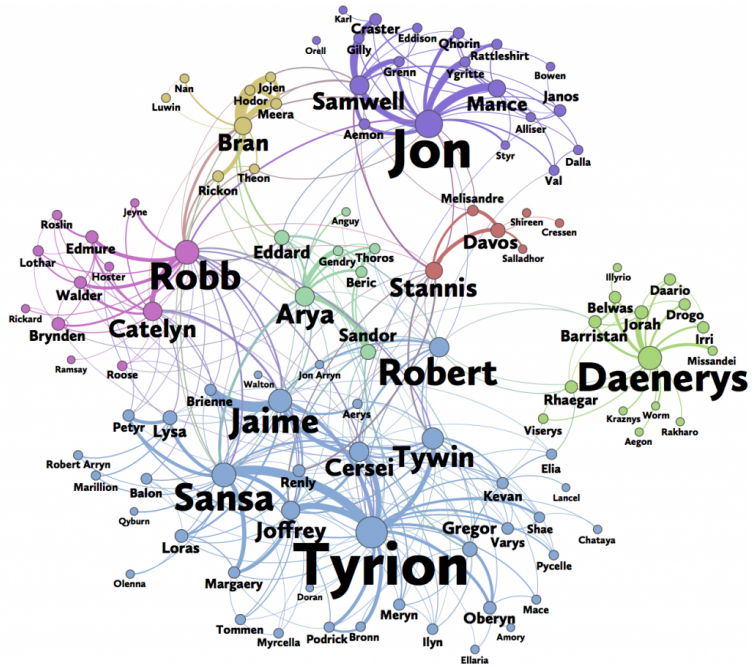


Il s'agit du graphe les liaisons ferroviaires directes entre grandes villes française. On note mathématiquement un tel graphe G = (S,A), avec S les sommets et A les arêtes.

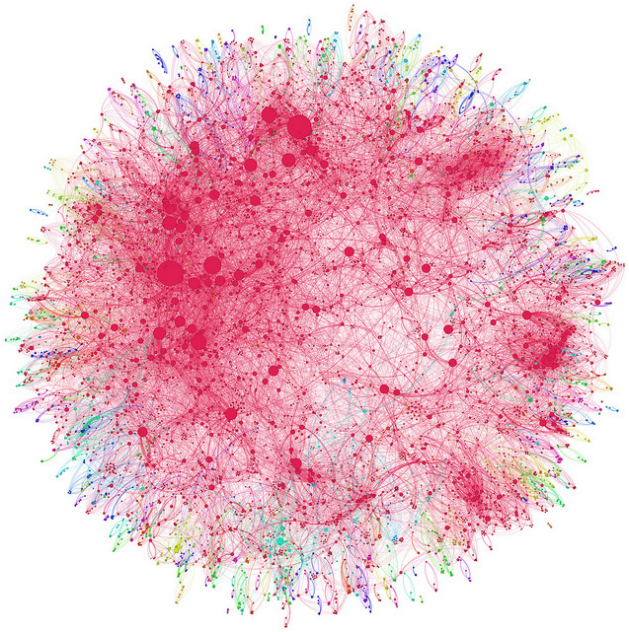
1.3 Quelques exemples réels

Réseaux sociaux : Chaque sommet est une personne, et les arêtes les liens entre personnes.

Un exemple fictionnel, avec le réseau des intéractions sociales entre les personnages de *Game of Throne* :

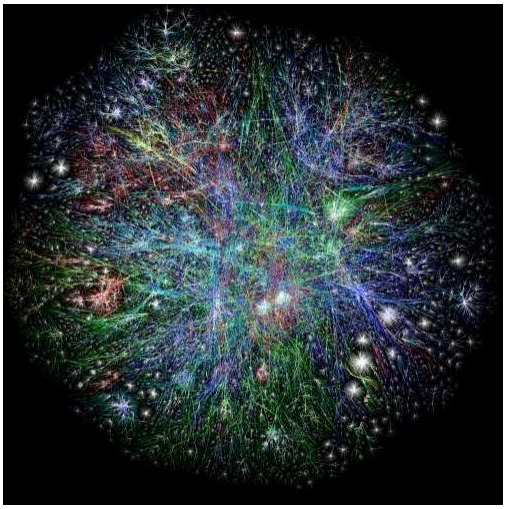


Enfin un dernier exemple : il s'agit des intéractions entre l'ensemble des chercheurs ayant publié des articles sur l'hépatite C, entre 2008 et 2012.

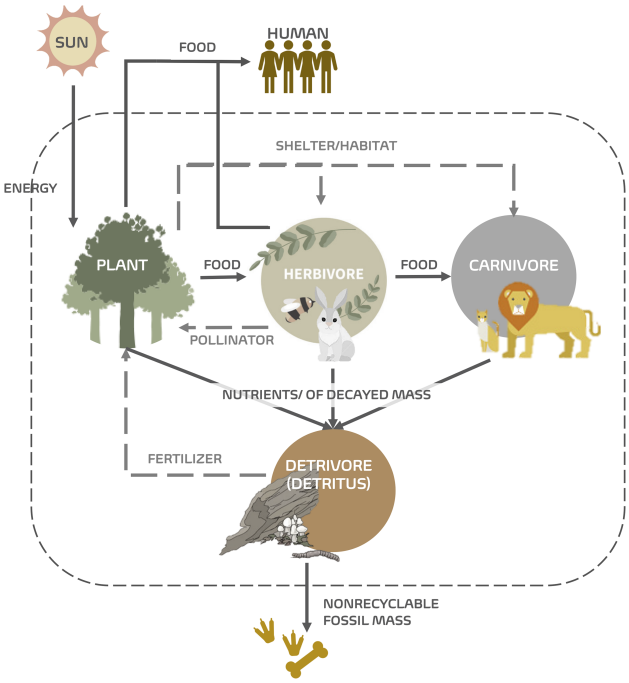


Réseaux de transport : Comme dans l'exemple de la partie précédente, chaque sommet est un arrêt de train / métro / tramway, etc. Les arrêtes sont les liaisons.

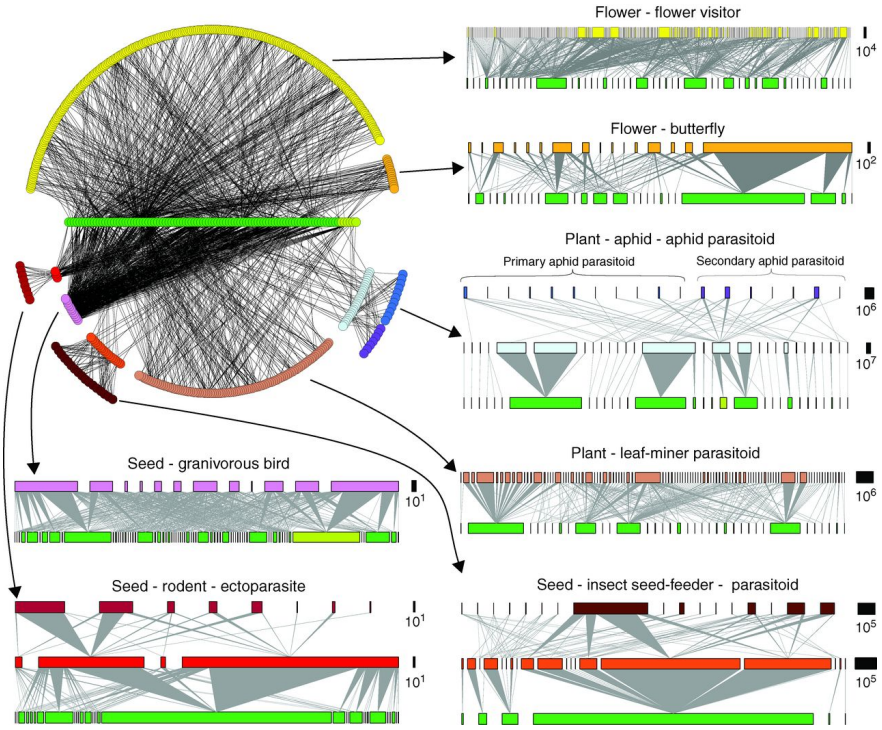
Réseaux technologiques : On peut faire des graphes des réseaux électrique, de communication (téléphone, fibre optique, ...) et bien evidemment du réseau internet (web). Voici par exemple un vue du réseau World Wide Web :



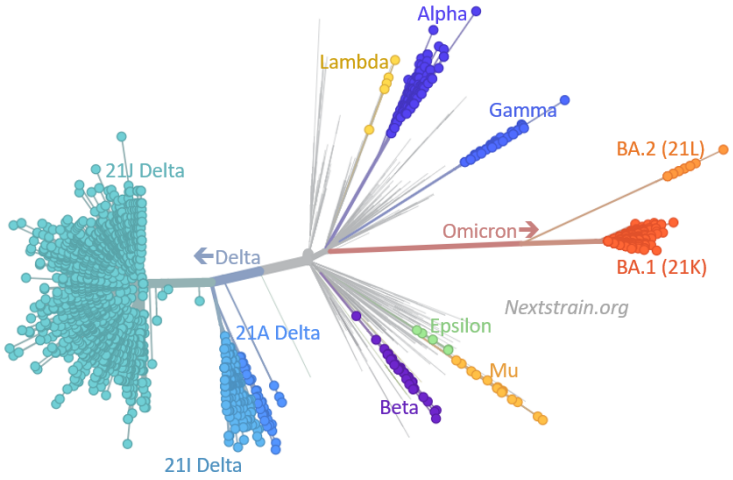
Réseaux écologiques : On peut réprésenter les liens existants entre les différents espèces vivantes sur Terre par des graphes :



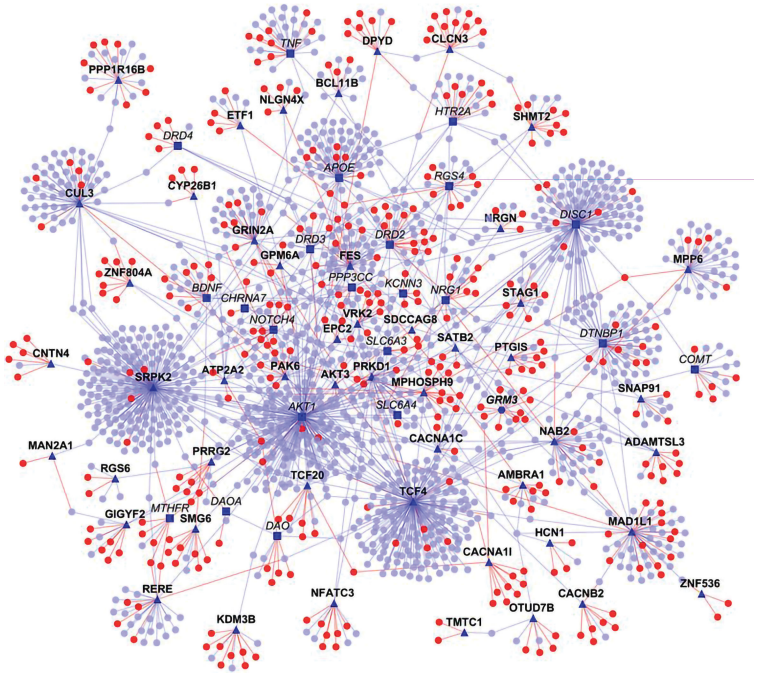
En version bien plus complexe :



Réseaux biologiques : Il existe de nombreux graphes dans les domaines biologiques. En voici certains d'entre eux, en commençant par un graphe des variants du covid : chaque sommet est un variant, et chaque arête un lien de "descendence".



Il existe aussi de nombreux graphes d'intéractions entre protéines, comme ci-dessous. Dans ce graphe, chaque sommet est une protéine intervenant dans le schizophrénie, et chaque arête une intéraction entre ces différentes protéines.

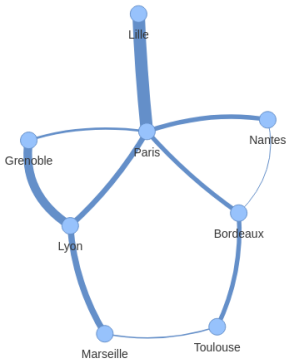


2 Autres caractéristiques et propriétés

Nous allons désormais caractériser ces graphes. Différentes questions seront posées, on peut par exemple trouver de genre de questions au tout début de l'épreuve **CCINP TPC Informatique 2024**

2.1 Pondération

On peut attribuer un "*poids*" à chaque arrête, qui dépend de l'usage du graphe. Par exemple, pour des graphes de transport, on peut mettre comme poids le temps de parcours de chaque arrête. Voici le premier graphe modifié pour tenir compte des temps de trajets :



Ici l'épaisseur du trait est inversement proportionnelle à la durée du parcours : on privilégiera donc les trajet els plus "pais".

Pour un moteur de recherche, sur le graphe du web, on peut mettre comme poids pour les liens (arêtes) entre chaque site le nombre de références et la popularité (voir https://fr.wikipedia.org/wiki/PageRank, algorithme qui a fait le succès de Google). Dans le cas

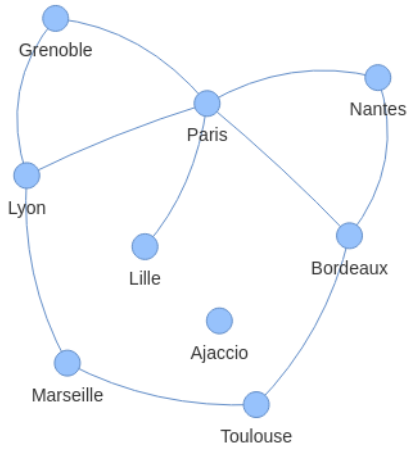
de réseaux sociaux, la pondération est généralement reliée aux liens d'affinité, et toute la difficulté repose à estimer ce *poids* (nombre de messages échangés, …).

2.2 Connexité, cycle et boucle

Le **degré** d'un sommet s, noté $d(s)$, est le nombre d'arêtes dont ce sommet est une extrémité. Un graphe **connexe** est un graphe dans lequel chaque sommet peut être relié à tout autre sommet par une arête ou une suite d'arêtes : on peut dire qu'un graphe connexe est *d'un seul tenant*.

Question 1 : Pour le graphe des liaisons ferroviaires, quel est l'ordre du graphe ? Est-il connexe ? Quel est le sommet de plus grand degré ? Quel sommet peut-on ajouter pour le rendre non connexe ?

Le graphe des laisons ferroviaires est connexe, et d'ordre 8. Paris est le sommet de plus grand degré ($d("Paris") = 5$). Ajoutons un sommet non relié aux autres :



Un **cycle** est une suite d'arêtes consécutives dont les deux sommets extrémités sont identiques. Une **boucle** est une arête ayant pour extremités le même sommet : il s'agit d'un cycle constitué d'une seule arête.

Question 2 : Dans le graphe des liaisons ferroviaires, donner les sommets constituant le plus grand cycle. Existe-t-il une boucle dans ce graphe ? Citer un type de graphe susceptible de comporter une boucle.

Plus grand cycle : Paris - Grenoble - Lyon - Marseille - Toulouse - Bordeaux - Nantes - Paris. Pas de boucle dans le graphe des liaisons ferroviaires, ce serait assez inutile en terme de transport… On peut trouver des boucles sur des graphes du web (lien d'un site vers lui-même, généralement une autre page), ou prédation si cannibalisme.

2.3 Orientation et degrés

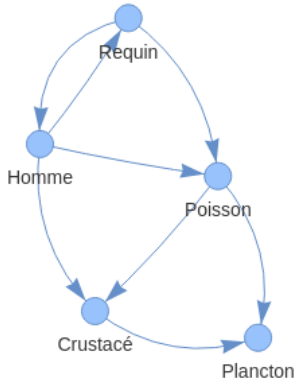
Un graphe est **orienté** si des arêtes sont orientées : ces arêtes sont alors nommées **arcs**, et représentés par une flèche.

Question 3 : Citer des exemples de graphes orientés. Représenter sur papier un tel graphe (très simple).

Graphe du web : les liens entre sites sont directionnels (d'un site vers un autre). Graphes écologiques de prédation (dans la plupart des cas : le requin mange (très peu) d'humains, et les humains mangent malheureusement des requins !). Graphe des variants du Covid, avec les liens de *descendance*. Arbres généalogiques + graphes routiers avec rue à sens unique.

On peut définir, pour un graphe orienté, les **degré sortant** $d_+(s)$ et **degré entrant** $d_-(s)$, qui sont respectivement le nombre d'arcs partant ou arrivant sur ce sommet s. On a alors $d(s) = d_-(s) + d_+(s)$.

Question 4 : On considère le graphe de prédation ci-dessous. Pour le sommet *Homme* du graphe de prédation, quels sont les degrés ?



$d_-(Homme) = 1$, $d_+(Homme) = 2$ et $d(Homme) = 3$

3 Construire et utiliser des graphes

D'un point vue *structure de données*, il existe différents moyens d'implémenter un graphe. Pour simplifier, nous étudierons ici seulement des graphes non orientés.

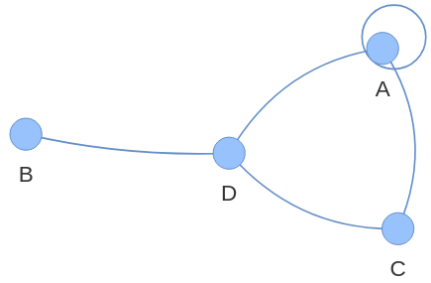
3.1 Première implémentation : en utilisant un dictionnaire d'adjacence

On définit alors le **dictionnaire d'adjacence** ainsi : il s'agit du dictionnaire contenant les *sommets* comme *clefs*, et à chacune de ces clefs, on associe comme *valeur* la liste des *autres sommets liés par une arête*, c'est-à-dire les *sommets adjacents*. Ainsi on peut écrire, pour le graphe des liaisons ferrovières :

[1]:
```
Dict_train = {'Lille': ['Paris'],
              'Paris': ['Lille', 'Nantes', 'Bordeaux', 'Lyon', 'Grenoble'],
              'Nantes': ['Paris', 'Bordeaux'],
              'Bordeaux' : ['Paris', 'Nantes', 'Toulouse'],
              'Toulouse' : ['Bordeaux', 'Marseille'],
              'Marseille' : ['Toulouse', 'Lyon'],
              'Grenoble' : ['Paris', 'Lyon'],
              'Lyon' : ['Paris', 'Marseille', 'Grenoble']}
```

Ce dictionnaire est fourni dans le fichier *I6_Graphes_p1.py*, aisni que la fonction `Dict_to_graphe` permettant de le visualiser à l'aide de la bibliothèque *NetworkX*.

Question 5 : On considère le graphe "simple" ci-dessous. Donner le code python permettant d'écrire son dictionnaire d'adjacence, puis le vérifier en visualisant le graphe (grâce à la fonction `Dict_to_graphe` fournie).



[17]:
```
# Ecriture sommet par sommet :
Dict_ABCD = dict()
Dict_ABCD['A'] = ['A', 'C', 'D']
Dict_ABCD['B'] = ['D']
Dict_ABCD['C'] = ['A', 'D']
Dict_ABCD['D'] = ['A', 'B', 'C']

# Ou bien écriture en une seule ligne :
Dict_ABCD = {'A': ['A', 'C', 'D'], # sommets adjacents de A
             'B': ['D'], # sommets adjacents de B
             'C': ['A', 'D'], # sommets adjacents de C
             'D': ['A', 'B', 'C']} # sommets adjacents de D

print(Dict_ABCD)
```

```
{'A': ['A', 'C', 'D'], 'B': ['D'], 'C': ['A', 'D'], 'D': ['A', 'B', 'C']}
```

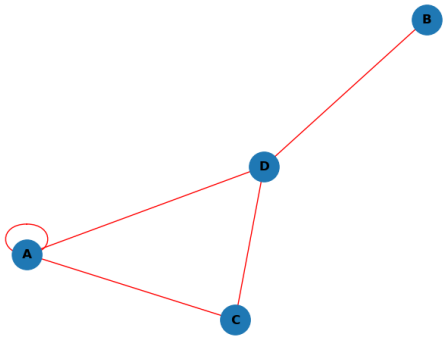
Remarque : on rapelle ici qu'un dictionnaire ***n'est pas ordonné****. L'ordre dans lequel on rentre les sommets (et même l'ordre des sommets adjacents) n'a aucune importance. Pour faciliter la lecture et la correction, ils seront néanmoins rentrés ici dans l'ordre alphabétique.*

Visualisons ce graphe :

[18]:
```
import networkx as nx
import matplotlib.pyplot as plt

def Dict_to_graph(Dict):
    """ Permet de représenter, à l'aide de NEtworkX, le graphe lié au␣
 ↪dictionnaire d'adjacence `Dic`
    donné sous la forme {Sommet : liste des sommets adjacents}"""
    global graphe
    graphe = nx.Graph()

    for sommet in Dict :
        graphe.add_node(sommet)
        for sommet_adj in Dict[sommet]:
```

```
            graphe.add_edge(sommet, sommet_adj)

    nx.draw(graphe, node_size=800, edge_color='red',␣
↪with_labels=True,font_weight='bold')
    plt.show()
```

[19]:
```
Dict_to_graph(Dict_ABCD)
```



3.2 Utiliser un dictionnaire d'adjacence

Nous allons dans cette partie écrire des fonctions qui permettent de déterminer certaines propriétés de graphe, losque ceux-ci ont une structure de dictionnaire d'adjacence.

Question 6 : Ecrire au moins les deux premières fonctions suivantes, pour des graphes non orientés qui ont pour structure un dictionnaire d'adjacence tel que décrit précédemment (`dict_graphe = {sommet : liste des sommets adjacents}`). On testera ensuite ces fonctions sur les deux graphes définis précedemment (liaisons ferrovières et ABCD) :

- une fonction `ordre(dict_graphe)` qui retourne l'ordre du graphe, soit le nombre de sommets contenu dans `dict_graphe`;
- une fonction `test_edge(sommet1, sommet2, dict_graphe)` qui retourne `True` s'il existe, dans le graphe implémenté par le dictionnaire `dict_graphe`, une arrête entre les sommets `sommet1` et `sommet2`;
- une fonction `ajout_sommet(sommet, list_sommets_adj, dict_graphe)` qui ajoute le sommet `sommet` au graphe implémenté par le dictionnaire `dict_graphe`, avec `list_sommets_adj` la liste de ses sommets adjacents ;
- une fonction `nb_aretes(dict_graphe)` qui retourne le nombres d'arêtes du graphe dont `dict_graphe` est l'implantation sous forme d'un dictionnaire (on pourra dans un premier temps considéré qu'il n'y a pas de boucles).
- une fonction `max_1_correspondance(sommet1,sommet2,dict_graphe)` qui retourne `True` si on peut relier les sommets sommet1 et sommet2 à l'aide de deux arêtes ou moins.

[16]:
```
def ordre(dict_graphe):
    """Retourne le nombre de sommet d'un graphe, à partir de l'écriture d'un␣
↪graphe sous forme d'un dictionnaire d'adjacence :
```

```
    dict_graphe = {sommet : liste des sommets adjacents} """
    return len(dict_graphe)

def test_edge(sommet1, sommet2, dict_graphe):
    """Retourne la variable booléenne True s'il existe une arête entre le␣
 ↪sommet 1 et le sommet 2, à partir de l'écriture d'un graphe sous forme d'un␣
 ↪dictionnaire d'adjacence :
    dict_graphe = {sommet : liste des sommets adjacents} """

    # Liste des voisins du sommet1:
    Voisins_1 = dict_graphe[sommet1]
    # Si sommet2 est dans cette liste :
    if sommet2 in Voisins_1:
        return True
    else :
        return False

    # version plus efficace, en 1 ligne :
    # return sommets2 in dict_graphe[sommet1]

def ajout_sommet(sommet, list_sommets_adj, dict_graphe):
    """Ajoute le sommet sommet ainsi que les arêtes list_sommets_adj au graphe␣
 ↪dict_graphe, implémenté sous forme d'un dictionnaire d'adjacence :
    dict_graphe = {sommet : liste des sommets adjacents} """

    # Ajout du nouveau sommet et de ses sommets adjacents :
    dict_graphe[sommet] = list_sommets_adj

    # A ne pas oublier : il faut ajouter ce nouveau sommet comme sommet␣
 ↪adjacent dans les autres sommets :
    for sommet_autre in list_sommets_adj :
        dict_graphe[sommet_autre].append(sommet)

def nb_aretes(dict_graphe):
    """Retourne l'ordre d'un graphe non orienté, à partir de l'écriture d'un␣
 ↪graphe sous forme d'un dictionnaire d'adjacence :
    dict_graphe = {"sommet" : liste des sommets adjacents} """
    somme = 0 # on va sommer le nombre de relation d'adjcence
    for sommet in dict_graphe : # pour chaque sommet
        somme += len(dict_graphe[sommet]) # on ajoute le nombre de sommets␣
 ↪adjacents
    return somme / 2 # on a compté chaque arête 2 fois !

def max_1_correspondance(sommet1,sommet2,dict_graphe):
    """Retourne la variable booléenne True s'il existe une ou deux arêtes␣
 ↪consécutives reliant le sommet sommet1 au sommet sommet2, à partir de␣
 ↪l'écriture d'un graphe sous forme d'un dictionnaire d'adjacence :
```

```
    dict_graphe = {"sommet" : liste des sommets adjacents} """


    if sommet2 in dict_graphe[sommet1]: #si sommet2 est directement adjacent de␣
 ↪sommet1
        return True
    else:
        for sommet_int in dict_graphe[sommet1] : # On cherche avec 1␣
 ↪correspondance, la correspondance étant effectué au sommet_int
            if sommet2 in dict_graphe[sommet_int] : # Si sommet2 est voisin du␣
 ↪sommet_int, c'est bon !
                return True
            # Autre version :
            #for sommet_int in dict_graphe :
            #    if sommet1 and sommet2 in dict_graphe[sommet_int]:
            #        return True
    return False
```

[17]:
```
print("Nombre de sommets du graphe des liaisons ferroviaires : ",␣
 ↪ordre(Dict_train))
print("Existe-t-il une liasons ferroviaire directe entre Bordeaux et Lyon ? ",␣
 ↪test_edge("Bordeaux", "Lyon", Dict_train))
print("Nombre d'arrêtes du graphe des liaisons ferroviaires : ",␣
 ↪nb_aretes(Dict_train))
print("Ajout d'un nouveau sommet : Montpellier")
ajout_sommet("Montpellier", ["Marseille", "Toulouse"], Dict_train)
print(Dict_train)

# Retablissons le graphe originel : on enlève le sommet qu'on vient de␣
 ↪rajouter, pour la suite :
Dict_train.pop("Montpellier")
for sommet in Dict_train :
    if "Montpellier" in Dict_train[sommet]:
        Dict_train[sommet].remove("Montpellier")
print(Dict_train)

print("Peut-on relier Bordeaux à Lyon avec max une correspondance ? ",␣
 ↪max_1_correspondance("Bordeaux", "Lyon", Dict_train))
```

```
Nombre de sommets du graphe des liaisons ferroviaires :  8
Existe-t-il une liasons ferroviaire directe entre Bordeaux et Lyon ?  False
Nombre d'arrêtes du graphe des liaisons ferroviaires :  10.0
Ajout d'un nouveau sommet : Montpellier
{'Lille': ['Paris'], 'Paris': ['Lille', 'Nantes', 'Bordeaux', 'Lyon',
'Grenoble'], 'Nantes': ['Paris', 'Bordeaux'], 'Bordeaux': ['Paris', 'Nantes',
'Toulouse'], 'Toulouse': ['Bordeaux', 'Marseille', 'Montpellier'], 'Marseille':
['Toulouse', 'Lyon', 'Montpellier'], 'Grenoble': ['Paris', 'Lyon'], 'Lyon':
```

```
['Paris', 'Marseille', 'Grenoble'], 'Montpellier': ['Marseille', 'Toulouse']}
{'Lille': ['Paris'], 'Paris': ['Lille', 'Nantes', 'Bordeaux', 'Lyon',
'Grenoble'], 'Nantes': ['Paris', 'Bordeaux'], 'Bordeaux': ['Paris', 'Nantes',
'Toulouse'], 'Toulouse': ['Bordeaux', 'Marseille'], 'Marseille': ['Toulouse',
'Lyon'], 'Grenoble': ['Paris', 'Lyon'], 'Lyon': ['Paris', 'Marseille',
'Grenoble']}
Peut-on relier Bordeaux à Lyon avec max une correspondance ?  True
```

3.3 Une autre implémentation : la liste d'adjacence

Le dictionnaire d'adjacence est un outil pratique car très "lisible" : en effet, pour les sommets, on peut utiliser des noms, chiffres, lettres, … ou même des objets plus complexes comme des tuples (voir *Info 7 : Théorie des jeux*). Cela rend les codes assez lisibles et plus facilement compréhensibles. cependant, le dictionnaire n'est pas la structure de donnée la plus efficace en terme de stockage et de temps d'accès. C'est pourquoi on peut aussi utiliser des listes python pour stocker un graphe.

Dans le cas d'une liste d'adjacence, il est nécessaire de numéroter les `i` sommets de 0 à $n-1$. La liste d'adjacence est alors est la liste `LA`, telle que `LA[i]` est la liste des sommets voisins du sommet `i`. Par convention, les voisins énumérés dans `LA[i]` seront listés dans l'ordre croissant.

Question 7 : Écrire le code python premettant de créer la liste d'adjacence pour le graphe ABCD. On numérotera les sommets ainsi : A:0, B:1, C:2 et D:3.

[31]:
```
LA_ABCD = [[0, 2, 3], # A = 0, C = 2 et D = 3 sont sommets adjacents de A = 0
           [3], # D = 3 est sommet adjacent de B = 1
           [0, 3], # A = 0 et D = 3 sommets adjacents de C = 2
           [0, 1, 2]] # A = 0, B = 1 et C = 2 sont sommets adjacents de D = 3
```

Dans le fichier *I6_Graphes_p1.py*, on donne le code pour créer la liste d'adjacence pour le graphe des liaisons ferrovières (on a numéroté les villes selon leur ordre alphabétique).

Question 8 : Ecrire la fonction `max_liaisons(LA)` qui renvoie, à partir d'une liste d'adjacence, le numéro du sommet possédant le plus d'arrêtes. Le tester avec le graphe ABCD et celui des liaisons ferrovières.

[36]:
```
def max_liaisons(LA):
    max = 0
    n = len(LA)
    for sommet in range(n):
        nb_liaisons = len(LA[sommet])
        if nb_liaisons > max :
            max = nb_liaisons
    return sommet
```

Testons avec le graphe des liaisons ferrovières, numéroté les villes dans l'ordre alphabétique :

[33]:
```
Villes = sorted(["Lille", "Paris", "Marseille", "Lyon", "Bordeaux", "Grenoble",␣
 ↪"Nantes", "Toulouse"])
[(i, Villes[i]) for i in range(len(Villes))]
```

[33]:
```
[(0, 'Bordeaux'),
 (1, 'Grenoble'),
 (2, 'Lille'),
 (3, 'Lyon'),
 (4, 'Marseille'),
 (5, 'Nantes'),
 (6, 'Paris'),
 (7, 'Toulouse')]
```

Ainsi la liste d'adjacence s'écrit ainsi :

[34]:
```
LA_train = [[5, 6, 7], # Sommets adjacents du sommet 0, Bordeaux
            [3, 6],    # Sommets adjacents du sommet 1, Grenoble
            [6],       # Sommets adjacents du sommet 2, Lille
            [1, 4, 6],
            [3, 7],
            [0, 6],
            [0, 1, 2, 3, 5],
            [0, 4]]
```

Et :

[37]:
```
max_liaisons(LA_train)
```

[37]:
```
7
```

Ce qui correspond à la ville :

[38]:
```
Villes[max_liaisons(LA_train)]
```

[38]:
```
'Toulouse'
```

3.4 Une dernière implémentation : la matrice d'adjacence

Comme pour la liste d'adjacence, la matrice d'adjacence repose sur un graphe pour lequel les n sommets sont numérotés de 0 à $n-1$. Commençons par le cas simple de la matrice d'adjacence pour un graphe non pondéré : à la ligne i et à la colonne j, elle contient 0 s'il n'y a pas d'arête entre les sommets numérotés i et j, et 1 sinon.

Question 9 : Ecrire la matrice d'adjacence `MA_ABCD` (liste de liste) pour l'exemple du graphe ABCD. Que peut-on dire de la matrice d'adjacence d'un graphe non pondéré et non orienté ?

[39]:
```
LA_ABCD = [[1, 0, 1, 1],
           [0, 0, 0, 1],
           [1 ,0, 0, 1],
           [1, 1, 1, 0]]
```

La matrice d'adjacence d'un graphe non pondéré et non orienté est forçement symétrique !

Question 10 : Ecrire une fonction `mat_to_list(M)` qui, à partir d'une matrice d'adjacence M, construit la liste d'adjacence. Pour tester cette fonction, on fournit dans le fichier *I6_Graphes_p1.py*

la matrice d'adjacence pour le graphe des liaisons ferrovières.

[44]:
```
def mat_to_list(M) :
    """Construit une liste d'adjacence (liste de liste),
    à partir d'une matrice d'adjacence en format liste de liste.
    Choix : les sommets sont listés dans l'ordre croissant
    """
    # nb de sommets
    n = len(M) # matrice matrice carré nxn
    L_adj = []

    for i in range(n) : # pour la ième ligne de la matrice
        L_i = [] # liste des sommets adjacents au sommet i
        for j in range(n) : # pour la jème colonne de la matrice
            if M[i][j] == 1 : # si le sommet j est bien adjacent avec le sommet␣
↪i :
                L_i.append(j)
        L_adj.append(L_i)
    return L_adj
```

[42]:
```
MA_train = [[0, 0, 0, 0, 0, 1, 1, 1], # sommets adjacents de Bordeaux
            [0, 0, 0, 1, 0, 0, 1, 0], # sommets adjacents de Grenoble
            [0, 0, 0, 0, 0, 0, 1, 0], # sommets adjacents de Lille
            [0, 1, 0, 0, 1, 0, 1, 0], # sommets adjacents de Lyon
            [0, 0, 0, 1, 0, 0, 0, 1], # sommets adjacents de Marseille
            [1, 0, 0, 0, 0, 0, 1, 0], # sommets adjacents de Nantes
            [1, 1, 1, 1, 0, 1, 0, 0], # sommets adjacents de Paris
            [1, 0, 0, 0 ,1, 0, 0 ,0]] # sommets adjacents de Toulouse
```

[46]:
```
mat_to_list(MA_train)
```

[46]:
```
[[5, 6, 7], [3, 6], [6], [1, 4, 6], [3, 7], [0, 6], [0, 1, 2, 3, 5], [0, 4]]
```

Dans le cas de **graphes pondérés**, il est évidemment intéressant d'ajouter le poids comme information. Prenons l'exemple du graphe des liaisons ferrovières : dans le fichier *I6_Graphes_p1.py*, on fournit le dictionnaire `Dict_train_pond` d'adjacence de ce graphe, en ajoutant le *poids* de chaque arrête. Ainsi, chaque sommet adjacent a été remplacé par un tuple contenant le sommet et le poids de l'arête (ici la durée du trajet).

[52]:
```
Dict_train_pond = {'Lille': [('Paris', 62)],
                   'Paris': [('Lille', 62), ('Nantes', 129), ('Bordeaux', 129),␣
↪('Lyon', 114), ('Grenoble', 183)],
                   'Nantes': [('Paris', 129), ('Bordeaux', 253)],
                   'Bordeaux' : [('Paris', 129), ('Nantes', 253), ('Toulouse',␣
↪129)],
                   'Toulouse' : [('Bordeaux', 129), ('Marseille', 228)],
                   'Marseille' : [('Toulouse', 228), ('Lyon', 104)],
                   'Grenoble' : [('Paris', 183), ('Lyon', 83)],
```

```
                    'Lyon' : [('Paris', 114), ('Marseille', 104), ('Grenoble',␣
 ↪83)]}
```

Pour un tel graphe pondéré, l'utilisation d'un dictionnaire d'adjacence est lourde : c'est pourquoi, hgénéralement, on utilise plutôt une matrice d'adjacence. Par rraport au même graphe non pondéré, on remplace les *1* de la matrice d'adjacence par le *poids* de chaque arrête. Mais par quoi remplace-t-on les 0, qui renseigne sur l'abscence d'arrête ? Cela dépend des graphes... On peut, au choix, laisser des 0, mettre des -1 ou même des ∞. C'est ce dernier cas que nous allons utiliser pour la matrice du graphe pondéré des liaisons ferroviaires : en effet, le cas d'abscence d'arête et donc de liaison correspond *physiquement* à un temps de trajet infini.

Question 11 : Compléter le code du fichier *I6_Graphes_p1.py* afin de créer la matrice d'adjacence`MA_train_pond` pour le graphe pondéré des liaisons ferroviaires, en partant du dictionnaire `Dict_train_pond` :

[58]:
```
from math import inf

# Liste des villes dans l'ordre alphabétique :
Villes = sorted([ville for ville in Dict_train_pond])
n = len(Villes)

# Création de la matrice :
MA_train_pond =[]

# Remplissage :
for sommet_i in range(n) :  # Pour le i-ème sommet, on remplit la i-ème ligne␣
 ↪de la matrice
    # Récupération du nom de la ville correspondant :
    ville = Villes[sommet_i]
    # Création de la i_ème ligne de la matrice, avec par défaut aucun sommet␣
 ↪adjacent (uniquement des poids infini) :
    MA_train_pond.append([inf]*n)
    # Parcours des sommets ou villes adjacentes sous forme de tuples␣
 ↪{ville_adj, poids} :
    for tuple_adj in Dict_train_pond[ville] :
        # On récupère  le nom de la ville adjacente et le poids de la liaison
        ville_adj, poids = tuple_adj
        # On recherche le numéro de la ville correspondante:
        sommet_adj = Villes.index(ville_adj)
        # On modifie alors le poids :
        MA_train_pond[sommet_i][sommet_adj] = poids

display(MA_train_pond)
```

```
[[inf, inf, inf, inf, inf, 253, 129, 129],
 [inf, inf, inf, 83, inf, inf, 183, inf],
 [inf, inf, inf, inf, inf, inf, 62, inf],
 [inf, 83, inf, inf, 104, inf, 114, inf],
```

```
 [inf, inf, inf, 104, inf, inf, inf, 228],
 [253, inf, inf, inf, inf, inf, 129, inf],
 [129, 183, 62, 114, inf, 129, inf, inf],
 [129, inf, inf, inf, 228, inf, inf, inf]]
```

Remarque : on pourrait "améliorer" cette matrice en plaçant des 0 sur la diagonale. En effet, le temps de trajet d'une ville vers elle-même est nul.

Quels sont les avantages (ou inconvénients) de l'écriture sous forme d'un dictionnaire par rapport à l'écriture d'une matrice ? Le choix d'un dictionnaire ou d'une matrice pourra dépendre de plusieurs facteurs :

- Les matrices permettent un accès *direct* (c'est-à-dire en $\mathcal{O}(1)$) à chacun de ses éléments : on peut ainsi accéder et/ou modifier de façon très efficace les arêtes et donc la structure du graphe. Pour un dictionnaire, l'accès et la modification se fait en $\mathcal{O}(n)$: il faut en effet parcourir les sommets dans le dictionnaire pour trouver une arête précise.
- Les dictionnaire sont cependant plus *pratique* et permettent une lecture plus aisée car ils permettent de stocker d'autres informations que des nombres, comme des noms de villes par exemple. Si l'accès à un élément de matrice est direct, il peut exister un lien moins direct entre le nom du sommet et son index…