Analyse de réseau avec Python et NetworkX

Laurent Beauguitte

Analyse de réseau avec Python et NetworkX

Laurent Beauguitte

07/13/2023

Sommaire

Introduction **3**
Installer Python 4
Deux mots sur Python 4
Documentation et installation du module 5

1 Importer ses données **7**

2 Filtrer, sélectionner **11**
2.1 Extraire les composantes connexes 11
2.2 Filtrage 13
2.3 Agrégation 17

3 Mesurer **18**
3.1 Mesures globales 18
3.2 Mesures locales 22

4 Partitionner **28**
4.1 Cliques et k-cores 28
4.2 Blockmodel 29
4.3 Détection de communautés 29

5 Visualiser ses données **32**

6 Pour aller plus loin **37**
6.1 Avec Python 37
6.2 Avec NetworkX 37

Introduction

Ceci est un tutoriel d'introduction à l'analyse de réseau avec Python centré sur le seul module `NetworkX`. Ce module étant très riche, toutes les fonctions ne sont pas évoquées ici.

Ce tutoriel s'adresse aux personnes :

- ayant une connaissance correcte de l'analyse de réseau. Si vous êtes débutante[1], je renvoie au petit guide pratique d'initiation à l'analyse de réseau disponible en ligne (Beauguitte (2023)) ;
- ayant l'habitude d'un logiciel autre, que ce logiciel soit à interface graphique (Cytoscape, Gephi, Pajek, Tulip, etc.) ou non (R) ;
- curieuses de savoir ce qu'il est possible de faire avec Python.

Le principal module d'analyse de réseau en Python est `NetworkX` (Hagberg et al. (2008)). Le module `igraph`, couramment utilisé pour l'analyse de réseau avec R, est également disponible en Python mais semble beaucoup moins utilisé. Des modules plus confidentiels sont disponibles et seront évoqués dans des billets ultérieurs.

Ce document ne prétend pas être exhaustif mais vise à présenter les points suivants :

- importer ses données (sommets et liens avec attributs)
- manipuler un graphe ;
- analyser son réseau (mesures globales, mesures portant sur les sommets, cliques et communautés, etc.) ;
- visualiser son réseau.

Étant débutant en Python, ce support est amené à être amélioré et enrichi dans les mois (années ?…) à venir. Le code proposé n'est pas nécessairement le plus efficace ni le plus court. Mais il fonctionne. Le script `2023_07_networkx.py` permet de reproduire toutes les opérations décrites dans ce tutoriel.

Données et script sont disponibles sur le git hub de l'auteur. La version html de ce document est accessible à cette adresse.

[1] Je n'utilise pas l'écriture épicène, j'écris le plus souvent au féminin et utilise l'accord de proximité.

Installer Python

Si vous n'avez jamais utilisé Python, le plus simple est sans doute d'installer la distribution Anaconda. C'est un peu une usine à gaz : ça installe plein de logiciels dont on n'a pas nécessairement besoin et avec des versions qui ne sont pas toujours les plus récentes mais au moins ça marche. Une fois Anaconda installée, ne le lancez pas : c'est lourd, lent et inutile. Il est plus rapide de lancer le `Anaconda Power Shell Prompt` puis, dans l'invite de commande, de taper `spyder`. L'IDE Spyder, assez similaire à RStudio, permet de créer ses scripts en Python dans un environnement agréable (rubrique d'aide, autocomplétion, feneêtre graphique, liste des objets importés ou créés, etc.).

Il est possible d'installer une version plus légère d'Anaconda, Miniconda. Il est également possible d'installer Python et Spyder de façon autonome. Les règlages pour que les logiciels fonctionnent les uns avec les autres peuvent être un peu plus longs. Idem si vous souhaitez utiliser Python dans un environnement RStudio - ce qui est le cas de ce tutoriel rédifgé en Quarto.

Dernier conseil : n'hésitez pas à tout désintaller quand ça bloque avant de réinstaller les briques une par une en feuilletant bien la documentation…

Deux mots sur Python

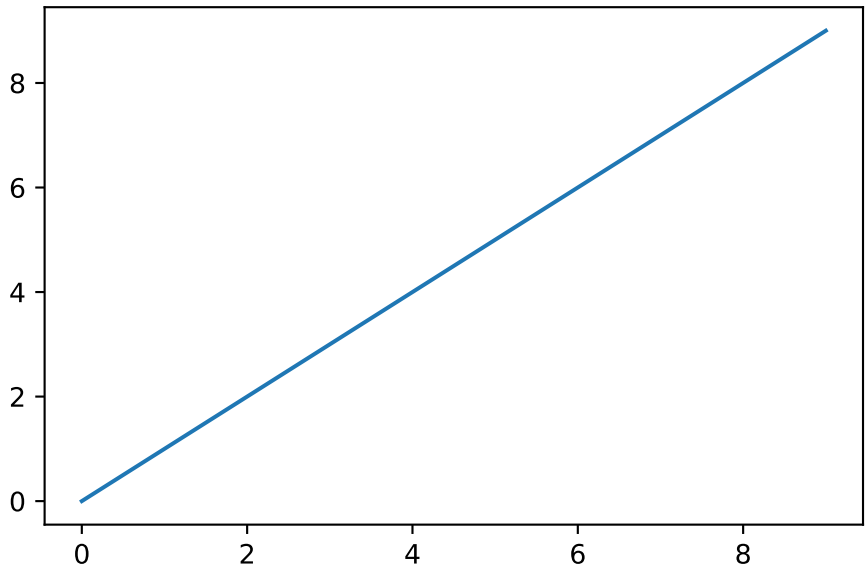
Si vous n'avez jamais utilisé Python, les informations suivantes devraient vous permettre de tester le script .py accompagnant ce tutoriel.

Python, comme R, est un logiciel modulaire. Il y a la base et des modules nécessaires pour faire telle ou telle manipulation. Comme R, certains modules sont quasi indispensables et très souvent utilisés ; l'immense majorité n'a aucun intérêt. L'ensemble des modules disponibles est accessible sur le Pypi. Un "bon" module est un module régulièrement mis à jour, avec une documentation intelligible et pour lequel on trouve des tutoriels en ligne. S'il est maintenu par plus d'une personne, c'est mieux.

Deux étapes pour les modules qui ne sont pas installés par défaut : les installer sur votre disque dur, les charger pour la session. Lorsqu'on charge un module, l'usage veut qu'on lui donne un nom abrégé. Certaines abréviations se retrouvent partout et je les utilise aussi. Pourquoi abréger les noms des modules ? Parce que pour appeler une fonction d'un module, on tape le nom du module, un point, puis le nom de la fonction.

Ce qui va donner des choses comme ça :

```
import matplotlib.pyplot as plt
x  = range(0,10)
plt.plot(x)
```



Je charge la librairie `matplotlib.pyplot` et je lui donne un nom abregé (`plt`). Je crée une suite d'entiers de 0 à 10. J'appelle la fonction `plot` du module `matplotlib.pyplot` pour visualiser x.

Dans `Spyder`, pour exécuter une ou plusieurs lignes de code, le plus simple est de la ou les sélectionner puis de taper sur F9 (équivalent du Ctrl + Entrée de RStudio). Si vous avez des instructions avec des retours à la ligne (arguments séparés par des virgules ou boucle avec : et indentation), sélectionnez l'ensemble des lignes avant d'exécuter le code. Idem pour les figures : sélectionner tout ce qui concerne la figure (titre, étiquettes des abscisses et ordonnées, etc.).

Documentation et installation du module

La page du module **NetworkX, Network Analysis with Python**, https://networkx.org/, permet d'accéder à toute la documentation nécessaire pour prendre en main le module. Mais dès que l'on va vouloir réaliser une opération précise, surtout si on n'est pas familier du fonctionnement des objets dans cet environnement logiciel, parcourir stackoverflow est presque obligatoire. Sur stackoverflow, faites attention à la date des sujets : certaines solutions proposées peuvent être obsolètes.

```
# installer le module
# pip install networkx
# si distribution ana ou miniconda
```

```
# conda install package-name

# charger les modules utiles
import pandas as pd                # manipulation de tableaux
import networkx as nx              # analyse de réseau
import numpy as np                 # statistiques
import matplotlib.pyplot as plt    # visualisation

# aide sur une fonction
# ?nom_module.nom_fonction
?pd.DataFrame.set_index
?nx.find_cliques
```

Le nombre de fonctions disponibles est impressionnant mais on obtient régulièrement des messages indiquant que telle fonction est “deprecated and will be removed”.

---

1 Importer ses données

Le jeu de données utilisé tout au long de ce document est l'extrait concernant les Bouches-du-Rhône de la base 2017 des mobilités scolaires intercommunales de l'INSEE.

La table **som_d13.csv** contient les attributs suivants :

- **CODGEO** : code INSEE de la commune ;
- **P19_POP** : population résidente en 2019 (entiers) ;
- **SUPERF** : superficie (flottants) ;
- **SUP_QUALI** : superficie inférieure ou supérieure à la moyenne départementale (chaîne de caractères). Des données ont été volontairement supprimés pour voir comment gérer les données manquantes ;
- **NOM** : nom de la commune ;
- **MARS** : variable booléenne permettant de savoir si la commune est un arrondissement de Marseille (1) ou non (0).

La table **liens_d13.csv** contient les attributs suivants :

- **Origine** : code INSEE de la commune de départ ;
- **Arrivee** : code INSEE de la commune d'arrivée ;
- **weight** : flux d'élèves entre communes (flux > 100).

Ces deux fichiers encodés en utf-8 permettent de créer un réseau simple, orienté et valué. Les réseaux autres (bimodaux, multiplexes, avec boucles) ne sont pas évoqués dans ce document.

Pour l'importation et la manipulation des données, soit vous savez utiliser Python et vous n'avez pas besoin de lire les paragraphes qui suivent, soit vous débutez en Python et utiliser le module `pandas` est recommmandé.

```
sommets = pd.read_csv("data/som_d13.csv", sep = ";")
liens = pd.read_csv("data/liens_d13.csv", sep = ";")
```

Avant de transformer ces tables en un réseau, il est prudent de contrôler le typage par défaut qu'a choisi `pandas` lors de l'importation avec la fonction `dtypes`. Il faut en effet que les codes INSEE soient considérés comme des chaînes de caractères (`str`) et non comme des entiers (`int64`), et que la variable **MARS** soit considérée comme une variable booléenne (`bool`).

```
print(sommets.dtypes)
print(liens.dtypes)
```

```
CODGEO         int64
P19_POP        int64
SUPERF       float64
SUP_QUALI     object
NOM           object
MARS           int64
dtype: object
Origine    int64
Arrivee    int64
weight     int64
dtype: object
```

Les lignes ci-dessous permettent de typer correctement les données.

```
# typage des variables
sommets[['CODGEO']] = sommets[['CODGEO']].astype('string')
sommets[['MARS']] = sommets[['MARS']].astype('bool')

liens[['Origine']] = liens[['Origine']].astype('string')
liens[['Arrivee']] = liens[['Arrivee']].astype('string')
```

La syntaxe pour créer un réseau orienté à partir d'une liste de liens est relativement simple. On précise quelle est la colonne des origines, celle des destinations, s'il existe une valuation des liens et si le graphe est orienté.

Pour que le réseau soit considéré comme valué, la colonne des intensités doit s'appeler `weight` ; il n'y a pas de contrainte particulière concernant les noms de colonnes correspondant aux sommets. Si le réseau est non orienté, on utilise la fonction `Graph()` et non `Digraph()`.

```
G = nx.from_pandas_edgelist(liens,
                            # data.frame des liens
                            source = "Origine",
                            # nom de la colonne origine
                            target = "Arrivee",
                            # nom de la colonne destination
                            edge_attr="weight",
                            # attribut poids pour un réseau valué
                            create_using=nx.DiGraph())
                            # création d'un réseau orienté
```

La gestion des attributs des sommets est un tout petit peu plus complexe et la documentation du module n'aide pas vraiment. La solution la plus simple, trouvée en ligne à cette adresse, est d'utiliser la fonction `set_index` du module `pandas`. On crée un objet correspondant aux attributs des sommets, on indique quel est l'identifiant (ici `CODGEO`) et on lui demande de transformer le tableau de données en dictionnaire.

```
nodes_attr = sommets.set_index('CODGEO').to_dict(orient = 'index')
nx.set_node_attributes(G, nodes_attr)

#contrôle
G.nodes("MARS")
```

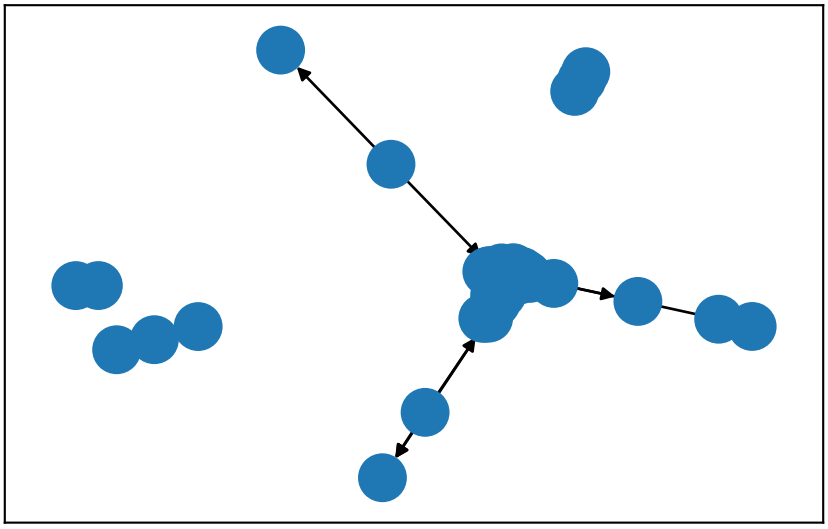
```
NodeDataView({'13001': False, '13041': False, '13201': True, '13205': True, '13208': True, '
```

Plusieurs moyens permettent de contrôler que l'importation s'est bien déroulée :

- afficher l'ordre (nombre de sommets) ;
- afficher la taille (nombre de liens) ;
- contrôler les caractéristiques du réseau ;
- visualiser le réseau obtenu.

```
print("Ordre : ", nx.number_of_nodes(G), "sommets")
print("Taille : ", nx.number_of_edges(G), "liens")
print("Réseau orienté ?", nx.is_directed(G))

# attribut des liens
print("attribut des liens : ", list(list(G.edges(data=True))[0][-1].keys()))

# attributs des sommets
print("attribut des sommets : ", list(list(G.nodes(data=True))[0][-1].keys()))

# visualisation basique
nx.draw_networkx(G, with_labels=False)
```

```
Ordre :  114 sommets
Taille :  317 liens
Réseau orienté ? True
attribut des liens :  ['weight']
attribut des sommets :  ['P19_POP', 'SUPERF', 'SUP_QUALI', 'NOM', 'MARS']
```



2 Filtrer, sélectionner

2.1 Extraire les composantes connexes

La visualisation montre un réseau non connexe. Le morceau de code qui suit décrit les différentes composantes du réseau et extrait la principale composante connexe. Le réseau étant orienté, on cherche une connexité faible (`weakly_connected_components`) ; si le réseau était non orienté, on utiliserait la fonction `connected_components`.

La logique est souvent similaire : on crée une liste correspondant au résultat ordonné d'une mesure puis on sélectionne des éléments de cette liste. Le numéro 0 entre crochets correspond au premier élément (Python numérote les éléments de 0 à **n - 1** et non de 1 à **n**).

```
# liste ordonnée des composantes connexes
CC = sorted(nx.weakly_connected_components(G),
            key=len,                  # clé de tri - len = longueur
            reverse=True)             # ordre décroissant
print("Nombre de composantes", len(CC))

# nombre de sommets par composantes
print("Nombre de sommets par composantes",
      [len(c) for c in sorted(nx.weakly_connected_components(G),
       key=len,
       reverse=True)])

# sélection de la composante connexe principale
GD = G.subgraph(CC[0])
```

```
Nombre de composantes 4
Nombre de sommets par composantes [106, 3, 3, 2]
```

Une version non orientée est créée car certaines mesures imposent un réseau de ce type. La valuation des liens est conservée dans la version non orientée : par défaut, elle ne correspond pas à la somme des intensités entrantes et sortantes. Ici, l'intensité du lien **ij** dans la version non orientée correspond à l'intensité du lien **ij** dans la version orientée.

```
# création d'une version non orientée
GU = nx.to_undirected(GD)

nx.is_weighted(GU)              #True

print("lien ij :", GD["13001"]["13201"]['weight'])
print("lien ji :", GD["13201"]["13001"]['weight'])
print("lien ij non orienté : ", GU["13001"]["13201"]['weight'])
```

```
lien ij : 135
lien ji : 499
lien ij non orienté :  135
```

Obtenir une valuation des liens dans le réseau non orienté correspondant à la somme des intensités **ij** + **ji** nécessite quelques étapes supplémentaires détaillées dans les lignes ci-dessous.

```
# créer une copie sans aucun lien
GU = nx.create_empty_copy(GD, with_data=True)
GU = nx.to_undirected(GU)

# éviter message "Frozen graph can't be modified"
GU = nx.Graph(GU)

# récupérer liens avec intensité nulle
GU.add_edges_from(GD.edges(), weight=0)

# pour chaque lien ij + ji
for u, v, d in GD.edges(data=True):
    GU[u][v]['weight'] += d['weight']

# contrôle

#a attributs liens et sommets
list(list(GU.edges(data=True))[0][-1].keys())
list(list(GU.nodes(data=True))[0][-1].keys())

# propriétés du réseau
nx.is_directed(GU)
nx.is_connected(GU)

print("lien ij :", GD["13001"]["13201"]['weight'])
```

```
print("lien ji :", GD["13201"]["13001"]['weight'])
print("lien ij non orienté : ", GU["13001"]["13201"]['weight'])
```

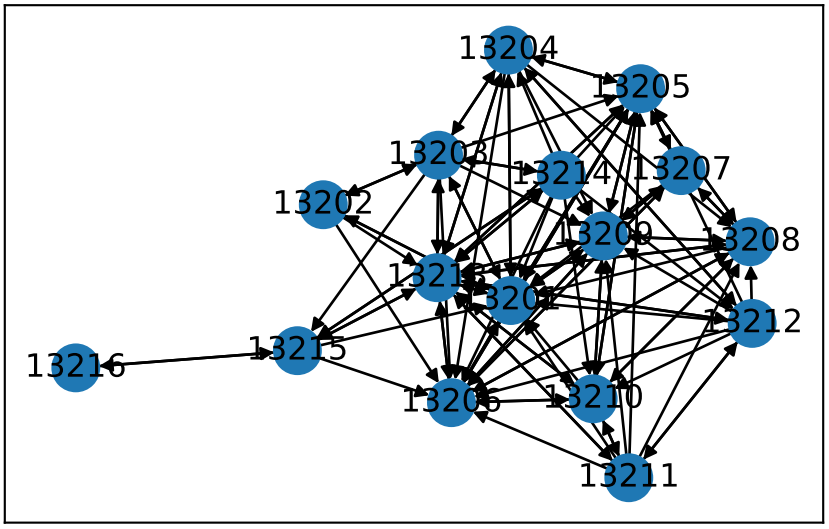
```
lien ij : 135
lien ji : 499
lien ij non orienté :  634
```

2.2 Filtrage

Les données sont importées, la plus grande composante connexe a été extraite dans deux versions, une orientée et une non orientée. On peut souhaiter faire des sélections autres, que ce soit sur les sommets ou sur les liens. Deux options sont possibles : supprimer liens ou sommets selon un critère donné (`remove_edges_from()`, `remove_nodes_from`) ; sélectionner liens ou sommets selon un critère donné (`subgraph()`).

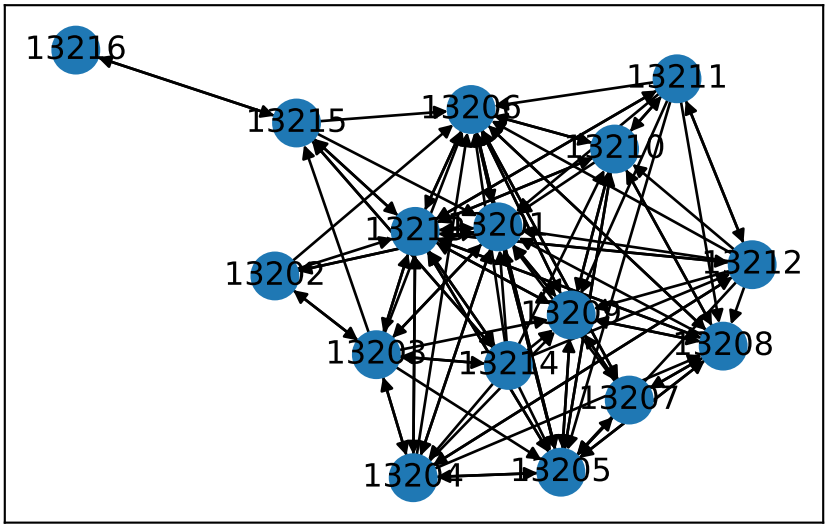
Si je souhaite travailler uniquement sur le cas marseillais :

```
# filtrage des sommets (1)
# sélection des sommets satisfaisant la condition
Mars = [n for n, v in G.nodes(data=True) if v['MARS'] == True]

# création d'un sous-graphe
Gmars = G.subgraph(Mars)

# visualisation
nx.draw_networkx(Gmars,
                 pos = nx.kamada_kawai_layout(Gmars),
                 with_labels=True)
```



On obtient le même résultat avec l'opération consistant à supprimer les sommets des communes hors Marseille :

```
# filtrage des sommets (2)
# sélection des sommets hors Marseille
nonmars = [n for n,v in G.nodes(data=True) if v['MARS'] == False]

# copier le réseau de départ
Gmars2 = G

# supprimer les communes hors Marseille
Gmars2.remove_nodes_from(nonmars)

# visualisation
nx.draw_networkx(Gmars2,
                 pos = nx.kamada_kawai_layout(Gmars2),
                 with_labels=True)
```

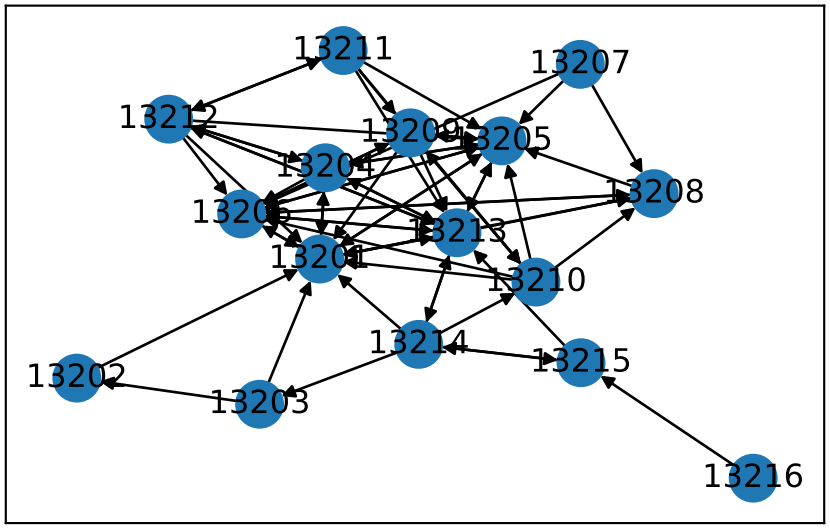


Si je souhaite travailler uniquement sur les flux les plus importants :

```
# filtrer les liens
# paramètres statistiques
liens.describe()

# fixer un seuil (ici la médiane)
seuil = 212

# identifier les liens sous ce seuil, récupérer les identifiants
long_edges = list(filter(lambda e: e[2] < seuil,
                 (e for e in G.edges.data('weight'))))
le_ids = list(e[:2] for e in long_edges)

# créer une copie du réseau de départ
Gsup = G

# supprimer les liens identifiés
Gsup.remove_edges_from(le_ids)

# ordre, taille et visualisation
print("Nb de sommets : ", nx.number_of_nodes(Gsup))
print("Nb de liens : ", nx.number_of_edges(Gsup))

nx.draw_networkx(Gsup,
```

```
                pos = nx.kamada_kawai_layout(Gsup),
                with_labels=True)
```

```
Nb de sommets :  16
Nb de liens :  64
```



Le fait de ne conserver que les liens entraîne la suppression des sommets devenant isolés. Le nombre de liens a très logiquement été divisé par deux dans la mesure où le seuil choisi ici est la médiane.

Si dans un réseau donné, j'ai des sommets isolés que je souhaite éliminer, j'utilise la fonction `remove_nodes_from`.

Soit un réseau aléatoire de 20 sommets contenant des isolés.

```
# générer un réseau aléatoire avec 2 isolés
rg = nx.gnp_random_graph(20, 0.05, seed = 1)
print("Nb de sommets (isolés compris) : ", nx.number_of_nodes(rg))

# liste des sommets avec un degré nul
isoles = [node for node,degree in dict(rg.degree()).items() if degree < 1]

# suppression des sommets concernés
rg.remove_nodes_from(isoles)
print("Nb de sommets  (isolés exclus) : ", nx.number_of_nodes(rg))
```

```
Nb de sommets (isolés compris) :  20
Nb de sommets  (isolés exclus) :  18
```

2.3 Agrégation

Il peut être intéressant d'agréger différents sommets. La fonction `contracted_nodes` prend en argument le réseau étudié et les deux sommets à fusionner, l'option `self_loops` permet de contrôler la création d'une boucle et l'option `copy` permet de créer un nouveau réseau sans écraser le premier.

Si je veux fusionner deux arrondissements marseillais, j'utilise le script suivant. Les lignes suivantes permettent de lister les liens entrants et sortants du sommet résultat de la fusion et de vérifier la présence (ici souhaitée) de la boucle.

```
GA = nx.contracted_nodes(G, '13215', '13216', self_loops=True, copy=True)
GA.in_edges('13215')   # liste des liens entrants
GA.out_edges('13215')  # liste des liens sortants
```

```
OutEdgeDataView([('13215', '13213'), ('13215', '13214'), ('13215', '13215')])
```

3 Mesurer

Les mesures sont testées sur deux réseaux, l'un orienté (`GD`) et l'autre non (`GU`), correspondant à la plus grande composante connexe. Lorsque la valuation et l'orientation des liens peuvent être prises en compte, les options utiles sont signalées. Par contre, le cas des liens multiples et des boucles n'est pas abordé.

`NetworkX` propose de nombreuses mesures et méthodes issues de la théorie des graphes au sens strict (centre, barycentre, rayon, etc.) et de l'analyse des flots dans les réseaux (optimisation) ; toutes ne sont pas évoquées ici et je me suis intéressé en priorité aux méthodes me semblant - peut-être à tort - les plus utilisées en sciences sociales.

3.1 Mesures globales

Dans les pages précédentes, on a déjà vu comment afficher l'ordre et la taille du réseau. La densité s'obtient avec la fonction `density`, le diamètre avec la fonction `diameter`.

```
# ordre et taille
print("nombre de sommets (ordre) : ",
      nx.number_of_nodes(GD),
      " sommets")
print("nombre de liens - orienté (taille) : ",
      nx.number_of_edges(GD),
      " liens")
print("nombre de sommets (ordre) : ",
      nx.number_of_nodes(GU),
      " sommets")
print("nombre de liens - non orienté (taille) : ",
      nx.number_of_edges(GU),
      " liens")

# densité et diamètre
print("densité (orienté) : ",
      round(nx.density(GD), 2))
print("densité (non orienté) :",
```

```
      round(nx.density(GU), 2))
# print("diamètre (non orienté et connexe) : ", nx.diameter(GD))
```

```
nombre de sommets (ordre) :  106  sommets
nombre de liens - orienté (taille) :  312  liens
nombre de sommets (ordre) :  106  sommets
nombre de liens - non orienté (taille) :  250  liens
densité (orienté) :  0.03
densité (non orienté) : 0.04
```

Si on cherche à mesurer le diamètre sur un réseau orienté non fortement connexe, on obtient le message d'erreur suivant : "Found infinite path length because the digraph is not strongly connected".

Mesures directement issues de la théorie des graphes, rayon et barycentre (appelé aussi point médian) correspondent respectivement à la distance entre le(s) sommet(s) ayant une excentricité minimale et tout autre sommet du graphe (cf *infra* les mesures locales) et à l'ensemble de sommets minimisant la fonction $\sum_{u \in V(G)} d_G(u, v)$, $d_G(u, v)$ étant la distance (topologique ou valuée) entre deux sommets u et v de G.

```
# rayon
print("rayon : ", nx.radius(GU))

# barycentre
print("barycentre : ", nx.barycenter(GU))
print("barycentre (valué) : ", nx.barycenter(GU, weight='weight'))
```

```
rayon :  4
barycentre :  ['13001']
barycentre (valué) :  ['13001']
```

La liste du ou des sommets centraux s'obtient avec `center`, la liste des sommets périphériques avec `periphery`. Les premiers correspondent aux sommets ayant une excentricité minimale, les seconds aux sommets ayant une excentricité maximale.

```
print("Sommets centraux : ", nx.center(GU))
print("Sommets périphériques : ", nx.periphery(GU))
```

```
Sommets centraux :  ['13001']
Sommets périphériques :  ['13083', '13076']
```

Comme toutes les mesures fondées sur la recherche de plus courts chemins (diamètre, rayon, etc.), ces fonctions supposent soit un réseau non orienté, soit un réseau orienté fortement connexe.

Il est possible de connaître l'ensemble des isthmes et des points d'articulation présents dans un réseau.

```
# isthmes et points d'articulation (réseau connexe)
# nombre et liste des isthmes
print("nb isthmes : ", len(list(nx.bridges(GU))))
print("isthmes : ",list(nx.bridges(GU)))

# nombre et liste des points d'articulation
print("nb points d'articulation : ", len(list(nx.articulation_points(GU))))
print("points d'articulation : ", list(nx.articulation_points(GU)))
```

```
nb isthmes :  32
isthmes :  [('13001', '13004'), ('13001', '13015'), ('13001', '13019'), ('13001', '13025'),
nb points d'articulation :  15
points d'articulation :  ['13046', '13028', '13005', '13054', '13103', '13050', '13067', '13
```

L'indice de Wiener, issu de l'étude des réseaux biologiques, correspond à la somme des plus courts chemins entre toutes les paires de sommets (équivaut à l'indice de dispersion de Shimbel parfois utilisé par les géographes des transports). Le réseau orienté n'étant pas fortement connexe, le résultat est une distance infinie (`inf`). Il est possible de prendre en compte la valuation des liens et de chercher la somme des plus courtes distances entre sommets.

```
# indice de wiener (somme des pcc)
print("Indice de Wiener (orienté) : ",
      nx.wiener_index(GD))     # distance topologique
print("Indice de Wiener (non orienté) : ",
      nx.wiener_index(GU)) # distance topologique
print("Indice de Wiener (non orienté, valué) : ",
      nx.wiener_index(GU, 'weight'))  # somme des intensités
```

```
Indice de Wiener (orienté) :  inf
Indice de Wiener (non orienté) :  15228.0
Indice de Wiener (non orienté, valué) :  3800738.0
```

Mesure plus récente et liée davantage aux études des physiciennes sur les réseaux, l'assortativité est mesurée par défaut en fonction du degré. Elle varie entre -1 (réseau disassortatif, les

sommets ayant un degré faible tendent à être voisins des sommets ayant un degré élevé et inversement) et 1 (réseau assortatif, les sommets ayant un degré faible sont voisins de sommets ayant un degré faible, les sommets ayant un degré élevé sont voisins de sommets ayant un degré élevé). Dans les lignes ci-dessous, les quatre assortativités possibles dans un réseau orienté sont calculées en fonction du degré.

```
# assortativité (degré par défaut)
# réseau orienté - 4 options possibles
print("assortativité in-in : ",
      round(nx.degree_assortativity_coefficient(GD, x="in", y='in'),3))

print("assortativité in-out : ",
      round(nx.degree_assortativity_coefficient(GD, x="in", y='out'),3))

print("assortativité out-in : ",
      round(nx.degree_assortativity_coefficient(GD, x="out", y='in'),3))

print("assortativité out-out : ",
      round(nx.degree_assortativity_coefficient(GD, x="out", y='out'),3))

# réseau non orienté
print("assortativité globale (non orienté) : ",
      round(nx.degree_assortativity_coefficient(GU),3))

# assortativité selon un critère autre que le degré
print("assortativité (Marseille vs autres communes) : ",
      round(nx.numeric_assortativity_coefficient(GD, "MARS"),3))
```

```
assortativité in-in :  -0.105
assortativité in-out :  0.262
assortativité out-in :  -0.198
assortativité out-out :  0.485
assortativité globale (non orienté) :  -0.234
assortativité (Marseille vs autres communes) :  0.707
```

Mesure relativement proche (voir la page wikipedia), le **rich club coefficient** s'obtient avec la fonction… `rich_club_coefficient` ; elle peut être utilisée uniquement sur des réseaux non orientés et ne tient pas compte des éventuelles boucles, liens multiples ou intensité des liens. La fonction renvoie une valeur par degré (rich club coefficient des sommets de degré 1, de degré 2, etc.).

```
# rich-club coefficient
print("rich-club coefficient : ",
      nx.rich_club_coefficient(GU, normalized=True, seed=42))
```

```
rich-club coefficient :  {0: 1.0, 1: 0.9955156950672646, 2: 1.0119047619047619, 3: 1.0546875
```

Les mesures “globales” fondées sur des mesures “locales” (degré ou transitivité moyenne par exemple) sont abordées dans la section suivante.

3.2 Mesures locales

Le degré se calcule à l’aide de la fonction `degree`. Si le réseau étudié est orienté, on peut utiliser `in_degree` et `out_degree`. L’argument optionnel `weight` permet de calculer le degré pondéré par l’intensité des liens.

```
# degré
print(GD.degree())

#degré entrant, sortant
print(GD.in_degree())
print(GD.out_degree())

# degré entrant pondéré
print(GD.in_degree(weight = "weight"))
```

```
[('13001', 72), ('13041', 6), ('13201', 24), ('13205', 23), ('13208', 20), ('13209', 22), ('
[('13001', 66), ('13041', 5), ('13201', 16), ('13205', 14), ('13208', 12), ('13209', 14), ('
[('13001', 6), ('13041', 1), ('13201', 8), ('13205', 9), ('13208', 8), ('13209', 8), ('13213
[('13001', 21875), ('13041', 926), ('13201', 4624), ('13205', 4649), ('13208', 2668), ('1320
```

Ajouter `_centrality` à la fonction utilisée permet d’obtenir les valeurs normalisées.

```
# degré normalisé
print(nx.degree_centrality(GD))
```

```
{'13001': 0.6857142857142857, '13041': 0.05714285714285715, '13201': 0.2285714285714286, '132
```

Pour transformer une mesure en attribut, on commence par créer un objet correspondondant au résultat puis on utilise la fonction `set_node_attributes` si la mesure porte sur les sommets et `set_edge_attributes` si elle porte sur les liens.

```
# transformer une mesure en attribut
deg = nx.degree(GU)
nx.set_node_attributes(GU, 'degree', deg)
print("attribut des sommets : ",
      list(list(GU.nodes(data=True))[0][-1].keys()))
```

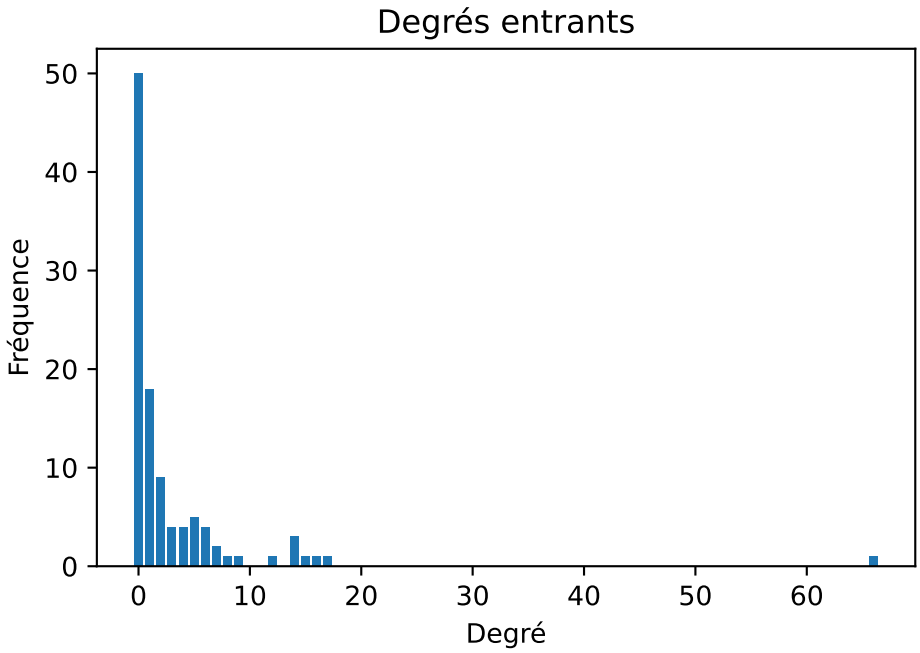
```
attribut des sommets :  ['P19_POP', 'SUPERF', 'SUP_QUALI', 'NOM', 'MARS', DegreeView({'13001
```

Les lignes ci-dessous proposent différentes solutions pour visualiser la distribution des degrés :

- histogramme ;
- courbe ;
- courbe avec une échelle log-log.

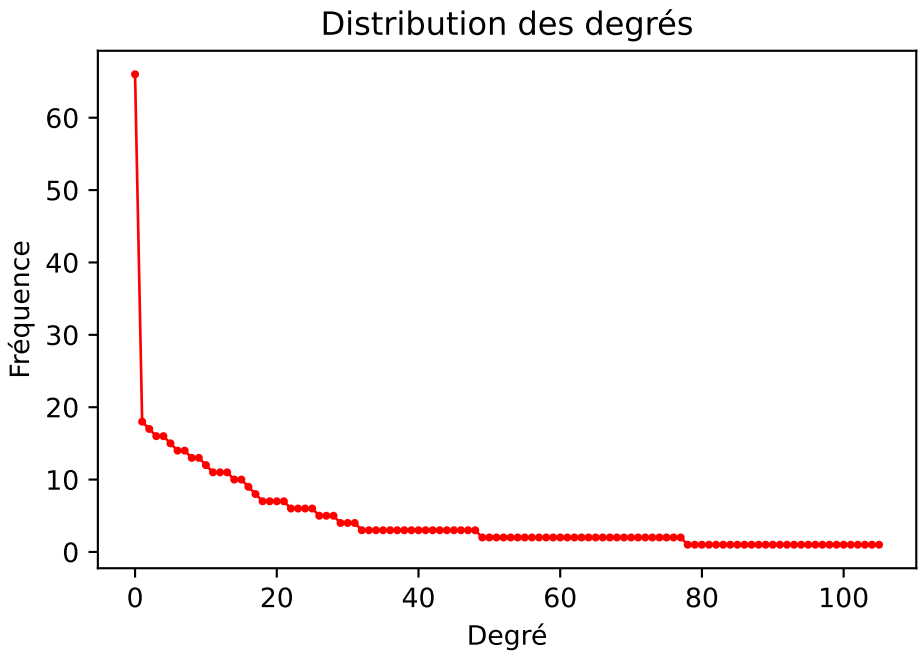
```
# distribution sous forme d'histogramme
degree_in = sorted((d for n, d in GD.in_degree()), reverse=True)
plt.bar(*np.unique(degree_in, return_counts=True))
plt.title("Degrés entrants")
plt.xlabel("Degré")
plt.ylabel("Fréquence")
```

```
Text(0, 0.5, 'Fréquence')
```



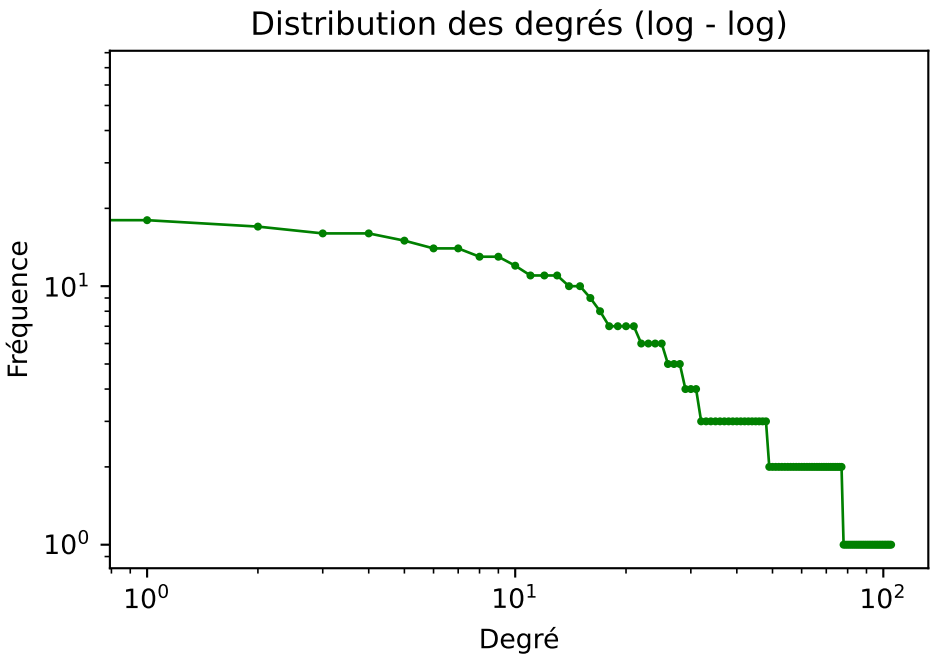

```
# courbe
degree_sequence = sorted((d for n, d in GU.degree()), reverse=True)

# distribution des degrés
plt.plot(degree_sequence, "r-", marker="o", linewidth=1, markersize=2)
plt.title("Distribution des degrés")
plt.xlabel('Degré')
plt.ylabel('Fréquence')
```

```
Text(0, 0.5, 'Fréquence')
```




```
# distribution degré - échelle log - log
plt.loglog(degree_sequence, "go-", linewidth=1, markersize=2)
plt.title("Distribution des degrés (log - log)")
plt.xlabel('Degré')
plt.ylabel('Fréquence')
```

```
Text(0, 0.5, 'Fréquence')
```



Il est possible de calculer le degré moyen des voisins à à l'aide de la fonction `average_neighbor_degree` (degré entrant, sortant, total, pondéré ou non).

```
# degré moyen des sommets voisins
nx.average_neighbor_degree(GD)
# degré entrant moyen des voisins entrants
nx.average_neighbor_degree(GD, source="in", target="in")
# degré moyen pondéré des voisins
nx.average_neighbor_degree(GD, weight="weight")
```

Les mesures de centralité autres s'obtiennent avec des fonctions aux noms attendus : `closeness`, `betweenness` (`edge_betweenness` pour l'intermédiarité des liens), `eigenvector_centrality` et `katz_centrality`.

```
# intermédiarité
nx.betweenness_centrality(GD)
L``
# intermédiarité des liens
nx.edge_betweenness_centrality(GD)

# proximité
```

```
nx.closeness_centrality(GD)

# centralité de vecteur propre
nx.eigenvector_centrality(GD)

# centralité de Katz
nx.katz_centrality(GD)
```

Concernant la recherche de la transitivité, on peut connaître le nom de triangles (triades fermées) contenant chaque sommet avec la fonction `triangles`. La transitivité locale s'obtient avec `clustering`, la transitivité globale avec `transitivity` et la transitivité moyenne avec `average_clustering`. Il est possible de prendre en compte l'orientation des liens et/ou leur intensité.

```
# nombre de triangles
nx.triangles(GU)

# typologie MAN (réseau orienté)
nx.triadic_census(GD)

# transitivité locale
nx.clustering(GD)

# prise en compte de l'intensité des liens
nx.clustering(GD, weight="weight")
```

```
# transitivité globale et moyenne
print("Transitivité globale (orienté) : ",
      round(nx.transitivity(GD), 2))
print("Transitivité moyenne (orienté) : ",
      round(nx.average_clustering(GD), 2))
print("Transitivité globale (non orienté) : ",
      round(nx.transitivity(GU), 2))
print("Transitivité moyenne (non orienté) : ",
      round(nx.average_clustering(GU), 2))
```

```
Transitivité globale (orienté) :  0.62
Transitivité moyenne (orienté) :  0.35
Transitivité globale (non orienté) :  0.29
Transitivité moyenne (non orienté) :  0.54
```

4 Partitionner

4.1 Cliques et k-cores

La page de la documentation consacrée aux cliques propose x fonctions dont plusieurs sont obsolotètes. Par défaut, `NetworkX` renvoit tous les ous-graphes complets d'ordre 1 à **n**, **n** correspondant au sous-graphe maximal complet et donc à la clique au sens strict du terme.

```
# cliques d'ordre 1 à n
print("Nombre de `cliques' : ", sum(1 for c in nx.find_cliques(GU)))
print("Nombre de sommets dans la plus grande clique : ",
      max(len(c) for c in nx.find_cliques(GU)))
print("Composition de la plus grande clique \n",
      max(nx.find_cliques(GU), key=len))
```

```
Nombre de `cliques' :  86
Nombre de sommets dans la plus grande clique :  10
Composition de la plus grande clique
 ['13001', '13213', '13206', '13201', '13205', '13209', '13212', '13210', '13208', '13211']
```

Nombre de variations des **k-cores** sont proposées (k-shell, k-crust, k-corona…), elles seront abordées dans une version ultérieure de ce tutoriel.

```
# ordre des k-cores
print(list(nx.k_components(GU)))

# composition du 8-core
list(nx.k_core(GU, k = 8))
```

```
[8, 7, 6, 5, 4, 3, 2, 1]
```

```
['13213',
 '13206',
 '13204',
```

```
 '13005',
 '13205',
 '13203',
 '13001',
 '13209',
 '13201',
 '13211',
 '13210',
 '13208',
 '13214',
 '13212']
```

4.2 Blockmodel

`NetworkX` ne semble pas proposer de méthodes de blockmodeling. La documentation fournit un script créant une partition des sommets créée par une CAH sur la matrice d'adjacence.

4.3 Détection de communautés

Plusieurs méthodes de détection de communautés sont implémentées dans le sous-module `community`. Seule la mesure de la modularité semble disponible pour évaluer la qualité de la partition obtenue.

```
# détection de communautés
louv = nx.community.louvain_communities(GU, seed=123)
print("nb de communautés (louvain) :", len(louv))
louv[1]

#mesure de la modularité
print("modularité (louvain) : ",
      round(nx.community.modularity(GU, louv),2))

# algorithme maximisant la modularité
greed = nx.community.greedy_modularity_communities(GD)
print("nb de communautés (greedy mod.) :", len(greed))
print("modularité (greedy mod.) : ",
      round(nx.community.modularity(GD, greed),2))
```

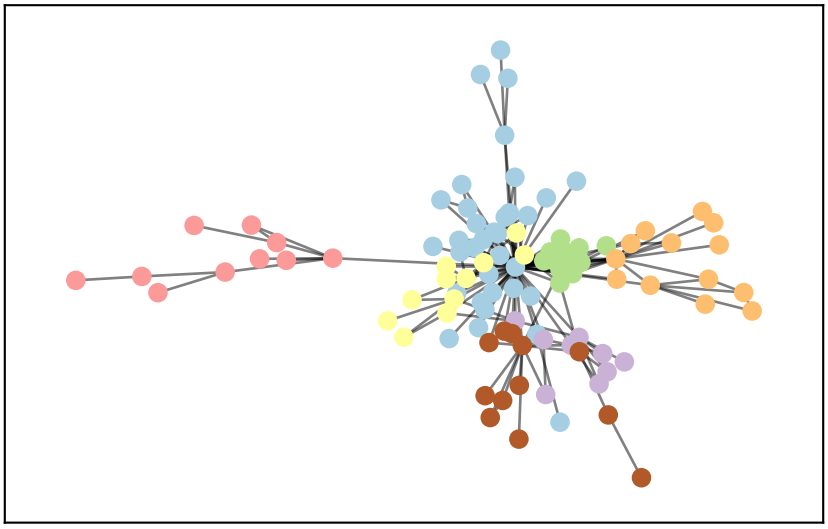
```
nb de communautés (louvain) : 7
```

```
modularité (louvain) :  0.5
nb de communautés (greedy mod.) : 6
modularité (greedy mod.) :  0.51
```

Les lignes qui suivent anticipent sur la section suivante et indiquent comment visualiser les communautés détectées. La première étape est de créer un dictionnaire attribuant à chaque sommet la communauté d'appartenance détectée avec l'algorithme utilisé.

```
# création d'un dictionnaire vide
louvain_dict = {}

# boucle qui remplit le dictionnaire
for i,c in enumerate(louv):
    for CODGEO in c:
        louvain_dict[CODGEO] = i

# ligne optionnelle pour ajouter cet attribut
# nx.set_node_attributes(GD, louvain_dict, 'louvain')

# choix de l'algorithme de placement des sommets
pos = nx.spring_layout(GU)

# importation d'une palette de couleurs
# nb de couleurs égal au nombre de classes + 1 (Python commence à 0)
cmap = plt.get_cmap('Paired', max(louvain_dict.values()) + 1)

# visualisation des sommets
nx.draw_networkx_nodes(GU,   # sommets à visualiser
                       pos,  # placement
                       louvain_dict.keys(),  # identifiants
                       node_size=40,         # taille
                       cmap=cmap,            # palette de couleur
                       node_color=list(louvain_dict.values()))
                       # affecter une couleur différente pour chaque classe
nx.draw_networkx_edges(GU,
                       pos,
                       alpha=0.5)  # transparence
```

```
<matplotlib.collections.LineCollection at 0x26d4fe4b340>
```

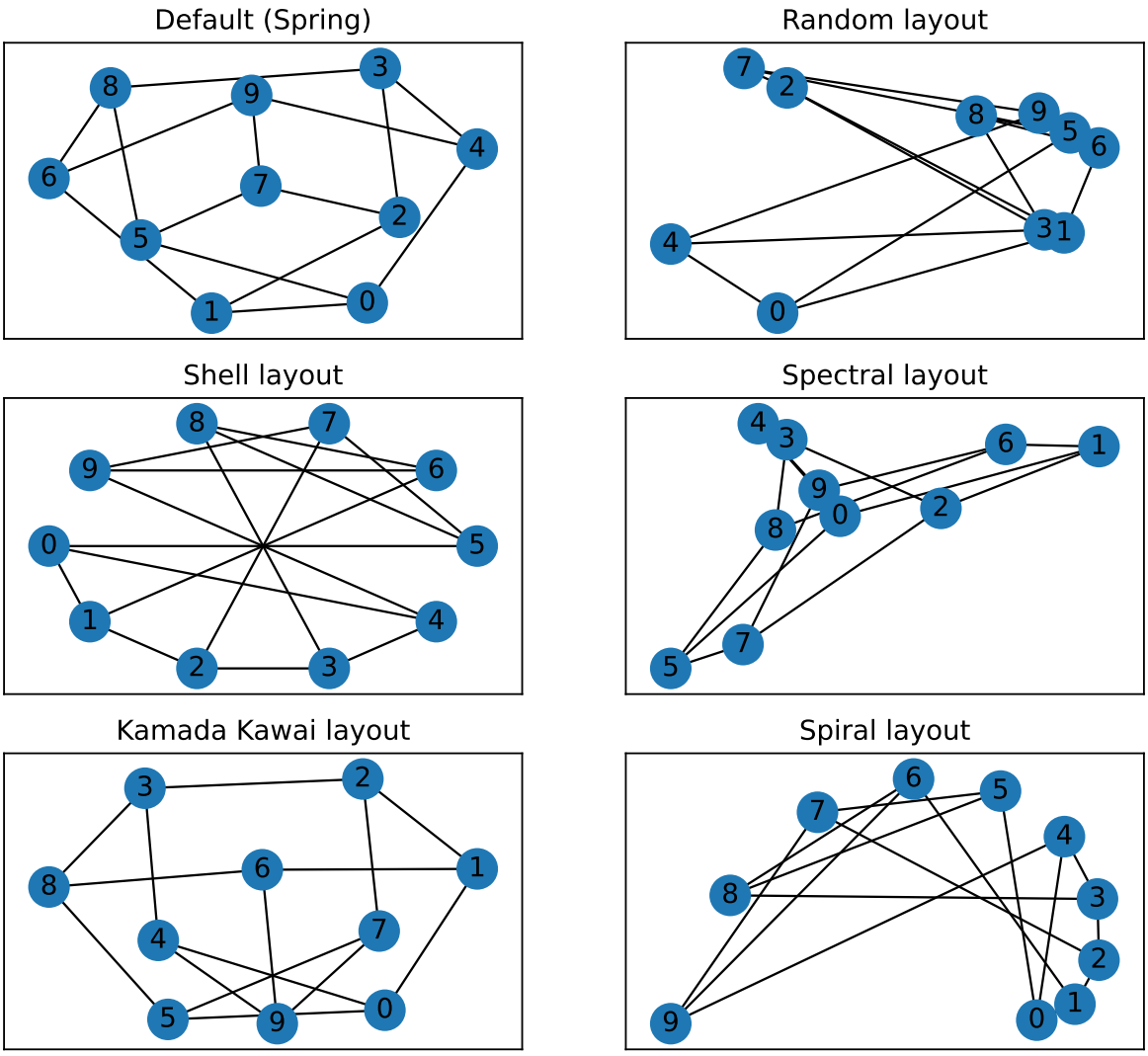


5 Visualiser ses données

Différents algorithmes de placement des sommets sont disponibles, certaines étant adaptés à des types précis de réseaux (`bipartite_layout`, `planar_layout`, `multipartite_layout`).

```
#graphe de Petersen
#pour les plus curieuses : https://fr.wikipedia.org/wiki/Graphe_de_Petersen
G = nx.petersen_graph()

# algorithmes de visualisation
# juxtaposer des fenêtres
fig, ax = plt.subplots(3, 2, figsize = (9,8)) # 3 lignes, 2 colonnes
nx.draw_networkx(G, ax = ax[0,0])
# ax[0,0] : à placer sur la première ligne, première colonne
ax[0,0].set_title('Default (Spring)')         # titre de la figure
nx.draw_networkx(G, pos = nx.random_layout(G), ax = ax[0,1])
ax[0,1].set_title('Random layout')
nx.draw_networkx(G, pos = nx.shell_layout(G), ax = ax[1,0])
ax[1,0].set_title('Shell layout')
nx.draw_networkx(G, pos = nx.spectral_layout(G), ax = ax[1,1])
ax[1,1].set_title('Spectral layout')
nx.draw_networkx(G, pos = nx.kamada_kawai_layout(G), ax = ax[2,0])
ax[2,0].set_title('Kamada Kawai layout')
nx.draw_networkx(G, pos = nx.spiral_layout(G), ax = ax[2,1])
ax[2,1].set_title('Spiral layout')
```

```
Text(0.5, 1.0, 'Spiral layout')
```

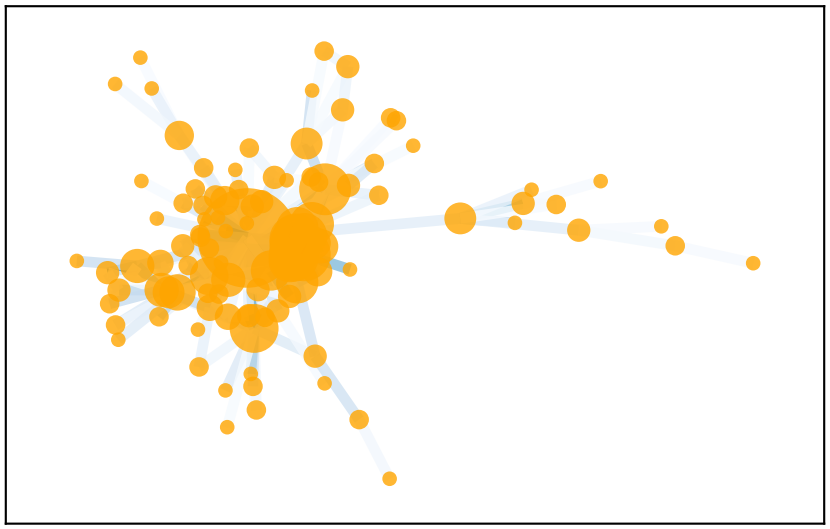


Il est évidemment possible de modifier l'apparence des sommets et des liens et, par exemple, de faire varier la taille ou la couleur en fonction d'un attribut. Il est souvent nécessaire de créer au préalable un dictionnaire permettant de faire varier le paramètre graphique voulu selon un indicateur donné, par exemple faire varier la taille des sommets selon le degré.

Les exemples ci-dessous ne prétendent pas à l'exhaustivité mais illustrent l'utilisation de quelques paramètres graphiques d'usage courant.

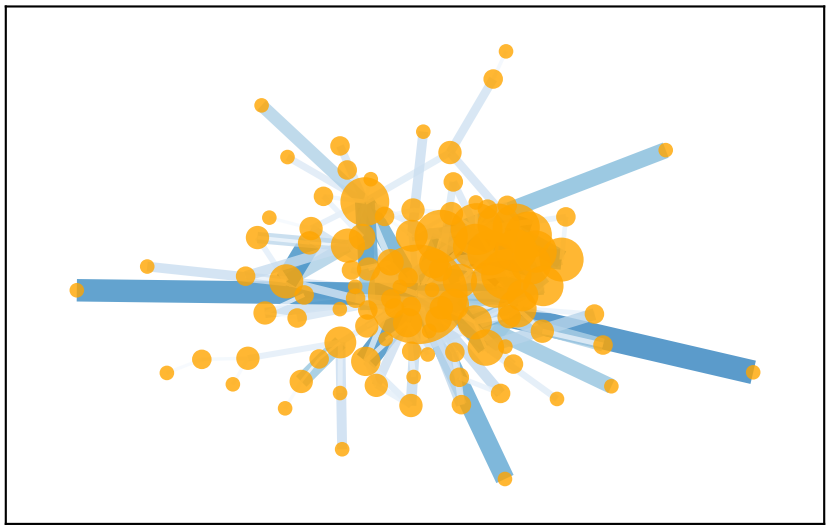
```
# création du dictionnaire pour les degrés
d = dict(GU.degree)
```

```
# création d'une liste des intensités
weights = [GU[u][v]['weight'] for u,v in GU.edges]

nx.draw_networkx(GU,
                 pos = nx.spring_layout(GU),  # algorithme de placement
                 node_color = 'orange',  # couleur des sommets
                 alpha = 0.8,            # transparence
                 nodelist= d.keys(),     # liste des sommets
                 node_size = [v * 20 for v in d.values()], # taille des sommets
                 edge_cmap=plt.cm.Blues, # palette de couleurs
                 edge_color = weights,   # couleur des liens
                 with_labels=False,      # affichage des labels
                 width=4)                # épaisseur des liens
```



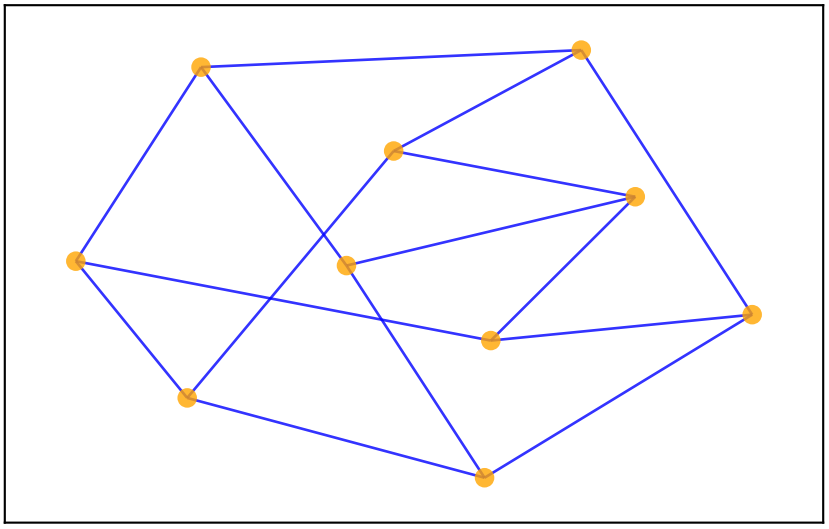
```
# faire varier teinte et épaisseur des liens
# diviser les intensités par 100

weigh2 = [i/100 for i in weights]

nx.draw_networkx(GU,
                 pos = nx.kamada_kawai_layout(GU),
                 node_color = 'orange',
                 alpha = 0.8,
                 nodelist= d.keys(),
```

```
                node_size = [v * 20 for v in d.values()],
                edge_cmap=plt.cm.Blues,
                edge_color = weights,
                with_labels = False,
                width=weigh2)
```



Si on souhaite produire plusieurs figures avec la même apparence, le plus simple est de définir au préalable les options de visualisation puis de les appeler.

```
# définir les options de visualisation
options = {
      'node_color' : 'orange',
      'node_size'  : 40,
      'edge_color' : 'blue',
      'width' : 1,
      'alpha' : 0.8,
      'with_labels': False
    }

# visualiser
nx.draw_networkx(G, **options)
```



On trouve en ligne des exemples permettant de tester d'autres types de visualisation (matrice, edge-bundling, etc.). Ces points seront évoqués dans une version ultérieure de ce tutoriel.

6 Pour aller plus loin

Pour aller plus loin en analyse de réseau (avec Python ou tout autre logiciel), commencer par la liste Awesome Network Analysis maintenue par François Briatte est un bon réflexe. La partie Python mériterait peut-être un coup de plumeau, notamment pour les modules conseillés - c'est inscrit sur ma to-do-list de l'été mais je ne promets rien…

6.1 Avec Python

C'est fou le nombre de livres payants proposant des initiations à des logiciels libres et gratuits. Les livres anglophones sont généralement disponibles sur libgen. Pour les livres francophones, vous pouvez 1. voir auprès de votre bibliothèque universitaire favorite ou 2. contactez directement l'auteur et l'autrice pour demander gentiment s'il n'y a pas une version pdf disponible quelque part.

Pour s'initier, Openclassrooms propose des cours plutôt corrects. Ce n'est pas pensé pour des utilisatrices en sciences sociales et on peut allégrement sauter certaines étapes mais ça reste utile.

Orienté sciences sociales, le dépôt github d'Émilien Schultz propose une liste de ressources utiles pour l'utilisation de Python et les supports des nombreuses formations qu'il a dispensées ces dernières années. Orienté sciences sociales mais en anglais, le site de Melanie Walsh est très chaudement recommandé.

L'excellent site **Programming Historian** propose 28 leçons sur Python, toutes disponibles en anglais, certaines sont traduites en espagnol, français et/ou portugais.

Le site Python Graph Gallery, copié sur The R Graph Gallery, permet de récupérer le code pour générer de belles figures. Ça reste cependant moins complet - et parfois moins joli - que ce qu'on peut faire avec R.

6.2 Avec NetworkX

Le site officiel est touffu (euphémisme). Nombre de fonctions sont obsolètes et la documentation n'est pas toujours d'une grande clarté. On trouve quelques tutoriels sur l'analyse de

réseaux avec `NetworkX` en ligne : j'ai notamment utilisé des morceaux de **Programming Historian** (Ladd et al. (2017)) et d'un cours de l'IUT de Reims (Blanchard (2018)). Beaucoup plus complet, le datacamp accessible à cette adresse est très complet mais commence à dater (2020), certaines parties ne fonctionnent plus très bien.

Beauguitte, L. (2023). *L'analyse de réseau en sciences sociales. Petit guide pratique*. Groupe fmr. https://hal.science/hal-04052709

Blanchard, F. (2018). Analyse des réseaux sociaux en python. https://iut-info.univ-reims.fr/users/blanchard/ISN20181218/index.html

Hagberg, A. A., Schult, D. A. et Swart, P. J. (2008). Exploring Network Structure, Dynamics, and Function using NetworkX. *Proceedings of the 7th Python in Science Conference*, 11-15.

Ladd, J. R., Otis, J., Warren, C. N. et Weingart, S. (2017). Exploring and Analyzing Network Data with Python. *Programming Historian*. https://doi.org/https://doi.org/10.46430/phen0064