femto-st SCIENCES & TECHNOLOGIES

Les graphes en python

C. Guyeux

UFC UNIVERSITÉ DE FRANCHE-COMTÉ cnrs ensmm utbm université de technologie Belfort-Montbéliard

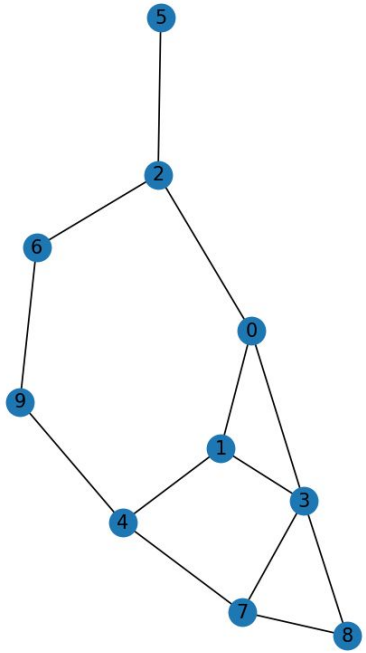
Préliminaire : installation de bibliothèques

Dans un terminal :

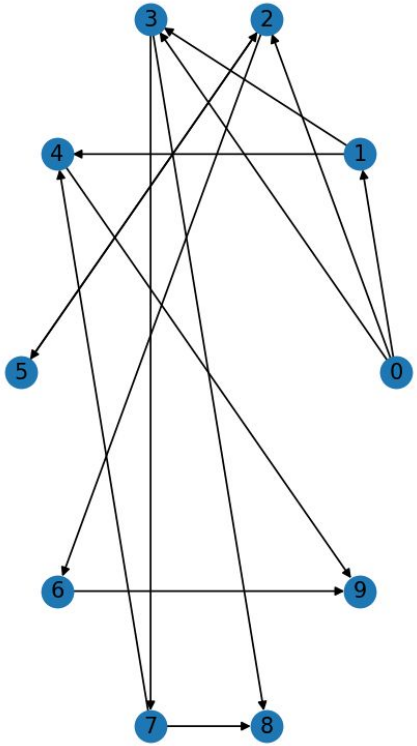
```
pip install ——user ——upgrade networkx
pip install ——user ——upgrade matplotlib
pip install ——user ——upgrade scipy
```

Zoologie des graphes

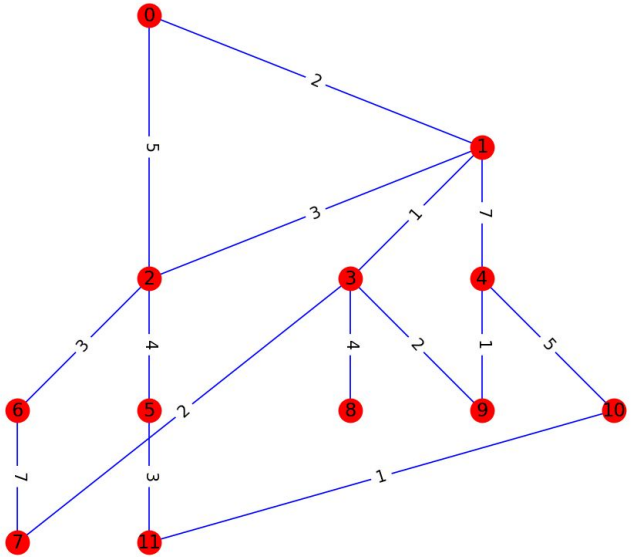
Graphe non orienté simple



Graphe orienté simple

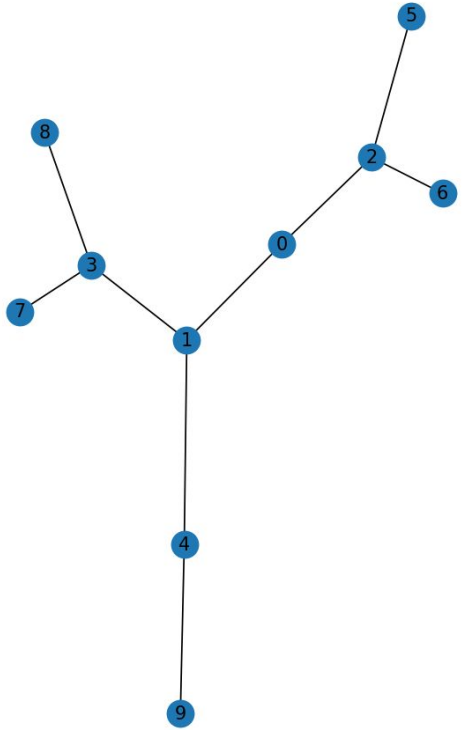


Graphe pondéré

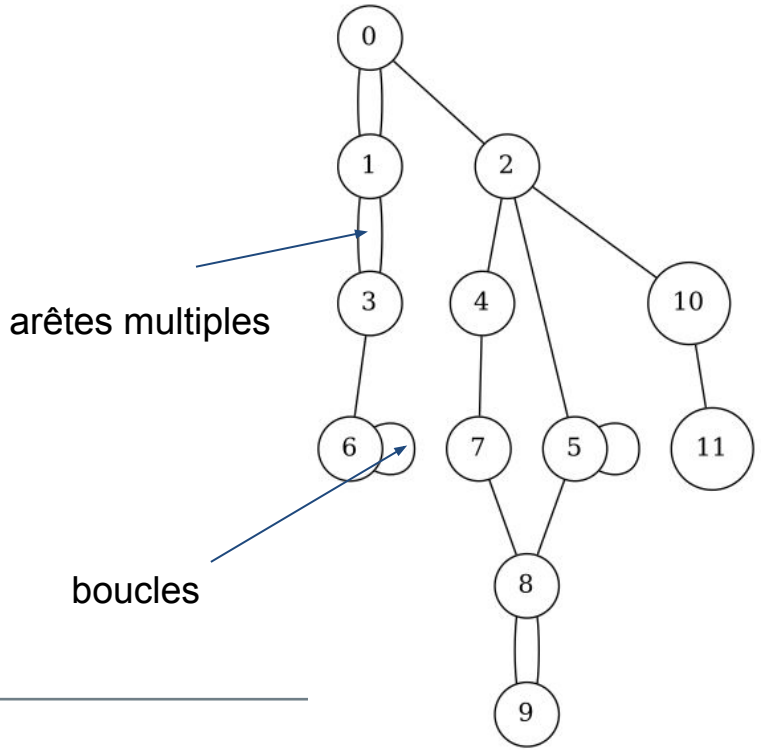


Graphes et arbres

Arbre



Graphe non orienté, pas simple



femto-st SCIENCES & TECHNOLOGIES

Motivation

Plus court chemin routier

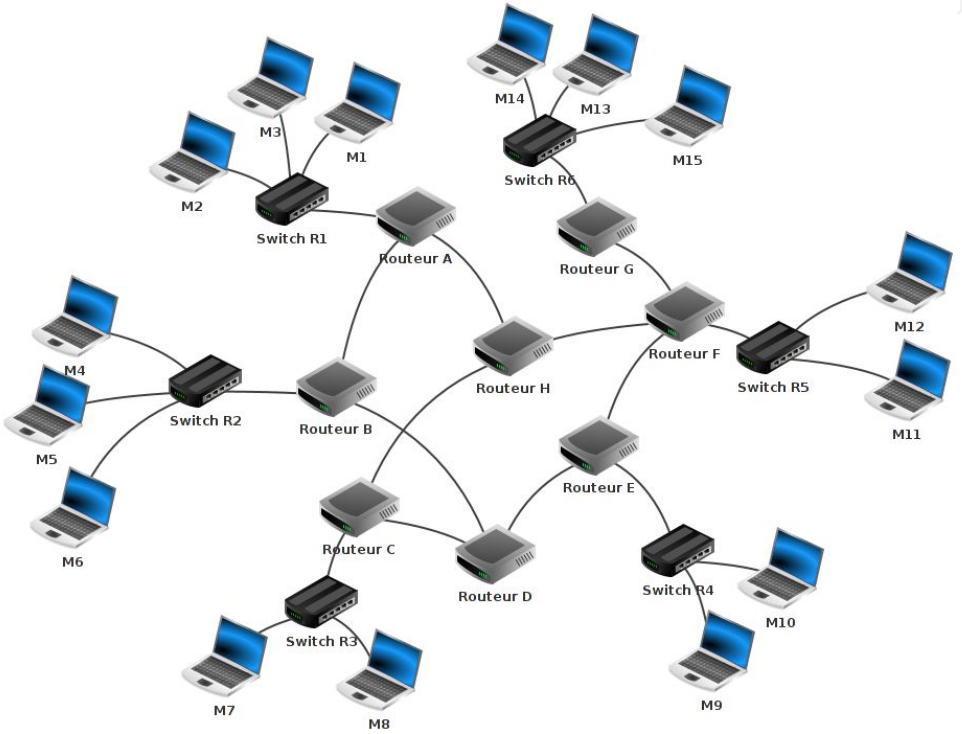
Quel est le plus court chemin pour se rendre de A à B…

…par exemple, dans le graphe OpenStreetMaps ?

Protocoles de routage

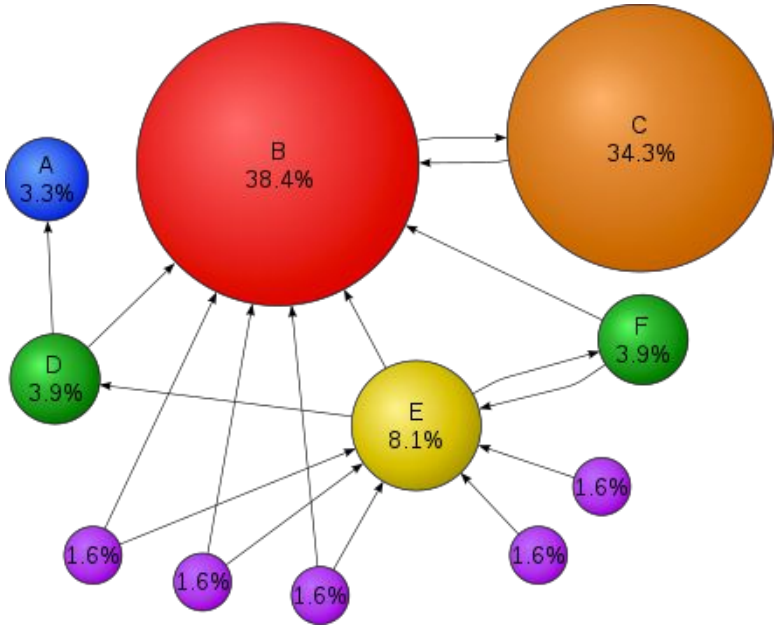
ping : le serveur cible est-il accessible ?

Si oui, trouver le plus rapide chemin vers lui.



Page rank de google

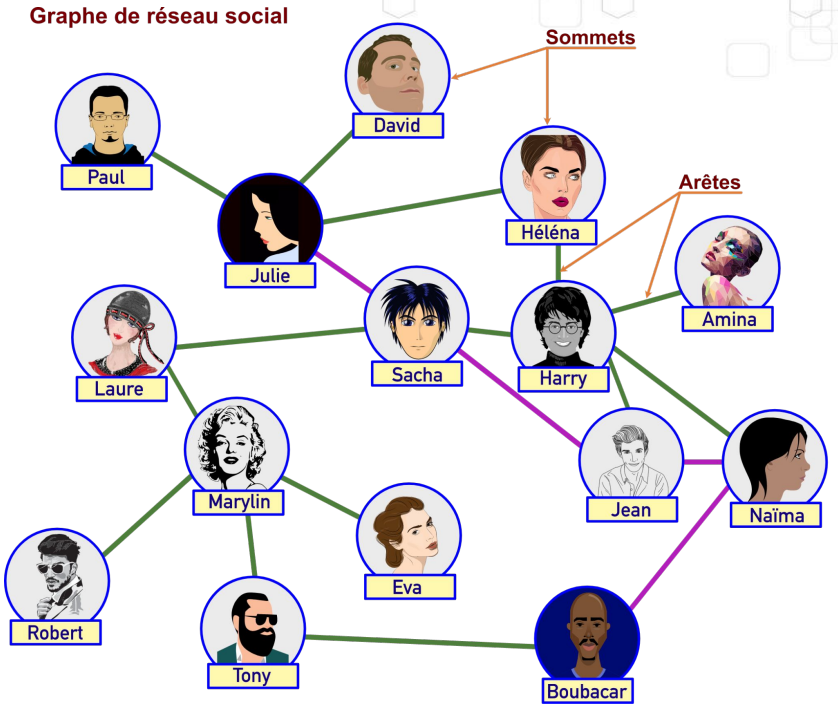
Un site web est d'autant plus important que des sites importants pointent sur lui.



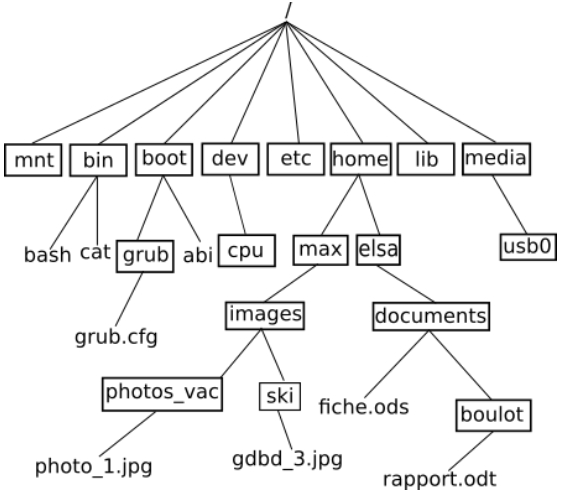
Analyse de réseaux sociaux

Cambridge Analytica :

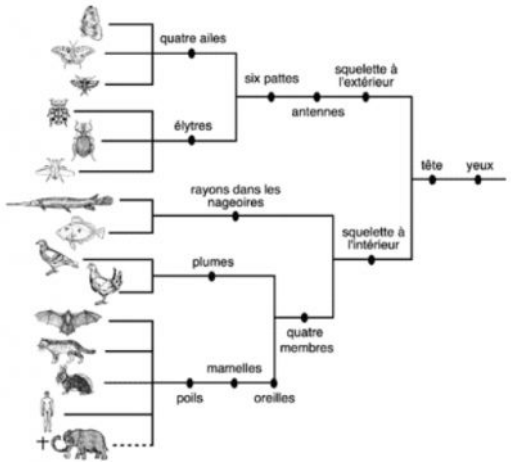
- Collecte de données (2014) : par un quiz, en récoltant par violation des infos sur les amis.
- Campagne présidentielle de 2016 : micro-ciblage (qui influencer ?)
- Diffusion de messages politiques pour faire basculer des “swing states”



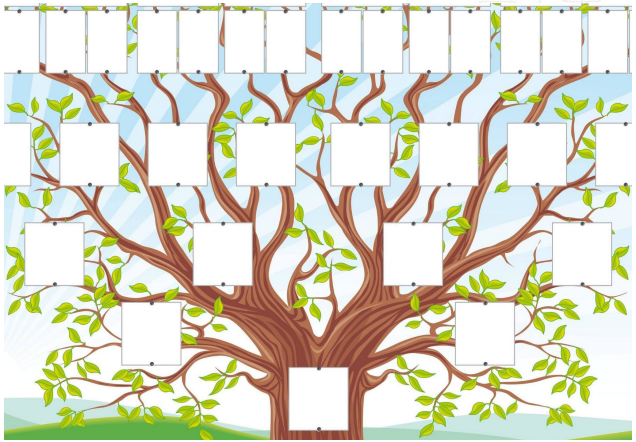
Arbres et arborescence



Arborescence GNU/Linux

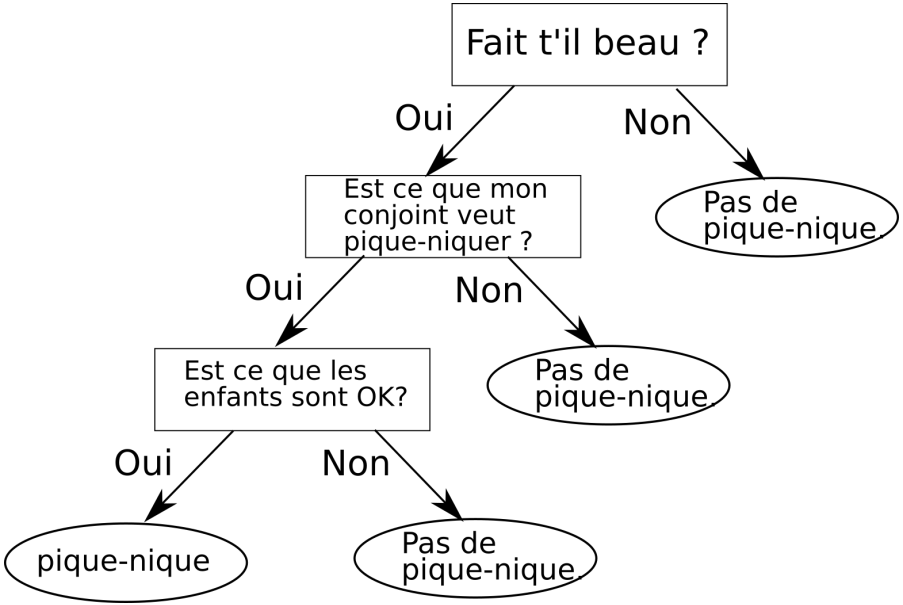


Arbre phylogénétique

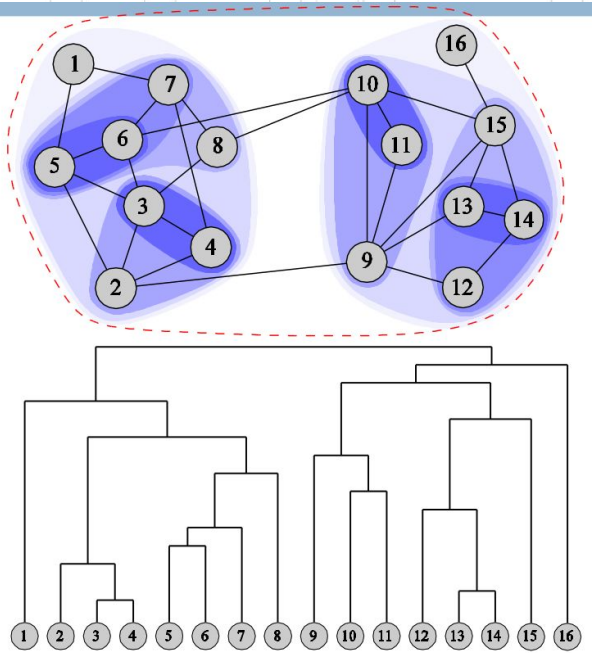


Arbre généalogique

Intelligence artificielle



Arbre de décision (apprentissage supervisé)



Clustering hiérarchique (non sup.)

Mais aussi…

- Intelligence artificielle :
  - apprentissage non supervisé : clustering hiérarchique
  - apprentissage supervisé : arbre de décision, méthodes d'ensemble (Forêts aléatoires, Extra trees, ADABoost, XGBoost, LightGBM…)
  - Graphes neural networks

- Compression sans perte : Huffmann (zip, mp3, jpg)

- Planification : méthode PERT

- Chaîne de Markov, Réseaux bayésiens…

Définitions

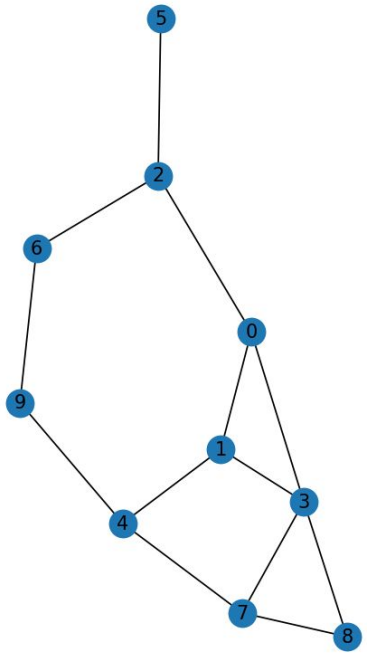
Graphe non orienté simple

Un **graphe non orienté simple** G est un couple ordonné (V, E), où :

1. V est un ensemble fini non vide de sommets (ou nœuds).
2. E est un ensemble d'arêtes, où chaque arête est un ensemble non ordonné de deux sommets distincts.

Dans un graphe non orienté simple :

- Il n'y a pas d'arêtes multiples entre deux sommets (c'est-à-dire qu'il y a au plus une seule arête entre deux sommets donnés).
- Il n'y a pas de boucles (c'est-à-dire qu'une arête ne peut pas relier un sommet à lui-même).



Graphe orienté simple

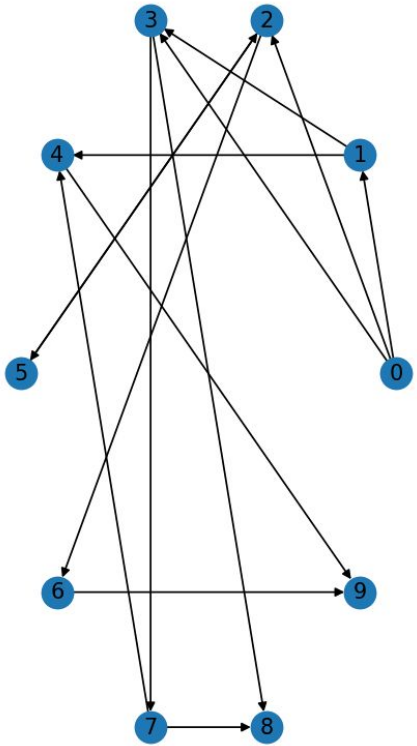
Un **graphe orienté simple** G est un couple ordonné (V, A), où :

1. V est un ensemble fini non vide de sommets (ou nœuds).
2. A est un ensemble d'arcs (ou flèches), où chaque arc est un couple ordonné de deux sommets distincts.

Dans un graphe orienté simple :

- Les relations entre les sommets sont directionnelles.
- Il n'y a pas d'arcs multiples ayant la même direction entre deux sommets .
- Il n'y a pas de boucles.

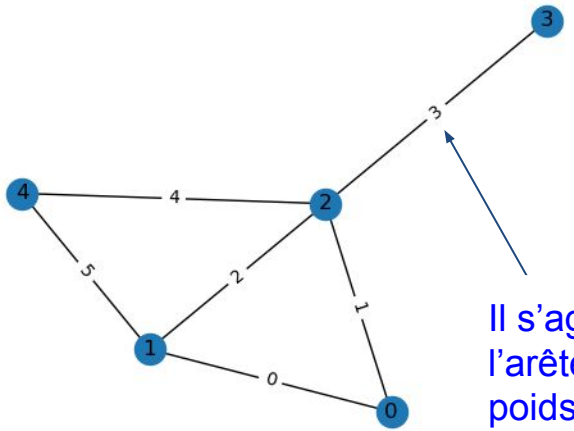
=> relations unidirectionnelles entre les sommets, sans redondance ni auto-connexion.



Matrice d’adjacence

Pour un graphe non orienté simple G avec *n* sommets, la **matrice d'adjacence** *A* est une matrice carrée de taille *n* x *n*, où l'élément a_{ij} est défini comme suit :

- a_{ij} = 1 (ou le poids) si une arête existe entre les sommets *i* et *j*,
- a_{ij} = 0 sinon.



Il s’agit du nom de l’arête, pas de son poids

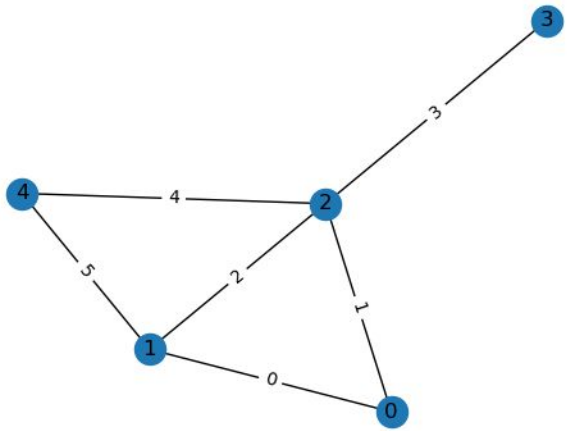
sommet 4

sommet 3

```
[[0, 1, 1, 0, 0],
 [1, 0, 1, 0, 1],
 [1, 1, 0, 1, 1],
 [0, 0, 1, 0, 0],
 [0, 1, 1, 0, 0]]
```

Liste d’adjacence

On appelle **liste d'adjacence** du graphe un tableau de listes, la *i*-ème liste contenant la liste des sommets adjacents au sommet *i* (l'ordre étant purement arbitraire).

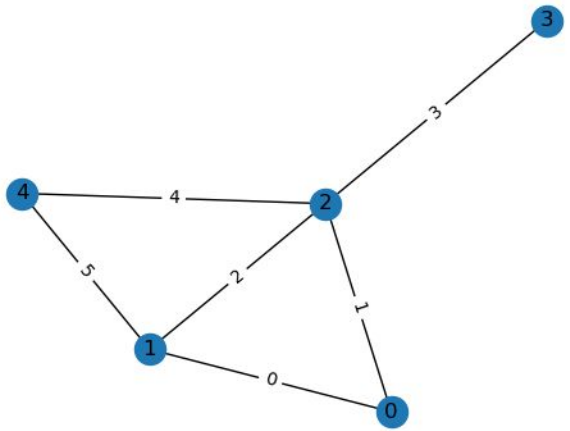



```
0: [1, 2]
1: [0, 2, 4]
2: [0, 1, 3, 4]
3: [2]
4: [2, 1]
```

Matrice d'incidence

La **matrice d'incidence** *M* d'un graphe non orienté à n sommets et m arêtes est une matrice *n* x *m*, où chaque ligne représente un sommet et chaque colonne représente une arête, et pour toute ligne i, colonne j :

- M_{ij} = 1 si le sommet *i* est connecté à l'arête *j*, et 0 sinon.
  (ou le poids, éventuellement signé, suivant le cas)



arête 3

sommet 3

```
[[1., 1., 0., 0., 0., 0.],
 [1., 0., 1., 0., 0., 1.],
 [0., 1., 1., 1., 1., 0.],
 [0., 0., 0., 1., 0., 0.],
 [0., 0., 0., 0., 1., 1.]]
```

Matrice d'incidence

La **matrice d'incidence** *M* d'un graphe non orienté à n sommets et m arêtes est une matrice *n* x *m*, où chaque ligne représente un sommet et chaque colonne représente une arête, et pour toute ligne i, colonne j :

- M_{ij} = 1 si le sommet *i* est connecté à l'arête *j*, et 0 sinon.

Et dans le cas orienté :

- M_{ij} = -1 si le sommet *i* est le sommet initial (d'où part l'arête) de l'arête *j*,
- M_{ij} = 1 si le sommet *i* est le sommet final (où arrive l'arête) de l'arête *j*,
- M_{ij} = 0 sinon.

Et dans le cas pondéré : le poids

Introduction à networkx

Création d'un graphe

Création d'un graphe non orienté :

```
>>> import networkx as nx
>>> G = nx.Graph()
```

Ajout de sommets (node) :

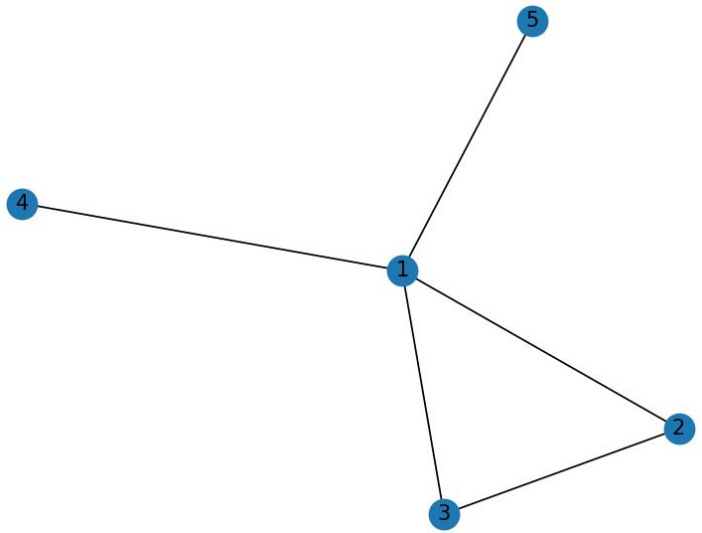
```
>>> G.add_node(1)
>>> G.add_nodes_from([2, 3, 4])
```

Ajout d'arêtes (edge) :

```
>>> G.add_edge(1,2)
>>> G.add_edges_from([(1, 3), (1, 4), (2, 3)])
>>> G.add_edge(1, 5) # Même si le noeud n'existe pas encore
```

Graphe : visualisation

```
>>> import matplotlib.pyplot as plt
>>> nx.draw(G, with_labels=True)
>>> plt.show() # plt.savefig("graphe.png")
```



Cas des graphes orientés, pondérés…

Graphe pondéré :

```
>>> G.add_edge(1, 2, weight=1)
>>> G.add_weighted_edges_from([(1, 2, 3), (2, 3, 4), (3, 4, 5)])
```

Graphe orienté (dirigé) :

```
>>> G = nx.DiGraph()
```

Les Graph et DiGraph ne permettent pas d'avoir d'arêtes multiples entre deux nœuds (les boucles simples sont autorisées). Pour des arêtes multiples :

```
>>> G = nx.MultiGraph()
>>> G = nx.MultiDiGraph()
```

Graphe : noeuds

```
>>> list(G.nodes())
[1, 2, 3, 4, 5]
```

Ordre du graphe (nombre de nœuds) :

```
>>> G.number_of_nodes()
5
```

Degré d'un noeud (nombre de voisins) :

```
>>> G.degree(1)
4
```

Degré moyen :

```
>>> degrees = G.degree()
>>> sum(dict(degrees).values()) / G.number_of_nodes()
2.0
```

Graphes réguliers : exercice

Un graphe est dit **régulier** quand tous ses sommets ont le même degré.

Faire une fonction qui teste si un graphe est régulier.

Graphes réguliers : solution

Un graphe est dit **régulier** quand tous ses sommets ont le même degré.

```
def is_regular(G):
    return len(set([G.degree(k) for k in G.nodes()])) == 1
```

tester la compréhension de liste :

```
[G.degree(k) for k in G.nodes()]
```

puis :

```
set([G.degree(k) for k in G.nodes()])  # set : ensemble
```

et conclure…

(len, sum, set… “battery included”)

A parte : set

```
>>> s=set([1,2,3])
>>> s.remove(3)
>>> 1 in s
True
>>> s.add(5)
>>> s
{1, 2, 5}
>>> for k in set([1, 2]):
        print(k)
```

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}
>>> s1.intersection(s2)
{2, 3}
>>> s1.union(s2)
{1, 2, 3, 4}
```

La liste a-t-elle des redondances ?

```
>>> len(set(L)) == len(L)
```

A parte : comprehension de liste

```
>>> from time import time
>>> t = time()
>>> S = 0
>>> for k in range(100000000):
        S += k**2

>>> time()-t
28.3428692817688
>>>
>>> t=time()
>>> sum([k**2 for k in range(100000000)])
>>> time()-t
25.112656354904175
```

A parte : comprehension de liste

```
>>> from time import time
>>> t = time()
>>> L = []
>>> for k in range(100000000):
        L.append(k**2)

>>> time()-t
25.168331146240234
>>>
>>> t=time()
>>> L = [k**2 for k in range(100000000)]
>>> time()-t
20.824453830718994
```

Graphe : arêtes, taille, diamètre

```
>>> print(G)
Graph with 5 nodes and 5 edges

>>> print(G.edges())
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3)]
```

Taille du graphe (nombre d'arêtes) :

```
>>> G.number_of_edges()
5
```

Diamètre du graphe : la plus longue distance (la plus courte) entre deux nœuds

```
>>> nx.diameter(G)
2
```

Diamètre d’un réseau social

Model: GPT-4

C

On considère le graphe où les sommets sont les humains, et il y a une arête entre deux sommets quand les individus se connaissent. Quel est le diamètre du graphe ?

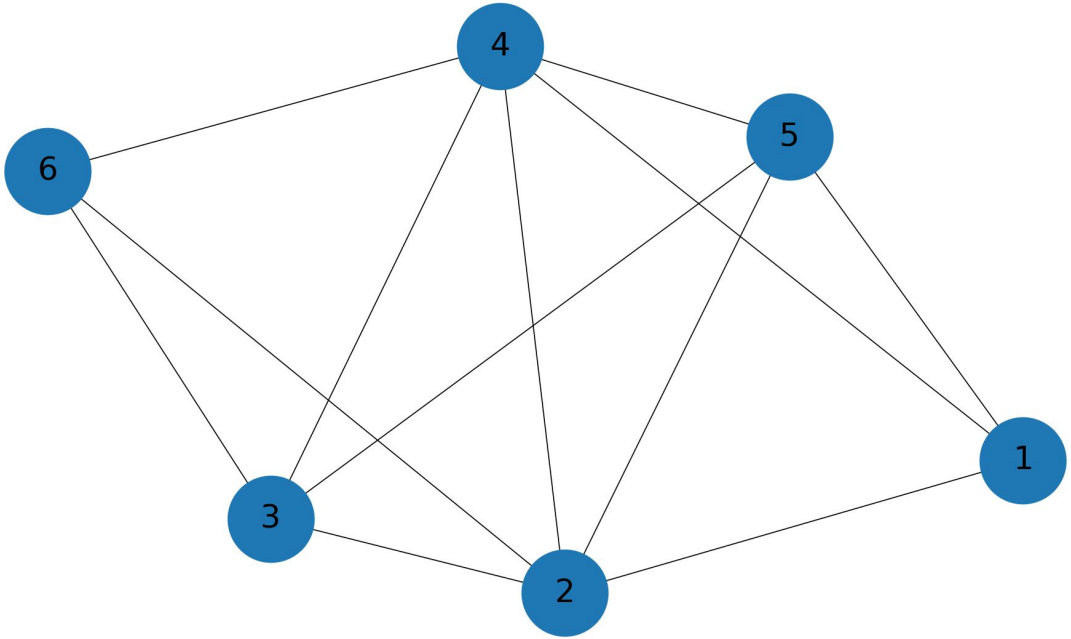
Il est très difficile de déterminer le diamètre exact d'un tel graphe, car il y a plusieurs facteurs à prendre en compte, notamment la taille et la complexité du réseau social humain. En théorie, le diamètre d'un graphe est la plus longue distance entre deux sommets du graphe. Dans le contexte d'un réseau social humain, cela correspondrait au nombre minimal d'étapes nécessaires pour passer d'une personne à une autre en passant uniquement par des relations de connaissance.

Dans la pratique, le concept des "six degrés de séparation" est souvent utilisé pour estimer cette distance dans les réseaux sociaux humains. Cette théorie, popularisée par l'expérience de Stanley Milgram dans les années 1960, suggère qu'en moyenne, toute personne sur Terre est reliée à une autre personne par une chaîne de six relations intermédiaires ou moins. Cependant, cela ne donne qu'une estimation moyenne du chemin le plus court entre deux personnes et non le diamètre exact du graphe.

Le diamètre exact du graphe serait probablement supérieur à six, mais il est difficile de le calculer précisément en raison de l'évolution constante et de la complexité des relations humaines.

Exercice

1. Créer le graphe ci-dessous.
2. Faire une fonction qui reçoit un graphe, et produit des informations sur lui (ordre, diamètre…)



Exercice : corrigé

```
>>> G = nx.Graph()
>>> G.add_edges_from([(1, 2), (1, 4), (1, 5), (2, 3), (2, 4),
(2, 5), (2, 6), (3, 6), (3, 5), (3, 4), (4, 6), (4, 5)])

>>> def graphe_info(G):
...     print(f"Ordre : {G.number_of_nodes()}")
...     print(f"Taille : {G.number_of_edges()}")
...     print(f"Diamètre : {nx.diameter(G)}")
...     degrees = G.degree()
...     print(f"Degré moyen : {sum(dict(degrees).values()) /
G.number_of_nodes()}")
```

A parte : les f-strings

```
>>> pi_val = 3.141592
>>> f"Exemple : {pi_val:.2f}"
Example 2: 3.14

>>> import datetime as dt
>>> day = dt.datetime.now()
>>> f"{day:%Y/%m/%d}"
'2023/03/28'
>>> f"{day:%Y %B %d (%A)}"
'2023 mars 28 (mardi)'

>>> f"blabla : {dd:>10} fois"
'blabla :        abc fois'
```

f"{variable:format}"

Graphes et matrices

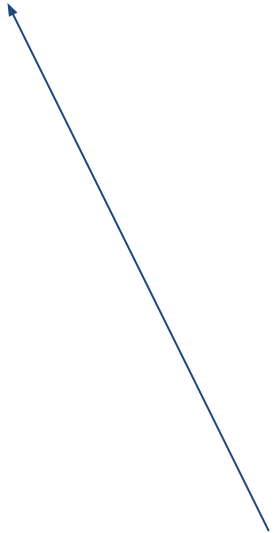
Matrice d'adjacence :

```
>>> nx.adjacency_matrix(G).toarray()
array([[0, 1, 1, 1, 0, 0],
       [1, 0, 1, 1, 1, 1],
       [1, 1, 0, 1, 1, 1],
       [1, 1, 1, 0, 1, 0],
       [0, 1, 1, 1, 0, 1],
       [0, 1, 1, 0, 1, 0]])
```

Pour la matrice d'incidence : `nx.incidence_matrix(G)`

Graphes et liste d'adjacences

```
>>> {n: list(neighbors) for n, neighbors in G.adjacency()}
{1: [2, 4, 5],
 2: [1, 3, 4, 5, 6],
 4: [1, 2, 3, 6, 5],
 5: [1, 2, 3, 4],
 3: [2, 6, 5, 4],
 6: [2, 3, 4]}
```



Compréhension de dictionnaire

Exercice

1. Faire une fonction qui, à une matrice d'adjacence, renvoie le graphe non orienté associé.

2. Faire de même à partir d'une liste d'adjacence

Exercice : corrigé

```
>>> def from_matrix(M):
...     G=nx.Graph()
...     for k in range(len(M)-1):
...         for l in range(k, len(M)):
...             if M[k,l]:
...                 G.add_edge(k, l)
...     return G
...
>>> def from_dict(dico):
...     G=nx.Graph()
...     for k in dico:
...         for l in dico[k]:
...             G.add_edge(k, l)
...     return G
```

Exercice : corrigé

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

G = nx.from_numpy_matrix(adj_matrix_np)

adj_list = {0: [1, 2],
            1: [0, 2, 3],
            2: [0, 1, 3],
            3: [1, 2]}

G = nx.from_dict_of_lists(adj_list)
```

Sur les constructeurs Graph(), DiGraph()...

On peut aussi passer au constructeur : la liste d'arêtes, un dictionnaire de listes (liste d'adjacence), un dictionnaire de dictionnaires, un array numpy (matrice d'adjacence)...

Création par matrice d'adjacence :

```
>>> import numpy as np
>>> G = nx.DiGraph(np.array([[0, 1, 0, 1], [1, 0, 0, 1], [1, 1,
0, 0], [1, 1, 0, 0]]))
```

Création par liste d'adjacence :

```
>>> G = nx.Graph({0: [1, 2], 1: [0, 2, 3], 2: [0, 1, 3], 3: [1,
2]})
```

Lemme des poignées de mains

Exercice : vérifier ce lemme, et sa conséquence, sur le graphe de votre choix

La somme des degrés des sommets est égale à deux fois le nombre d'arêtes

Conséquence : un graphe simple a un nombre pair de sommets de degré impair.

Lemme des poignées de mains

Exercice : vérifier ce lemme, et sa conséquence, sur le graphe de votre choix

La somme des degrés des sommets est égale à deux fois le nombre d'arêtes

```
len([k for k in G.nodes() if G.degree(k)%2 == 1])%2 == 0
```

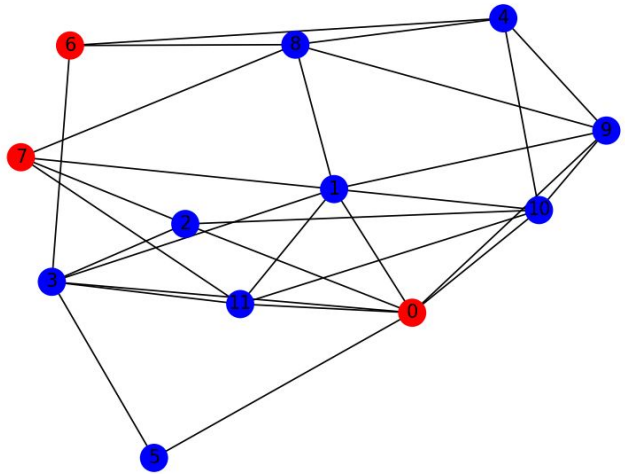
Conséquence : un graphe simple a un nombre pair de sommets de degré impair.

```
sum([G.degree(k) for k in G.nodes()]) == 2*G.number_of_edges()
```

L’affichage de graphes

Colorer les noeuds

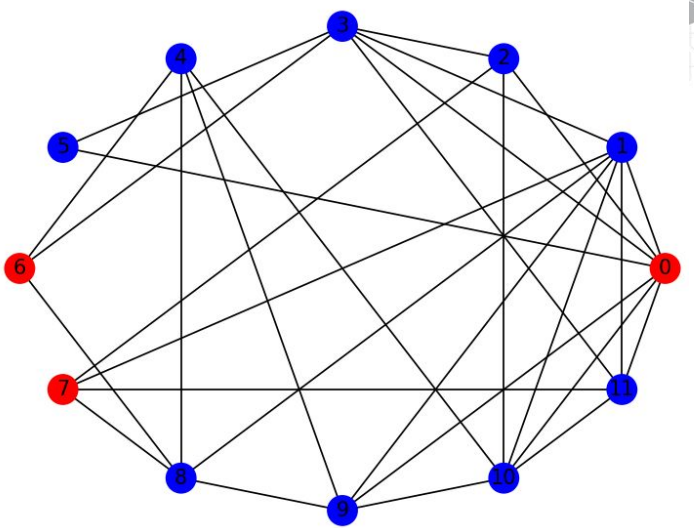
```
>>> node_color = ['red', 'blue', 'blue', 'blue', 'blue', 'blue',
'red', 'red', 'blue', 'blue', 'blue', 'blue']
>>> nx.draw(G, node_color=node_color, with_labels=True)
>>> plt.show()
```



femto-st SCIENCES & TECHNOLOGIES

Layout (mise en page)

```
>>> pos = nx.circular_layout(G)
>>> nx.draw(G, pos, node_color=node_color,
            with_labels=True)
>>> plt.show()
```



Autres layouts :

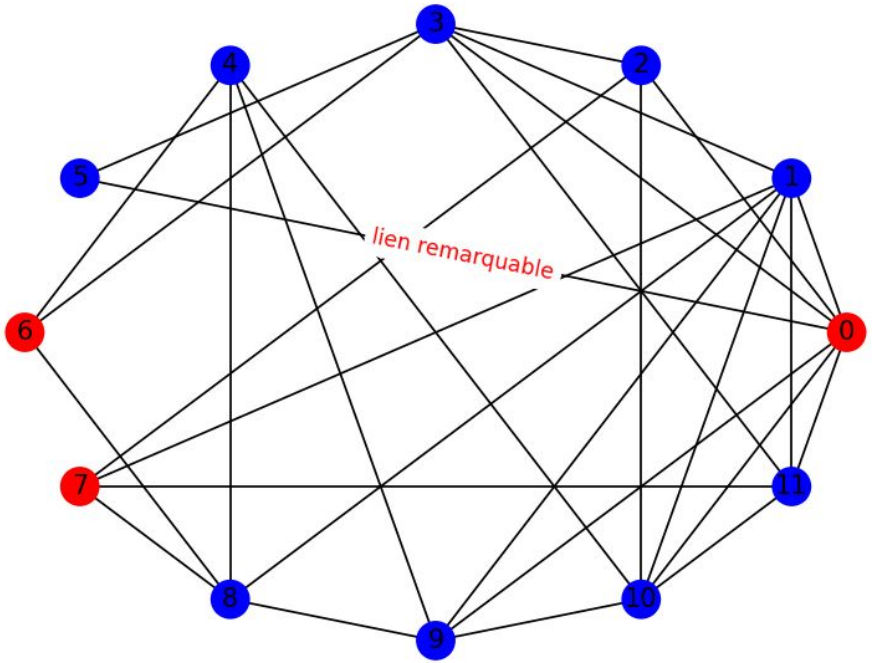
- `nx.kamada_kawai_layout(G)`
- `nx.planar_layout(G)`
- `nx.random_layout(G)`
- `nx.spectral_layout(G)`
- `nx.spring_layout(G)`
- `nx.shell_layout(G)`

Layout (mise en page) : exercice

Tester les différentes mises en pages

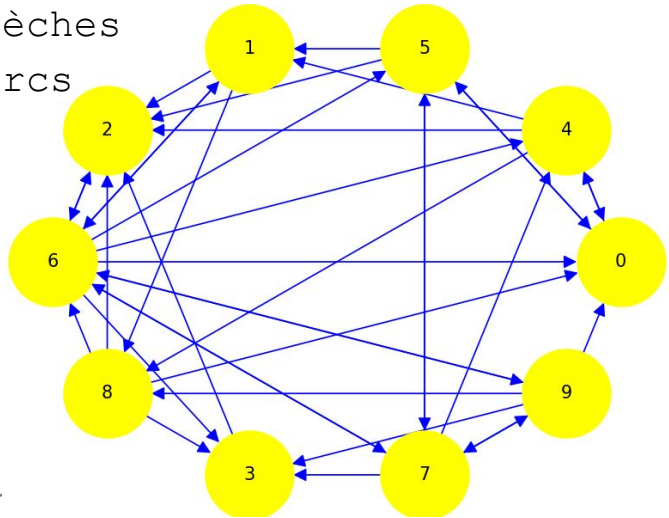
Nommer les arêtes

```
nx.draw(G, pos, node_color=node_color, with_labels=True)
nx.draw_networkx_edge_labels(G, pos,
      edge_labels={(0, 5): "lien remarquable"},
      font_color='red')
plt.show()
```



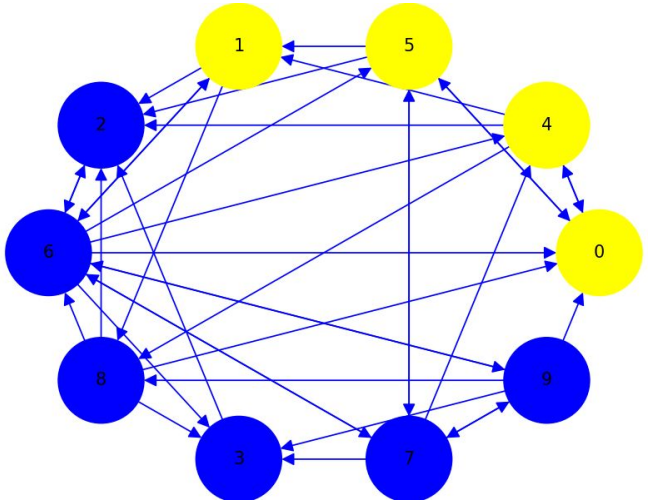
Paramétrer les noeuds

```
options = {
    'node_color': 'yellow',      # couleur du noeud
    'node_size': 3500,           # taille du noeud
    'width': 1,                  # épaisseur de l'arc
    'arrowstyle': '-|>',         # style des flèches (orienté)
    'arrowsize': 18,             # taille des flèches
    'edge_color':'blue',         # couleur des arcs
}

nx.draw(G, pos, with_labels = True,
        arrows=True, **options)
```



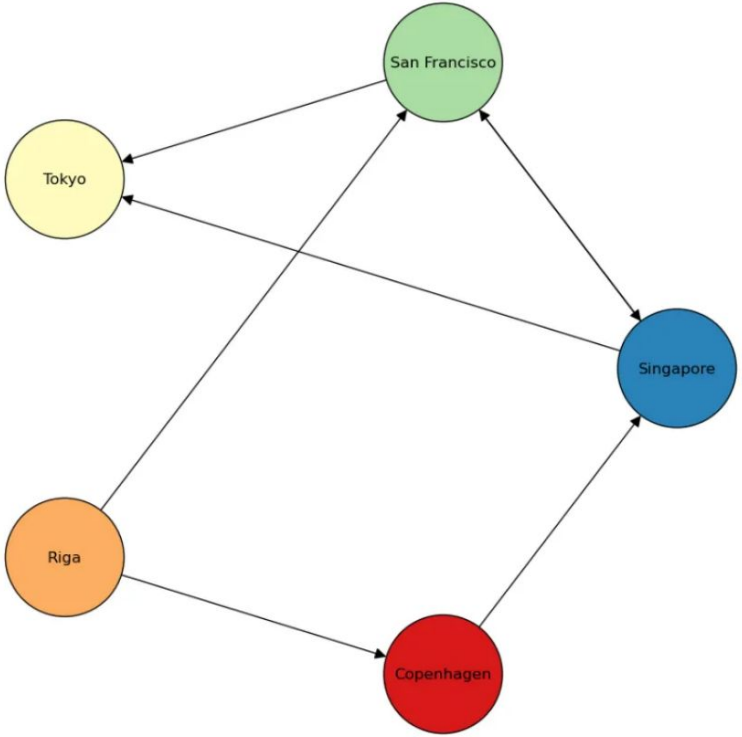
Paramétrer les noeuds

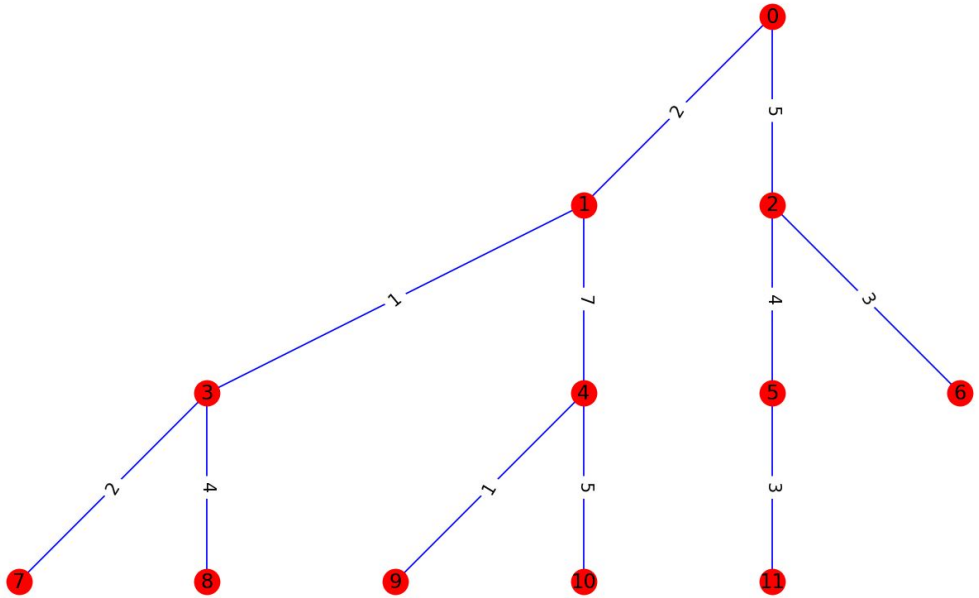
```
options = {
    'node_color': ['yellow']*4+['blue']*6,
    'node_size': 3500,
    'width': 1,
    'arrowstyle': '-|>',
    'arrowsize': 18,
    'edge_color':'blue',
}

nx.draw(G, pos, with_labels = True,
        arrows=True, **options)
```



Exercice

Tracer les graphes suivants :





Autre moteur de rendu (graphviz)

```
import networkx as nx
import matplotlib.pyplot as plt

# Création d'un graphe pondéré
G = nx.Graph()
G.add_weighted_edges_from([(0, 1, 2), (0, 2, 5), (1, 3, 1),
      (1, 4, 7), (2, 5, 4), (2, 6, 3), (3, 7, 2), (3, 8, 4),
      (4, 9, 1), (4, 10, 5), (5, 11, 3), (1, 2, 3), (3, 9, 2),
      (10, 11, 1), (6, 7, 7)])

# On utilise l'interface Python pour Graphviz, pour générer
# des dispositions de nœuds, moteur de rendu "dot"
pos = nx.nx_agraph.graphviz_layout(G, prog='dot')
```

Autre moteur de rendu (graphviz)

```
fig, ax = plt.subplots(figsize=(6, 6))
ax.set_aspect('equal') # Même échelle pour les axes

nx.draw_networkx_edges(G, pos, ax=ax, edge_color='b', width=1)
nx.draw_networkx_nodes(G, pos, ax=ax, node_size=300,
                       node_color='r')
nx.draw_networkx_labels(G, pos,
          {i: f'{i}' for i in range(len(G.nodes()))},
          ax=ax, font_size=14)
nx.draw_networkx_edge_labels(G, pos,
          {(u, v): d['weight'] for u, v, d in G.edges(data=True)},
          font_size=12, ax=ax, label_pos=0.5)
plt.axis('off')
plt.show()
```

Graphviz : autres moteurs de rendu

Représenter automatiquement un graphe, d'une manière "jolie", n'est pas simple. D'où plusieurs moteurs de rendus :

- **dot** : Dessine des graphes dirigés acycliques (DAG), en produisant des dessins hiérarchiques ou en couches qui minimisent les arêtes croisées.

- **neato** : Dessine des graphes non orientés, à partir d'un algorithme d'optimisation représentant les nœuds comme des particules chargées et les arêtes comme des ressorts, minimisant l'énergie du système pour obtenir une disposition équilibrée.

- **Circo (ou "circular")** : Place les nœuds sur un cercle et minimise les arêtes croisées.
  Adapté pour représenter les graphes qui ont des cycles ou des sous-graphes circulaires.

Graphviz : autres moteurs de rendu

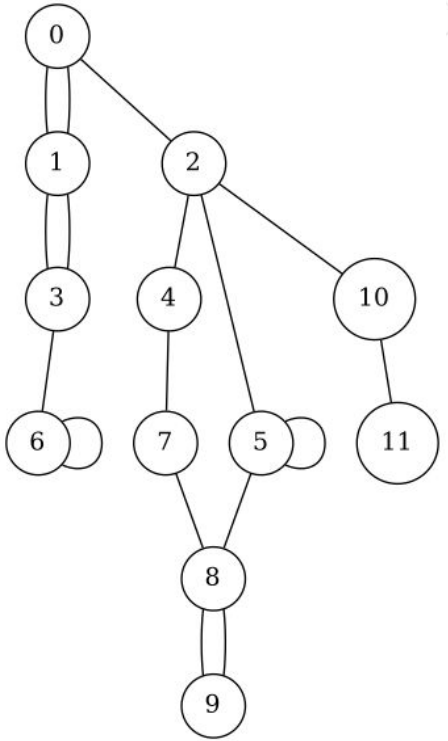
- **twopi** : Dessine les graphes en utilisant une disposition radiale, où les nœuds sont placés sur des cercles concentriques en fonction de leur distance par rapport à un nœud central.
  Particulièrement utile pour visualiser les graphes avec une structure hiérarchique ou radiale.

- **fdp** : Utilise un algorithme différent de neato pour minimiser l'énergie du système et générer une disposition équilibrée.
  Adapté pour les graphes non orientés.

- **sfdp** : extension de fdp pour gérer de manière efficace les graphes de grande taille.

Cas des graphes non simples

```
import graphviz as gv
G = gv.Graph()
G.edge_attr.update(arrowsize='0.8')
G.node_attr.update(shape='circle')
for i in range(12):
    G.node(str(i))

G.edge('0', '1', key='1')
G.edge('0', '1', key='2')
:
:
G.edge('10', '11', key='1')
# Afficher le graphe
G.view()
```



Divers types de (sous-)graphes

Graphes simples : exercice

Les objets Graph() et DiGraph() n'ont pas d'arêtes (arcs) multiples, mais ils peuvent avoir des boucles. Faire une fonction qui teste si un graphe donné est simple.

Graphes simples : correction

Les objets Graph() et DiGraph() n'ont pas d'arêtes (arcs) multiples, mais ils peuvent avoir des boucles. Pour savoir si un tel graphe est simple :

```
>>> def is_simple(G):
...     return all(len(set(edge))==2 for edge in G.edges)
```

Graphes simples et complets : exercice

Les objets Graph() et DiGraph() n'ont pas d'arêtes (arcs) multiples, mais ils peuvent avoir des boucles. Pour savoir si un tel graphe est simple :

```
>>> def is_simple(G):
...     return all(len(set(edge))==2 for edge in G.edges)
```

Un graphe simple est **complet** quand tous les sommets sont adjacents. Il a donc n(n-1)/2 arêtes pour n sommets, et c'est une CNS. Faire une fonction qui teste cela.

Graphes simples et complets

Les objets Graph() et DiGraph() n'ont pas d'arêtes (arcs) multiples, mais ils peuvent avoir des boucles. Pour savoir si un tel graphe est simple :

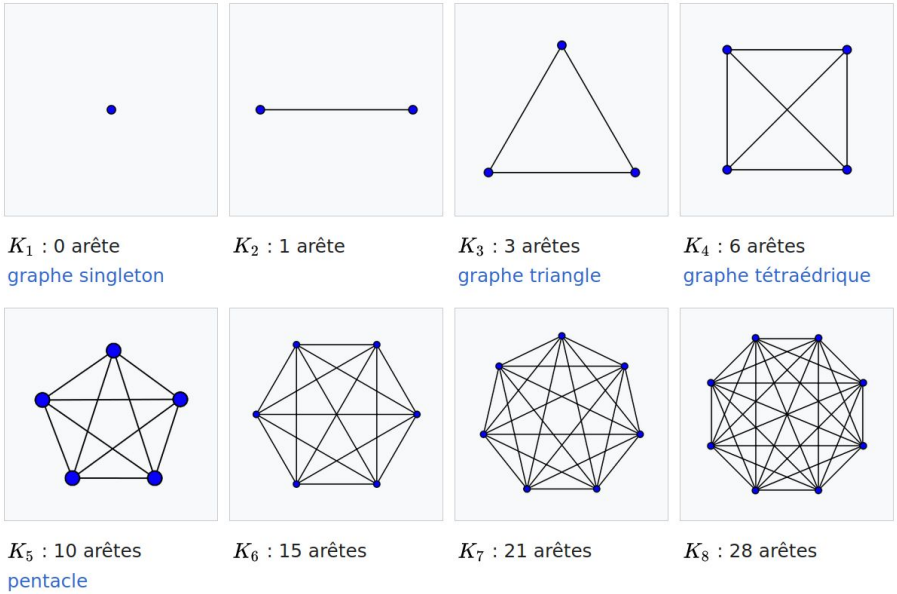
```
>>> def is_simple(G):
...     return all(len(set(edge))==2 for edge in G.edges)
```

Un graphe simple est **complet** quand tous les sommets sont adjacents. Il a donc n(n-1)/2 arêtes pour n sommets, et c'est une CNS :

```
>>> def is_complete(G):
...     n = G.order()
...     return is_simple(G) and (n*(n-1)//2 == G.size())
```

Les graphes K_n

À isomorphisme près, il n'existe qu'un seul graphe complet non orienté d'ordre n, que l'on note K_n (galerie : Wikipedia) :



H est-il un graphe partiel de G ?

H est un **graphe partiel** de G si on peut obtenir H en enlevant une ou plusieurs arêtes à G (sans toucher à ses sommets). Comment tester cela ?

H est-il un graphe partiel de G ?

H est un **graphe partiel** de G si on peut obtenir H en enlevant une ou plusieurs arêtes à G (sans toucher à ses sommets).

```
>>> nodes_equal = G.nodes == H.nodes
>>> edges_present = all(edge in G.edges for edge in H.edges)
>>> is_partial = nodes_equal and edges_present
```

H est-il sous-graphe de G ?

Un **sous-graphe** d'un graphe donné est obtenu en enlevant certains sommets, et toutes les arêtes incidentes à ces sommets.

Il faut donc tester que :

- Les sommets de H sont des sommets de G (inclusion)
- Les arêtes de H sont des arêtes de G (inclusion)
- Les arêtes de G constituées de sommets de H sont des arêtes de H

Le faire…

H est-il sous-graphe de G ?

Un **sous-graphe** d'un graphe donné est obtenu en enlevant certains sommets, et toutes les arêtes incidentes à ces sommets.

```
def is_subgraph(H, G):
    nodes_present = all(node in G.nodes for node in H.nodes)
    edges_present = all(edge in G.edges for edge in H.edges)
    # Les arêtes de G constituées de sommets de H sont des arêtes de H
    last_cond = all(edge in H.edges
                    for edge in G.edges
                    if set(edge).issubset(set(H.nodes)))
    return nodes_present and edges_present and last_cond
```

Sous-graphe : exercice

Faire une fonction qui, à un graphe G et à une liste de ses sommets, produit le sous-graphe constitué de ces sommets (on pourra regarder la méthode subgraph).

Sous-graphe : exercice

Faire une fonction qui, à un graphe G et à une liste de ses sommets, produit le sous-graphe constitué de ces sommets (on pourra regarder la méthode subgraph).

```
def induced_subgraph(G, vertices):
    return G.subgraph(vertices).copy()
```