femto-st SCIENCES & TECHNOLOGIES

Les graphes en python

C. Guyeux

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

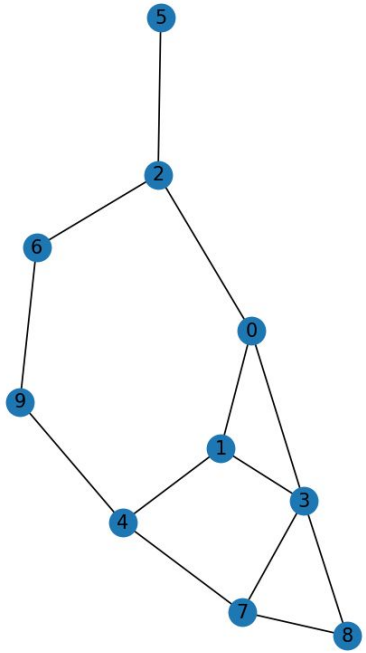
Préliminaire : installation de bibliothèques

Dans un terminal :

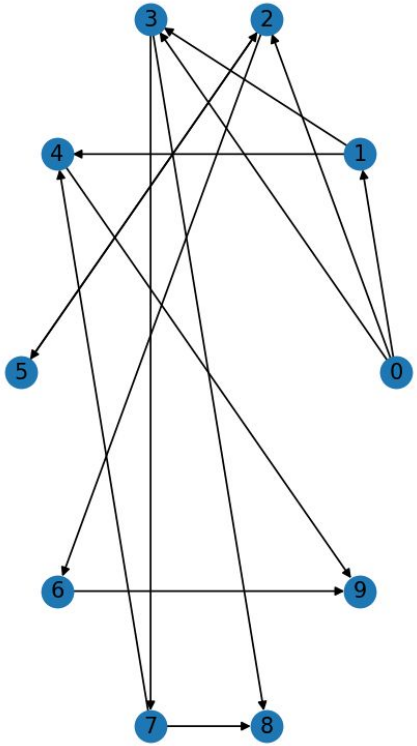
```
pip install ——user ——upgrade networkx
pip install ——user ——upgrade matplotlib
pip install ——user ——upgrade scipy
```

Zoologie des graphes

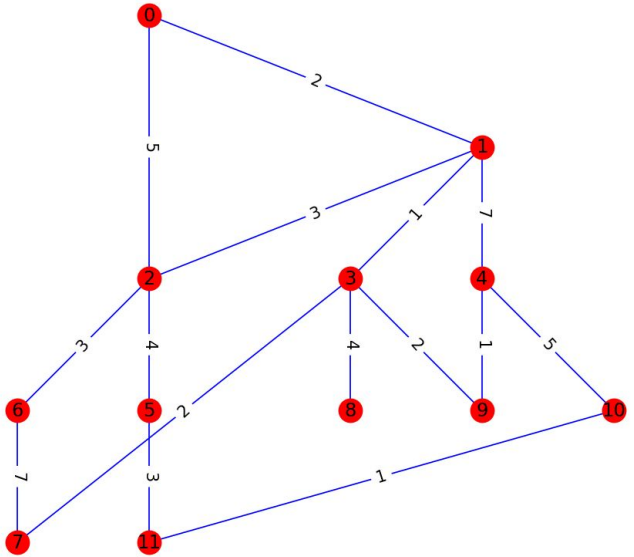
Graphe non orienté simple



Graphe orienté simple

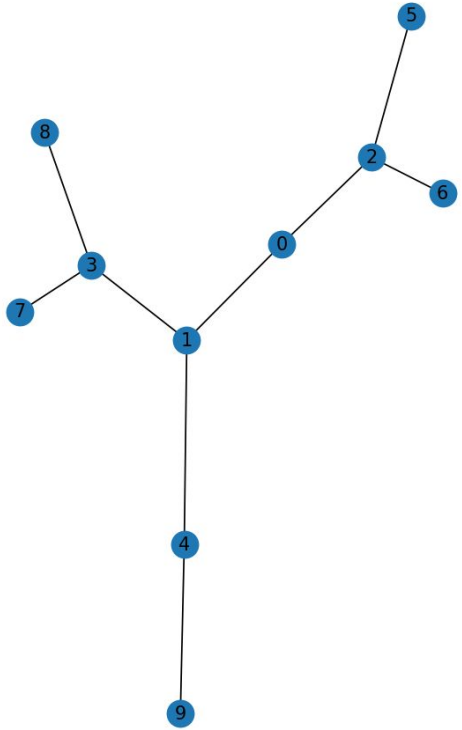


Graphe pondéré

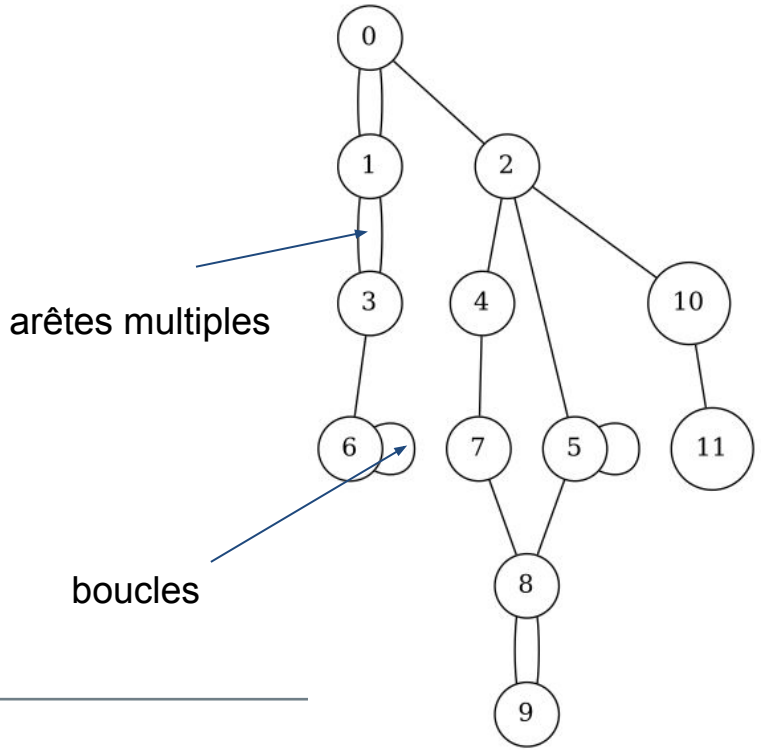


Graphes et arbres

Arbre



Graphe non orienté, pas simple



femto-st SCIENCES & TECHNOLOGIES

Motivation

Plus court chemin routier

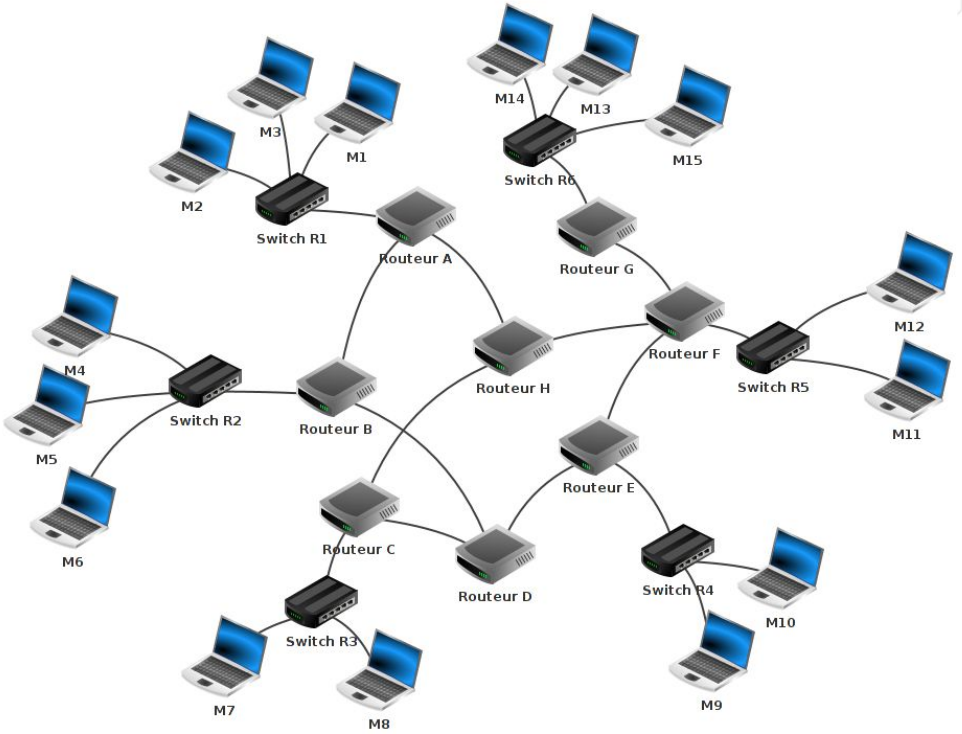
Quel est le plus court chemin pour se rendre de A à B…

…par exemple, dans le graphe OpenStreetMaps ?

Protocoles de routage

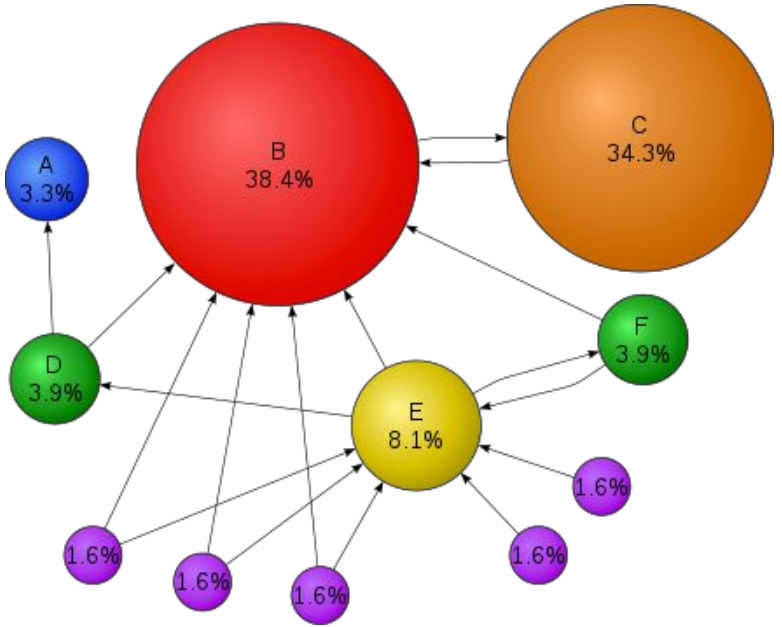
ping : le serveur cible est-il accessible ?

Si oui, trouver le plus rapide chemin vers lui.



Page rank de google

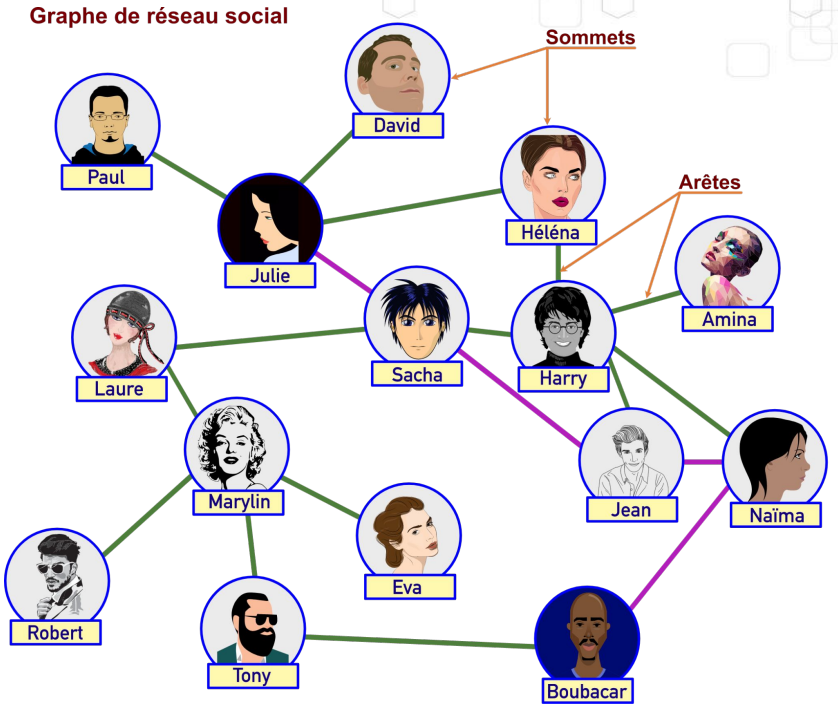
Un site web est d'autant plus important que des sites importants pointent sur lui.



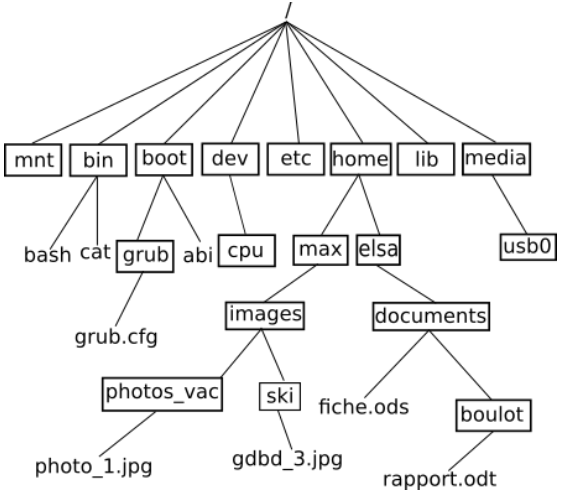
Analyse de réseaux sociaux

Cambridge Analytica :

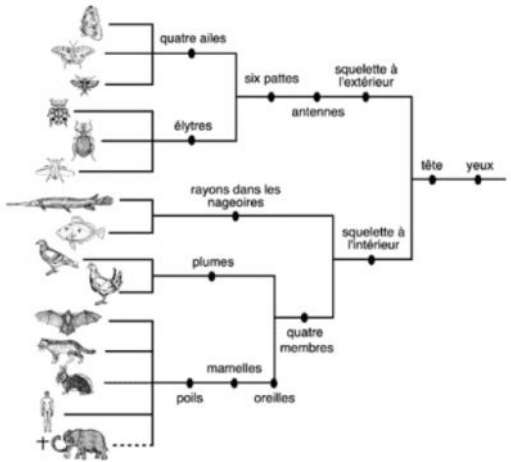
- Collecte de données (2014) : par un quiz, en récoltant par violation des infos sur les amis.
- Campagne présidentielle de 2016 : micro-ciblage (qui influencer ?)
- Diffusion de messages politiques pour faire basculer des “swing states”



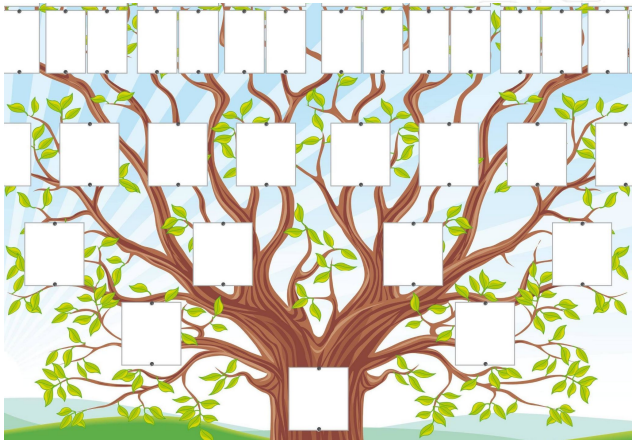
Arbres et arborescence



Arborescence GNU/Linux

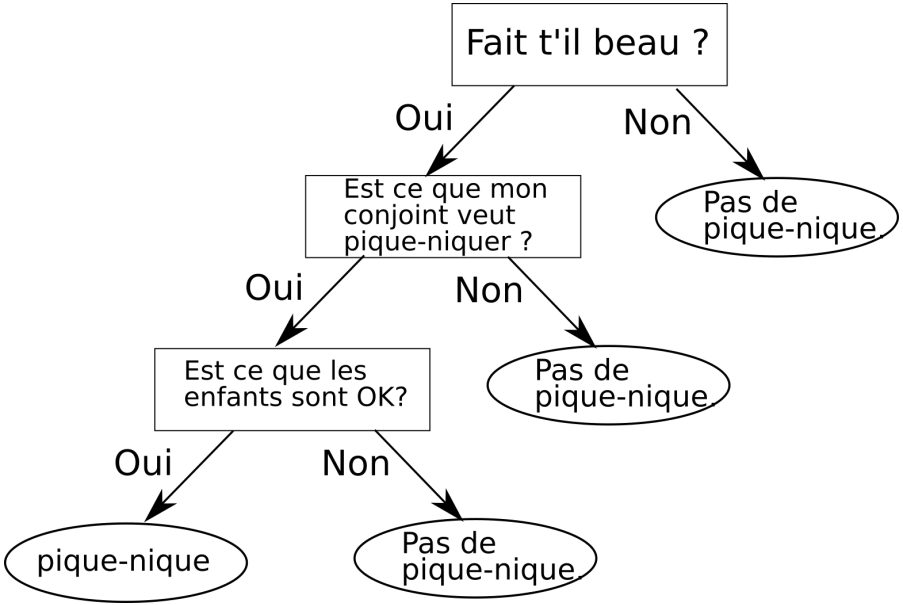


Arbre phylogénétique

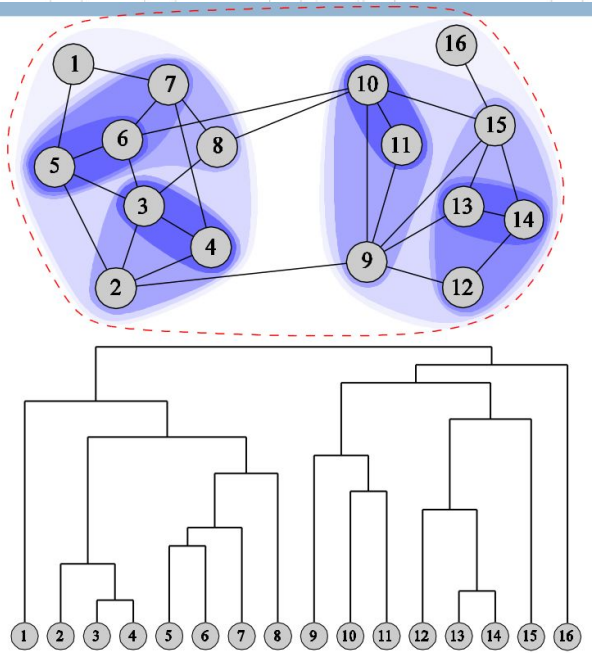


Arbre généalogique

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

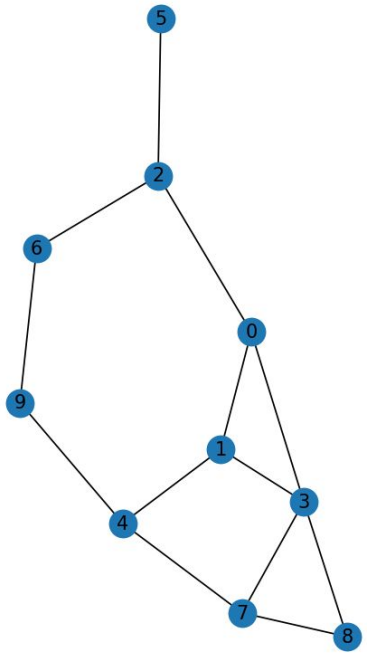
Graphe non orienté simple

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

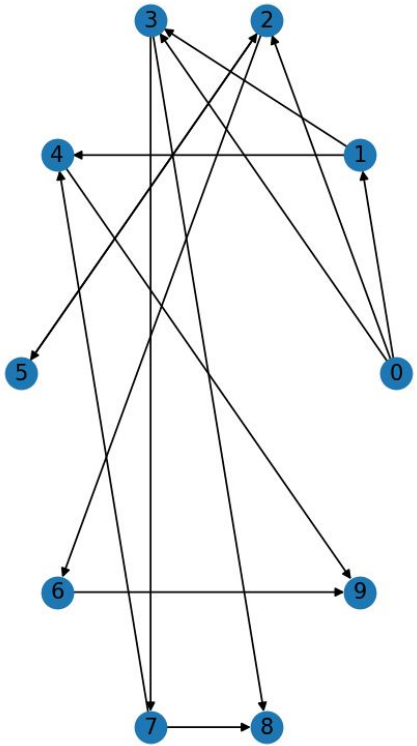
Un **graphe orienté simple** G est un couple ordonné (V, A), où :

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.



Adjacence, incidence, degré

Cas orienté :

- Deux arcs sont **adjacents** s'ils ont au moins une extrémité commune
- Pour un arc (x, y), y est **successeur** de x, et x est **prédécesseur** de y. x et y sont dits **voisins**, ou **adjacents**.
- L'arc (x, y) est **incident extérieurement** à x, et **incident intérieurement** à y.
- Le demi-degré extérieur $d^+(x)$ (resp. intérieur $d^-(x)$)est le nombre d'arc incident extérieurement (resp. intérieurement) à x. Enfin, le degré $d(x)$ vaut $d^+(x)+d^-(x)$.

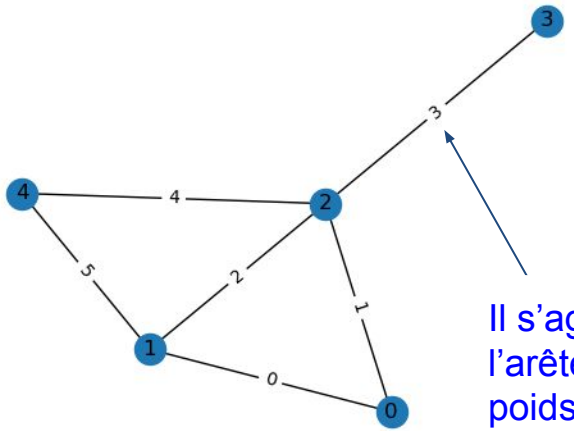
Cas non orienté :

- Une arête [x, y] a pour extrémités x et y, elle est incidente à x et y. Ces derniers sont dits **voisins**.
- Le degré d(x) d'un sommet x est son nombre d'arêtes incidentes.

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

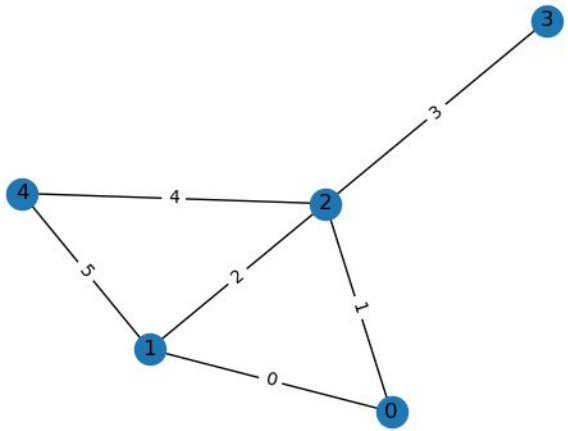
sommet 4

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).

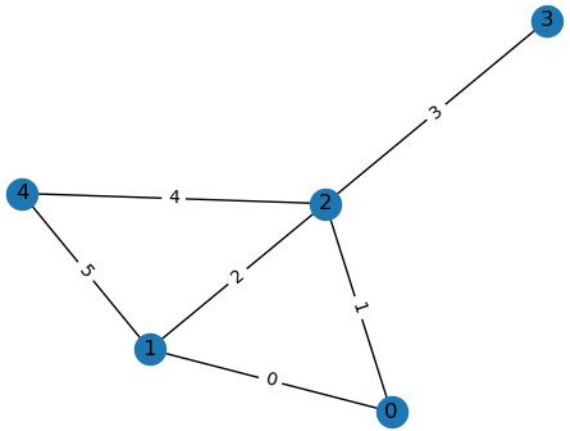



```
0: [1, 2]
1: [0, 2, 4]
2: [0, 1, 3, 4]
3: [2]
4: [2, 1]
```

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) :

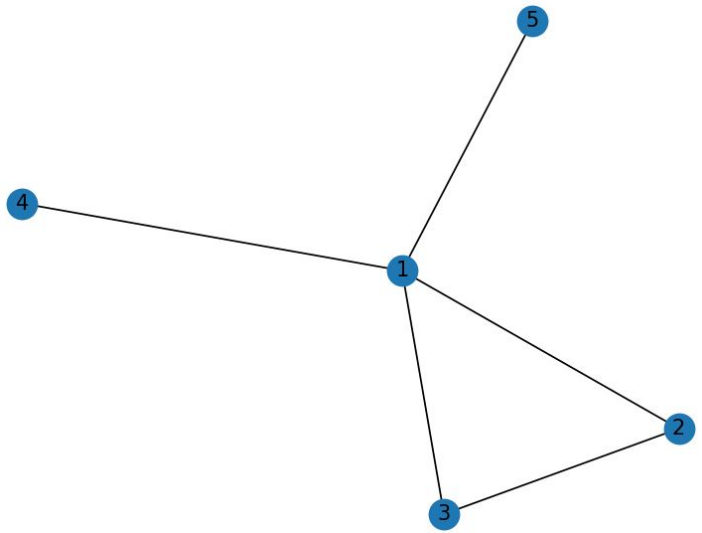
```
>>> G.add_node(1)
>>> G.add_nodes_from([2, 3, 4])
```

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
```

Compréhension de liste : efficacité

```
>>> from time import time
>>> t=time()
>>> nums = ""
>>> for n in range(20000000):
...     nums += str(n)
...
>>> print(time()-t)
4.4729392528533936

>>> t=time()
>>> nums = [str(n) for n in range(20000000)]
>>> s = "".join(nums)
>>> print(time()-t)
2.752303123474121
```

On modifie un objet par nature immuable.

Les compréhension de listes sont aussi plus rapides que map ou filter, tout en étant plus claires.

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 ?

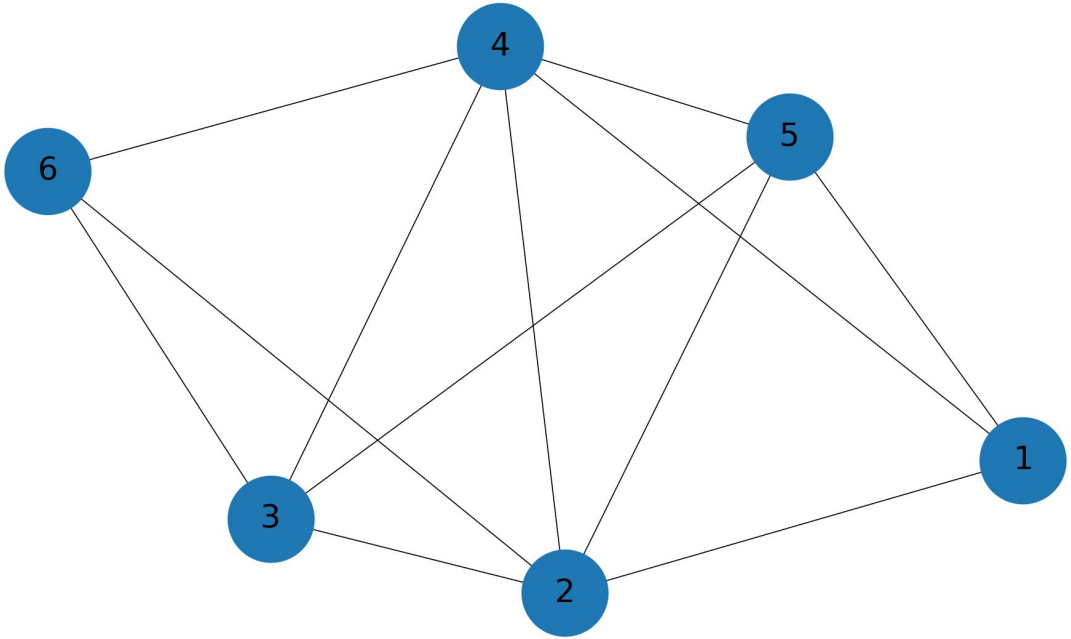
Il est très difficile de déterminer le diamètre exact d'un tel graphe, car il y a plusieurs facteurs à prendre en compte, notamment la taille et la complexité du réseau social humain. En théorie, le diamètre d'un graphe est la plus longue distance entre deux sommets du graphe. Dans le contexte d'un réseau 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)'

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

f"{variable:format}"

Graphes et matrices

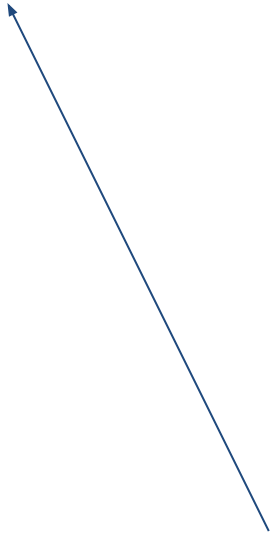
Matrice d'adjacence :

```
>>> 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

3. Faire une fonction qui, à partir de la matrice d'adjacence, renvoie la liste des successeurs et celle des prédécesseurs d'un sommet x.
   a.

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

```
>>> 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
```

TODO : gérer les sommets de degré 0

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_array(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 (non orienté)

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 (non orienté)

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

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

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

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

Lemme des poignées de mains (non orienté)

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

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

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

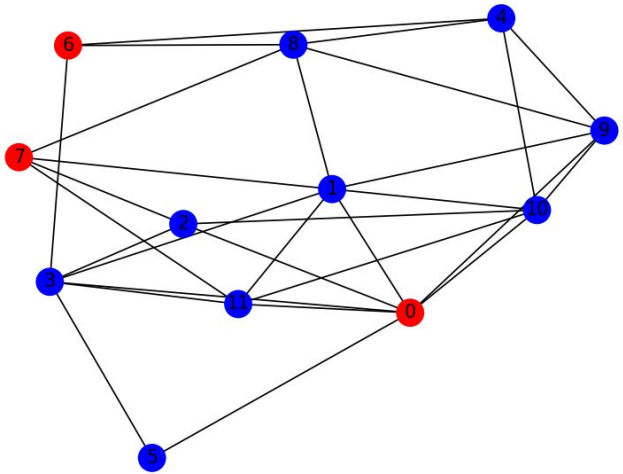
```
len([k for k in G.nodes() if G.degree(k)%2 == 1])%2 == 0
```

=> Il n'est pas possible de relier 15 ordinateurs, de sorte que chaque appareil soit relié avec exactement trois autres.

L’affichage de graphes

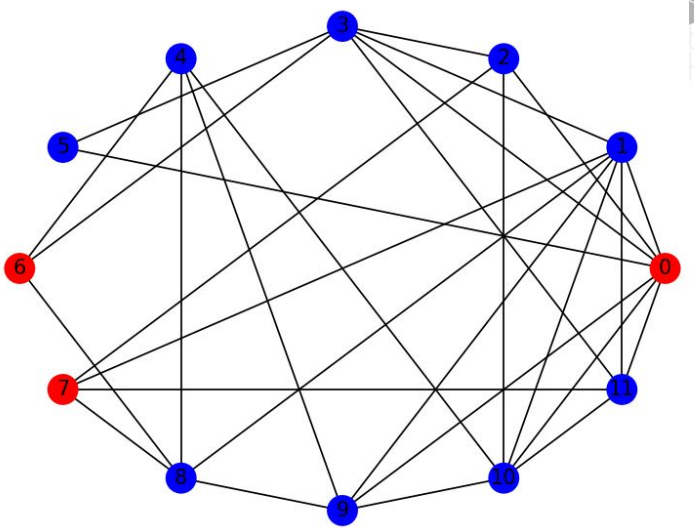
Colorer les sommets

```
>>> 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()
```



Layout (mise en page)

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

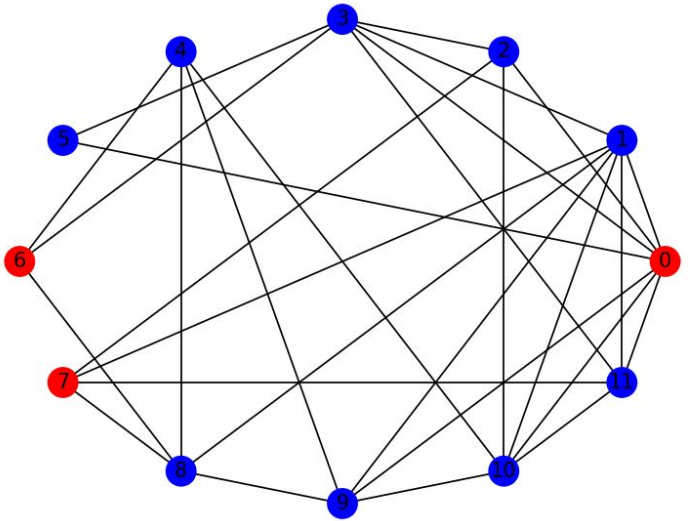


Autres layouts :

- `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)

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



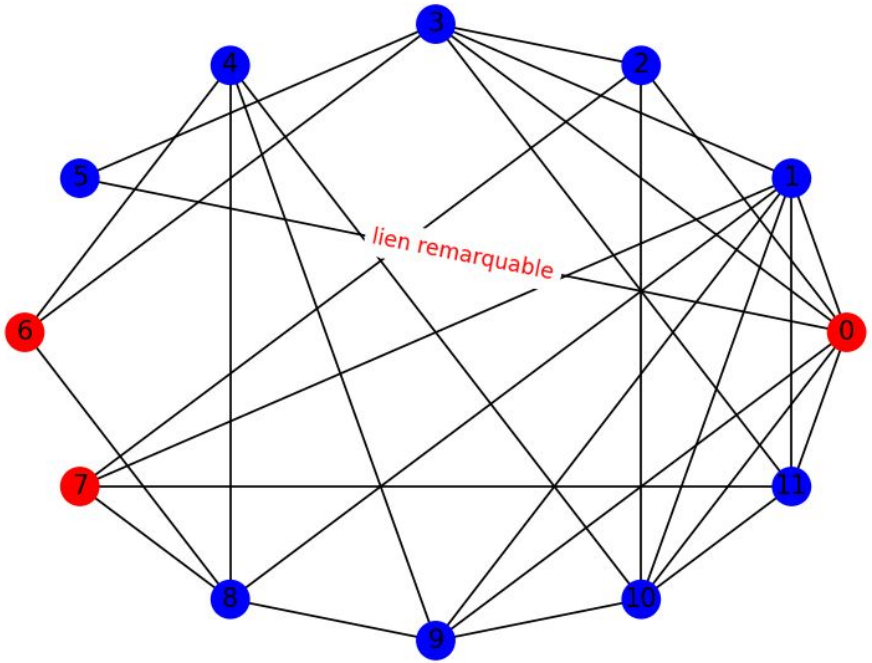
Autres layouts :

- `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)`

Tester les différentes mises en pages

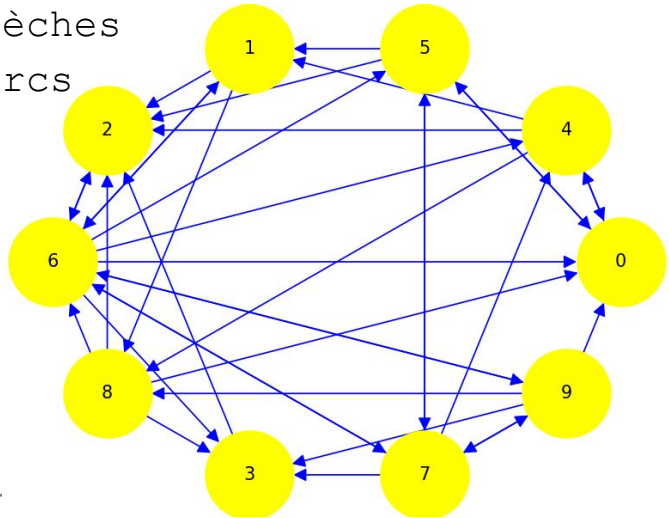
Nommer les arêtes

```
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()
```



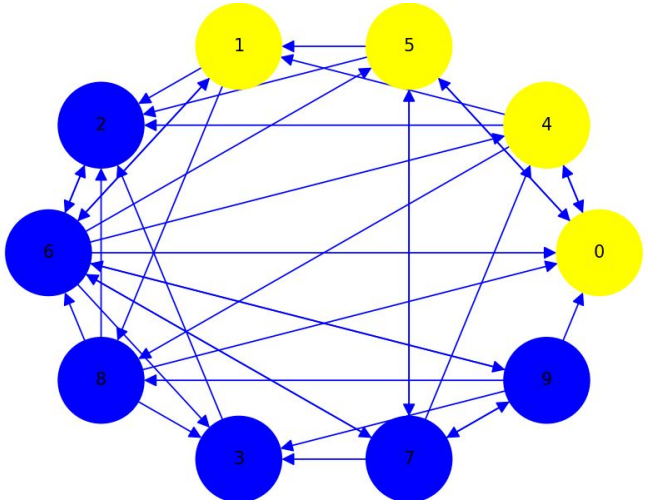
Paramétrer les noeuds

```
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)
```



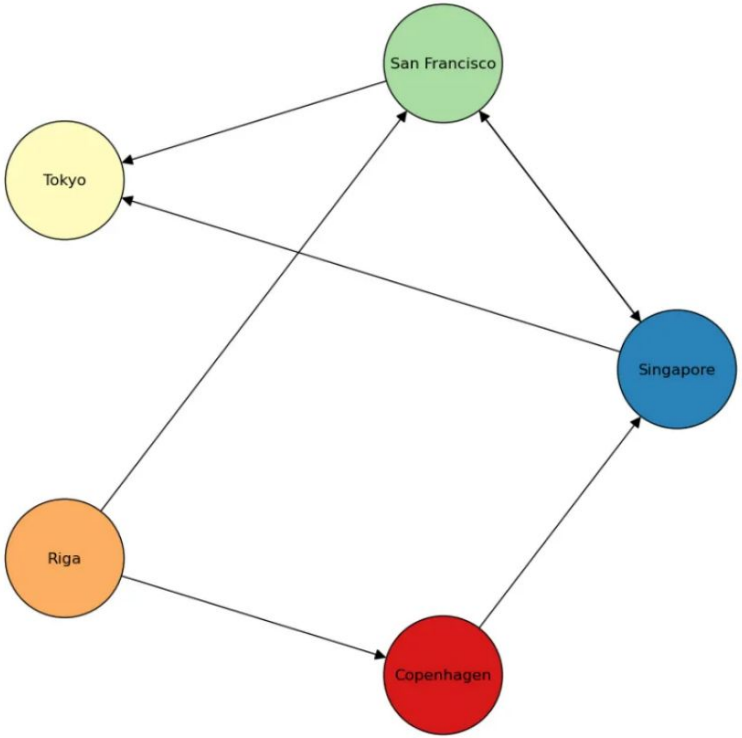
Paramétrer les noeuds

```
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 le graphe suivant :

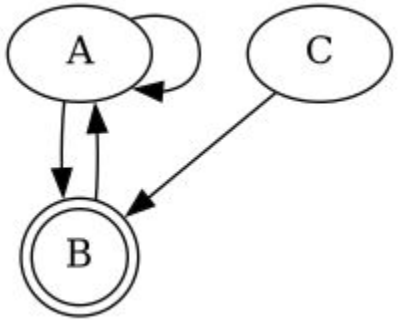


L’affichage de graphes graphviz

Graphviz : le format DOT

Le format DOT est un langage de description de graphe utilisé par le logiciel de visualisation de graphe Graphviz.

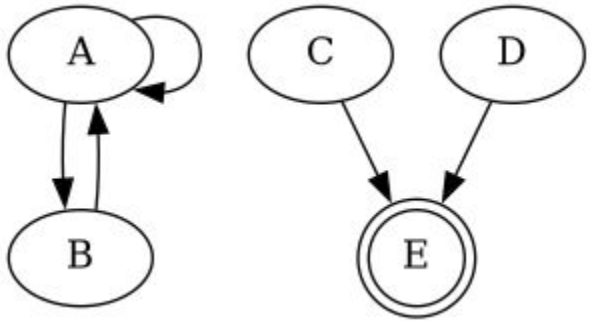
```
digraph G {
        A [shape=ellipse];
        B [shape=doublecircle];
        C [shape=ellipse];

        A -> A;
        A -> B;
        B -> A;
        C -> B;
}
```



dot -Tpng input.dot -o output.png

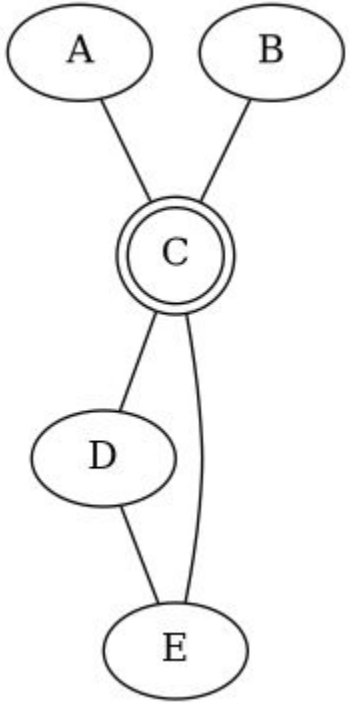
Graphviz : le format DOT

```
digraph G {
        A [shape=ellipse];
        B [shape=ellipse];
        C [shape=ellipse];
        D [shape=ellipse];
        E [shape=doublecircle];

        A -> A;
        A -> B;
        B -> A;
        C -> E;
        D -> E;
}
```



Graphviz : graphe non orienté

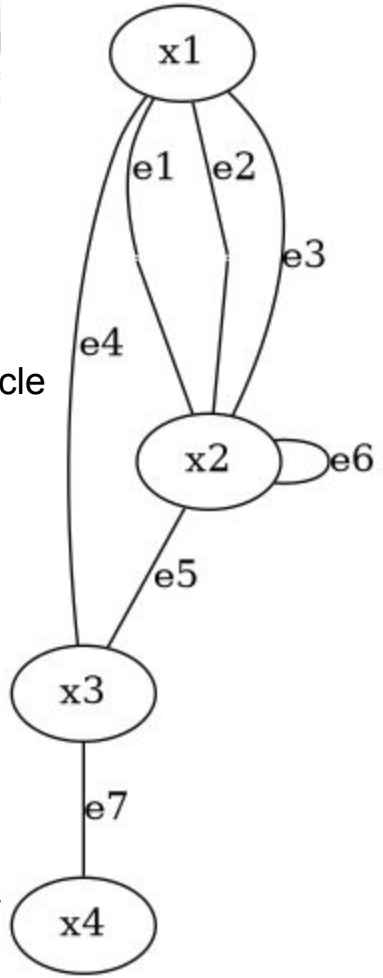
```
graph G {
        A [shape=ellipse];
        B [shape=ellipse];
        C [shape=doublecircle];
        D [shape=ellipse];
        E [shape=ellipse];

        A -- C;
        B -- C;
        C -- D;
        C -- E;
        D -- E;
}
```



Graphviz : les multigraphes

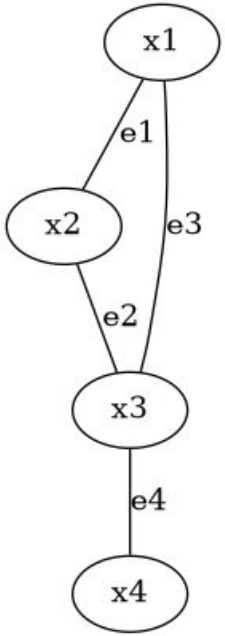
Création de sommets invisibles pour simuler plusieurs arêtes entre x1 et x2

```
graph G {
        x1 [shape=ellipse];
        x2 [shape=ellipse];
        x3 [shape=ellipse];
        x4 [shape=ellipse];

        i1 [shape=point, width=0, height=0, label=""];
        i2 [shape=point, width=0, height=0, label=""];

        // Création des arêtes
        x1 -- i1 [label="e1"];
        i1 -- x2;
        x1 -- i2 [label="e2"];
        i2 -- x2;
        x1 -- x2 [label="e3"];
        x1 -- x3 [label="e4"];
        x2 -- x3 [label="e5"];
        x2 -- x2 [label="e6"]; // Boucle sur x2
        x3 -- x4 [label="e7"];
}
```



Graphviz : exercice

Reproduire ce graphe :



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.

Autre moteur de rendu (graphviz)

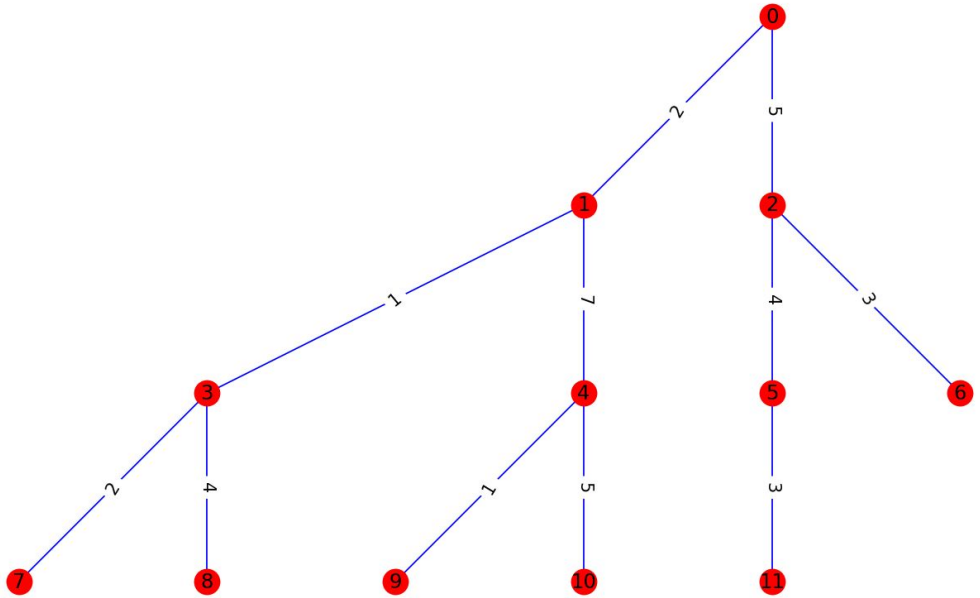
```
import networkx as nx
import matplotlib.pyplot as plt

# 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()
```

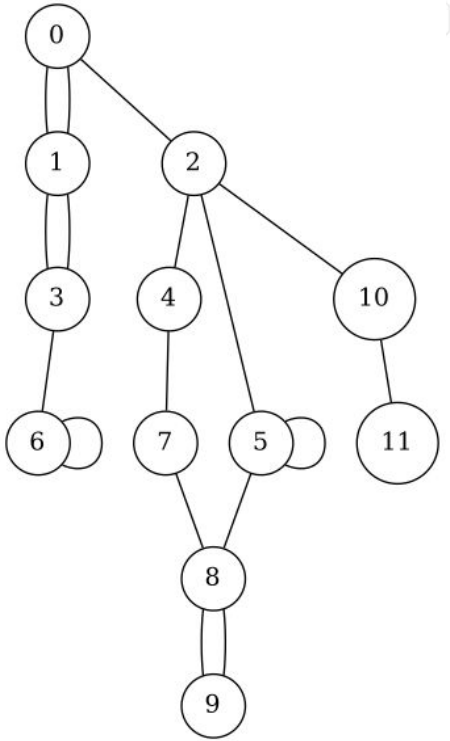
Exercice

Tracer le graphe suivant :



Bibliothèque graphviz

```
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()
```



Quelques graphes particuliers

(Anti)symétrique, complémentaire, inverse

Un graphe orienté est :

- **symétrique** si l'existence de l'arc (x,y) implique celle de (y,x). Il est **antisymétrique** si l'existence de l'arc (x,y) implique que l'arc (y,x) n'existe pas.

- le **complémentaire** d'un graphe orienté G si ses arcs sont ceux qui ne sont pas dans G.

- l'**inverse** d'un graphe orienté G quand on inverse tous ses arcs.

(Anti)symétrique, complémentaire, inverse

Un graphe orienté est :

- **symétrique** si l'existence de l'arc (x,y) implique celle de (y,x). Il est **antisymétrique** si l'existence de l'arc (x,y) implique que l'arc (y,x) n'existe pas.

- le **complémentaire** d'un graphe orienté G si ses arcs sont ceux qui ne sont pas dans G.

- l'**inverse** d'un graphe orienté G quand on inverse tous ses arcs.

=> Faire des fonctions testant ou produisant cela.

(Anti)symétrique : correction

```
def est_symetrique(G):
    for a,b in G.edges():
        if (b,a) in G.edges():
            return False
    return True

def est_anti_symetrique(G):
    for a,b in G.edges():
        if (b,a) not in G.edges():
            return False
    return True
```

Complémentaire : correction

```
from itertools import product

def complementaire(G):
    comp = nx.DiGraph()
    comp.add_nodes_from(G.nodes)
    for a,b in product(G.nodes(), G.nodes()):
        if a != b and (a,b) not in list(G.edges()):
            comp.add_edge(a, b)
    return comp
```

Inverse : correction

```
def inverse(G):
    comp = nx.DiGraph()
    comp.add_nodes_from(G.nodes)
    for a,b in G.edges():
        comp.add_edge(b, a)
    return comp
```

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 :

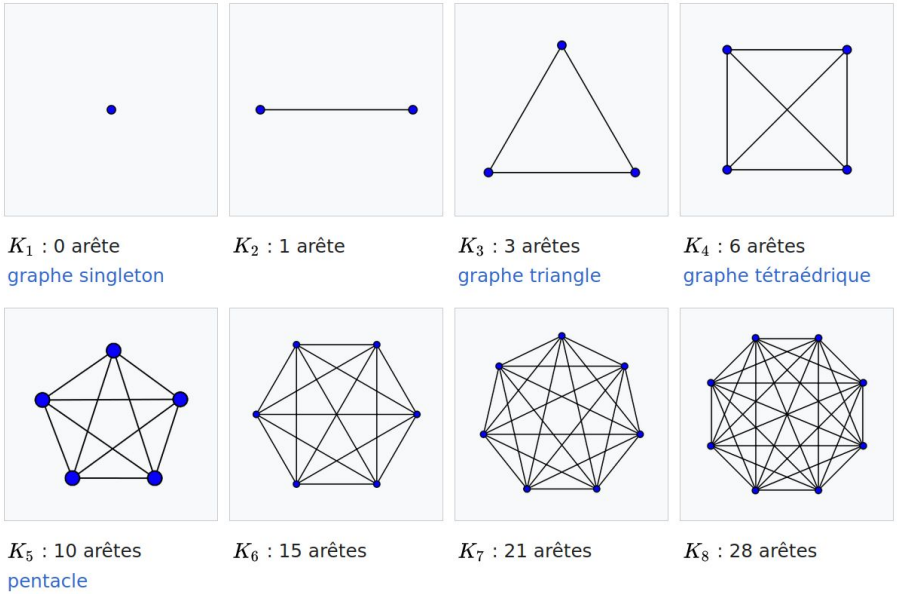
```
>>> def is_simple(G):
...     return all(len(set(edge))==2 for edge in G.edges)
```

Un graphe simple est **complet** quand tous les sommets (du graphe non orienté sous-jacent) sont adjacents. Il a donc n(n-1)/2 arêtes pour n sommets, 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 en l'honneur de Kuratowski (galerie : Wikipedia) :




```
nx.complete_graph(n)
```

Graphes et densité

La **densité** d'un graphe est son nombre d'arcs/arêtes rapporté au nombre maximal d'arc/arêtes théoriquement possible.

- Pour les graphes simples à N sommets (donc sans boucles), cela revient à diviser par N(N-1) dans le cas orienté, et par N(N-1)/2 dans le cas non orienté.

- Un graphe complet a donc une densité de 1.

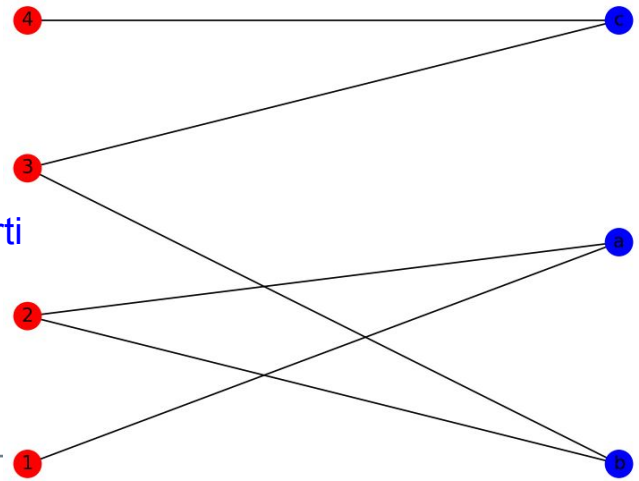
- `nx.density(G)`

Graphes biparti

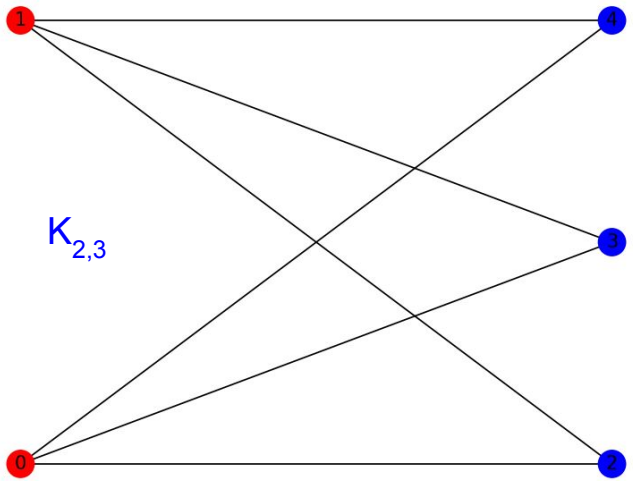
Un graphe est **biparti** si on peut diviser ses sommets en deux sous-ensembles X et Y avec aucune connexion entre deux sommets de X ou deux de Y.

Il est **biparti complet** si chaque sommet de X est connecté à tous les sommets de Y. On le note alors $K_{m,n}$, où m est le nombre de sommets de X, et n celui de Y.

Graphe biparti



$K_{2,3}$




```
nx.complete_bipartite_graph(2,
3)
```

Graphes biparti : applications

- En mathématique, représenter une fonction
- Modéliser des problèmes d'affectation
- …

Divers types de (sous-)graphes

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()
```

Graphes connexes

Networkx : compléments

Arêtes incidentes à un noeud :

```
list(G.edges(mon_noeud))
```

pour les graphes orientés, arc entrant / sortant :

```
list(G.in_edges(mon_noeud)), list(G.out_edges(mon_noeud))
```

Supprimer un sommet du graphe :

```
G.remove_node(3)
```

et pour les arêtes :

```
G.remove_edge(2, 3)
```

Copier un graphe :

```
G_copy = G.copy()
```

Sur les parcours

- Un **chemin** est une séquence d'arcs, dans laquelle les extrémités initiales/finales coïncident. Un chemin fermé est un **circuit**.

- Un sommet y est **descendant** de x s'il est situé sur un chemin d'origine x. x est alors **ascendant**, ou **ancêtre** de y.

- La **chaîne** et le **cycle** sont l'équivalent pour les graphes non orientés.

- Un **parcours** (chaîne ou chemin) est **élémentaire** s'il n'emprunte qu'une fois ses sommets.

- La **fermeture transitive** de x est l'ensemble des descendants de x.

Networkx et les parcours

Fermeture transitive de x :

```
nx.descendants(G, x)
```

et savoir si y est un descendant de x :

```
y in nx.descendants(G, x)
```

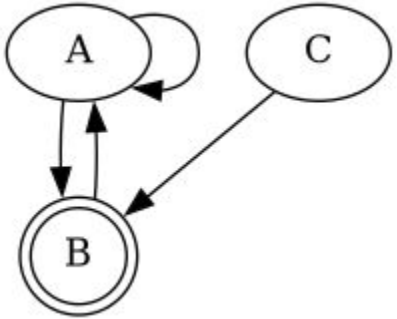
Pour obtenir les ancêtres de x :

```
nx.ancestors(G, x)
```

Sur la connexité

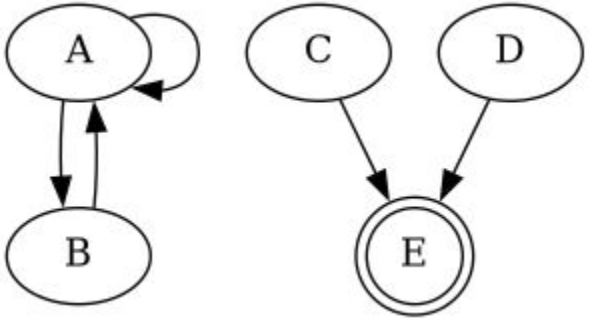
- Un graphe non orienté est **connexe** s'il existe une chaîne entre toute paire de sommets.
- S'il n'est pas connexe, on peut identifier plusieurs sous-graphes connexes, maximaux au sens de l'inclusion : les **composantes connexes**.
  Ce sont les classes d'équivalence de la relation binaire :
  $x \mathcal{R} y \Leftrightarrow$ il existe une chaîne entre x et y
- Cas des graphes orientés :
  - **Connexité faible** : si le graphe non orienté, obtenu en remplaçant arcs par arêtes, est connexe.
  - **Connexité forte** : s'il existe un chemin entre toute paire de sommets
    $x \mathcal{R} y \Leftrightarrow$ il existe un chemin de x à y et de y à x
- Un **point d'articulation** est un sommet qui augmente le nombre de composantes connexes si on l'enlève. Un **isthme** est un arc (arête) qui a la même propriété.

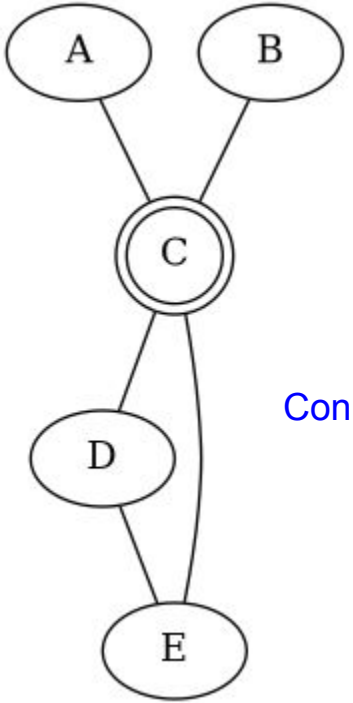
Sur la connexité : exemples



Connexe

Non connexe





Connexe

Les points d'articulation sont les sommets doublement cerclés

Sur la connexité : networkx

Tester la connexité d'un graphe non orienté :

```
nx.is_connected(G)
```

Et pour les graphes orientés :

```
nx.is_weakly_connected(G)
nx.is_strongly_connected(G)
```

Pour les composantes connexes d'un graphe non orienté :

```
nx.number_connected_components(G)
for component in nx.connected_components(G)
    print(component)
```

et dans le cas orienté :

```
nx.weakly_connected_components(G)
nx.strongly_connected_components(G)
```

Sur la connexité : exercice

1. Faire une fonction qui teste si un noeud donné est un point d'articulation d'un graphe
2. Idem pour tester si une arête est un isthme.

Sur la connexité : exercice

1. Faire une fonction qui teste si un noeud donné est un point d'articulation d'un graphe
2. Idem pour tester si une arête est un isthme.

```
def is_articulation_point(G, node):
    G_copy = G.copy()
    G_copy.remove_node(node)
    return nx.number_connected_components(G_copy) >
nx.number_connected_components(G)

def is_isthme(G, edge):
    G_copy = G.copy()
    G_copy.remove_edge(edge)
    return nx.number_connected_components(G_copy) >
nx.number_connected_components(G)
```

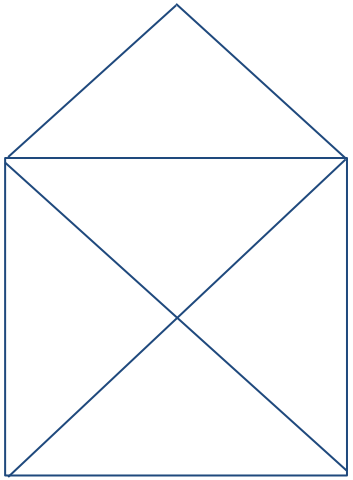
Parcours eulérien : définition

Un parcours est **eulérien** s'il passe exactement une fois par chaque arc (arête).

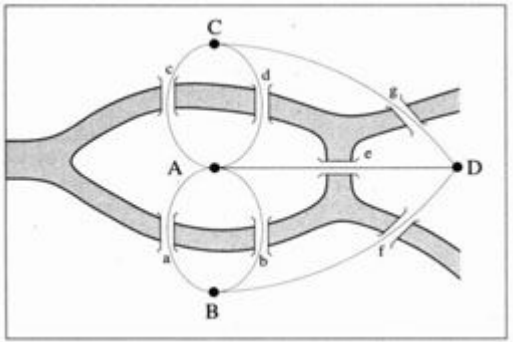
Parcours eulérien : exemples

Un parcours est **eulérien** s'il passe exactement une fois par chaque arc (arête).

Exemples :



jeu de l'enveloppe



Ponts de Konigsberg (Euler)

Parcours eulérien : théorème (Euler)

Un multigraphe admet un parcours eulérien si et seulement s'il est connexe et possède 0 ou deux nœuds de degré impairs.

- S'il y a 0 nœuds impairs, le parcours est un cycle, et on peut donc partir de tout nœud.

- S'il a 2 nœuds impairs, le parcours est une chaîne reliant ces 2 nœuds.

=> Il suffit de savoir trouver un cycle dans le cas 0, car on peut se ramener du cas 2 au cas 0 en ajoutant une arête.

=> Le problème de l'enveloppe a donc une solution : il existe une chaîne eulérienne dont les extrémités sont les coins inférieurs (seuls sommets impairs).

Parcours eulérien : exercice

1. Faire une fonction qui teste si un graphe possède un parcours eulérien
2. Faire une fonction qui teste si un cycle donné est (ou non) eulérien

Parcours eulérien : exercice

1. Faire une fonction qui teste si un graphe possède un parcours eulérien
2. Faire une fonction qui teste si un cycle donné est (ou non) eulérien

```
def has_eulerian_path(G):
    odd_degree_nodes = [v for v, d in G.degree() if d % 2 == 1]
    return len(odd_degree_nodes) == 0 or len(odd_degree_nodes) == 2
```

```
def is_eulerian_cycle(G, path):
    return sorted([sorted(u) for u in path]) == sorted([sorted(u) for
u in G.edges()])
```

Cycle eulérien : en trouver un (exercice)

On choisit un sommet s_0 quelconque.

- On parcourt le graphe à partir de s_0 en suivant les arêtes « au hasard » mais sans passer deux fois par la même arête, jusqu'à « retomber » sur s0 (c'est toujours le cas si le graphe contient un cycle eulérien)
- On a donc un cycle $(s_0\ s_1\ \ldots\ s_p)$ sans répétition d'arête. Si ce cycle couvre toutes les arêtes du graphe, alors on a fini.
- Dans le cas contraire, comme le graphe est connexe, il existe forcément un sommet s_i du cycle ayant une arête incidente hors de ce cycle.
  - Soit alors $G' = (S', A')$ avec A' l'ensemble des arêtes non couvertes par ce cycle et S' l'ensemble des sommets qui sont des extrémités d'arêtes de A'.
  - On applique récursivement l'algorithme en partant du sommet si pour obtenir ainsi un cycle eulérien de G' : $(s_{i,0}\ s_{i,1}\ \ldots\ s_{i,q})$, où $s_{i,0} = s_{i,q} = s_i$ et $q = n - p$.
  - On obtient alors un cycle eulérien de G :
    $(s_0\ s_1\ \ldots\ s_i = s_{i,0}\ s_{i,1}\ \ldots\ s_{i,q} = s_i\ s_{i+1}\ \ldots\ s_p)$.

Cycle eulérien : en trouver un (solution)

```
import networkx as nx
from random import choice

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

def find_cycle(G, s0):
    odd_degree_nodes = [v for v, d in G.degree() if d % 2 == 1]
    assert len(odd_degree_nodes) == 0 or len(odd_degree_nodes) == 2
    if len(odd_degree_nodes) == 2:
        G.add_edge(*odd_degree_nodes)
    liste = [s0]
    G_copy = G.copy()
    a = s0
    arete = choice(list(G_copy.edges(a)))
    b = arete[(arete.index(a)+1)%2]
```

Cycle eulérien : en trouver un (solution)

```
G_copy.remove_edge(*arete)
while b != s0:
        liste.append(b)
        a = b
        arete = choice(list(G_copy.edges(a)))
        b = arete[(arete.index(a)+1)%2]
        G_copy.remove_edge(*arete)
if len(G_copy.edges())==0:
        return liste
else:
        for si in liste:
                if any([si in edge for edge in G_copy.edges()]):
                        break
        for k in list(G_copy.nodes()):
                if G_copy.degree(k) == 0:
                        G_copy.remove_node(k)
```

Cycle eulérien : en trouver un (solution)

```
        new_liste = find_cycle(G_copy, si)
        ni = liste.index(si)
        print(liste, si, ni, new_liste)
        return liste[:ni]+new_liste+liste[ni:]

find_cycle(G, choice(list(G.nodes())))
```

Parcours hamiltoniens : définition

- Un parcours est **hamiltonien** s'il passe exactement une fois par tout sommet du graphe.
  => et donc, au plus une fois par chaque arc/arête.

- On ne connaît pas d'algorithme efficace pour savoir (dans tous les cas) si un graphe admet un parcours hamiltonien.

- Exemples :
  - Trouver, sur un échiquier, un parcours du cavalier visitant une fois chaque case (un tel parcours existe).
  - Problème voisin : le voyageur de commerce (circuit hamiltonien de coût optimal).

Parcours hamiltoniens : exercice

Tester si un parcours donné est hamiltonien

Parcours hamiltoniens : exercice

Tester si un parcours donné est hamiltonien

```
def is_hamiltonian_path(G, path):
    sommets_visites = [k[0] for k in path]
    une_fois = len(sommets_visites) == len(set(sommets_visites))
    tous = set(sommets_visites) == set(G.nodes())
    return une_fois and tous
```

Parcours de graphes

Nombre de chaînes/chemins de longueur k

Dans le cas des graphes simples (orientés ou non), la puissance k-ième de la matrice d'adjacence A produit les chaînes simples (resp. chemins) de longueurs k dans un graphe non orienté (resp. orienté) :

> l'élément (i, j) de la matrice A^k est le nombre de chaînes simples/chemins de longueur k menant de i à j.

Nombre de chemins de longueur k : exercice

Faire une fonction qui, à :

- un graphe G,
- un couple de sommets (i, j),
- une longueur k,

retourne le nombre de chemins de longueur k dans G menant de i à j.

Parcours d'un graphe

Il existe deux principales manières de parcourir, systématiquement et efficacement, tous les noeuds d'un graphe, à partir d'un noeud donné :

- le **parcours en largeur** : exploration "horizontale"
  on visite tous ses voisins, puis les voisins de ses voisins, etc.

```
list(nx.bfs_tree(G, start_node))
```

- le **parcours en profondeur** : exploration "verticale"
  on part d'un nœud, on explore un de ses voisins, puis on continue à explorer les voisins de ce voisin jusqu'à atteindre un nœud sans voisins non visités, puis on revient en arrière pour explorer les autres voisins du nœud précédent non encore visités, et ainsi de suite.

```
list(nx.dfs_tree(G, start_node))
```

Parcours en largeur (breadth-first search BFS)

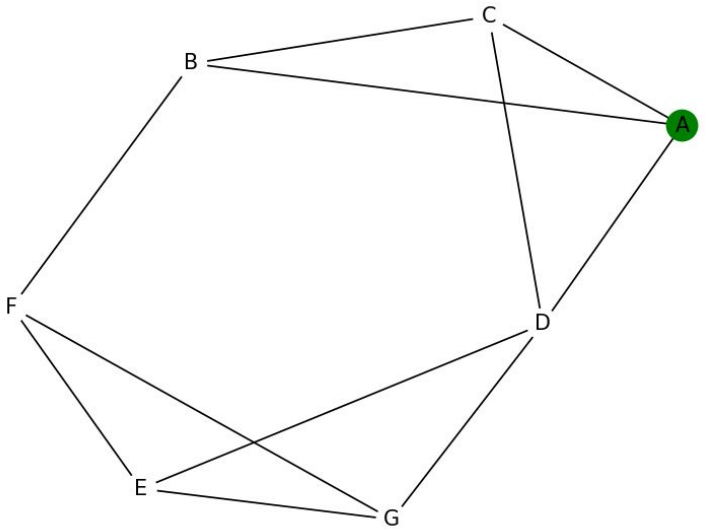
Algorithme du parcours en largeur :

1. mettre le nœud source dans la file ;
2. retirer le nœud du début de la file pour le traiter ;
3. mettre tous ses voisins non explorés dans la file (à la fin) ;
4. si la file n'est pas vide reprendre à l'étape 2.

Parcours en largeur : illustration

On part (par exemple) du nœud A dans le graphe suivant, comme étant le premier nœud à voir.

```
visite = []
a_voir = ['A']
```

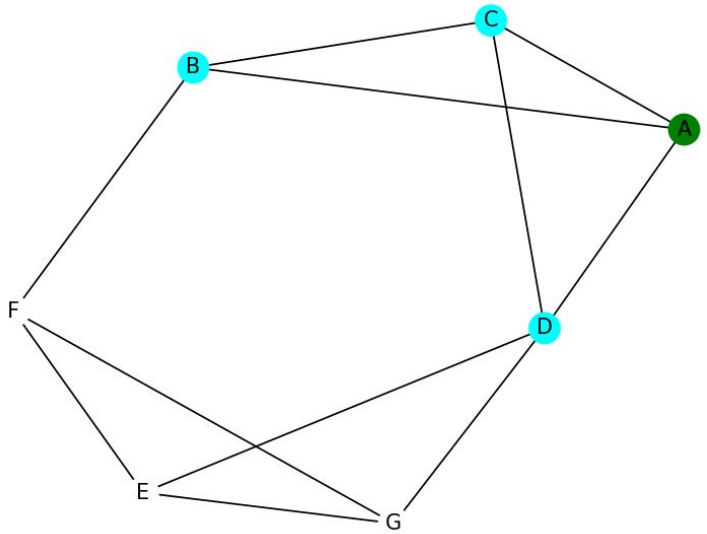


Parcours en largeur : illustration

On part (par exemple) du nœud A dans le graphe suivant, comme étant le premier nœud à voir.

On le visite, en récoltant tous ses voisins non visités, grâce à `list(G.adj['A'])`.

```
visite = ['A']
a_voir = ['B', 'C', 'D']
```



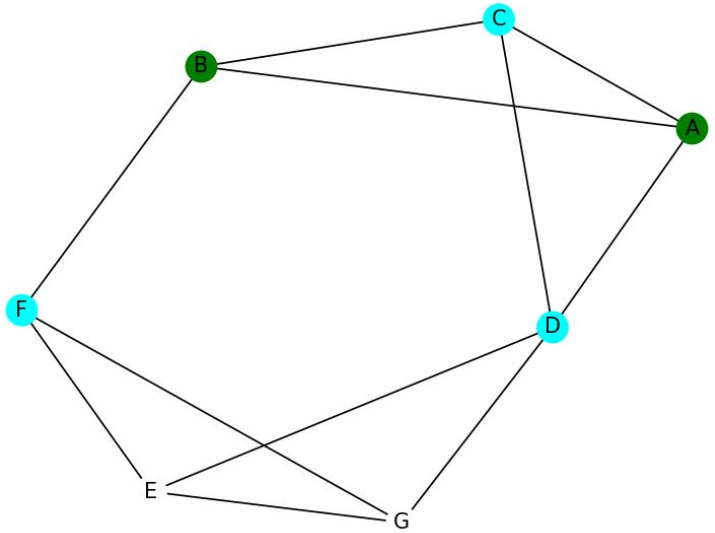
Parcours en largeur : illustration

On part (par exemple) du nœud A dans le graphe suivant, comme étant le premier nœud à voir.

On le visite, en récoltant tous ses voisins non visités, grâce à `list(G.adj['A'])`.

On visite le prochain nœud de la liste, ici B, en récoltant ses voisins non visités, puis en le marquant comme “visité”.

```
visite = ['A', 'B']
a_voir = ['C', 'D', 'F']
```



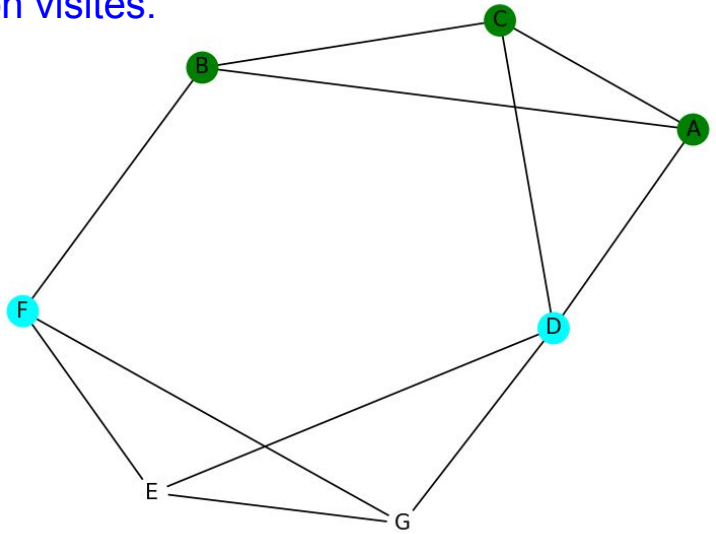
Parcours en largeur : illustration

On part (par exemple) du nœud A dans le graphe suivant, comme étant le premier nœud à voir.

On le visite, en récoltant tous ses voisins non visités, grâce à `list(G.adj['A'])`.

On visite le prochain nœud de la liste, ici B, en récoltant ses voisins non visités, puis en le marquant comme “visité”.

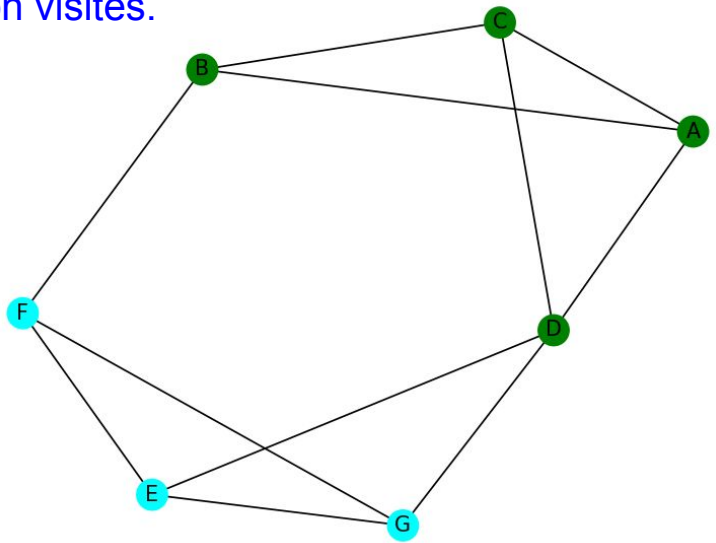
On passe au suivant de la liste (C), qui n’a pas de voisins non visités.




```
visite = ['A', 'B', 'C']
a_voir = ['D', 'F']
```

Parcours en largeur : illustration

On part (par exemple) du nœud A dans le graphe suivant, comme étant le premier nœud à voir.
On le visite, en récoltant tous ses voisins non visités, grâce à `list(G.adj['A'])`.
On visite le prochain nœud de la liste, ici B, en récoltant ses voisins non visités, puis en le marquant comme “visité”.
On passe au suivant de la liste (C), qui n’a pas de voisins non visités.
Puis à D, etc.




```
visite = ['A', 'B', 'C', 'D']
a_voir = ['F', 'E', 'G']
```

Parcours en largeur : exercice

Programmer le parcours en largeur.

On rappelle que :

- `list(G.adj['A'])` permet d'accéder aux voisins d'un nœud.
- Pour la gestion des listes de voisins, on doit tantôt extraire le premier de la liste, tantôt en ajouter à la fin. Il s'agit donc d'une file. Deux manières efficaces de réaliser cela :
  - avec `liste.pop(0)` et `liste.extend(nouvelle_liste)`
  - avec `collections.deque`, qui implémente les files.

Parcours en largeur : solution

```
from collections import deque

def bfs(G, start_node):
    visite = []
    a_voir = deque([start_node])

    while a_voir:
        node = a_voir.popleft()
        if node not in visite:
            visite.append(node)
            a_voir.extend(neighbor for neighbor in G.adj[node]
                          if neighbor not in visite)
    return visite
```

A parte : deque versus list.pop(0)

- list.pop(0) a une complexité temporelle de O(n), où n est le nombre d'éléments dans la liste.

  Lorsqu'on utilise pop(0) sur une liste, tous les éléments restants de la liste doivent être décalés d'une position vers la gauche pour combler l'espace laissé par l'élément supprimé.

- collections.deque.popleft() a une complexité temporelle de O(1), c'est-à-dire qu'elle est indépendante de la taille de la structure de données.

  Les objets deque sont implémentés en utilisant une structure de données de liste chaînée double, ce qui permet de supprimer rapidement et efficacement les éléments du début (ou de la fin) de la structure, sans avoir à déplacer les autres éléments.

Parcours en largeur : applications

Le parcours en largeur peut être utilisé pour :

1. Trouver le **plus court chemin** entre deux sommets dans un graphe non pondéré.
2. Détecter la présence de **cycles** dans un graphe.
3. Déterminer le **niveau** (la distance) de chaque sommet par rapport au sommet de départ.
4. Identifier les **composantes connexes** d'un graphe non orienté.
5. Analyser les **réseaux sociaux** en mesurant la distance entre les individus, les groupes ou les pages.

Parcours en largeur : exercices

Faire des fonctions pour…

1. Trouver le **plus court chemin** entre deux sommets dans un graphe non pondéré.
2. Détecter la présence de **cycles** dans un graphe.
3. Déterminer le **niveau** (la distance) de chaque sommet par rapport au sommet de départ.
4. Identifier les **composantes connexes** d'un graphe non orienté.

Parcours en profondeur (depth-first search DFS)

Dans le parcours en profondeur, on va chercher à aller le plus loin possible avec les descendants des descendants, avant de revenir au dernier sommet non traité.

Cela revient à considérer une pile plutôt qu'une file, ou à procéder récursivement sur les sommets visités.

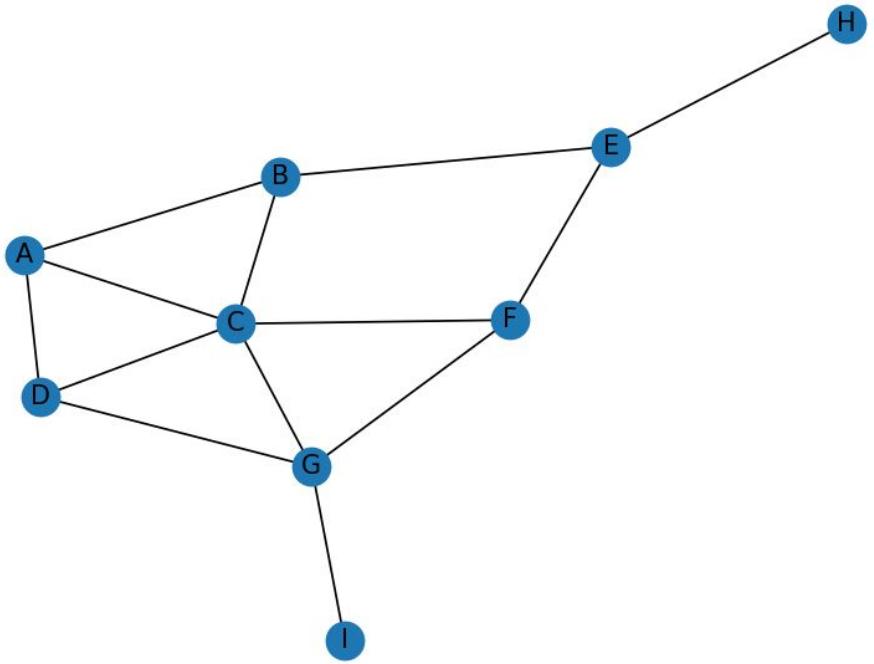
Parcours en profondeur : algorithme

```
parcours, deja_visites, a_traiter = [], [], [depart]
while a_traiter:
    sommet = a_traiter.pop()
    if sommet not in deja_visites:
        parcours.append(sommet)
        deja_visites.append(sommet)
        for voisin in G.adj[sommet]:
            if voisin not in deja_visites:
                a_traiter.append(voisin)
```

Parcours en profondeur : exercice

Faire un parcours en profondeur du graphe ci-contre.

Une solution :
['A', 'D', 'G', 'I', 'F', 'E', 'H', 'B', 'C']



Parcours en profondeur : applications

Le parcours en profondeur est utilisé pour :

1. Explorer complètement un graphe, en visitant tous les sommets et les arêtes.
2. Détecter la présence de cycles dans un graphe.
3. Identifier les ponts (arêtes dont la suppression augmente le nombre de composantes connexes) dans un graphe.
4. Identifier les points d'articulation (sommets dont la suppression augmente le nombre de composantes connexes) dans un graphe.

Parcours de graphe et labyrinthe

- Chaque intersection du labyrinthe correspond à un nœud du graphe, et chaque passage entre deux intersections correspond à une arête.

- Lorsqu'on cherche à résoudre un labyrinthe, on veut trouver un chemin entre un point de départ (entrée) et un point d'arrivée (sortie), ce qui peut se faire par un parcours de graphe.

Parcours de graphe et labyrinthe

- Le DFS explore les nœuds de manière "verticale", en suivant un chemin jusqu'à ce qu'il ne puisse plus aller plus loin avant de revenir en arrière pour explorer d'autres chemins.

- Cela revient par exemple à tourner à droite à chaque croisement.

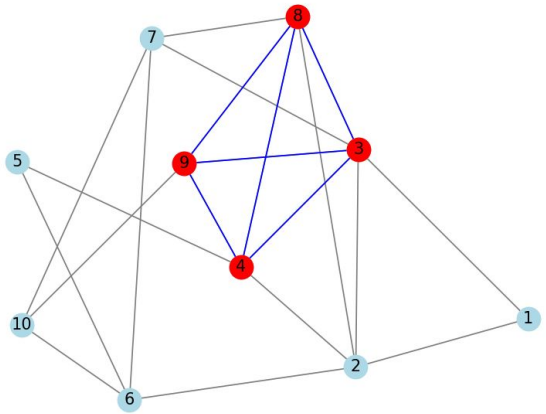
- Le BFS explore les nœuds de manière "horizontale", en visitant tous les nœuds (croisements) d'un même niveau avant de passer au niveau suivant. On arrête une fois à la sortie.

- Elle permet d'en déduire aussi le chemin le plus court entre l'entrée et la sortie, si celui-ci existe (ce que ne permet pas le DFS).

Cliques et stables

Les cliques de G

Une **clique** est un sous-graphe complet d'un graphe donné : en enlevant des sommets (et leurs arêtes), on arrive à quelque chose de complet.



Les cliques de G

Une **clique** est un sous-graphe complet d'un graphe donné : en enlevant des sommets (et leurs arêtes), on arrive à quelque chose de complet.

=> Comment le tester ?

Les cliques de G

Une clique est un sous-graphe complet d'un graphe donné : en enlevant des sommets (et leurs arêtes), on arrive à quelque chose de complet.

```
def is_clique(H, G):
    return is_complete(H) and is_subgraph(H, G)
```

Avec networkx :

```
nx.find_cliques(G)
```

Les cliques de G : exercice

Sept employés (A, B, C, D, E, F et G) se sont rendus au bureau aujourd'hui. Le tableau suivant précise "qui a rencontré qui" :

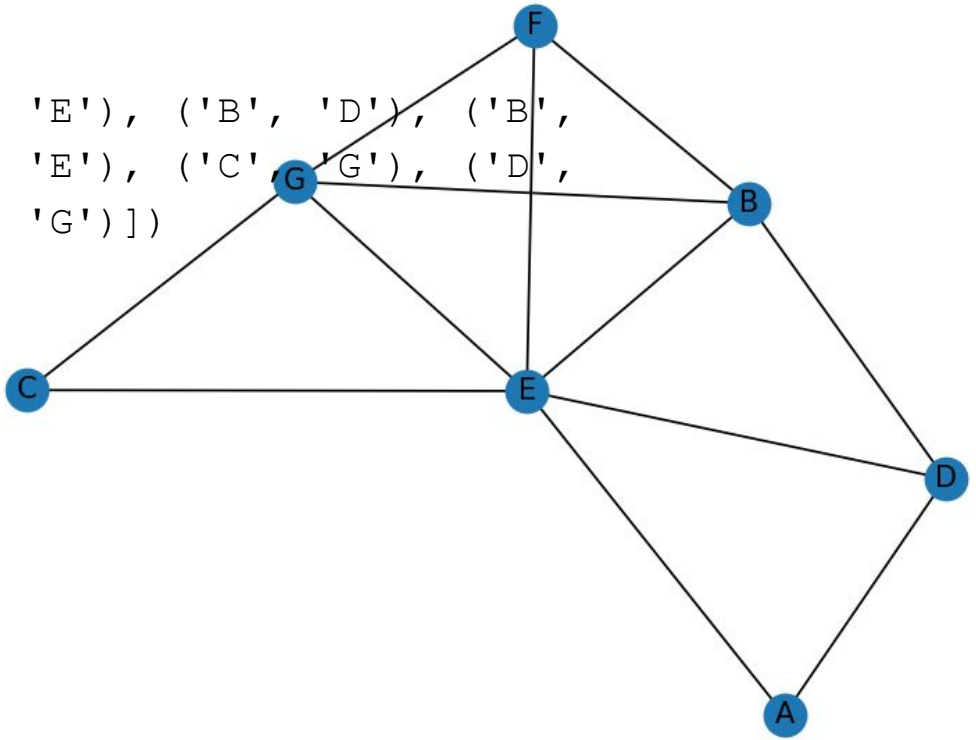
Employé	A	B	C	D	E	F	G
a rencontré	D,E	D,E,F,G	E,G	A,B,E	A,B,C,D,F,G	B,E,G	B,C,E,F

- De combien de places assises dispose au minimum le bureau, sachant que chacun a pu travailler assis pendant sa période de présence ?

- Qui était sur ces chaises au moment où le bureau était le plus occupé ?

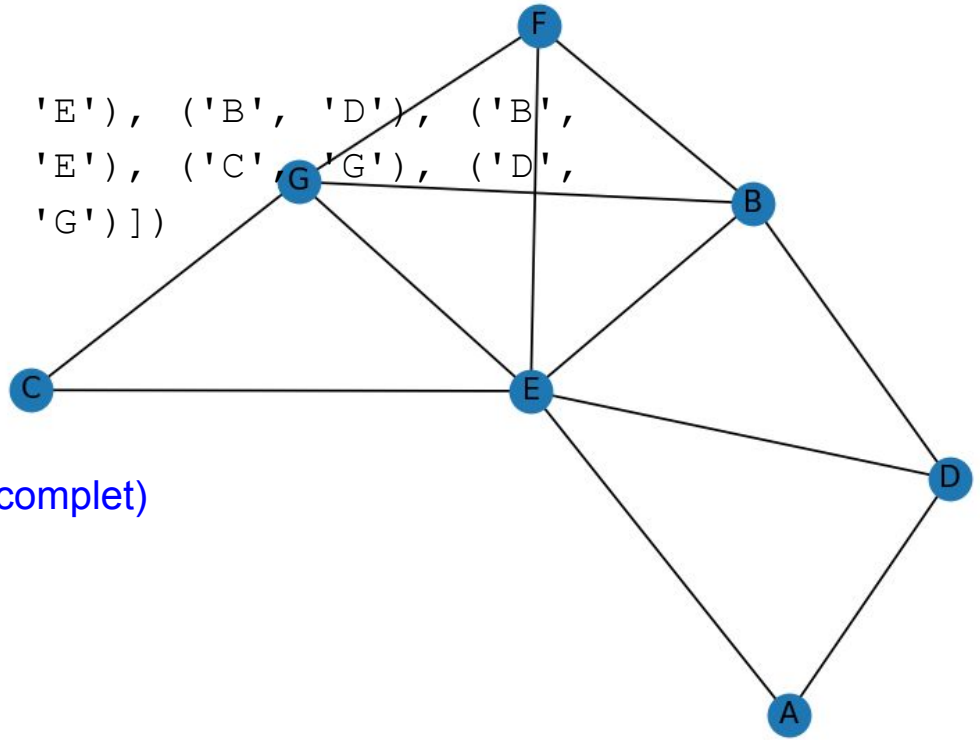
Les cliques de G : exercice

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

G = nx.Graph()
G.add_edges_from([('A', 'D'), ('A', 'E'), ('B', 'D'), ('B',
'E'), ('B', 'F'), ('B', 'G'), ('C', 'E'), ('C', 'G'), ('D',
'E'), ('E', 'F'), ('E', 'G'), ('F', 'G')])
nx.draw(G, with_labels=True)
plt.show()
```



Les cliques de G : exercice

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

G = nx.Graph()
G.add_edges_from([('A', 'D'), ('A', 'E'), ('B', 'D'), ('B',
'E'), ('B', 'F'), ('B', 'G'), ('C', 'E'), ('C', 'G'), ('D',
'E'), ('E', 'F'), ('E', 'G'), ('F', 'G')])
nx.draw(G, with_labels=True)
plt.show()
```



=> recherche de la plus grande clique (pb. NP-complet)

Les cliques de G : exercice

```
>>> from itertools import combinations
>>> sorted(combinations(G.nodes, 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('A', 'E'), ('A', 'F'),
('A', 'G'), ('B', 'C'), ('B', 'F'), ('B', 'G'), ('D', 'B'),
('D', 'C'), ('D', 'E'), ('D', 'F'), ('D', 'G'), ('E', 'B'),
('E', 'C'), ('E', 'F'), ('E', 'G'), ('F', 'C'), ('F', 'G'),
('G', 'C')]
>>> len(sorted(sum([list(combinations(G.nodes, k))
                    for k in range(2, G.number_of_nodes())],
                   [])))
119
```

Les cliques de G : exercice

```
>>> for nodes in sum([list(combinations(G.nodes, k))
                      for k in range(2, G.number_of_nodes())],
                     []):
...     H = induced_subgraph(G, nodes)
...     if is_clique(H, G):
...         print(len(nodes), nodes)
...
2 ('A', 'D')
2 ('A', 'E')
:
4 ('E', 'B', 'F', 'G')
```

Application des cliques

Elles servent notamment à la détection de communautés dans les réseaux sociaux :

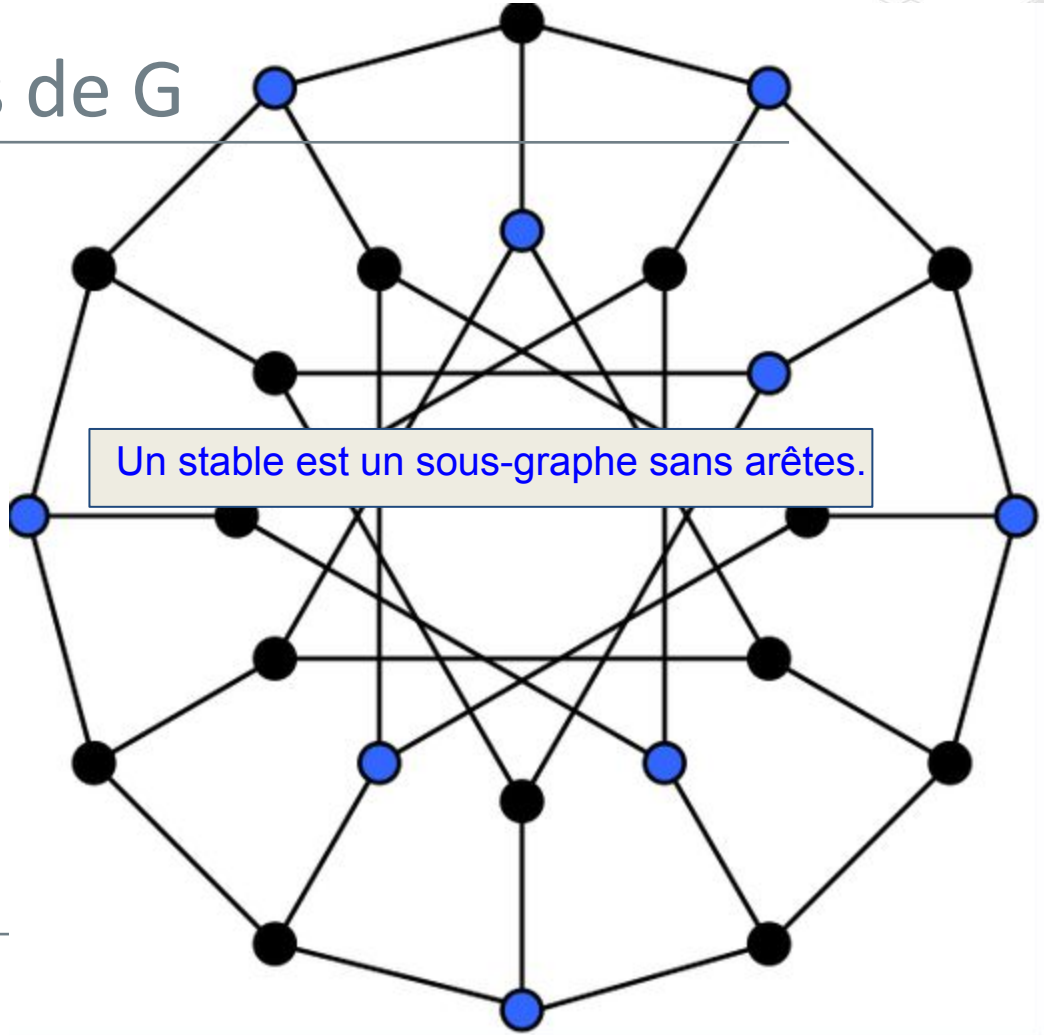
- Dans un réseau social, les individus sont représentés par des nœuds, et les relations (amitiés, collaborations, etc.) entre les individus sont représentées par des arêtes.

- Dans ce graphe, tous les membres d'une clique sont connectés les uns aux autres, formant ainsi une communauté resserrée.

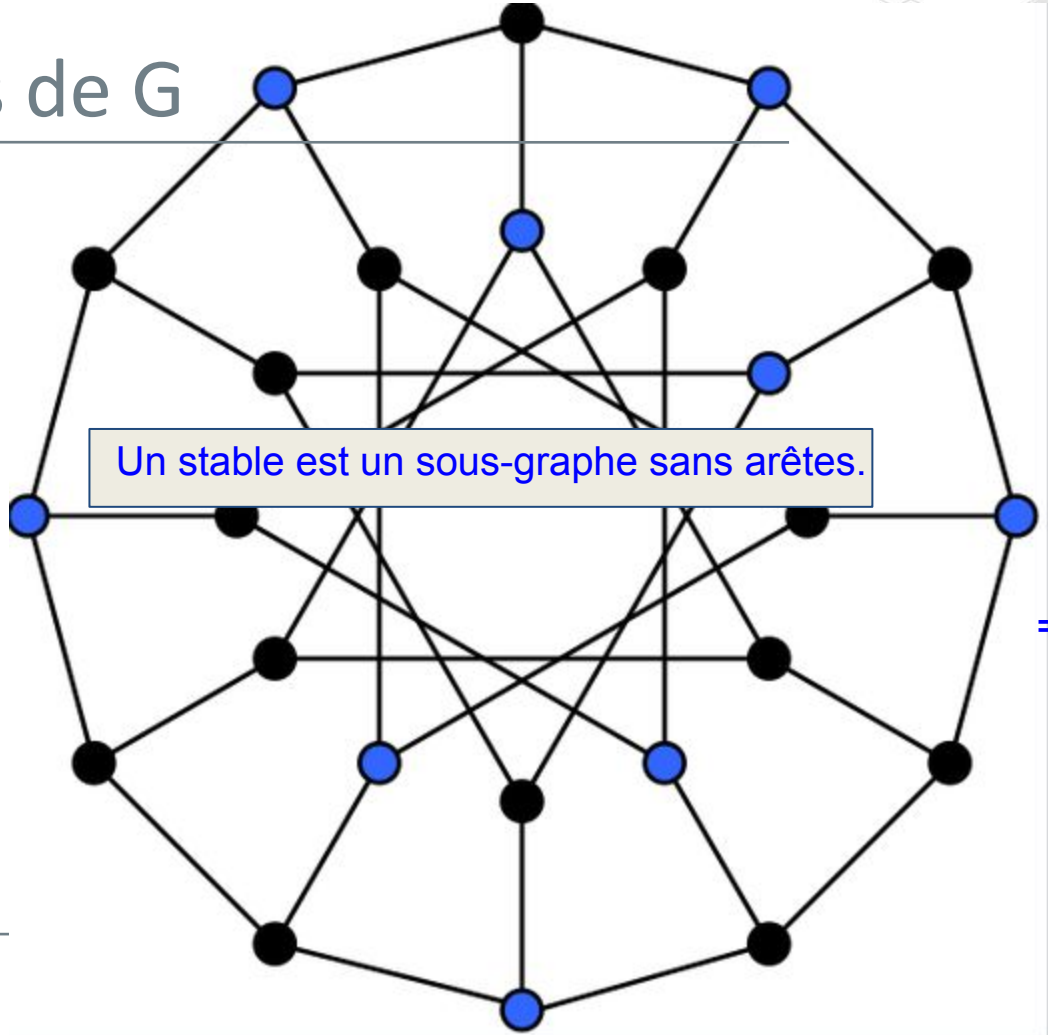
- La détection de cliques dans un réseau social peut révéler des groupes d'individus ayant des relations étroites, des centres d'intérêt communs ou des caractéristiques similaires.

- On peut par exemple améliorer les recommandations d'amis ou de contenu en se basant sur les membres d'une clique existante.

Les stables de G



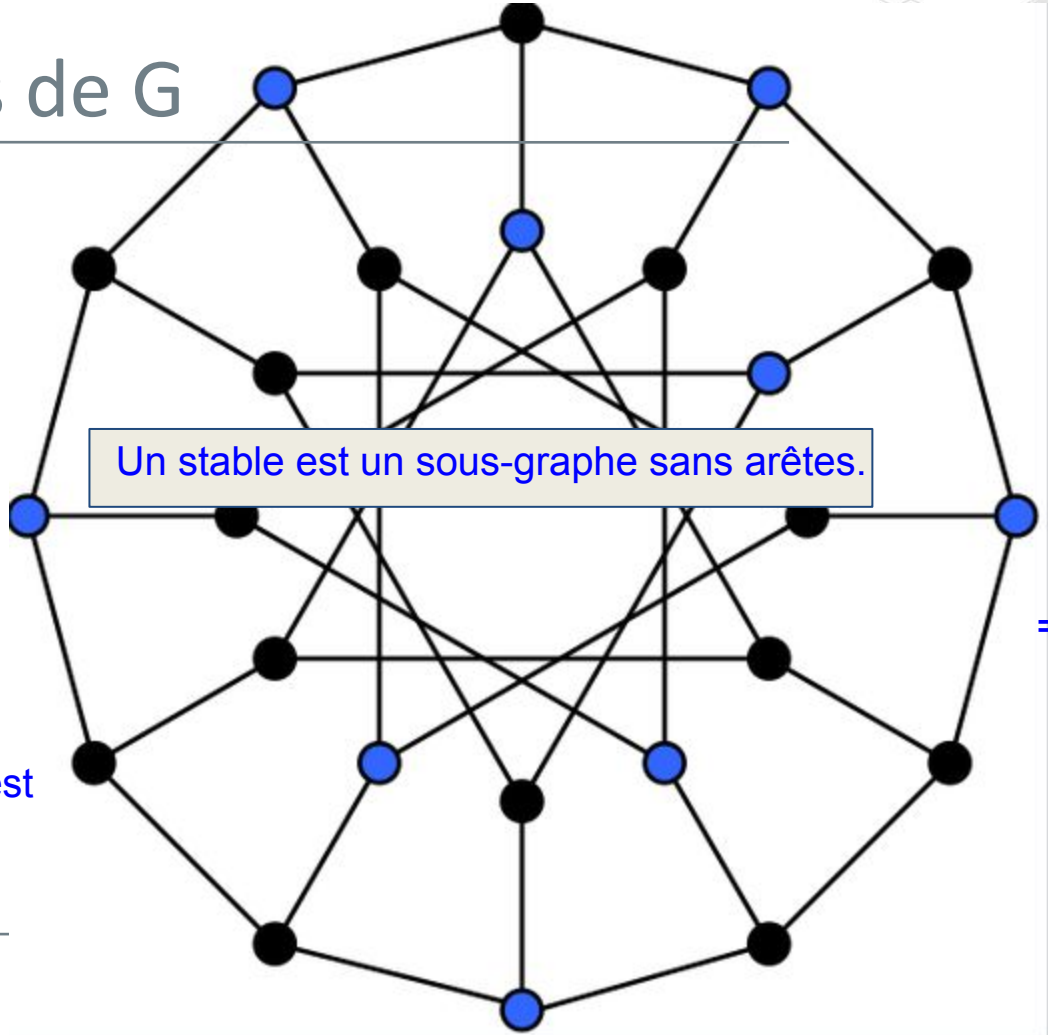
Les stables de G



Un stable est un sous-graphe sans arêtes.

=> C'est une clique dans le graphe complémentaire

Les stables de G



Un stable est un sous-graphe sans arêtes.

(Faire une fonction testant si un sous-graphe est un stable.)

=> C'est une clique dans le graphe complémentaire

Les stables de G

Un stable est un sous-graphe sans arêtes.

```
def is_stable(H, G):
    return is_subgraph(H, G) and H.number_of_edges()==0
```

Calcul d'un stable maximal avec networkx :

```
>>> nx.maximal_independent_set(G)
```

Les stables de G : application

- Supposons qu'on veuille organiser un nombre maximal de matchs en parallèle, dans un tournoi sportif.

- Les sommets sont les matchs, et une arête entre deux matchs signifie que ces derniers sont incompatibles.

- Trouver une solution revient à chercher un stable maximal.

=> problème difficile

Transversal

- Un sous-ensemble T de sommets est un transversal de G si toute arête de G a au moins une extrémité dans T.
  => Un transversal “couvre” donc toutes les arêtes

- Si T est un transversal de G, alors l’ensemble des autres sommets est un stable.

Transversal : application

- Supposons qu'on veuille surveiller les couloirs d'un bâtiment par des caméras pivotantes.

- Une caméra à une intersection permet de surveiller tous les couloirs adjacents.
  - Les sommets sont les intersections
  - Les arêtes sont les couloirs.

- L'ensemble minimal de caméras à installer est un transversal minimal.

Graphes planaires

Notion de graphe planaire

Un graphe est **planaire** si on peut le dessiner dans le plan sans croisement d'arcs.

- tous les graphes à moins de 5 sommets sont planaires
- tous les graphes bipartis à moins de 6 sommets sont planaires.
- K_5 est le plus petit graphe complet non biparti qui ne soit pas planaire.
- $K_{3,3}$ est le plus petit graphe biparti complet non planaire.

Formule d'Euler pour les graphes planaires :

Si un graphe simple connexe avec au moins 3 arêtes est planaire, alors son nombre d'arêtes M vérifie : $M \leq 3N-6$, où N est le nombre de sommets.

Graphe planaire et formule d'Euler : exercice

A partir de la formule d'Euler, faire une fonction qui renvoie :

- non, ce graphe ne peut pas être planaire
- il peut l'être, éventuellement.

Ajouter les autres conditions évoquées ci-dessus.

Graphe planaire et formule d'Euler : solution

```
def is_possibly_planar(graph):
    N = graph.number_of_nodes()
    M = graph.number_of_edges()

    # tous les graphes à moins de 5 sommets sont planaires
    if N < 5:
            return "il est planaire"

    # tous les graphes bipartis à moins de 6 sommets sont planaires
    if nx.is_bipartite(graph) and N < 6:
        return "il est planaire"
…
```

Graphe planaire et formule d'Euler : solution

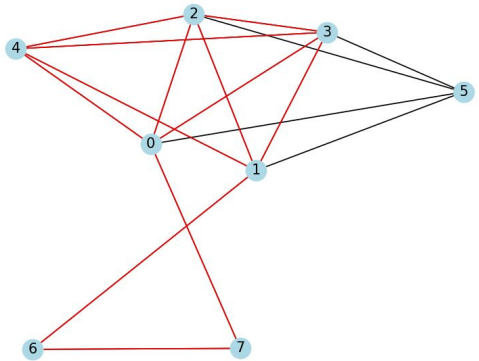
```
    # K5 est le plus petit graphe complet non biparti qui ne soit pas
planaire, K3,3 est le plus petit graphe biparti complet non planaire
    if nx.is_complete(graph):
        if nx.is_bipartite(graph) and N == 6:
            return "non, il n'est pas planaire"
        elif not nx.is_bipartite(graph) and N == 5:
            return "non, il n'est pas planaire"

    # Formule d'Euler pour les graphes planaires
    if M <= 3 * N - 6:
        return "il peut éventuellement l'être"
    else:
        return "non, il n'est pas planaire"
```

Théorème de Kuratowski

Un graphe est planaire si et seulement s'il ne contient aucun sous-graphe réductible à $K_{3,3}$ ou K_5.

G1 est **réductible** à G2 si on peut le rendre égal en utilisant les opérations :

- enlever des arêtes,
- renuméroter des sommets,
- contracter chaque sommet de degré 2 et ses deux arêtes incidentes en une arête unique.



Application des graphes planaires

Le schéma théorique d'un circuit électronique peut être représenté par un graphe : les nœuds sont les composants, et les arêtes les connexions entre ces derniers.

- si le graphe est planaire, le circuit est implantable sur une carte de circuit imprimé réalisable automatiquement;

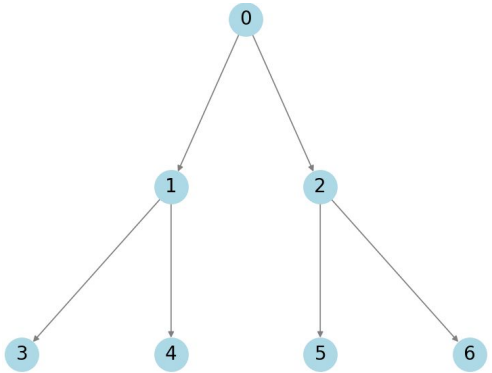
- sinon, il faut des soudures manuelles pour faire passer un fil au-dessus d'un autre.

Arbres et ete3

Arbre, arborescence

- Un **arbre** est un graphe connexe sans cycle.
  - C'est aussi un graphe connexe à N-1 arête.
  - C'est encore un graphe connexe, qui ne l'est plus si on enlève une arête.
    => il a juste ce qu'il faut pour être connexe.

- Une **arborescence** est un arbre orienté où tous les sommets sont descendants d'un sommet unique, appelé la **racine**.

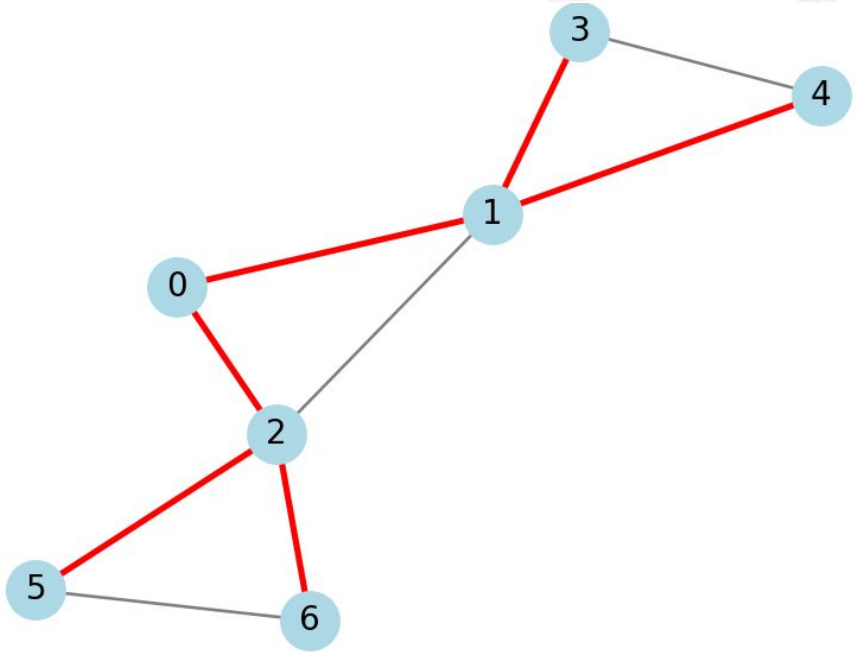


Arbre couvrant

- Un graphe partiel de G qui est un arbre est dit **arbre recouvrant** de G (spanning tree).

  `=> nx.minimum_spanning_tree(G)`

Il s'agit d'un arbre qui relie tous les sommets de G, en utilisant uniquement des arêtes de G.



Osmnx

Divers

Fusionner deux graphes

```
nx.compose(F,H)
```

Coloration de graphes

Nombre chromatique

Déterminer le nombre chromatique d'un graphe est :

- dans la classe P pour 2 couleurs : il suffit de rechercher si un cycle impair existe,
- pour plus de deux couleurs : NP complet.

Cliques et stables : exercice

Dans un groupe de six personnes, il y en :

- soit au moins trois qui se connaissent mutuellement,
- soit au moins trois qui ne se connaissent pas.

Cliques et stables : exercice

Dans un groupe de six personnes, il y en :

- soit au moins trois qui se connaissent mutuellement,
- soit au moins trois qui ne se connaissent pas.

Cela revient à dire que, quel que soit le graphe G à six sommets, si on génère tous ses sous-graphes de 3 sommets, alors il en existe forcément au moins un qui est :

- soit une clique (les trois sommets sont adjacents deux à deux),
- soit un stable (aucun arc).

=> Vérifiez cela