# TP 2 : Social Network


## General Remarks

- This lab is an opportunity to discover networkx Python package to represent graphs, to calculate centrality metrics and then apply them on a case study.

TUTORIAL : https://www.youtube.com/watch?v=PouhDHfssYA (https://www.youtube.com/watch?v=PouhDHfssYA)

### 1 : Graph Representation in Python

NetworkX is the most popular Python package for manipulating and analyzing graphs. Let's try to create the following simple graph:

Entrée [1]:

```
pip install decorator==5.0.9
```

```
Requirement already satisfied: decorator==5.0.9 in c:\users\sarah\anaconda3
\lib\site-packages (5.0.9)
Note: you may need to restart the kernel to use updated packages.
```

Entrée [2]:

```
#imports
import networkx as nx
```

Entrée [3]:

```
#create an empty graph
graph = nx.Graph()

#create a single vertex "Mike" vertex=node
graph.add_node("Mike")

#add a bunch of vertices : “Amine”, “Rémi”, “Nick”
graph.add_node("Amine")
graph.add_node("Rémi")
graph.add_node("Nick")

#add an edge between “Mike” and “Amine”
graph.add_edge("Mike", "Amine")

#add an edge between “Amine” and “Rémi”
graph.add_edge("Amine", "Rémi")

#add an edge between “Mike” and “Christophe”
graph.add_edge("Mike", "Christophe")
```

Entrée [4]:

```
#display nodes list
list(graph.nodes)
```

Out[4]:

```
['Mike', 'Amine', 'Rémi', 'Nick', 'Christophe']
```

Entrée [5]:

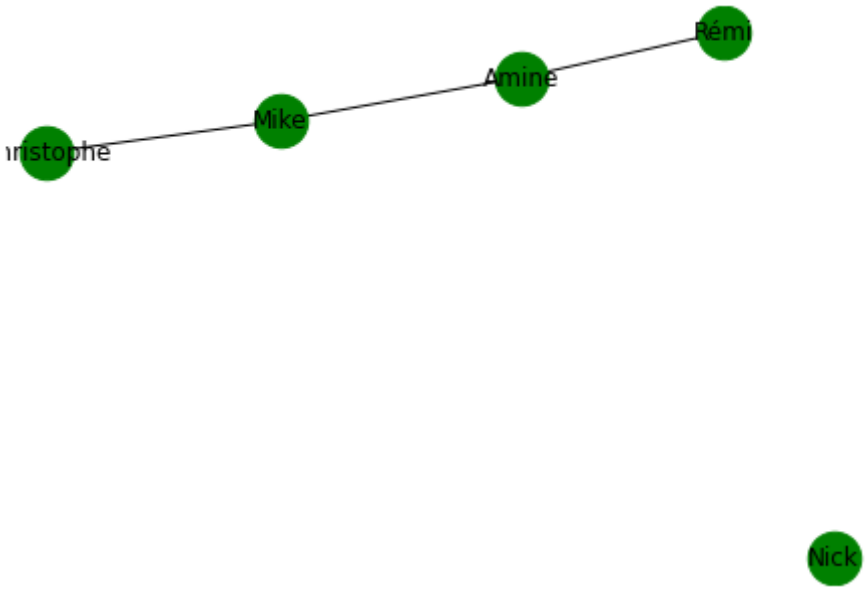
```
#display edges (relations) list
list(graph.edges)
```

Out[5]:

```
[('Mike', 'Amine'), ('Mike', 'Christophe'), ('Amine', 'Rémi')]
```

Entrée [6]:

```
#show graph
nx.draw(graph, node_size=700, with_labels=True, node_color='green')
```



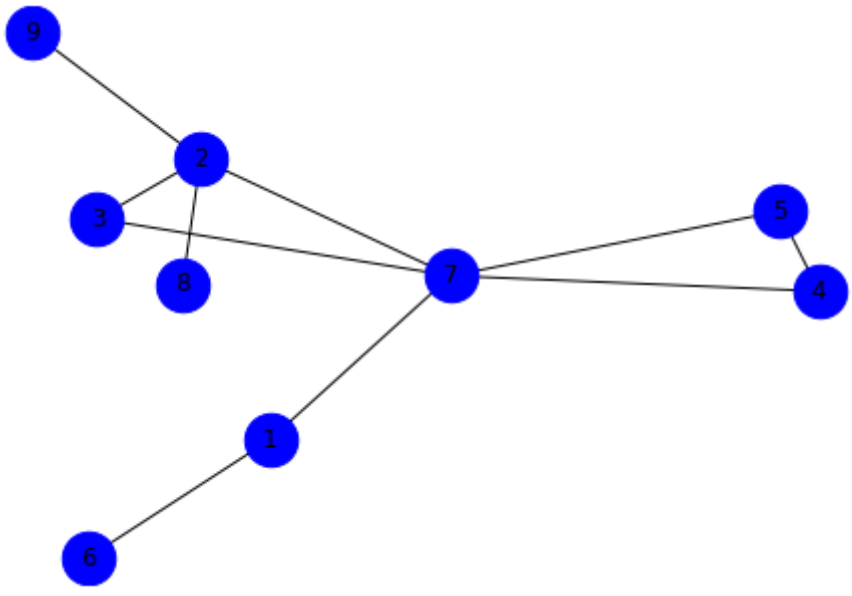

## 2 : Calculating Centrality Metrics w/ Python

Let's consider a network with 10 vertices and edges distributed as following: [(7,2), (2,3), (7,4), (4,5), (7,3), (7,5), (1,6),(1,7),(2,8),(2,9)]

Entrée [7]:

```
#create the associated graph by using networkx package
graph1 = nx.Graph()

#create nodes
graph1.add_node(1)
graph1.add_node(2)
graph1.add_node(3)
graph1.add_node(4)
graph1.add_node(5)
graph1.add_node(6)
graph1.add_node(7)
graph1.add_node(8)
graph1.add_node(9)

#display nodes list
print(list(graph1.nodes))

#create edges
graph1.add_edge(7,2)
graph1.add_edge(2,3)
graph1.add_edge(7,4)
graph1.add_edge(4,5)
graph1.add_edge(7,3)
graph1.add_edge(7,5)
graph1.add_edge(1,6)
graph1.add_edge(1,7)
graph1.add_edge(2,8)
graph1.add_edge(2,9)

#display edges (relations) list
print(list(graph1.edges))

#show graph
nx.draw(graph1, node_size=700, with_labels=True, node_color='blue')
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[(1, 6), (1, 7), (2, 7), (2, 3), (2, 8), (2, 9), (3, 7), (4, 7), (4, 5), (5,
7)]
```



Entrée [8]:

```
#calculate its degree centrality
print("Degree centrality (graph1) : "+str(nx.degree_centrality(graph1)))

#calculate its betweenness centrality
print("Betweenness centrality (graph1) : "+str(nx.betweenness_centrality(graph1)))

#calculate its closeness centrality
print("Closeness centrality (graph1) : "+str(nx.closeness_centrality(graph1)))
```

```
Degree centrality (graph1) : {1: 0.25, 2: 0.5, 3: 0.25, 4: 0.25, 5: 0.25, 6:
0.125, 7: 0.625, 8: 0.125, 9: 0.125}
Betweenness centrality (graph1) : {1: 0.25, 2: 0.46428571428571425, 3: 0.0,
4: 0.0, 5: 0.0, 6: 0.0, 7: 0.7142857142857142, 8: 0.0, 9: 0.0}
Closeness centrality (graph1) : {1: 0.5, 2: 0.6153846153846154, 3: 0.5333333
333333333, 4: 0.47058823529411764, 5: 0.47058823529411764, 6: 0.347826086956
52173, 7: 0.7272727272727273, 8: 0.4, 9: 0.4}
```

Comment your result :

- The degree centrality for a node v is the fraction of nodes it is connected to.
- Compute the shortest-path betweenness centrality for nodes. To calculate betweenness centrality, you take every pair of the network and count how many times a node can interrupt the shortest paths (geodesic distance) between the two nodes of the pair.
- To obtain information, one should be near from everyone. In this sense, the node in the nearest position on average can most efficiently obtain information. Thus, closeness centrality is the most appropriate.

## 3 : Fraud Analytics

Let's look at how we can use SNA to detect fraud.

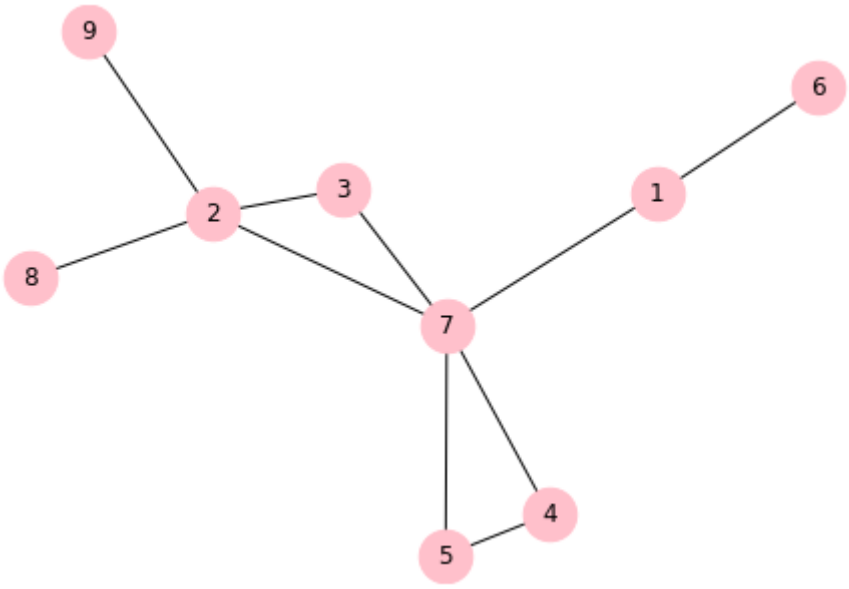
The word homophily has been coined to represent the effect human social network has on a person. Extending this concept, a homophilic network is a group of people who are likely to be associated with each other due to some common factor; for example, having the same origin or hobbies, being part of the same gang or the same university, or some combination of other factors.

If we want to analyze fraud in a homophilic network, we can take advantage of the relationships between the person under investigation and other people in the network, whose risk of involvement in fraud has already been carefully calculated. Flagging a person due to their company is sometimes also called guilt by association.

In an effort to understand the process, let's first look at a simple case.

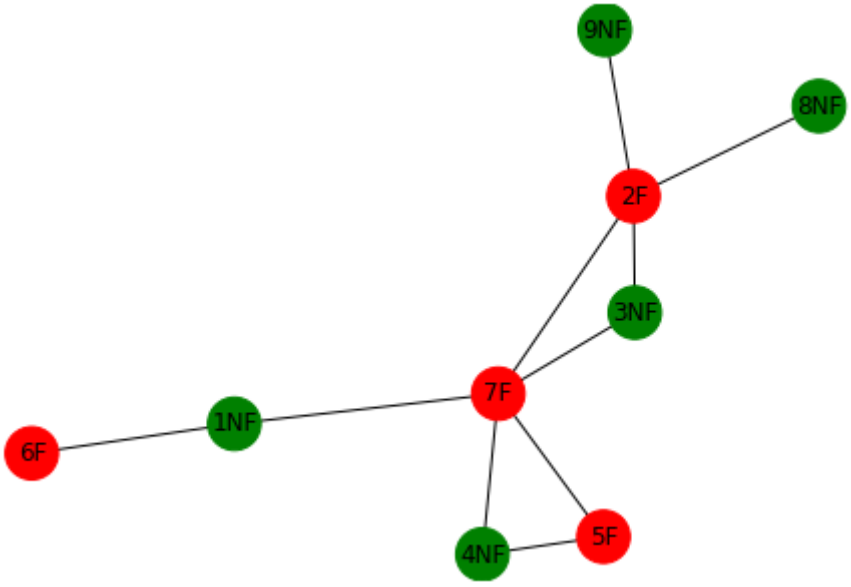
Entrée [9]:

```
#create a network with nine vertices and eight edges w/ fraud and non-fraud flags
graph2 = nx.Graph()
graph2.add_node(1, role='NF')
graph2.add_node(2, role='F')
graph2.add_node(3, role='NF')
graph2.add_node(4, role='NF')
graph2.add_node(5, role='F')
graph2.add_node(6, role='F')
graph2.add_node(7, role='F')
graph2.add_node(8, role='NF')
graph2.add_node(9, role='NF')

graph2.add_edge(7,2)
graph2.add_edge(2,3)
graph2.add_edge(7,4)
graph2.add_edge(4,5)
graph2.add_edge(7,3)
graph2.add_edge(7,5)
graph2.add_edge(1,6)
graph2.add_edge(1,7)
graph2.add_edge(2,8)
graph2.add_edge(2,9)

color = ['red', 'green']

nx.draw(graph2, node_size=700, with_labels=True, node_color='pink')
```

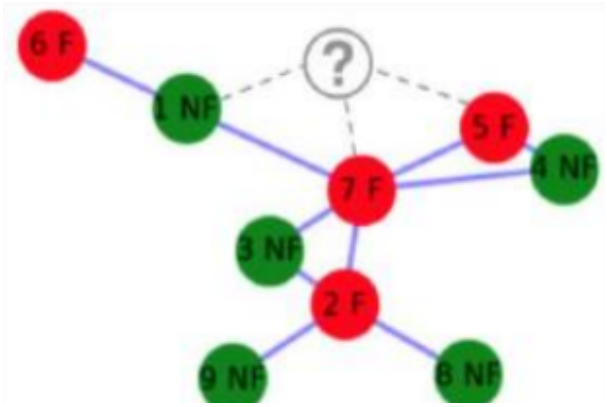


Entrée [10]:

```
color = []
liste = list(graph2.nodes) #numéro des nodes
index_liste = [0,1,2,3,4,5,6,7,8]
role = nx.get_node_attributes(graph2, 'role') #fraude ou pas fraude (l'indice commence à 1)
label = {} #combinaison des 2 à intégrer dans le graphe

for i in liste:
    if graph2.nodes[i] == {'role': 'NF'}:
        color.append('green')
    else:
        color.append('red')

for i in range(9):
    label[i] = (str(liste[i]) + role[i+1])

#on a un décalage de clé, on incrémente donc les clés du dict "label" de 1
label = {k+1 if k >= 0 else k:v for k,v in label.items()}

nx.draw(graph2, node_size=700, with_labels=True, node_color=color, labels=label)
```




Let's assume that we add another vertex, named q, to the network, as shown in the following figure. We have no prior information about this person and whether this person is involved in fraud or not. We want to classify this person as NF or F based on their links to the existing members of the social network:




Two ways to do it:

- Using a simple method that does not use centrality metrics and additional information about the type of fraud
- Using a watchtower methodology, which is an advanced technique that uses the centrality metrics of the existing nodes, as well as additional information about the type of fraud

## Simple Fraud Analytics

The simple technique of fraud analytics is based on the assumption that in a network, the behaviour of a person is affected by the people they are connected to. In a network, two vertices are more likely to have similar behaviour if they are associated with each other. Based on this assumption, we devise a simple technique. If we want to find the probability that a certain node, "a", belongs to F, the probability is represented by P(F/q) and is calculated as follows:

### 1 : Calculate the likelihood that node q being involved in fraud.

$P(F|q) = 1/degree(q) * \Sigma w(n, nj)$

- with $w(n, nj)$ = the weight of the connection between n and nj

The node would be in F if $P(F/q) > T$ (T is a given threshold).

Entrée [11]:

```
#on donne juste la liste de voisins et le seuil fixé T
def likelihood(neighbors, T):

    #calcul des poids
    #le poids = 1 si la relation est vers F et à 0 si vers NF
    weights = []
    for i in neighbors:
        if graph2.nodes[i] == {'role': 'F'}:
            weights.append(1)
        else:
            weights.append(0)

    #calcul de la probabilité
    P = sum(weights)/len(neighbors)

    #choix et affichage
    if P >= T:
        print ("The likelihood is :", P ,"which means that q is involved in fraud.")
    else:
        print ("The likelihood is :", P ,"which means that q is not involved in fraud.")

    #on réinitialise les poids
    weights = []
```

Entrée [12]:

```
#on observe que le "?" a 3 relations directes dont 3 voisins qui sont : 1, 7 et 5
#on fixe le seuil à 0.5 car on estime qu'une probabilité supérieure à 50% est suffisante po

likelihood([1,7,5], 0.5)
```

```
The likelihood is : 0.6666666666666666 which means that q is involved in fra
ud.
```

# The Watchtower Fraud Analytics Methodology

The previous, simple fraud analytics technique has the following two limitations:

- It does not evaluate the importance of each vertex in the social network. A connection to a hub that is involved in fraud may have different implications than a relationship with a remote, isolated person.
- When labeling someone as a known case of fraud in an existing network, we do not consider the severity of the crime. The watchtower fraud analytics methodology addresses these two limitations employing the following concepts:

## 1 : Scoring negative outcomes.

If a person is known to be involved in fraud, we say that there is a negative outcome associated with this individual. The scores are based on an analysis of fraud cases and their impact from historical data. From a score of 1 to 10, some negative outcomes can be rated as follows:

| Negative outcome | Negative outcome score |
|---|---|
| Impersonation | 10 |
| Involvement in credit card theft | 8 |
| Fake check submission | 7 |
| Criminal record | 6 |
| No record | 0 |

## 2 : Degree of Suspicion (DOS)

The degree of suspicion (DOS) quantifies our level of suspicion that a person may be involved in fraud. A DOS value of 0 means that this is a low-risk person and a DOS value of 9 means that this is a high-risk person. Analysis of historical data shows that professional fraudsters have important positions in their social networks. To incorporate this:

- Calculate all of the four centrality metrics of each vertex in our network: degree of centrality, betweenness, closeness and eigenvector.
- Take the average of these vertices.
- If a person associated with a vertex is involved in fraud, illustrate this negative outcome by scoring the person using the pre-determined values shown in the preceding table.
- Multiply the average of the centrality metrics and the negative outcome score to get the value of the DOS.
- Finally, normalize the DOS by dividing it by the maximum value of the DOS in the network.

**Calculate for each vertex, in a given network, its normalized DOS according to the process described above.**

Entrée [13]:

```
def average_centrality_metrics():
    moy = {}
    for i in list(graph2.nodes):
        moy[i] = (((nx.degree_centrality(graph2))[i] + (nx.betweenness_centrality(graph2))[
    return moy
```

Entrée [14]:

```
average_centrality_metrics()
```

Out[14]:

```
{1: 0.3092973867583357,
 2: 0.5063384184950667,
 3: 0.2869334781183585,
 4: 0.2602569741151565,
 5: 0.2602569741151565,
 6: 0.13901924248080227,
 7: 0.6647705525324459,
 8: 0.1703577895739195,
 9: 0.1703577895739195}
```

Entrée [15]:

```
import random

#on attribue un score au hasard parmi les choix disponibles pour ceux ayant fraudé, pour le

def negative_score_nodes():
    neg_score = [6, 7, 8, 10] #scores disponibles
    neg_score_nodes = {} #scores des noeuds

    for i in list(graph2.nodes):
        if graph2.nodes[i] == {'role': 'F'}:
            neg_score_nodes[i] = random.choice(neg_score)
        else:
            neg_score_nodes[i] = 0

    return neg_score_nodes
```

Entrée [16]:

```
negative_score_nodes()
```

Out[16]:

```
{1: 0, 2: 8, 3: 0, 4: 0, 5: 8, 6: 8, 7: 7, 8: 0, 9: 0}
```

Entrée [17]:

```
def compute_DOS():
    dos_nodes = {}
    normalized_dos = {}
    for i in list(graph2.nodes):
        dos_nodes[i] = average_centrality_metrics()[i]*negative_score_nodes()[i]

    for i in list(graph2.nodes):
        normalized_dos[i] = dos_nodes[i]/max(dos_nodes.values())

    #print('The DOS nodes (not normalized) are :\n',dos_nodes)
    #print('The maximum DOS (not normalized) is :\n', max(dos_nodes.values()))

    return normalized_dos
```

Entrée [18]:

```
normalized_dos = compute_DOS()
normalized_dos
```

Out[18]:

```
{1: 0.0,
 2: 0.6664647140210972,
 3: 0.0,
 4: 0.0,
 5: 0.4893736890189153,
 6: 0.20912364717601878,
 7: 1.0,
 8: 0.0,
 9: 0.0}
```

## 2 : Degree of Suspicion (DOS)

In order to calculate the DOS of the new node that has been added, the following formula is used :

$$DOS_k = \frac{1}{degree_k} \sum_{n_j \in Neighborhood_n} w(n, n_j) DOS_{normalized_j}$$

This will indicate the risk of fraud associated with this new node added to the system. It is possible to create different risk bins for the DOS, as follows:

| Value of the DOS | Risk classification |
|---|---|
| DOS = 0 | No risk |
| 0<DOS<=0.10 | Low risk |
| 0.10<DOS<=0.3 | Medium risk |
| DOS>0.3 | High risk |

**Calculate the DOS of the new node q.**

Entrée [19]:

```
def dos_q(neighbors):
    weights = []
    dos_neighbors = []
    weights_dos = []

    #on ajoute les poids des voisins
    for i in neighbors:
        if graph2.nodes[i] == {'role': 'F'}:
            weights.append(1)
        else:
            weights.append(0)

    #on ajoute les DOS de ces mêmes voisins
        dos_neighbors.append(normalized_dos[i])

    #on veut multiplier les poids des voisins par leur dos respectif
    for i in [0, 1, 2]:
        weights_dos.append(weights[i]*dos_neighbors[i])

    #print(weights)
    #print(dos_neighbors)
    #print(weights_dos)

    dos_q = sum(weights_dos)/len(neighbors)

    print("DOS(q) = ",dos_q)

    if dos_q == 0:
            print("No risk.")
    elif 0 < dos_q <= 0.10:
            print("Low risk.")
    elif 0.10 < dos_q <= 0.3:
            print("Medium risk.")
    else:
            print("High risk.")
```

Entrée [20]:

```
dos_q([1,7,5])
```

```
DOS(q) =  0.49645789633963844
High risk.
```

Entrée [ ]:

```
#sur le anaconda prompt, taper :
#jupyter-nbconvert --to PDFviaHTML TP2_SocialNetwork_MADAD_NOUAR.ipynb
```