Entrée [ ]:

(https://www.aims-senegal.org/)

AFRICAN INSTITUTE FOR MATHEMATICAL SCIENCES (AIMS)

Title of the course : Complex Networks

Lecturers by : Franck Kalala MUTUMBO, from Aims Senegal

Individuel

Mikhaël Presley KIBINDA-MOUKENGUE

ASSIGNMENT n°1 Review block n°4

Entrée [899]:
```
import networkx as nx
import numpy as np
import collections
import matplotlib.pyplot as plt
dir(nx)
```

Out[899]:
```
['AmbiguousSolution',
 'DiGraph',
 'ExceededMaxIterations',
 'Graph',
 'GraphMLReader',
 'GraphMLWriter',
 'HasACycle',
 'LCF_graph',
 'LFR_benchmark_graph',
 'MultiDiGraph',
 'MultiGraph',
 'NetworkXAlgorithmError',
 'NetworkXError',
 'NetworkXException',
 'NetworkXNoCycle',
 'NetworkXNoPath',
 'NetworkXNotImplemented',
 'NetworkXPointlessConcept',
 'NetworkXTreewidthBoundExceeded',
```

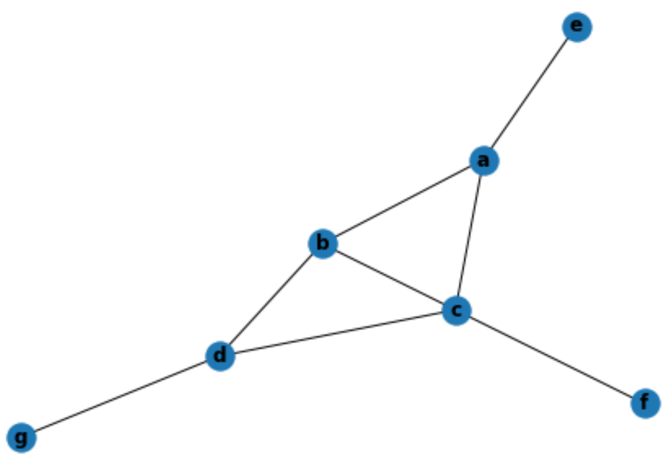
Entrée [900]:
```
nx.__version__
```

Out[900]:
```
'2.4'
```

For the non oriented graph

Entrée [901]:
```
G_1=nx.Graph()
G_1.add_nodes_from(["a", "b", "c", "d", "e", "f", "g"])
G_1.add_edges_from([("a","b"),("a","c"),("a","e"),("b","c"),("b","d"),("c","d"
```

Entrée [933]:
```
nx.draw(G_1, with_labels=True, font_weight='bold')
```



Entrée [ ]:

Distance matrix

This matrix also called the all-pairs shortest path matrix calculates the short distance between the nodes i to a node j.

Entrée [903]:
```
D1=list(nx.shortest_path_length(G_1))
D1
m=list()
for i in range(len(D1)):
    k=sorted(D1[i][1].items())
#     print(k)
    for l in range(len(k)):
        b=k[l][1]
#         print(b)
        m.append(b)
print(m)
D=np.array(m)
D=D.reshape(len(D1),len(D1))
D=np.array(m)
D=D.reshape(len(D1),len(D1))
D
```

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

Out[903]:
```
array([[0, 1, 1, 2, 1, 2, 3],
       [1, 0, 1, 1, 2, 2, 2],
       [1, 1, 0, 1, 2, 1, 2],
       [2, 1, 1, 0, 3, 2, 1],
       [1, 2, 2, 3, 0, 3, 4],
       [2, 2, 1, 2, 3, 0, 3],
       [3, 2, 2, 1, 4, 3, 0]])
```

Entrée [ ]:

We can also use :

Entrée [904]:
```
(nx.floyd_warshall_numpy(G_1, nodelist=None, weight=None)).astype(int)
```

Out[904]:
```
matrix([[0, 1, 1, 2, 1, 2, 3],
        [1, 0, 1, 1, 2, 2, 2],
        [1, 1, 0, 1, 2, 1, 2],
        [2, 1, 1, 0, 3, 2, 1],
        [1, 2, 2, 3, 0, 3, 4],
        [2, 2, 1, 2, 3, 0, 3],
        [3, 2, 2, 1, 4, 3, 0]])
```

It is the matrix of the smallest path or shortest distance from a point i to a point j in the network.

Entrée [ ]:

Eccentricity

$e(u) = \max_{v \in V(G)} d(u, v)$, means the maximum distance from u to any other vertex of the Network. Here for example, the maximale distance which separates a and others nodes is 3, and this node of the network is g, and so one for nodes b, c...

Entrée [905]:
```
nx.eccentricity(G_1, v=None, sp=None)
```

Out[905]:
```
{'a': 3, 'b': 2, 'c': 2, 'd': 3, 'e': 4, 'f': 3, 'g': 4}
```

The eccentricity of a vertex is the maximum distance between this vertex and the other vertices of the graph. Let's take the example of a social network, we can say that a is friends with b, c and e, b is friends with a, c and d, c is friends with a, b, d and f, d is friends with b, c and g, e is friendly with a, f is friendly with c, f is friendly with c and g is friendly with d, the eccentricity of the a node is 3 ; the point is distant from all others at a maximum distance of 3. So the point a to be friendly with g must pass through three groups of friends in order to get there.

Entrée [ ]:

Radius

$r(G) = \min_{u,v \in V(G)} d(u, v)$, this is the opposite of eccentricity of the graph. It's the minimum distance between the pair of vertices. It's also defined as the minimum of eccentricity. So, we have clearly 2.

Entrée [906]:
```
nx.radius(G_1, e=None, usebounds=False)
```

Out[906]:
```
2
```

Contrary to eccentricity, the radius is the minimum of the eccentricity. In order for everyone in our social network to be friends, everyone has to be in at least two groups in order to chat with each other.

Entrée [ ]:

Center

$C(G) = \{u \in V(G), u\ is\ central\ node\}$, is defined as the set of nodes in which the eccentricity is equal to the radius. So we have only the nodes b and c.

Entrée [907]:
```
nx.center(G_1, e=None, usebounds=False)
```

Out[907]:
```
['b', 'c']
```

In the case of a social network, the centre is defined as the largest group of subscribers. It connects all the friends, it is a point that is linked in maximum with all the others.

Entrée [ ]:

Wiener_index

$W(G) = \sum_{i<j} d_{ij}$, Used otfen in chemical grapy theory, means the sum of the lengths of the shortest paths between all pairs of vertices.

Entrée [908]:
```
nx.wiener_index(G_1, weight=None)
```

Out[908]: 40.0

Often used in Chemistry, it is the sum of the shortest friendship links (in terms of layers, direct or non-direct friendship) between all pairs of subscribers.

Entrée [ ]:

Eulerian_path

Eulerian Path is a path in graph that visits every edge or link exactly once. Here, we can't do it.

Entrée [909]:
```
def has_eulerian_path(G):
    if G.is_directed():
        # Every node must have equal in degree and out degree and the
        # graph must be strongly connected
        return (all(G.in_degree(n) == G.out_degree(n) for n in G) and
                nx.is_strongly_connected(G))
    # An undirected Eulerian graph has no vertices of odd degree and
    # must be connected.
    return all(d % 2 == 0 for v, d in G.degree()) and nx.is_connected(G)
```

Entrée [910]:
```
has_eulerian_path(G_1)
```

Out[910]: False

We can also use :

Entrée [911]:
```
nx.is_eulerian(G_1)
```

Out[911]: False

Here it means that, is it possible for a subscriber to be friends with everyone else but by going to a group (safest place to make friends) once and only once? In our network case, this is impossible.

Entrée [ ]:

Hamiltonian_path

Hamiltonian path is a path visits every node in a graph exactly once. Here also, we can't find a hamiltonian path.

Entrée [912]:
```
def has_hamiltonian_path(G):
    F = [(G,[list(G.nodes())[0]])]
    n = G.number_of_nodes()
    while F:
        graph,path = F.pop()
        confs = []
        for node in graph.neighbors(path[-1]):
            conf_p = path[:]
            conf_p.append(node)
            conf_g = nx.Graph(graph)
            conf_g.remove_node(path[-1])
            confs.append((conf_g,conf_p))
        for g,p in confs:
            if len(p)==n:
                return p
            else:
                F.append((g,p))
    return 'No hamiltonian path'
```

Entrée [913]:
```
has_hamiltonian_path(G_1)
```

Out[913]:
```
'No hamiltonian path'
```

This means, is it possible that all groups of invidual can manage to unite and this through one and only one person per group? This is not possible in our graph case.

Entrée [ ]:

Local_Clustering

$C_i = \frac{2e_i}{k_i(k_i - 1)}$, local clustering of a node in a graph quantifies the proximity (in terms of probability) of its neighbors to a group often seen as a triangle.

Entrée [914]:
```
nx.clustering(G_1, nodes=None, weight=None)
```

Out[914]:
```
{'a': 0.3333333333333333,
 'b': 0.6666666666666666,
 'c': 0.3333333333333333,
 'd': 0.3333333333333333,
 'e': 0,
 'f': 0,
 'g': 0}
```

It is the probability that each node or subscriber belongs to a group, this is also explained by the different friendships shared with neighbors, for this purpose we identify triangular groups.

Entrée [ ]:

Global_Clustering

$C = \frac{3 \times number\ of\ triangles\ in\ the\ network}{number\ of\ connected\ triplets\ in\ the\ network}$, It can be seen as the relative number of transitive triples, it's based on triplets of nodes. Here it's defined as the average of the clusters nodes of the network G.

Entrée [915]:
```
nx.transitivity(G_1)
```

Out[915]:
```
0.4
```

Here it is just a matter of finding a general membership probability by transitivity, i.e. finding a general coefficient in terms of probability on the average membership of each subscriber. 0.4 means here that in general all individuals in our network have a weak membership link to the same groups simultaneously.

Entrée [ ]:
```

```

Degree_distribution

$P_k = \frac{N_k}{N}$, where N_k is the number of the nodes which have the degree k and N is the total number of the nodes in the network. The degree of a node in a network is the number of links it has with other nodes, and the distribution of degrees is the probability distribution of these degrees over all the nodes in the network.

Entrée [916]:
```
nx.degree(G_1)
```

Out[916]:
```
DegreeView({'a': 3, 'b': 3, 'c': 4, 'd': 3, 'e': 1, 'f': 1, 'g': 1})
```

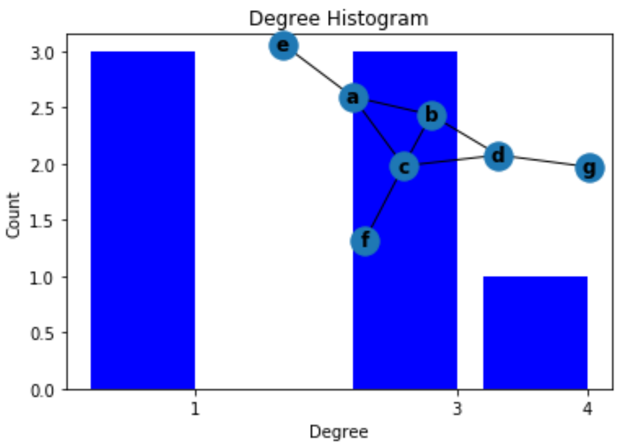
$P_1 = \frac{3}{7}$, $P_2 = 0$, $P_3 = \frac{3}{7}$, $P_4 = \frac{1}{7}$, $P_5 = 0$, $P_6 = 0$ and $P_7 = 0$.

Entrée [917]:
```
degree_sequence = sorted([d for n, d in G_1.degree()], reverse=True)  # degree
# print "Degree sequence", degree_sequence
degreeCount = collections.Counter(degree_sequence)
deg, cnt = zip(*degreeCount.items())

fig, ax = plt.subplots()
plt.bar(deg, cnt, width=0.80, color='b')

plt.title("Degree Histogram")
plt.ylabel("Count")
plt.xlabel("Degree")
ax.set_xticks([d + 0.4 for d in deg])
ax.set_xticklabels(deg)

# draw graph in inset
plt.axes([0.4, 0.4, 0.5, 0.5])
Gcc = G_1.subgraph(nx.draw(G_1, with_labels=True, font_weight='bold'))
pos = nx.spring_layout(G_1)
plt.axis('off')
plt.show()
```



The histogram tells us that three subscribers of the network namely e, f and g have exactly one friendship link each with the others, three others namely a, b and d have three friendship links with the others (degree 3) as well as only one c subscriber who has more than 4 friendship link from all the others (degree 4).

Entrée [ ]:

Average_path_length

$a = \sum_{u,v \in V(G)} \frac{d(u,v)}{N(N-1)}$, means the average number of steps along the shortest paths for all possible pairs of network nodes.

Entrée [918]:
```
nx.average_shortest_path_length(G_1)
```

Out[918]: 1.9047619047619047

This is the average of the closest possible friendships or relationships between subscribers, actually in our case, there are a large number of individuals who are very close possible.

Entrée [ ]:

Entrée [ ]:

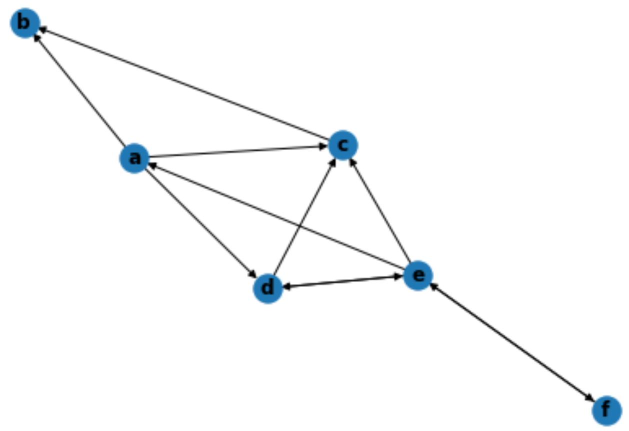
For the oriented graph

Entrée [919]:

```
G_2=nx.DiGraph()
pos=nx.spring_layout(G_2)
G_2.add_nodes_from(["a", "b", "c", "d", "e", "f"])
G_2.add_edges_from([("a","b"),("a","c"),("a","d"),("c","b"),("d","c"),("d","e"
```

Entrée [920]:

```
nx.draw(G_2, with_labels=True, font_weight='bold')
```



Entrée [ ]:

Distance matrix

This matrix also called the all-pairs shortest path matrix calculates the short distance between the nodes i to a node j.

Entrée [921]:

```
nx.floyd_warshall_numpy(G_2, nodelist=None, weight=None)
```

Out[921]:

```
matrix([[ 0.,  1.,  1.,  1.,  2.,  3.],
        [inf,  0., inf, inf, inf, inf],
        [inf,  1.,  0., inf, inf, inf],
        [ 2.,  2.,  1.,  0.,  1.,  2.],
        [ 1.,  2.,  1.,  1.,  0.,  1.],
        [ 2.,  3.,  2.,  2.,  1.,  0.]])
```

It is the matrix of the smallest path or shortest distance connecting a node i to a node j of the network, here infinity means that there is no link between the node i and j.

Entrée [ ]:

Eccentricity

This oriented graph is not strongly connected, we cannot calculate the eccentricity, by definition it is the maximum distance to leave from node b to all other nodes. In our case there is no path to leave from b to f for example.

Entrée [ ]:

Radius

Since radius is the opposite of eccentricity, since the graph is not strongly connected, then no minimum distance between pairs of nodes can be found.

Entrée [ ]:

Center

We can't find the most important node, because it means that the node where its eccentricity is equal to the raduis of the network. Above, there are no eccentricity and no radius.

Entrée [ ]:

Wiener_index

$W(G) = \sum_{i<j} d_{ij}$, Used otfen in chemical grapy theory, means the sum of the lengths of the shortest paths between all pairs of vertices.

Entrée [922]:

```
nx.wiener_index(G_2, weight=None)
```

Out[922]:

```
inf
```

Often used in Chemistry, it sums up the shortest paths between all pairs of nodes in a network, but here it turns out that there are nodes that are very close together but do not communicate with each other, so we will sum some values with infinity, which will of course give infinity. If this graph represents, for example, the organization chart of a country's treasury, we will see that, on average, even the shortest or closest services do not always communicate with each other.

Entrée [ ]:

Eulerian_path

Eulerian Path is a path in graph that visits every edge or link exactly once. Here, we can't do it.

Entrée [923]:
```
def has_eulerian_path(G):
    if G.is_directed():
        # Every node must have equal in degree and out degree and the
        # graph must be strongly connected
        return (all(G.in_degree(n) == G.out_degree(n) for n in G) and
                nx.is_strongly_connected(G))
    # An undirected Eulerian graph has no vertices of odd degree and
    # must be connected.
    return all(d % 2 == 0 for v, d in G.degree()) and nx.is_connected(G)
```

Entrée [924]:
```
has_eulerian_path(G_2)
```

Out[924]:
```
False
```

We can also use :

Entrée [925]:
```
nx.is_eulerian(G_2)
```

Out[925]:
```
False
```

Here the question we are asking ourselves is, is it possible to be able to pass a file for finance, using one and only one procedure? Here, in our case, the answer is NO. Because in our organization chart there is not always mutual cooperation between services.

Entrée [ ]:
```

```

Hamiltonian_path

Hamiltonian path is a path visits every node in a graph exactly once. Here also, we can't find a hamiltonian path.

Entrée [926]:
```
def has_hamilton_path(graph, start_u):
    size = len(graph)
  # if None we are -unvisiting- comming back and pop v
    to_visit = [None, start_u]
    path = []
    while(to_visit):
        u = to_visit.pop()
        if u :
            path.append(u)
            if len(path) == size:
                break
            for x in set(graph[u])-set(path):
                to_visit.append(None) # out
                to_visit.append(x) # in
        else: # if None we are comming back and pop v
            path.pop()
    return path
```

Entrée [927]:
```
has_hamilton_path(G_2,list(G_2.nodes())[5])
```

Out[927]:
```
['f', 'e', 'a', 'd', 'c', 'b']
```

Here the question we ask ourselves is, is it possible to pass a file through to finance, by passing through a department once and only once? Here, in our case, the answer is YES. Because in our organization chart a file can leave the f service by going through the e, a, d, c and b services once and only once.

Entrée [ ]:

Local_Clustering

$C_i = \frac{2e_i}{k_i(k_i - 1)}$, local clustering of a node in a graph quantifies the proximity (in terms of probability) of its neighbors to a group often seen as a triangle.

Entrée [928]:

```
nx.clustering(G_2, nodes=None, weight=None)
```

Out[928]:

```
{'a': 0.4166666666666667,
 'b': 0.5,
 'c': 0.4166666666666667,
 'd': 0.5,
 'e': 0.19230769230769232,
 'f': 0}
```

It is the proximity that exists between a service and all the others in terms of probability, we can see that in our case the a and c services are grouped together, we can say that in these services we do the same operations on the files, as well as the b and d services. As for the other services e and f, they make operations particular to themselves, they form their own clusters.

Entrée [ ]:

Global_Clustering

$C = \frac{3 \times number\ of\ triangles\ in\ the\ network}{number\ of\ connected\ triplets\ in\ the\ network}$, It can be seen as the relative number of transitive triples, it's based on triplets of nodes. Here it's defined as the average of the clusters nodes of the network G.

Entrée [929]:

```
nx.transitivity(G_2)
```

Out[929]:

```
0.3
```

It is the probability of communication between services in terms of transitivity, that is to say finding the exchanges in average of all the groups of services. The similarity existing between these different groups.

Entrée [ ]:

Degree_Distribution

$P_k = \frac{N_k}{N}$, where N_k is the number of the nodes which have the degree k and N is the total number of the nodes in the network. The degree of a node in a network is the number of links it has with other nodes, and the distribution of degrees is the probability distribution of these degrees over all the nodes in the network. Here $k = k^{in} + k^{out}$.

Entrée [930]:
```
nx.degree(G_2)
```

Out[930]:
```
DiDegreeView({'a': 4, 'b': 2, 'c': 4, 'd': 4, 'e': 6, 'f': 2})
```

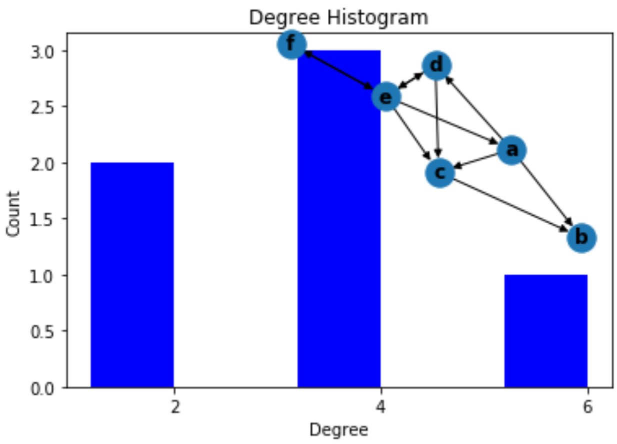
$P_1 = 0$, $P_2 = \frac{2}{6}$, $P_3 = 0$, $P_4 = \frac{3}{6}$, $P_5 = 0$ and $P_6 = \frac{1}{7}$.

Entrée [932]:
```
degree_sequence = sorted([d for n, d in G_2.degree()], reverse=True)  # degree
# print "Degree sequence", degree_sequence
degreeCount = collections.Counter(degree_sequence)
deg, cnt = zip(*degreeCount.items())

fig, ax = plt.subplots()
plt.bar(deg, cnt, width=0.80, color='b')

plt.title("Degree Histogram")
plt.ylabel("Count")
plt.xlabel("Degree")
ax.set_xticks([d + 0.4 for d in deg])
ax.set_xticklabels(deg)

# draw graph in inset
plt.axes([0.4, 0.4, 0.5, 0.5])
Gcc = G_2.subgraph(nx.draw(G_2, with_labels=True, font_weight='bold'))
pos = nx.spring_layout(G_2)
plt.axis('off')
plt.show()
```



This histogram justifies the frequency of communication of one service in relation to the others. The histogram tells us that two services, namely b and f, communicate with each other for the reception (incoming degree 2) and reception+transfer (incoming degree 1 and outgoing degree 1) of files, respectively, three services, namely a, c and d, have a degree of exchange with the others equal to 4 (incoming and outgoing degrees combined) and one e service has a degree of collaboration with the others equal to 6 (incoming and outgoing degrees combined).

Entrée [ ]:

Average_Path

$a = \sum_{u,v \in V(G)} \frac{d(u,v)}{N(N-1)}$, means the average number of steps along the shortest paths for all possible pairs of network nodes.

$a = \infty$, Since in this network the Wienner's index is infinite. So, We have an infinite number of average path.

Entrée [ ]:

Entrée [ ]: