DSTA

Chapter III - The Internet Network

This solution notebook is taken from the Python 2 notebook for Ch. 3 of Caldarelli-Cheesa's textbook (CC).

```
import networkx as nx

import matplotlib.pyplot as plt
```

Network from SVG with the best node positioning

```
from bs4 import BeautifulSoup
```

```
DATFILE = "./data/test_graph.dat"
```

```
def Graph_from_SVG(stream):

    G = nx.Graph()

    attrs = {
        "line" :  ["x1", "y1", "x2", "y2"]
    }

    op = open(stream, "r")

    xml = op.read()

    soup = BeautifulSoup(xml)

    count = 0
```

```
    node_dict = {}
    pos = {}

    for attr in attrs.keys():
        tmps = soup.findAll(attr)

        for t in tmps:
            node1 = (t['x1'], t['y1'])
            node2 = (t['x2'], t['y2'])

            if not node1 in node_dict:
                node_dict[node1] = str(count)
                pos[str(count)] = (float(node1[0]), float(node1[1]))
                count += 1

            if not node2 in node_dict:
                node_dict[node2] = str(count)
                pos[str(count)] = (float(node2[0]),float(node2[1]))
                count += 1

            G.add_edge(node_dict[node1], node_dict[node2])

    #save the graph in an edge list format
    nx.write_edgelist(G, DATFILE, data = False)

    return G, pos
```

Plotting the test Networks

```
# getting the network in the SVG format
# NOTICE how the SVG file sits inside the imgs folder
FILE = "./imgs/test_graph.svg"

(G, pos) = Graph_from_SVG(FILE)

#plot the optimal node distribution
nx.draw(G, pos, node_size = 150, node_color='black')
```

```
#save the graph on a figure file
#savefig("./data/test_network_best.png", dpi=200)
```

Degree Centrality

```
degree_centrality = nx.degree(G)

print (degree_centrality)
```

```
l = []

res = degree_centrality
for n in G.nodes():
    l.append(res[n])

nx.draw_networkx_edges(G, pos)
for n in G.nodes():
    list_nodes = [n]
    color = str( (res[n]-min(l))/float((max(l)-min(l))) )
    nx.draw_networkx_nodes(G, {n:pos[n]}, [n], node_size = 100, node_color = color)
```

Define a function that calculates the distance from a root node

```
def node_distance(G, root_node):
    queue = []
    list_distances = []
    queue.append(root_node)

    # reset the old keys
    if 'distance' in G.nodes[root_node]:
        for n in G.nodes():
            del G.nodes[n]['distance']

    G.nodes[root_node]["distance"] = 0

    # now...
    while len(queue):
        working_node = queue.pop(0)

        for n in G.neighbors(working_node):
```

```
            if len(G.nodes[n]) == 0:
                G.nodes[n]["distance"] = G.nodes[working_node] \
                ["distance"]+1

                queue.append(n)

    for n in G.nodes():
        list_distances.append(((root_node,n), G.nodes[n]["distance"]))

    return list_distances
```

Closeness Centrality

```
norm = 0.0
diz_c = {}
l_values = []

for n in G.nodes():
    l = node_distance(G,n)

    ave_length = 0

    for path in l:
        ave_length += float(path[1])/(G.number_of_nodes()-1-0)

    norm += 1/ave_length

    diz_c[n] = 1/ave_length

    l_values.append(diz_c[n])

#visualization
nx.draw_networkx_edges(G, pos)
for n in G.nodes():
    list_nodes=[n]
    color = str((diz_c[n]-min(l_values))/(max(l_values)- \
                                          min(l_values)))

    nx.draw_networkx_nodes(G, {n:pos[n]}, [n], node_size = \
                           100, node_color = color)
```

```
#savefig("./closeness_200.png",dpi=200)
```

Notice how lighter colors indicate higher closeness centrality.

Q1. Take the adjacency matrix and plot closeness centrality

```
norm = 0.0

diz_c = {}

l_values = []

adjacency_matrix = [
                     [0,1,0,1],
                     [1,0,1,1],
                     [0,1,0,0],
                     [1,1,0,0]
                   ]
```

```
A = np.asarray(adjacency_matrix)

#GM=nx.adjacency_matrix(adjacency_matrix)
GM = nx.Graph(A, nodetype = int)

print(GM.nodes())

for n in GM.nodes():
    l = node_distance(GM, n)
    print(l)

    ave_length=0

    for path in l:
        ave_length += float(path[1])/(GM.number_of_nodes()-1-0)

    norm += 1/ave_length

    diz_c[n] = 1/ave_length
```

```
    l_values.append(diz_c[n])

pos = {0: (87.823, 235.898), 1: (135.772, 207.394), 2: (132.986, 255.878), 3: (155.0, 101.0)}

nx.draw_networkx_edges(GM, pos)

for n in GM.nodes():
    list_nodes = [n]
    color = str((diz_c[n]-min(l_values))/(max(l_values) - min(l_values)))
    nx.draw_networkx_nodes(GM, {n:pos[n]}, [n], node_size = 100, node_color = color)
```

Betweenness Centrality

```
list_of_nodes = list(G.nodes())

num_of_nodes = G.number_of_nodes()

bc = {} #we will need this dictionary later on

for i in range(0, num_of_nodes-1):
    for j in range(i+1, num_of_nodes):
        print(list_of_nodes[i])
        print(list_of_nodes[j])

        paths = nx.all_shortest_paths(G,source = list_of_nodes[i], \
                                      target = list_of_nodes[j])

        count = 0.0

        path_diz = {}

        for p in paths:
            #print p
            count += 1.0

            for n in p[1:-1]:
                if not n in path_diz:
                    path_diz[n] = 0.0
                path_diz[n] += 1.0

        for n in path_diz.keys():
```

```
            path_diz[n] = path_diz[n]/count

            if not n in bc:
                bc[n] = 0.0

            bc[n] += path_diz[n]
```

```
#visualization
l = []

res = bc

for n in G.nodes():
    if not n in res:
        res[n] = 0.0

    l.append(res[n])

nx.draw_networkx_edges(G, pos)

for n in G.nodes():
    list_nodes = [n]

    color = str( (res[n]-min(l))/(max(l)-min(l)) )

    nx.draw_networkx_nodes(G, {n:pos[n]}, [n], node_size = 100, node_color = color)

#savefig("./data/betweenness_200.png",dpi=200)
```

Eigenvector Centrality

```
#networkx eigenvector centrality
centrality = nx.eigenvector_centrality_numpy(G)

#visualization
l = []

res = centrality

for n in G.nodes():
```

```
    if not n in res:
        res[n] = 0.0

    l.append(res[n])

nx.draw_networkx_edges(G, pos)

for n in G.nodes():
    list_nodes = [n]

    color = str( (res[n]-min(l))/(max(l)-min(l)) )

    nx.draw_networkx_nodes(G, {n:pos[n]}, [n], node_size = 100, node_color = color)

#savefig("eigenvetor_200.png",dpi=200)
```

Computing the Giant Connected Component

```
#Generating the test graph with two components

G_test = nx.Graph()

G_test.add_edges_from([('A','B'),('A','C'),('C','D'),('C','E'),
                       ('D','F'), ('D','H'),('D','G'),('E','G'),
                       ('E','I')])
#disconnetted node
G_test.add_node('X')

nx.draw(G_test, label = True)

#savefig("components_200.png",dpi=200)
```

Giant Component through a Breadth First Search

```
def giant_component_size(G_input):

    G = G_input.copy()

    components = []
    #print(G)
```

```
    node_list = list(G.nodes())
    #print(node_list.type)

    while len(node_list) != 0:
        #print(list(node_list))
        #print(node_list[0])
        root_node = node_list[0]
        component_list = []
        component_list.append(root_node)
        queue = []

        queue.append(root_node)

        G.nodes[root_node]["visited"] = True

        while len(queue):

            working_node = queue.pop(0)

            for n in G.neighbors(working_node):

                #check if any node attribute exists
                if len(G.nodes[n]) == 0:
                    G.nodes[n]["visited"] = True
                    queue.append(n)
                    component_list.append(n)

        components.append((len(component_list), component_list))

        #remove the nodes of the component just discovered
        for i in component_list: node_list.remove(i)

    components.sort(reverse = True)

    GiantComponent = components[0][1]
    SizeGiantComponent = components[0][0]

    return GiantComponent, len(components)
```

```
GCC, num_components = giant_component_size(G_test)
```

```
print ("Giant Connected Component:", GCC)

print ("Number of components:", num_components)
```

Q2. Use networkx built in function to calculate Giant Component and the list of edges.

```
#giant = max(nx.connected_component_subgraphs(G_test), key=len)
#giant=max(G_test.subgraph(c) for c in nx.strongly_connected_components(G))
#print(giant.nodes())
#print(giant.edges())

connected_component_subgraphs = (G_test.subgraph(c) for c in nx.connected_components(G_test))

largest_subgraph = max(connected_component_subgraphs, key = len)

largest_subgraph.edges()
```

Q3. Do the same as Q2. for a given adjacency matrix.

```
adjacency_matrix = [
                    [0,1,0,1],
                    [1,0,1,1],
                    [0,1,0,0],
                    [1,1,0,0],
                   ]
```

```
A = np.asarray(adjacency_matrix)

G = nx.Graph(A, nodetype = int)

connected_component_subgraphs = (G.subgraph(c) for c in nx.connected_components(G))

largest_subgraph = max(connected_component_subgraphs, key = len)

largest_subgraph.edges()
#G=nx.from_numpy_matrix(A)
#giant = max(nx.connected_component_subgraphs(G), key=len)
```

```
#giant=
#print(giant.nodes())
#print(giant.edges())
#print(nx.connected_component_subgraphs(G_test))
```