SOCIAL NETWORK ANALTICS LAB

DIGITAL ASSIGNMENT-1

Name: Matam Tejaswini
RegNo: 22MCB0034

AIM: To draw a directed and undirected graph from the edge.csv file.

Sol: To draw a directed and undirected graph from the edge file, we can use python and networkx library.

Code:

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

# Load the edge list from the CSV file
with open('edge.csv', 'r') as f:edges = [tuple(map(int, line.strip().split(','))) for line in f]

# Create a new graph object
G = nx.Graph()

# Add the edges to the graph
G.add_edges_from(edges)

# Draw the graph
nx.draw(G, with_labels=True)

# Show the graph
plt.show()
```

In this example, the code assumes that the edge.csv file is in the same directory as the Python script. If your edge.csv file is in a different directory, you will need to modify the file path accordingly.

The with **open()** block reads the CSV file and converts each row into a tuple of two integers representing an edge in the graph.

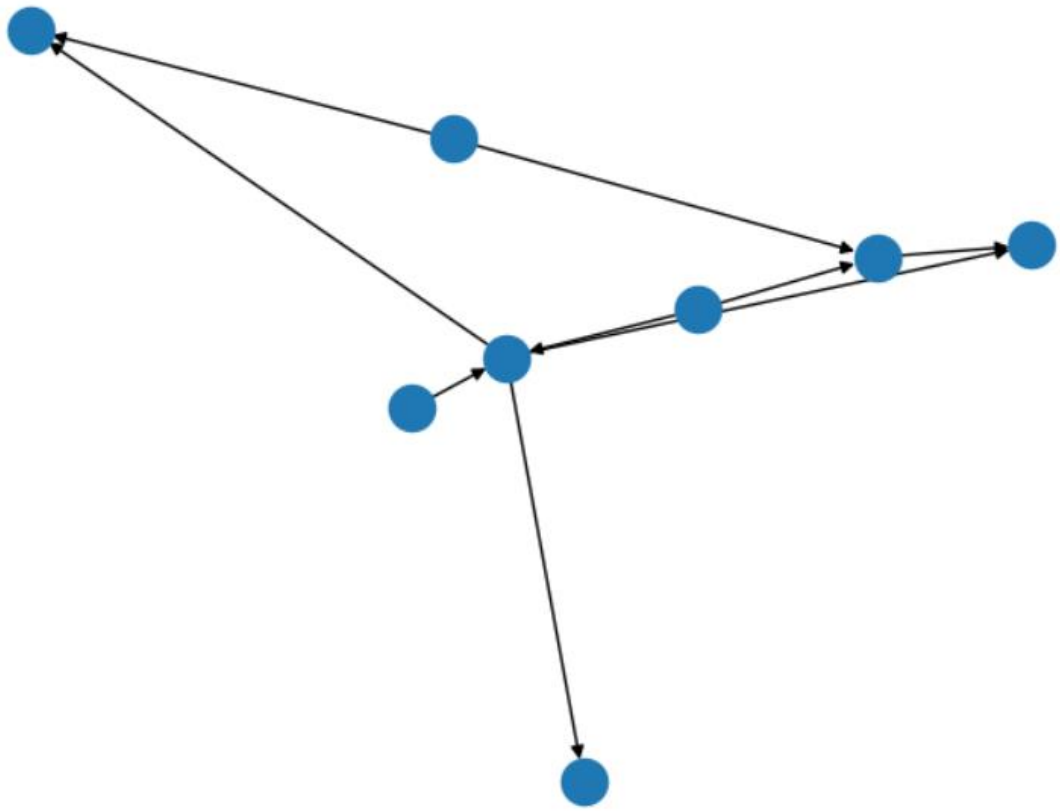
The **nx.Graph()** function creates a new graph object, which we then add edges to using **G.add_edges_from().** Depending on whether you want a directed or

undirected graph, you can use **nx.DiGraph()** or **nx.MultiDiGraph()** for directed graphs, or **nx.Graph()** or **nx.MultiGraph()** for undirected graphs.

Finally, we use **nx.draw()** to draw the graph, and **plt.show()** to display it on the screen. The **with_labels**=True option tells the **nx.draw()** function to include node labels in the graph visualization**.**

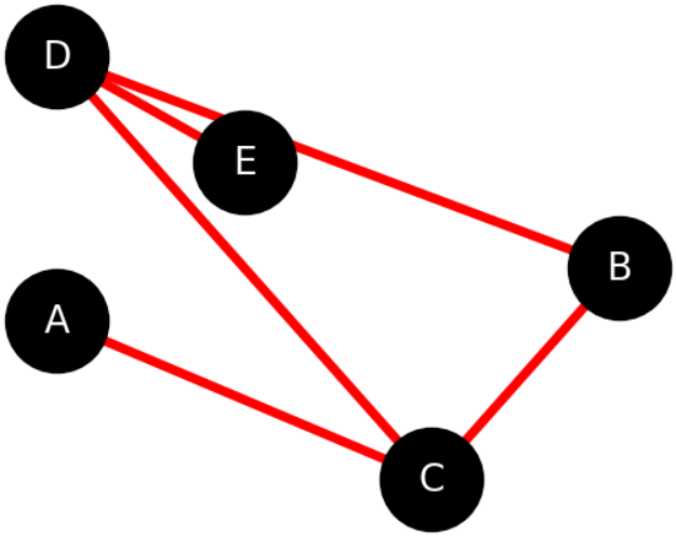
Directed graph:

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

G = nx.DiGraph()
G.add_edges_from(
    [('A', 'B'), ('A', 'C'), ('D', 'B'), ('E', 'C'), ('E', 'F'),
     ('B', 'H'), ('B', 'G'), ('B', 'F'), ('C', 'G')])

nx.draw(G)

plt.show()
```



Undirected graph:

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

G=nx.Graph()

G.add_node("A")
G.add_node("B")
G.add_node("C")
G.add_node("D")
G.add_node("E")

G.add_edge("A","C")
G.add_edge("B","C")
G.add_edge("B","D")
G.add_edge("C","D")
G.add_edge("D","E")

pos={
     "A":(1,5),
     "B":(4,6.),
     "C":(3.,2),
     "D":(1,10),
     "E":(2,8)
}

nx.draw(G,pos=pos,with_labels=True,node_color="black",node_size=3000,fo
nt_color="white",font_size=20,width=5,edge_color="red")
plt.margins(0.2)
plt.show()
```



AIM: To find the total number of nodes, edges, nodes with maximum degree and nodes with minimum degree in the graphs.

Sol: for finding the total number of nodes, edges, nodes with maximum degree and nodes with minimum degree in the graphs we have to use networkx library along with python.

```
import networkx as nx

# Load the edge list from the CSV file
with open('edge.csv', 'r') as f
edges = [tuple(map(int, line.strip().split(','))) for line in f]

# Create a new graph object
G = nx.Graph()

# Add the edges to the graph
G.add_edges_from(edges)

# Calculate the total number of nodes and edges
num_nodes = G.number_of_nodes()
num_edges = G.number_of_edges()

# Find the node with the maximum degree
max_degree_node = max(G, key=G.degree)

# Find the node with the minimum degree
min_degree_node = min(G, key=G.degree)

# Print the results
print(f"Total number of nodes: {num_nodes}")
print(f"Total number of edges: {num_edges}")
print(f"Node with maximum degree: {max_degree_node} (degree =
{G.degree(max_degree_node)})")
print(f"Node with minimum degree: {min_degree_node} (degree =
{G.degree(min_degree_node)})")
```

REMOVAL OF NODES:

```
import networkx as nx

g = nx.Graph()
g.add_nodes_from([1, 2, 3, 4, 5, 6])
g.add_edges_from([(1, 2), (2, 3), (3, 4), (3, 5), (4, 5), (4, 6), (5,
 6)])
print(g.nodes)
print(g.edges)

g.remove_node(1)
print(g.nodes)
print(g.edges)

g.remove_nodes_from([2, 3])
print(g.nodes)
print(g.edges)

g.remove_edges_from([(4, 5), (4, 6)])
print(g.nodes)
print(g.edges)
```

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

```
import networkx as nx

g = nx.Graph()
g.add_nodes_from([1, 2, 3, 4, 5, 6])
g.add_edges_from([(1, 2), (2, 3), (3, 4), (3, 5), (4, 5), (4, 6), (5,
 6)])
print(G.nodes)
print(G.edges)
print(G.number_of_nodes())
print(G.number_of_edges())

min_degree_node = min(G, key=G.degree)
```

```
max_degree_node = max(G, key=G.degree)

print("Total number of nodes: {number_of_nodes}")
print("Total number of edges: {number_of_edges}")
print("Node with maximum degree: {max_degree_node} (degree = {G.degre
e(max_degree_node)})")
print("Node with minimum degree: {min_degree_node} (degree = {G.degre
e(min_degree_node)})")
```

```
[1, 7, 2, 3, 6, 5, 4, 8, 9]
[(1, 1), (1, 7), (7, 2), (7, 6), (2, 1), (2, 2), (2, 3), (2, 6), (3, 5), (6, 4), (5, 4), (5, 8), (5, 9), (4, 3), (8,
9
15
Total number of nodes: {number_of_nodes}
Total number of edges: {number_of_edges}
Node with maximum degree: {max_degree_node} (degree = {G.degree(max_degree_node)})
Node with minimum degree: {min_degree_node} (degree = {G.degree(min_degree_node)})
```

In this example, we first load the edge list from the CSV file and create a new graph object using **nx.Graph()**. We then add the edges to the graph using **G.add_edges_from()**.

To find the total number of nodes and edges, we use the **G.number_of_nodes()** and **G.number_of_edges()** functions.

To find the node with the maximum degree, we use the **max()** function with the **key** argument set to **G.degree**, which returns the degree of each node. The **max()** function will then return the node with the maximum degree.

To find the node with the minimum degree, we do the same thing but use the **min()** function instead.

Finally, we print out the results using formatted strings.

AIM: Create an adjacency matrix for a directed graph from an edge.csv file

To create an adjacency matrix for a directed graph from an edge.csv file using Python and numpy.

Code:

```
import numpy as np

# Load the edge list from the CSV file
with open('edge.csv', 'r') as f:
    edges = [tuple(map(int, line.strip().split(','))) for line in f]

# Determine the number of nodes in the graph
num_nodes = max(max(edges)) + 1

# Create an empty adjacency matrix
adj_matrix = np.zeros((num_nodes, num_nodes))

# Populate the adjacency matrix
for edge in edges:
    adj_matrix[edge[0], edge[1]] = 1

# Print the adjacency matrix
print(adj_matrix)
```

In this example, we first load the edge list from the CSV file and determine the number of nodes in the graph by finding the maximum node ID and adding 1 (since node IDs start from 0). We then create an empty adjacency matrix using **np.zeros()** with dimensions **(num_nodes, num_nodes)**.

We then iterate through the edge list and populate the adjacency matrix by setting the appropriate entries to 1 (since this is a directed graph). Specifically, for each edge **(u, v)** in the edge list, we set **adj_matrix[u, v] = 1**.

Finally, we print the resulting adjacency matrix.

```
import numpy

edges = numpy.array([[0,1],[0,3],[1,2],[1,4],[2,5],[3,4],[3,5],[4,5]])

matrix = numpy.zeros((edges.max()+1, edges.max()+1))
matrix[edges[:,0], edges[:,1]] = 1
matrix
```

```
array([[0., 1., 0., 1., 0., 0.],
       [0., 0., 1., 0., 1., 0.],
       [0., 0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 1., 1.],
       [0., 0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0., 0.]])
```

TO CREATE A GRAPH USING ADJACENCY MATRIX

```
# Add a vertex to the set of vertices and the graph
def add_vertex(v):
  global graph
  global vertices_no
  global vertices
  if v in vertices:
    print("Vertex ", v, " already exists")
  else:
    vertices_no = vertices_no + 1
    vertices.append(v)
    if vertices_no > 1:
       for vertex in graph:
          vertex.append(0)
    temp = []
    for i in range(vertices_no):
       temp.append(0)
    graph.append(temp)

# Add an edge between vertex v1 and v2 with edge weight e
def add_edge(v1, v2, e):
    global graph
    global vertices_no
    global vertices
    # Check if vertex v1 is a valid vertex
    if v1 not in vertices:
       print("Vertex ", v1, " does not exist.")
    # Check if vertex v1 is a valid vertex
    elif v2 not in vertices:
       print("Vertex ", v2, " does not exist.")
    # Since this code is not restricted to a directed or
```

```
    # an undirected graph, an edge between v1 v2 does not
    # imply that an edge exists between v2 and v1
    else:
        index1 = vertices.index(v1)
        index2 = vertices.index(v2)
        graph[index1][index2] = e

# Print the graph
def print_graph():
  global graph
  global vertices_no
  for i in range(vertices_no):
    for j in range(vertices_no):
      if graph[i][j] != 0:
        print(vertices[i], " -> ", vertices[j], \
        " edge weight: ", graph[i][j])

# Driver code
# stores the vertices in the graph
vertices = []
# stores the number of vertices in the graph
vertices_no = 0
graph = []
# Add vertices to the graph
add_vertex(1)
add_vertex(2)
add_vertex(3)
add_vertex(4)
# Add the edges between the vertices by specifying
# the from and to vertex along with the edge weights.
add_edge(1, 2, 1)
add_edge(1, 3, 1)
add_edge(2, 3, 3)
add_edge(3, 4, 4)
add_edge(4, 1, 5)
print_graph()
print("Internal representation: ", graph)
```

```
1  ->  2  edge weight:  1
1  ->  3  edge weight:  1
2  ->  3  edge weight:  3
3  ->  4  edge weight:  4
4  ->  1  edge weight:  5
Internal representation:  [[0, 1, 1, 0], [0, 0, 3, 0], [0, 0, 0, 4], [5, 0, 0, 0
```

AIM: To find the centrality measures (betweenness, eigenvector, pagerank, closeness and degree) for each node in undirected graph from an csv file where source and target nodes are character values.

Sol: To find the centrality measures (betweenness, eigenvector, pagerank, closeness and degree) for each node in undirected graph from an csv file where

source and target nodes are character values, we can use python and networkx libraries.

Betweenness centrality: This measures the importance of a node in a network based on how many shortest paths between pairs of other nodes pass through it. A node with high betweenness centrality lies on many of the shortest paths between other nodes in the network**.**

Eigen vector centrality :This measures the importance of a node in a network eigenvector centrality is connected to other nodes that are also well-connected in the network.

Pagerank centrality: This measures the importance of a node in a network based on the concept of "random surfer" in the PageRank algorithm used by Google Search. A node with high PageRank is one that is likely to be visited by a random surfer who starts at a random node and follows outgoing links from each node.

Closeness centrality :This measures the importance of a node in a network based on how quickly it can reach all other nodes in the network. A node with high closeness centrality is one that is close to all other nodes in the network in terms of the shortest path distance.

Degree centrality: This measures the importance of a node in a network based on how many edges it has. A node with high degree centrality is one that is directly connected to many other nodes in the network**.**

Code:

```
import networkx as nx

# Load the edge list from the CSV file
with open('edge.csv', 'r') as f:
    edges = [tuple(line.strip().split(',')) for line in f]

# Create a new graph object
G = nx.Graph()

# Add the edges to the graph
G.add_edges_from(edges)

# Calculate the centrality measures for each node
```

```
betweenness = nx.betweenness_centrality(G)
eigenvector = nx.eigenvector_centrality(G)
pagerank = nx.pagerank(G)
closeness = nx.closeness_centrality(G)
degree = nx.degree_centrality(G)

# Print the results
for node in G.nodes():
    print(f"Node {node}:")
    print(f"Betweenness centrality: {betweenness[node]}")
    print(f"Eigenvector centrality: {eigenvector[node]}")
    print(f"PageRank: {pagerank[node]}")
    print(f"Closeness centrality: {closeness[node]}")
    print(f"Degree centrality: {degree[node]}")
    print()
```

In this example, we first load the edge list from the CSV file, where the source and target nodes are represented by character values. We create a new undirected graph object using **nx.Graph()** and add the edges to the graph using **G.add_edges_from()**.

We then calculate the centrality measures for each node using the following functions:

- **nx.betweenness_centrality()**: calculates the betweenness centrality for each node.
- **nx.eigenvector_centrality()**: calculates the eigenvector centrality for each node.
- **nx.pagerank()**: calculates the PageRank for each node.
- **nx.closeness_centrality()**: calculates the closeness centrality for each node.
- **nx.degree_centrality()**: calculates the degree centrality for each node.

We then print out the results for each node using a **for** loop over the nodes in the graph. For each node, we print out its character value, followed by its betweenness centrality, eigenvector centrality, PageRank, closeness centrality, and degree centrality.

Pagerank centrality:

```
def pagerank(G, alpha=0.85, personalization=None,
                        max_iter=100, tol=1.0e-6, nstart=None, weight='weight',
                        dangling=None):

        if len(G) == 0:
                return {}

        if not G.is_directed():
                D = G.to_directed()
        else:
                D = G

        # Create a copy in (right) stochastic form
        W = nx.stochastic_graph(D, weight=weight)
        N = W.number_of_nodes()

        # Choose fixed starting vector if not given
        if nstart is None:
                x = dict.fromkeys(W, 1.0 / N)
        else:
                # Normalized nstart vector
                s = float(sum(nstart.values()))
                x = dict((k, v / s) for k, v in nstart.items())

        if personalization is None:

                # Assign uniform personalization vector if not given
                p = dict.fromkeys(W, 1.0 / N)
        else:
                missing = set(G) - set(personalization)
                if missing:
                        raise NetworkXError('Personalization dictionary '
                                                        'must have a value for every
node. '
                                                        'Missing nodes %s' %
missing)
                s = float(sum(personalization.values()))
```

```
            p = dict((k, v / s) for k, v in personalization.items())

        if dangling is None:

            # Use personalization vector if dangling vector not specified
            dangling_weights = p
        else:
            missing = set(G) - set(dangling)
            if missing:
                raise NetworkXError('Dangling node dictionary '
                                    'must have a value for every
node. '
                                    'Missing nodes %s' %
missing)
            s = float(sum(dangling.values()))
            dangling_weights = dict((k, v/s) for k, v in dangling.items())
        dangling_nodes = [n for n in W if W.out_degree(n, weight=weight) ==
0.0]

        # power iteration: make up to max_iter iterations
        for _ in range(max_iter):
            xlast = x
            x = dict.fromkeys(xlast.keys(), 0)
            danglesum = alpha * sum(xlast[n] for n in dangling_nodes)
            for n in x:

                # this matrix multiply looks odd because it is
                # doing a left multiply x^T=xlast^T*W
                for nbr in W[n]:
                    x[nbr] += alpha * xlast[n] * W[n][nbr][weight]
                x[n] += danglesum * dangling_weights[n] + (1.0 - alpha) *
p[n]

            # check convergence, l1 norm
            err = sum([abs(x[n] - xlast[n]) for n in x])
            if err < N*tol:
                return x
        raise NetworkXError('pagerank: power iteration failed to converge '
                            'in %d iterations.' % max_iter)
```

```
{0: 0.15279451509556327,
 1: 0.08030389976086605,
 2: 0.09509770589511787,
 3: 0.08030389976086605,
 4: 0.08748987253768038,
 5: 0.06679084611307022,
 6: 0.126541905206293,
 7: 0.10778982486771263,
 8: 0.10778982486771264,
 9: 0.09509770589511787}
```

Eigan vector centrality:

```
def eigenvector_centrality(G, max_iter=20, tol=1.0e-6, nstart=None,
                           weight='weight'):

     from math import sqrt
    if type(G) == nx.MultiGraph or type(G) == nx.MultiDiGraph:
        raise nx.NetworkXException("Not defined for multigraphs.")

    if len(G) == 0:
        raise nx.NetworkXException("Empty graph.")

    if nstart is None:

        # choose starting vector with entries of 1/len(G)
        x = dict([(n,1.0/len(G)) for n in G])
    else:
        x = nstart

    # normalize starting vector
    s = 1.0/sum(x.values())
    for k in x:
        x[k] *= s
    nnodes = G.number_of_nodes()

    # make up to max_iter iterations
    for i in range(max_iter):
        xlast = x
        x = dict.fromkeys(xlast, 0)

        # do the multiplication y^T = x^T A
        for n in x:
            for nbr in G[n]:
                x[nbr] += xlast[n] * G[n][nbr].get(weight, 1)

        # normalize vector
```

```
        try:
            s = 1.0/sqrt(sum(v**2 for v in x.values()))

        # this should never be zero?
        except ZeroDivisionError:
            s = 1.0
        for n in x:
            x[n] *= s

        # check convergence
        err = sum([abs(x[n]-xlast[n]) for n in x])
        if err < nnodes*tol:
            return x

    raise nx.NetworkXError("""eigenvector_centrality():
power iteration failed to converge in %d iterations."%(i+1))""")

import networkx as nx
G = nx.path_graph(4)
centrality = nx.eigenvector_centrality(G)
print(['%s %0.2f'%(node,centrality[node]) for node in centrality])
```

```
['0 0.37', '1 0.60', '2 0.60', '3 0.37']
```

Betweenness centrality:

```
def betweenness_centrality(G, k=None, normalized=True, weight=None,
             endpoints=False, seed=None): r
  betweenness = dict.fromkeys(G, 0.0) # b[v]=0 for v in G
  if k is None:
    nodes = G
  else:
    random.seed(seed)
    nodes = random.sample(G.nodes(), k)
  for s in nodes:

    # single source shortest paths
    if weight is None: # use BFS
      S, P, sigma = _single_source_shortest_path_basic(G, s)
    else: # use Dijkstra's algorithm
      S, P, sigma = _single_source_dijkstra_path_basic(G, s, weight)

    # accumulation
    if endpoints:
      betweenness = _accumulate_endpoints(betweenness, S, P, sigma, s)
    else:
```

```
      betweenness = _accumulate_basic(betweenness, S, P, sigma, s)

  # rescaling
  betweenness = _rescale(betweenness, len(G), normalized=normalized,
            directed=G.is_directed(), k=k)
  return betweenness
import networkx as nx
G=nx.erdos_renyi_graph(20,1)
b=nx.betweenness_centrality(G)
print(b)
```

```
{0: 0.0, 1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0, 8: 0.0, 9: 0.0, 10: 0.0, 11: 0.0, 12: 0.0, 13: 0.0, 14: 0.0, 15: 0.0, 16: 0.0, 17: 0.0, 18: 0.0, 19: 0.0}
```

Closeness centrality:

```
def closeness_centrality(G, u=None, distance=None, normalized=True):
  r
  if distance is not None:

    # use Dijkstra's algorithm with specified attribute as edge weight
    path_length = functools.partial(nx.single_source_dijkstra_path_leng
th,
                    weight=distance)
  else:
    path_length = nx.single_source_shortest_path_length

  if u is None:
    nodes = G.nodes()
  else:
    nodes = [u]
  closeness_centrality = {}
  for n in nodes:
    sp = path_length(G,n)
    totsp = sum(sp.values())
    if totsp > 0.0 and len(G) > 1:
      closeness_centrality[n] = (len(sp)-1.0) / totsp

      # normalize to number of nodes-1 in connected part
      if normalized:
        s = (len(sp)-1.0) / ( len(G) - 1 )
        closeness_centrality[n] *= s
    else:
      closeness_centrality[n] = 0.0
  if u is not None:
    return closeness_centrality[u]
  else:
```

```
    return closeness_centrality

import networkx as nx
G=nx.erdos_renyi_graph(20,0.6)
c=nx.closeness_centrality(G)
print(c)
```

```
{0: 0.5757575757575758, 1: 0.76, 2: 0.6785714285714286, 3: 0.76, 4: 0.7916666666666666, 5: 0.7037037037037037, 6: 0.7307692307692307, 7: 0.7307692307692307, 8: 0.7037037037037037,
```

Degree centrality:

```
import networkx as nx

def degree_centrality(G, nodes):
    r
    top = set(nodes)
    bottom = set(G) - top
    s = 1.0/len(bottom)
    centrality = dict((n,d*s) for n,d in G.degree_iter(top))
    s = 1.0/len(top)
    centrality.update(dict((n,d*s) for n,d in G.degree_iter(bottom)))
    return centrality
import networkx as nx
G=nx.erdos_renyi_graph(20,0.6)
d=nx.degree_centrality(G)
print(d)
```

```
{0: 0.7368421052631579, 1: 0.8421052631578947, 2: 0.7894736842105263, 3: 0.5263157894736842, 4: 0.6842105263157894, 5: 0.6842105263157894, 6:
```