SmallSEOTools

PLAGIARISM SCAN REPORT

Words	874	Date	April 03,2021
Characters	7898	Excluded URL	

0% Plagiarism	100% Unique	0 Plagiarized Sentences	34 Unique Sentences

Content Checked For Plagiarism

```
graph =nx.read_weighted_edgelist("reachability.txt", comments='#', delimiter=None, create_using=nx.DiGraph() ,
nodetype=int, encoding='utf-8')
graph
# Setting the weights for all edges to 1
for e in graph.edges():
graph[e[0]][e[1]]['weight'] = 1
# Edited
#To import reachability-meta dataset
import pandas as pd
with open('reachability-meta.csv','r') as csv_file:
lines = csv_file.readlines()
lines = lines[1:]
city = []
k=0
city_no=dict()
#Storing the names of each airport city that corresponds to a node
for line in lines:
data = line.split(',')
temp = (data[1]+data[2])[1:-1].lower()
city.append(temp)
city_no[temp]=k
k+=1
print(city)
# 2- Generating summary of Graph
print(nx.info(graph))
# 3- Visualizing Betweeness Centrality ---> Airports that connects remote places
pos = nx.spring_layout(graph)
btwn_ctly = nx.betweenness_centrality(graph, normalized=True, endpoints=True)
# To Sort the nodes in descending order based on the Betweeness centrality measure
max_betweeness = sorted(btwn_ctly, key=btwn_ctly.get, reverse=True)[:5] #RESULT: Node no. with top 5 highest
betweeness
print("Airport Cities which connect remote places")
#Displaying the Results
for i in max_betweeness:
print(city[i])
# 4 a - Visualizing in-Degree Centrality ---> Busiest & Central HUB like Airports
degCent = nx.in_degree_centrality(graph) # get the in-degree centerality of each node
#sort the nodes in-degree centrality list
max_in_degree = sorted(degCent, key=degCent.get, reverse=True)[:5]
```

```
print('Airport Cities with High Arrivals')
for i in max_in_degree:
print(city[i]) # display top 5 cities name
# 4 b - Visualizing out-Degree Centrality ---> Busiest & Central HUB like Airports
degCent = nx.out_degree_centrality(graph)# get the out-degree centerality of each node
max_out_degree = sorted(degCent, key=degCent.get, reverse=True)[:5] # sort the out-degree centrality list
print('Airport Cities with High Departures')
for i in max_out_degree:
print(city[i]) # display top 5 cities name
# 5- Visualizing Closeness Centrality ---> Airport having many closest Airports(highest Accessibility)
cloCent = nx.closeness_centrality(graph) # get the Closeness centerality of each node
close=sorted(cloCent, key=cloCent.get, reverse=True)[:5] # sort the Closeness centrality list
for i in close:
print(city[i]) # display top 5 cities name
# 6- Visualizing Eigen vector Centrality ---> Influential Nodes (Airports)
eigCent = nx.eigenvector_centrality(graph) # get the Eigen vector centerality of each node
max_eigen_values = sorted(eigCent, key=eigCent.get, reverse=True)[:5] # sort the Eigen vector centrality list
for i in max_eigen_values:
print(city[i]) # display top 5 cities name
# 7 Shortest Path from a source to destination
source = input('Enter the Source City : ')
destination = input('Enter the destination City :')
source_node = city_no[source.lower()] # find the source city id
destination_node = city_no[destination.lower()] # find destination city id
path = nx.shortest_path(graph,source_node,destination_node,method='dijkstra') # finding the shortest path
sample_graph = graph
for i in path:
print(city[i],end=" -> ") # display the result
print()
try:
rm = input("Enter the city to remove : ")
# Finding Shortest Path after removing a node
sample_graph.remove_node(city_no[rm]) # find the city id which we want to remove
path = nx.shortest_path(sample_graph,source_node,destination_node,method='dijkstra') # finding the shortst path after
removed the node
for i in path:
print(city[i],end=" -> ") # display the result
except:
pass
# graph represent no of nodes VS degree( both in-degree and out-degree separatly)
print("graph represent no of nodes VS degree( both in-degree and out-degree separatly)")
plt.rcParams["figure.figsize"] = (20,10) # adjusting result graph size
in_degrees = G.in_degree() # dictionary node:degree
in_degrees = [v for k, v in in_degrees] # separating degree
in_values = sorted(set(in_degrees)) # sorting the degree values
in_hist = [in_degrees.count(x) for x in in_values] # finding no of nodes have a x degree for all x in calculated degree
plt.plot(in_values,in_hist,'ro-') # ploting in-degree
out_degrees = G.out_degree() #calculating out-degree
out_degrees = [v for k, v in out_degrees] # separating degree
out_values = sorted(set(out_degrees)) # sorting the degree values
out_hist = [out_degrees.count(x) for x in out_values] # finding no of nodes have a x degree for all x in calculated degree
plt.plot(out_values,out_hist,'bo-') # ploting out-degree
plt.legend(['In-degree','Out-degree']) # scale and reference
plt.xlabel('Degree') # x label
plt.ylabel('Number of nodes') # y label
plt.title('network') # title of the graph
plt.show() # plot the graph
sc=int(input("source")) # input : source
tar=int(input("target")) # input : target
dis=int(input("distance")) # input : no of hubs
ls1=list(nx.all_simple_edge_paths(G,sc,tar,dis)) # get the list of [(source,s1)...(sn,target)]
print(ls1) # print the list
g=nx.DiGraph() #generate the Digraph
for i in ls1:
for j in i:
```

```
g.add_edge(j[0],j[1]) # add the edges in the digraph based on calculated list
nodes=list(g.nodes()) #change the object datatype to list
no_col=[]
for i in nodes:
if i==sc:
no_col.append("green") # color the source node as green color
elif i==tar:
no_col.append("red") #color target node as red color
else:
no_col.append("yellow") # color other intermidiate nodes as yellow colour
plt.figure(figsize=(18,18)) # Width&Height of output Square box
nx.draw_networkx(g, with_labels=True,node_color=no_col, node_size=2000 ) # generate and visualization of graph
plt.show() #ploat the graph
sc=int(input("source")) #input : source node
lim=int(input("limit")) #INPUT : hub distance
dic=nx.dfs_successors(G, source=sc, depth_limit=lim) # perform depth first search on gicen source node and with given depth limit
print(dic) # display the list of [from node : { to nodes}]
g=nx.DiGraph() # generate the digraph for calulated list
for i in dic.keys():
for j in dic[i]:
g.add_edge(i,j) # add edge between the node based on the calculated list
nodes=list(g.nodes()) # converting object type to list type
no_col=[]
for i in nodes:
if i==sc:
no_col.append("green")
else:
no_col.append("yellow")
plt.figure(figsize=(18,18))
nx.draw_networkx(g, with_labels=True,node_color=no_col, node_size=1000 ) # generate the finall graph
plt.show() # plot the graph
G1=G.to_undirected(reciprocal=False, as_view=False) # converting digraph into undirected graph
list(nx.bridges(G1, root=None)) # finding and displaying the bridges
G1=G.to_undirected(reciprocal=True, as_view=False) # converting digraph into undirected graph
G1.remove_nodes_from(list(nx.isolates(G1))) #remomve the isolated nodes
plt.figure(figsize=(18,18)) # Width&Height of Square box
pos = nx.random_layout(G) #layout
nx.draw_networkx(G1, pos=pos,with_labels=True,node_color="yellow",edge_color=
("red","green","blue","black","orange","yellow"),node_size=1000 ) # generate graph
plt.show()
```

Sources	Similarity