# Pomoć oko funkcija

Python funkcija help dohvaća službenu dokumentaciju predane funkcije (koja često sadrži i primjere korištenja). Primjer:

```
help(networkx.Graph)
```

# Potrebni importovi

```
import networkx as nx

import matplotlib.pyplot as plt

import numpy as np

import itertools

import random

from simulation import Simulation
```

# Inicijalizacija mreže

```
graph = nx.Graph()

graph = nx.DiGraph()
```

# Dodavanje čvorova i veza

```
graph.add_node()

graph.add_nodes_from()


graph.add_edge()

graph.add_edges_from()
```

# Iteriranje po čvorovima ili vezama

```
graph.nodes()

graph.edges()
```

## Dodavanje i pristupanje atributima čvorova i veza

nx.set_node_attributes(graph, value, attribute)

nx.get_node_attributes(graph, attribute)


nx.set_edge_attributes(graph, value, attribute)

nx.get_edge_attributes(graph, attribute)


## Prikaz mreže

nx.draw(graph)

nx.draw_networkx(graph, node_color, labels, pos)


## Podmreže

nx.subgraph(graph, subgraph_nodes)

nx.find_cliques(graph)

nx.connected_components(graph)


## Stupnjevi čvorova

graph.degree()

graph.in_degree()

graph.out_degree()


## Čitanje i pisanje
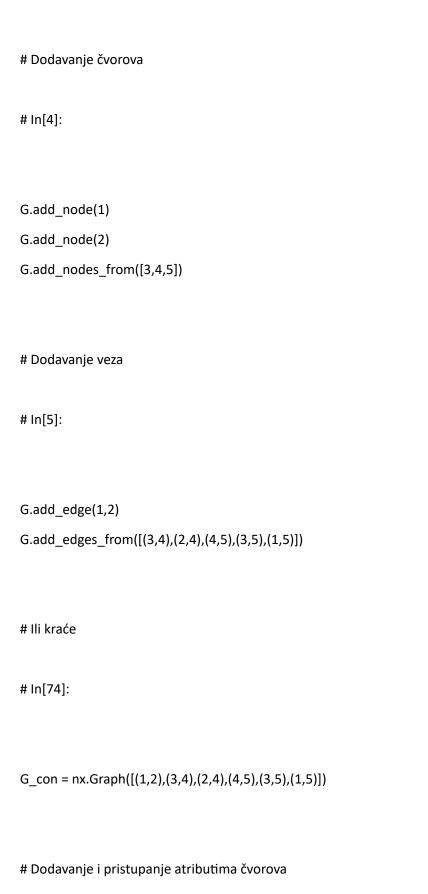
nx.read_edgelist(path)

nx.write_edgelist(graph, path)


## Svojstvene vrijednosti mreže

graph.number_of_nodes()

graph.number_of_edges()

nx.degree_assortativity_coefficient(graph)

nx.average_shortest_path_length(graph)

nx.diameter(graph)

nx.average_clustering(graph)

nx.degree_centrality(graph)

nx.betweenness_centrality(graph)

nx.is_connected(graph)

nx.density(graph)

## K-jezgrena dekompozicija

nx.k_core(graph, k)

nx.k_shell(graph, k)

## Slučajni modeli

nx.gnp_random_graph(n_nodes, probability)

nx.gnm_random_graph(n_nodes, n_edges)

nx.watts_strogatz_graph(n_nodes, n_neighbours_to_join, rewiring_probability)

nx.barabasi_albert_graph(n_nodes, n_connections)

## Kombinacije elemenata

itertools.combinations(iterator, combination_length)

## Simulacija

sim = Simulation(G, initial_state, state_transition, name)

sim.state()

sim.run()

sim.plot()

# Auditorne 1

```python
#!/usr/bin/env python
# coding: utf-8
```

```python
# In[1]:
```

```python
#pip install matplotlib
#pip install networkx
#pip install numpy
```

```python
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
import random
```

## # Inicijalizacija mreže

```python
# Inicijalizacija neusmjerene mreže
```

```python
# In[3]:
```

```python
G = nx.Graph()
```

# Dodavanje čvorova

# In[4]:

```python
G.add_node(1)
G.add_node(2)
G.add_nodes_from([3,4,5])
```

# Dodavanje veza

# In[5]:

```python
G.add_edge(1,2)
G.add_edges_from([(3,4),(2,4),(4,5),(3,5),(1,5)])
```

# Ili kraće

# In[74]:

```python
G_con = nx.Graph([(1,2),(3,4),(2,4),(4,5),(3,5),(1,5)])
```

# Dodavanje i pristupanje atributima čvorova

```python
# In[7]:
```
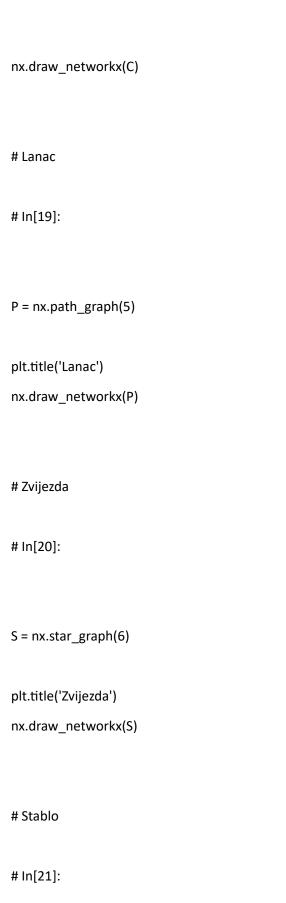
```python
nx.set_node_attributes(G, "vrijednost", "atribut")
print(nx.get_node_attributes(G, "atribut"))
```

```python
# Pristupanje i dodavanje atributa pojedinom čvoru
```

```python
# In[8]:
```

```python
G.nodes[1]["atribut"] = "nova vrijednost"
G.nodes[2]["atribut 2"] = "druga vrijednost"
```

```python
# Dodavanje atributa čvora rječnikom rječnika
```

```python
# In[9]:
```

```python
node_attributes_dictionary = {1 : {"Atribut 1" : "Vrijednost 11"},
                2 : {"Atribut 1" : "Vrijednost 12", "Atribut 2" : "Vrijednost 22"},
                3 : {"Atribut 2" : "Vrijednost 23", "Atribut 1" : "Vrijednost 13"},
                4 : {"Atribut 3" : "Vrijednost 34"}}

nx.set_node_attributes(G, node_attributes_dictionary)
print(nx.get_node_attributes(G, "Atribut 1"))
```

# Dodavanje i pristupanje atributima vezama

# In[10]:

```python
nx.set_edge_attributes(G, "vrijednost", "atribut")
print(nx.get_edge_attributes(G, "atribut"))
```

# Pristupanje i dodavanje atributa pojedinoj vezi

# In[11]:

```python
G.edges[(1,2)]["atribut"] = "nova vrijednost"
```

# Dodavanje atributa veza rječnikom rječnika

# In[12]:

```python
edge_attributes_dictionary = {(1,2) : {"Atribut 1" : "Vrijednost 112"},
                (2,4) : {"Atribut 1" : "Vrijednost 122", "Atribut 2" : "Vrijednost 222"},
                (4,5) : {"Atribut 2" : "Vrijednost 245", "Atribut 1" : "Vrijednost 145"},
                (3,5) : {"Atribut 3" : "Vrijednost 335"}}

nx.set_edge_attributes(G, edge_attributes_dictionary)
```

```python
print(nx.get_edge_attributes(G, "Atribut 1"))
```

# Crtanje mreže

# In[13]:

```python
plt.title('Mreža G')
nx.draw_networkx(G)
```

# Podešavanje parametara crtanja

# In[14]:

```python
pos = nx.spring_layout(G_con, k = 0.1, seed = 7)
node_labels = nx.get_node_attributes(G, 'atribut')
```

# In[15]:

```python
plt.title('Mreža G_con')
nx.draw_networkx(G_con, labels = node_labels, pos = pos)
```

# Inicijalizacija usmjerene mreže

```
# In[16]:


D = nx.DiGraph([(1,2),(2,1),(4,3),(2,4),(4,5),(3,5),(1,5)])


plt.title('Usmjerena mreža')

nx.draw_networkx(D)


# Bipartitna mreža


# In[17]:


B = nx.complete_bipartite_graph(4,5)


plt.title('Bipartitna mreža')

nx.draw_networkx(B)


# Ciklička mreža


# In[18]:


C = nx.cycle_graph(4)


plt.title('Ciklička mreža')
```

```python
nx.draw_networkx(C)
```

```python
# Lanac
```

```python
# In[19]:
```

```python
P = nx.path_graph(5)
```

```python
plt.title('Lanac')
nx.draw_networkx(P)
```

```python
# Zvijezda
```

```python
# In[20]:
```

```python
S = nx.star_graph(6)
```

```python
plt.title('Zvijezda')
nx.draw_networkx(S)
```

```python
# Stablo
```

```python
# In[21]:
```

```python
T = nx.generators.balanced_tree(2, h = 3)


plt.title('Stablo')

nx.draw_networkx(T)



# Inicijalizacija mreže s težinama


# In[22]:



D_w = nx.DiGraph()

D_w.add_weighted_edges_from([(1,2,3),(2,1,4),(4,3,1),(2,4,2),(4,5,2),(3,5,6),(1,5,1)])



# Crtanje mreže s težinama


# In[23]:



pos = nx.spring_layout(D_w, seed=7)

labels = nx.get_edge_attributes(D_w,'weight')


plt.title('Mreža D_w')

nx.draw_networkx(D_w,pos)

nx.draw_networkx_edge_labels(D_w, pos = pos, edge_labels = labels)
```

# Prolazak po čvorovima i vezama

# In[24]:

```python
print(D_w.number_of_nodes())
print(D_w.number_of_edges())
```

# In[25]:

```python
print(D_w.nodes())
print(D_w.edges())
```

# In[26]:

```python
for node in D_w.neighbors(4):
    print(node)
```

# In[27]:

```python
for node in D_w.predecessors(4):
    print(node)
```

```
# In[28]:


for node in D_w.successors(4):

    print(node)



# Podmreže


# In[29]:



D_w_subgraph = nx.subgraph(D_w, (3,4,5))

D_w_subgraph_V2 = D_w.subgraph((3,4,5))



# In[30]:



plt.subplot(131)

plt.title('Cijela mreža')

pos = nx.spring_layout(D_w, seed = 7)

labels = nx.get_edge_attributes(D_w, 'weight')

nx.draw_networkx(D_w, pos)

nx.draw_networkx_edge_labels(D_w, pos, labels)


plt.subplot(132)

plt.title('Podmreža')
```

```python
pos = nx.spring_layout(D_w_subgraph, seed = 7)

labels = nx.get_edge_attributes(D_w_subgraph, 'weight')

nx.draw_networkx(D_w_subgraph, pos)

nx.draw_networkx_edge_labels(D_w_subgraph, pos, labels)


plt.subplot(133)

plt.title('Podmreža V2')

pos = nx.spring_layout(D_w_subgraph_V2, seed = 7)

labels = nx.get_edge_attributes(D_w_subgraph_V2, 'weight')

nx.draw_networkx(D_w_subgraph_V2, pos)

nx.draw_networkx_edge_labels(D_w_subgraph_V2, pos, labels)



# Klike


# In[31]:



G_clique_example = nx.Graph([(1,2),(1,3),(2,3),(3,4),(4,5),(4,6),(5,6)])


plt.title('Mreža G_clique_example')

nx.draw_networkx(G_clique_example, pos = nx.spring_layout(G_clique_example, seed = 7))



# In[32]:



G_cliques = nx.find_cliques(G_clique_example)
```

```python
clique_n = 0
clique_colors = ['red', 'green']
for clique in G_cliques:
    print(clique)
    if len(clique) > 2:
        nx.set_node_attributes(G_clique_example.subgraph(clique), clique_colors[clique_n], "color")
        clique_n+=1


plt.title('Klike s više od 2 člana')
nx.draw_networkx(G_clique_example,  node_color = list(nx.get_node_attributes(G_clique_example,
"color").values()), pos = nx.spring_layout(G_clique_example, seed = 7))
```

```python
# Gustoća
```

```python
# In[33]:
```

```python
print(f"Gustoća mreže G : {nx.density(G)}")
print(f"Gustoća mreže D_w : {nx.density(D_w)}")
```

```python
# Stupanj čvora
```

```python
# In[34]:
```

```python
print(f"Stupanj čvora 4 mreže G : {G.degree(4)}")
```

```python
print(f"Stupnjevi svih čvorova mreže G : {G.degree()}")


G_nodes_degrees = G.degree()

nx.set_node_attributes(G, dict(G_nodes_degrees), "degree")


plt.title('Mreža G s naznačenim stupnjevima')

nx.draw_networkx(G, labels = nx.get_node_attributes(G, "degree"), pos = nx.spring_layout(G, seed = 7))




# Stupanj čvora za usmjerene mreže


# In[35]:



print(f"Stupanj čvora 4 mreže D : {D.degree(4)}")

print(f"Ulazni stupanj čvora 4 mreže D : {D.in_degree(4)}")

print(f"Izlazni stupanj čvora 4 mreže D : {D.out_degree(4)}")

print(f"Stupnjevi svih čvorova mreže D : {D.degree()}")

print(f"Ulazni stupnjevi svih čvorova mreže D : {D.in_degree()}")

print(f"Izlazni stupnjevi svih čvorova mreže D : {D.out_degree()}")


D_nodes_degrees = D.in_degree()

nx.set_node_attributes(D, dict(D_nodes_degrees), "in_degree")


plt.title('Mreža D s naznačenim stupnjevima')

nx.draw_networkx(D, labels = nx.get_node_attributes(D, "in_degree"), pos = nx.spring_layout(D, seed = 7))
```

# Prosječni stupanj čvora

# In[36]:

```python
total_nodes = D.number_of_nodes()

total_degree = 0
total_in_degree = 0
total_out_degree = 0

for (node, degree) in D.degree():
    total_degree += degree

for (node, in_degree) in D.in_degree():
    total_in_degree += in_degree

for (node, out_degree) in D.out_degree():
    total_out_degree += out_degree


print(f"Prosječni stupanj čvora : {total_degree/total_nodes}")
print(f"Prosječni ulazni stupanj čvora : {total_in_degree/total_nodes}")
print(f"Prosječni izlazni stupanj čvora : {total_out_degree/total_nodes}")
```

# Mrežna reprezentacija

# In[37]:

```python
print(f"Lista susjedstva za mrežu G : \n {list(nx.generate_adjlist(G))}")
print(f"Matrica susjedstva za mrežu G : \n {nx.adjacency_matrix(G).todense()}")
print(f"Lista veza za mrežu G :  \n {list(nx.generate_edgelist(G))}")
```

# In[38]:

```python
print(f"Lista susjedstva za mrežu D_w : \n {list(nx.generate_adjlist(D_w))}")
print(f"Matrica susjedstva za mrežu D_w : \n {nx.adjacency_matrix(D_w).todense()}")
print(f"Lista veza za mrežu D_w :  \n {list(nx.generate_edgelist(D_w))}")
```

# Spremanje i učitavanje mreža pomoću liste veza

# In[39]:

```python
nx.write_edgelist(D_w, 'D_w.edgelist')
with open('D_w.edgelist') as f:
    head =[next(f) for x in range(5)]
print(head)
D_w_copy = nx.read_edgelist('D_w.edgelist')
```

# # Primjeri mreža

```python
# Dodatni primjeri dostupni na: <br>
# https://snap.stanford.edu/data/ <br>
# http://vlado.fmf.uni-lj.si/pub/networks/data/


# Članova karate kluba


# In[40]:



G_karate = nx.karate_club_graph()
print(f"Broj čvorova : {G_karate.number_of_nodes()}")
print(f"Broj rubova : {G_karate.number_of_edges()}")



# In[41]:



plt.title('Mreža članova karate kluba')
nx.draw_networkx(G_karate, node_size = [v * 100 for v in dict(G_karate.degree).values()], pos = nx.spring_layout(G_karate, seed = 7))



# Mreža interakcije proteina


# In[42]:



G_protein = nx.read_edgelist('protein_interaction.edgelist')
d = dict(G_protein.degree)
```

```python
print(f"Broj čvorova : {G_protein.number_of_nodes()}")

print(f"Broj rubova : {G_protein.number_of_edges()}")
```

# In[43]:

```python
plt.title('Mreža interakcije proteina')

nx.draw_networkx(G_protein, node_size = [v*100 for v in d.values()],pos = nx.spring_layout(G_protein, seed = 7))
```

# Peer-to-peer mreža

# In[44]:

```python
G_network = nx.read_edgelist('p2p-Gnutella08.txt')

print(f"Broj čvorova : {G_network.number_of_nodes()}")

print(f"Broj rubova : {G_network.number_of_edges()}")
```

# Provjera povezanosti mreže i dobivanje dobivanje najveće komponente

# In[45]:

```python
print(f"Mreža je povezana? {nx.is_connected(G_network)}")
```

```python
G_network_components = sorted([G_network.subgraph(c).copy() for c in
nx.connected_components(G_network)], key= len, reverse = True)


for (index, network_components) in enumerate(G_network_components):

    print(f"Komponenta indexa {index} je povezana? {nx.is_connected(G_network_components[index])}")
```

# In[46]:

```python
plt.figure(figsize = (30,30))

plt.title('Velika komponenta P2P mreže')

nx.draw_networkx(G_network_components[0], node_size = [v*100 for v in
dict(G_network_components[0].degree).values()], pos = nx.spring_layout(G_network_components[0], k
= 10) , alpha = 0.05, with_labels = False)
```

# In[47]:

```python
plt.title('Druga komponenta P2P mreže')

nx.draw_networkx(G_network_components[1], pos = nx.spring_layout(G_network_components[1], k =
0.1), node_size = 300)
```

# Asortativnost

# In[48]:

```python
print(f"Asortativnost karate : {nx.degree_assortativity_coefficient(G_karate)}")

print(f"Asortativnost proteini : {nx.degree_assortativity_coefficient(G_protein)}")

print(f"Asortativnost P2P mreža : {nx.degree_assortativity_coefficient(G_network)}")
```

```python
# Prosječan najkraći put
```

```python
# In[49]:
```

```python
print(f"Prosječan najkraći put karate : {nx.average_shortest_path_length(G_karate)}")

print(f"Prosječan najkraći put proteini : {nx.average_shortest_path_length(G_protein)}")

print(f"Prosječan najkraći put najveće komponente P2P mreže :
{nx.average_shortest_path_length(G_network_components[0])}")
```

```python
# Dijametar
```

```python
# In[50]:
```

```python
print(f"Dijametar karate : {nx.diameter(G_karate)}")

print(f"Dijametar proteini : {nx.diameter(G_protein)}")

print(f"Dijametar najveće komponente P2P mreže : {nx.diameter(G_network_components[0])}")
```

```python
# Koeficijent klasteriranja čvorova
```

```python
# In[51]:
```

```python
print(f"Koeficijent klasteriranja čvorova karate : {nx.average_clustering(G_karate)}")
print(f"Koeficijent klasteriranja čvorova protein : {nx.average_clustering(G_protein)}")
print(f"Koeficijent klasteriranja čvorova mreža : {nx.average_clustering(G_network)}")


# Breadth first search

# Prolazak rubova BFS-om počevsi od čvora 0

# In[52]:


for edge in nx.bfs_edges(G_karate, 0):
    print(edge)


# Konstrukcija stabla pomoću BFS-a

# In[53]:


some = nx.bfs_tree(G_karate, 0)

plt.title('Stablo konstruirano pomoću BFS-a')
nx.draw_networkx(some)
```

# Distribucija stupnja


# In[54]:


```python
def degree_calc(G):
    degrees = [val for (node, val) in G.degree()]
    avg_degree = np.mean(degrees)
    med_degree = np.median(degrees)

    return degrees, avg_degree, med_degree
```


# In[55]:


```python
def degree_distribution_plot(degree_list, avg_degree, med_degree, cumulative, title):
    plt.hist(degree_list,label='Distribucija stupnja', cumulative = cumulative)
    plt.axvline(avg_degree,color='r',linestyle='dashed',label='Prosječni stupanj')
    plt.axvline(med_degree,color='g',linestyle='dashed',label='Medijan stupanj')
    plt.legend()
    plt.ylabel('Postotak čvorova')
    plt.xlabel('Iznos stupnja')
    plt.title(title)
```


# In[56]:

```python
d_list_karate, avg_d_karate, med_d_karate = degree_calc(G_karate)

d_list_protein, avg_d_protein, med_d_protein = degree_calc(G_protein)

d_list_mreza, avg_d_mreza, med_d_mreza = degree_calc(G_network)
```

# In[57]:

```python
degree_distribution_plot(d_list_karate, avg_d_karate, med_d_karate, False, 'Karate')
```

# In[58]:

```python
degree_distribution_plot(d_list_protein, avg_d_protein, med_d_protein, False, 'Proteini')
```

# In[59]:

```python
degree_distribution_plot(d_list_mreza, avg_d_mreza, med_d_mreza, False, 'P2P mreža')
```

# Distribucija bliskosti

# In[60]:

```python
def closeness_calc(G):
```

```python
    closeness = [val for (node, val) in nx.closeness_centrality(G).items()]

    avg_closeness = np.mean(closeness)

    med_closeness = np.median(closeness)


    return closeness, avg_closeness, med_closeness
```

# In[61]:

```python
def closeness_distribution_plot(closeness_list, avg_closeness, med_closeness, cumulative, title):

    plt.hist(closeness_list,label='Distribucija bliskosti', cumulative = cumulative)

    plt.axvline(avg_closeness, color='r' ,linestyle='dashed', label = "Prosječna bliskost")

    plt.axvline(med_closeness, color='g' ,linestyle='dashed', label = "Medijan bliskosti")

    plt.legend()

    plt.ylabel('Postotak čvorova')

    plt.xlabel('Iznost bliskosti')

    plt.title(title)
```

# In[62]:

```python
c_list_karate, avg_c_karate, med_c_karate = closeness_calc(G_karate)

c_list_protein, avg_c_protein, med_c_protein = closeness_calc(G_protein)

c_list_mreza, avg_c_mreza, med_c_mreza = closeness_calc(G_network)
```

# In[63]:

```python
closeness_distribution_plot(c_list_karate, avg_c_karate, med_c_karate, False, 'Karate')
```

# In[64]:

```python
closeness_distribution_plot(c_list_protein, avg_c_protein, med_c_protein, False, 'Proteini')
```

# In[65]:

```python
closeness_distribution_plot(c_list_mreza, avg_c_mreza, med_c_mreza, False, 'P2P mreža')
```

# Distribucija međupoloženosti

# In[66]:

```python
def calc_betweenness(G):
    betweenness = [val for (node, val) in nx.betweenness_centrality(G).items()]
    avg_betweenness = np.mean(betweenness)
    med_betweenness = np.median(betweenness)

    return betweenness, avg_betweenness, med_betweenness
```

```
# In[67]:


def betweenness_distribution_plot(betweenness_list, avg_betweenness, med_betweenness,
cumulative, title):

    plt.hist(betweenness_list,label='Distribucija međupoloženosti', cumulative = cumulative)

    plt.axvline(avg_betweenness,color='r',linestyle='dashed',label='Prosječna međupoloženost')

    plt.axvline(med_betweenness,color='g',linestyle='dashed',label='Medijan međupoloženosti')

    plt.legend()

    plt.ylabel('Postotak čvorova')

    plt.xlabel('Iznos međupoloženosti')

    plt.title(title)


# In[68]:


b_list_karate, avg_b_karate, med_b_karate = calc_betweenness(G_karate)

b_list_protein, avg_b_protein, med_b_protein = calc_betweenness(G_protein)

b_list_mreza, avg_b_mreza, med_b_mreza = calc_betweenness(G_network)


# In[69]:


betweenness_distribution_plot(b_list_karate, avg_b_karate, med_b_karate, False, 'Karate')
```

# In[70]:

```python
betweenness_distribution_plot(b_list_protein, avg_b_protein, med_b_protein, False, 'Proteini')
```

# In[71]:

```python
betweenness_distribution_plot(b_list_mreza, avg_b_mreza, med_b_mreza, False, 'P2P mreža')
```

# Heterogenost

# In[72]:

```python
def calc_heterogenity(g):
    average_squared_degree = sum([degree**2 for (node, degree) in g.degree()])/g.number_of_nodes()
    average_degree = sum(degree for (node, degree) in g.degree())/g.number_of_nodes()

    heterogenity = average_squared_degree/(average_degree**2)

    return heterogenity
```

# In[73]:

```python
print(f"Karate heterogenost : {calc_heterogenity(G_karate)}")
print(f"Proteini heterogenost : {calc_heterogenity(G_protein)}")
print(f"P2P mreža heterogenost : {calc_heterogenity(G_network)}")
```

# Auditorne 2

```python
#!/usr/bin/env python
# coding: utf-8
```

# In[1]:

```python
import networkx as nx
import random
import matplotlib.pyplot as plt
import numpy as np
```

# Robusnost

# In[2]:

```python
g_karate = nx.karate_club_graph()
g_protein = nx.read_edgelist('protein_interaction.edgelist')
```

# In[3]:

```python
def get_giant_component_size(g):
    return len(max(nx.connected_components(g), key = len))
```

```python
# In[4]:


def failure(g, n_steps):
    c = g.copy()
    n_nodes_start = c.number_of_nodes()
    n_nodes_to_remove = max(n_nodes_start//n_steps, 1)
    #n_nodes_to_remove = np.ceil(n_node_start/n_steps)

    relative_giant_component_size = []
    relative_giant_component_size.append(1)

    relative_n_nodes_removed = []
    relative_n_nodes_removed.append(0)


    for step in range(1, n_steps + 1):
        if c.number_of_nodes() > n_nodes_to_remove:
            nodes_to_leave = random.sample(list(c.nodes), c.number_of_nodes() - n_nodes_to_remove)
            c = nx.subgraph(c, nodes_to_leave)

            giant_component_size = get_giant_component_size(c)

            relative_giant_component_size.append(giant_component_size/n_nodes_start)
            relative_n_nodes_removed.append(1- (c.number_of_nodes()/n_nodes_start))

        else:
            relative_giant_component_size.append(0)
```

```python
        relative_n_nodes_removed.append(1)

    return relative_giant_component_size, relative_n_nodes_removed


# In[5]:


def attack(g, n_steps):
    c = g.copy()
    n_nodes_start = c.number_of_nodes()
    n_nodes_to_remove = max(n_nodes_start//n_steps, 1)
    #n_nodes_to_remove = np.ceil(n_node_start/n_steps)

    relative_giant_component_size = []
    relative_giant_component_size.append(1)

    relative_n_nodes_removed = []
    relative_n_nodes_removed.append(0)


    for step in range(1, n_steps + 1):
        if c.number_of_nodes() > n_nodes_to_remove:
            nodes_to_leave = sorted(c.nodes, key = c.degree, reverse = True)[n_nodes_to_remove:]
            c = nx.subgraph(c, nodes_to_leave)

            giant_component_size = get_giant_component_size(c)

            relative_giant_component_size.append(giant_component_size/n_nodes_start)
```

```python
        relative_n_nodes_removed.append(1- (c.number_of_nodes()/n_nodes_start))


    else:
        relative_giant_component_size.append(0)
        relative_n_nodes_removed.append(1)


    return relative_giant_component_size, relative_n_nodes_removed
```

# In[6]:

```python
def plot_comparison(n_nodes_removed, giant_component_failure, giant_component_attack):
    plt.plot(n_nodes_removed, giant_component_failure, color = 'b', label = "Kvar")
    plt.plot(n_nodes_removed, giant_component_attack, color = 'r', label = "Napad")
    plt.legend()
    plt.ylabel("Veličina gigantske komponente")
    plt.xlabel("Uklonjenih čvorova")
```

# In[7]:

```python
giant_component_failure_karate, n_nodes_removed_karate = failure(g_karate, 30)
giant_component_attack_karate, g_nodes_removed_karate = attack(g_karate, 30)
```

# In[8]:

```python
plot_comparison(g_nodes_removed_karate, giant_component_failure_karate,
giant_component_attack_karate)
```

# In[9]:

```python
giant_component_failure_protein, n_nodes_removed_protein = failure(g_protein, 30)
giant_component_attack_protein, n_nodes_removed_protein = attack(g_protein, 30)
```

# In[10]:

```python
plot_comparison(n_nodes_removed_protein, giant_component_failure_protein,
giant_component_attack_protein)
```

# K-jezgrena dekompozicija

# In[11]:

```python
max_core = max(list(nx.core_number(g_karate).values()))
```

# In[12]:

```python
k_core_dict = {}
k_shell_dict = {}


for core in range(0, max_core + 2):
    k_core_dict[core] = nx.k_core(g_karate, core)
    k_shell_dict[core] = nx.k_shell(g_karate, core)



# In[13]:


pos = nx.spring_layout(g_karate, seed = 2)


plt.figure(figsize = (30, 30))
plt.subplot(321)
plt.title('0-jezgra')
nx.draw_networkx(k_core_dict[0])


plt.subplot(322)
plt.title('1-jezgra')
nx.draw_networkx(k_core_dict[1])


plt.subplot(323)
plt.title('2-jezgra')
nx.draw_networkx(k_core_dict[2])


plt.subplot(324)
plt.title('3-jezgra')
```

```python
nx.draw_networkx(k_core_dict[3])


plt.subplot(325)

plt.title('4-jezgra')

nx.draw_networkx(k_core_dict[4])


plt.subplot(326)

plt.title('5-jezgra')

nx.draw_networkx(k_core_dict[5])
```

# In[14]:

```python
plt.figure(figsize = (30, 30))

plt.subplot(321)

plt.title('0-ljuska')

nx.draw_networkx(k_shell_dict[0])


plt.subplot(322)

plt.title('1-ljuska')

nx.draw_networkx(k_shell_dict[1])


plt.subplot(323)

plt.title('2-ljuska')

nx.draw_networkx(k_shell_dict[2])


plt.subplot(324)

plt.title('3-ljuska')
```

```
nx.draw_networkx(k_shell_dict[3])


plt.subplot(325)

plt.title('4-ljuska')

nx.draw_networkx(k_shell_dict[4])


plt.subplot(326)

plt.title('5-ljuska')

nx.draw_networkx(k_shell_dict[5])
```

# # 2.1. USMJERENE MREŽE SOCIJALNIH INTERAKCIJA


# U drugom djelu ove pokazne vježbe koristimo znanstvene Twitter podatke za stvaranje i istraživanje usmjerenih mreža društvenih interakcija.


# ## DATASET:


# In[15]:


```
import json

search_tweets = json.load(open('science_tweets.json'))
```


# Svaki tweet zapravo je jedna instanca Tweet objekta (https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model/tweet)


# In[16]:

search_tweets[:2]


# # Twitter retweetanje


# Temeljna interakcija u ekosustavu Twittera je "retweet" -- ponovno emitiranje tweeta drugog korisnika vašim pratiteljima.


# ## Filtriranje retweetova


# U našem skupu podataka nalaze se retweetovi. Objekt tweeta koji se nalazi u našem datasetu je retweet ako uključuje 'retweeted_status'. Napravit ćemo novi skup podataka koji će se sastojati samo od retweetova:


# In[17]:


```
retweets = []
for tweet in search_tweets:
    if 'retweeted_status' in tweet:
        retweets.append(tweet)
len(retweets)
```


# ## Izrada DiGrafa


# Prikazat ćemo tweetove na ovom popisu retweetova u smjeru protoka informacija: od korisnika koji je retweetao do retweetara, korisnika čija je objava retweetana. Budući da korisnik može retweetati objave

drugog korisnika više puta, želimo da ovaj graf bude težinski, s brojem retweeta kao težinom - brojimo koliko je puta neki korisnik retweetao objave nekog drugog korisnika.

# In[18]:

```
import networkx as nx

D = nx.DiGraph() #inicijalizacija usmjerenog grafa

for retweet in retweets:
    retweeted_status = retweet['retweeted_status']
    retweeted_sn = retweeted_status['user']['screen_name'] #ime korisnika koji je retweetao
    retweeter_sn = retweet['user']['screen_name'] #ime ciji je tweet retweetan

    if D.has_edge(retweeted_sn, retweeter_sn):
        D.edges[retweeted_sn, retweeter_sn]['weight'] += 1
    else:
        D.add_edge(retweeted_sn, retweeter_sn, weight=1)
```

# In[19]:

```
D.edges
```

# Logika dodavanja bridova je povećati težinu brida za 1 ako brid postoji ili stvoriti brid s težinom 1 ako ne postoji.

#

# Kada pišete kod kao što je ovaj koji se više puta referira na isti usmjereni brid, pazite da budete u skladu sa smjerom brida.

# ## Analiza grafa

# ### Najviše retweetani korisnik

# Budući da su bridovi u smjeru protoka informacija, out-degree nam daje broj korisnika koji retweetaju određenog korisnika. Možemo dobiti korisnika s najvišim stupnjem izlaza pomoću ugrađene max funkcije (korisnika čije se objave najviše retweetaju):

# In[20]:

```
max(D.nodes, key=D.out_degree)
```

# ali možemo dobiti i više informacija za "najboljih N" korisnika:

# In[21]:

```
from operator import itemgetter

sorted(D.out_degree(), key=itemgetter(1), reverse=True)[:5]
```

# U ovom dijelu koda koristimo činjenicu da D.out_degree() vraća niz (ime, stupanj) tuplova; key=itemgetter(1) govori sortiranoj funkciji da sortira ove tuplove prema njihovoj vrijednosti na indeksu 1. Postavljanje reverse=True govori sortiranoj funkciji da to želimo u silaznom redoslijedu, a [:5] daje nam prvih 5 stavki s rezultirajuće liste.

#

# Međutim, ovo je težinski graf! Prema zadanim postavkama, out_degree() zanemaruje težine rubova. Možemo dobiti izlaznu težinu tako da kažemo funkciji out_degree() da uzme u obzir težinu bridova:

# In[22]:

```
sorted(D.out_degree(weight='weight'), key=itemgetter(1), reverse=True)[:5]
```

# U nekim će slučajevima ova dva rezultata biti ista, npr. ako niti jednog od ovih korisnika nije više puta retweetao isti korisnik. Ovisno o vašem slučaju upotrebe, možete ili ne morate uzeti težine u obzir.

# ### Detekcija anomalija

# Jedna vrsta manipulacije društvenih medija uključuje račune koji stvaraju vrlo malo originalnog sadržaja, umjesto toga "spammaju" retweetove svih sadržaja u određenom razgovoru. To su potencijalno korisnici koji puno više retweetaju od ostalih. Možemo li otkriti da neki korisnici znatno više retweetaju od ostalih? Pogledajmo N korisnika koji najčešće retweetaju:

# In[23]:

```
sorted(D.in_degree(weight='weight'), key=itemgetter(1), reverse=True)[:5]
```

# ### Povezanost

# Možemo se pitati predstavljaju li tweetovi jedan veliki razgovor ili mnogo malih razgovora; općenito govoreći, svaka slabo povezana komponenta predstavlja razgovor.


# In[24]:


nx.is_weakly_connected(D)


# Tweetovi definitivno ne predstavljaju jedan veliki razgovor, no ono što možemo očekivati je da postoji velik broj malih razgovora. Pogledajmo koliko:


# In[25]:


nx.number_weakly_connected_components(D)


# ### Crtanje grafa

# Možemo pokušati nacrtati ovaj graf s čvorovima veličine prema njihovoj izlaznoj snazi:


# In[26]:


node_sizes = [D.out_degree(n, weight='weight') * 50 for n in D.nodes] # množimo s 50 da bi nam čvorovi na slici izgledali veće


get_ipython().run_line_magic('matplotlib', 'inline')
nx.draw(D, node_size=node_sizes)

# Imajte na umu da u ovom pojednostavljenom crtežu čvorovi s nultom vanjskom težinom nisu nacrtani na grafu jer je njihova veličina 0. To nam odgovara; ovdje su izvučeni samo korisnici koji su retweetani, ne i oni čije objave nitko nikad nije retweetao.

# # Twitter spominjanja

# Druga Twitter interakcija između korisnika događa se kada jedan korisnik spomene drugog u tweetu pod svojim @screen_name. Kao primjer, razmotrite sljedeći hipotetski tweet od @osome_iu:

#

# "Check out the new @IUSICE and @USC_ISI research https://..."

#

# Od ovog tweeta stvorili bismo dva brida:

#

# ('osome_iu', 'IUSICE')

# ('osome_iu', 'USC_ISI')

#

# Na nama je u kojem ćemo smjeru povući te rubove, ali moramo biti dosljedni. U ovom primjeru nacrtat ćemo rubove u smjeru toka pozornosti: @osome_iu posvećuje pozornost @IUSICE i @USC_ISI.

# ## Izrada DiGrafa

# Kao što smo na početku spomenuli, svaki tweet predstavljen je značajkom Tweet Object i svaki tweet ima Entitete (https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model/entities#entitiesobject) koji uvijek sadržavaju popis 'user_mentions' pa čak i kad je taj popis prazan. Zbog toga nije potrebno filtrirati tweetove koji sadrže spominjanja.

# In[ ]:

```python
import networkx as nx

D = nx.DiGraph()

for tweet in search_tweets:
    tweet_sn = tweet['user']['screen_name']
    for user_mention in tweet['entities']['user_mentions']:
        mentioned_sn = user_mention['screen_name']


        edge = (tweet_sn, mentioned_sn)
        if D.has_edge(*edge):
            D.edges[edge]['weight'] += 1
        else:
            D.add_edge(*edge, weight=1)


D.edges
```

# ## Analiza grafa

# ### Najpopularniji korisnici

# Budući da su ti bridovi u smjeru protoka pažnje, in-degree nam daje broj drugih korisnika koji spominju određenog korisnika. Možemo dobiti korisnika s najvišim stupnjem pomoću ugrađene max funkcije - korisnika koji se najčešće spominje od strane drugih:

# In[ ]:

```
max(D.nodes, key=D.in_degree)
```

# ali opet možemo dobiti i više informacija za "najboljih N" korisnika - korisnika koji se najčešće spominju:

# In[ ]:

```
from operator import itemgetter

sorted(D.in_degree(), key=itemgetter(1), reverse=True)[:5]
```

# Korištenjem weight='weight' možemo dobiti prvih 5 korisnika prema ulaznoj težini umjesto prema ulaznom stupnju:

# In[ ]:

```
sorted(D.in_degree(weight='weight'), key=itemgetter(1), reverse=True)[:5]
```

# U nekim će slučajevima ova dva rezultata biti ista,npr. ako nijednog od ovih korisnika nije više puta spomenuo isti korisnik. Ovisno o vašem slučaju upotrebe, možete ili ne morate uzeti težine u obzir.

# ### Pokretaći razgovora - Conversation drivers

# Korisnik koji spominje mnoge druge u razgovoru možda "pokreće" razgovor i pokušava uključiti druge u dijalog. Takav korisnik može biti i spam. Da vidimo tko ovdje najviše spominje - ovdje gledamo vrijednost out degree:

# In[ ]:

```python
sorted(D.out_degree(weight='weight'), key=itemgetter(1), reverse=True)[:5]
```

# ### Povezanost

# Možemo pitati predstavljaju li tweetovi dobiveni pretraživanjem jedan veliki razgovor ili mnogo malih razgovora; općenito govoreći, svaka slabo povezana komponenta predstavlja razgovor.

# In[ ]:

```python
nx.is_weakly_connected(D)
```

# In[ ]:

```python
nx.number_weakly_connected_components(D)
```

# ### Crtanje grafa

# In[ ]:

```
node_sizes= [D.in_degree(n, weight='weight') * 20 for n in D.nodes]
```

```
get_ipython().run_line_magic('matplotlib', 'inline')
```

```
nx.draw(D, node_size=node_sizes)
```

# Kao i u prethodnom primjeru, u ovom pojednostavljenom crtežu čvorovi s nultom vanjskom težinom nisu nacrtani na grafu jer je njihova veličina 0. To nam odgovara; ovdje su izvučeni samo korisnici koji su bili nekad spomenuti od strane drugih korisnika.

## ## 2.2. PAGE RANK ALGORITAM

# PageRank je algoritam za izračunavanje mjere centralnosti koja ima za cilj uhvatiti važnost svakog čvora. Obično se koristi u usmjerenim grafovima (mrežama). Kada se primijeni na webu, algoritam svakoj stranici dodjeljuje PageRank vrijednost. Algoritam za rangiranje tražilice tada može koristiti ovu vrijednost, u kombinaciji s mnogim drugim čimbenicima — kao što je podudaranje između upita i teksta stranice — za sortiranje rezultata upita. Stranica s visokim PageRank-om smatra se važnom, a algoritam za rangiranje joj daje prednost: ako su ostale stvari iste, stranice s većim PageRank-om rangirane su više.

# Učitat ćemo novi dataset kao DiGraf: math Wikipedia dataset:

# In[ ]:

```
D = nx.read_graphml('enwiki_math.graphml')
```

# In[ ]:

```
len(D) # broj čvorova -> isto kao da pise len(D.nodes)
```

# In[ ]:

```python
sorted(D.degree, key=lambda x: x[1], reverse=True)[:5]
```

# Nad učitanim datasetom pokrenut ćemo PageRank algoritam i izračunati PageRank za svaki od članaka:

# In[ ]:

```python
pr = nx.pagerank(D, alpha=0.85)
```

# Zanima nas kojih je top 10 članaka po izračunatom PageRank-u:

# In[ ]:

```python
sorted(pr, key=itemgetter(1), reverse=True)[:10]
```

# Želimo usporediti top 10 članaka po PageRanku s top 10 članaka po in degree-u. Hoće li to biti isti članci?

# In[ ]:

```
sorted(D.in_degree(weight='weight'), key=itemgetter(1), reverse=True)[:10]
```

# Distribucija PageRanka je prilično slična distribuciji in-degree-a na webu. Zašto onda jednostavno ne upotrijebite in-degree za rangiranje? Moramo uzeti u obzir da nisu sve staze jednake. Putevi sa stranica koje se često posjećuju daju veći doprinos. Drugim riječima, na važnost stranice utječe važnost stranica koje povezuju na nju.

# In[ ]:

# Auditorne 3

#!/usr/bin/env python
# coding: utf-8

# # 2.1. USMJERENE MREŽE SOCIJALNIH INTERAKCIJA

# U drugom djelu ove pokazne vježbe koristimo znanstvene Twitter podatke za stvaranje i istraživanje usmjerenih mreža društvenih interakcija.

# ## DATASET:

# In[69]:

```
import json
search_tweets = json.load(open('science_tweets.json'))
```

# Svaki tweet zapravo je jedna instanca Tweet objekta (https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model/tweet)

# In[70]:

```
search_tweets[:2]
```

# # Twitter retweetanje

# Temeljna interakcija u ekosustavu Twittera je "retweet" -- ponovno emitiranje tweeta drugog korisnika vašim pratiteljima.

# ## Filtriranje retweetova

# U našem skupu podataka nalaze se retweetovi. Objekt tweeta koji se nalazi u našem datasetu je retweet ako uključuje 'retweeted_status'. Napravit ćemo novi skup podataka koji će se sastojati samo od retweetova:

# In[71]:

```
retweets = []
for tweet in search_tweets:
    if 'retweeted_status' in tweet:
        retweets.append(tweet)
len(retweets)
```

# ## Izrada DiGrafa

# Prikazat ćemo tweetove na ovom popisu retweetova u smjeru protoka informacija: od korisnika koji je retweetao do retweetara, korisnika čija je objava retweetana. Budući da korisnik može retweetati objave drugog korisnika više puta, želimo da ovaj graf bude težinski, s brojem retweeta kao težinom - brojimo koliko je puta neki korisnik retweetao objave nekog drugog korisnika.

# In[72]:

```python
import networkx as nx

D = nx.DiGraph() #inicijalizacija usmjerenog grafa

for retweet in retweets:
    retweeted_status = retweet['retweeted_status']
    retweeted_sn = retweeted_status['user']['screen_name'] #ime korisnika koji je retweetao
    retweeter_sn = retweet['user']['screen_name'] #ime ciji je tweet retweetan

    if D.has_edge(retweeted_sn, retweeter_sn):
        D.edges[retweeted_sn, retweeter_sn]['weight'] += 1
    else:
        D.add_edge(retweeted_sn, retweeter_sn, weight=1)
```

# In[73]:

```python
D.edges
```

# Logika dodavanja bridova je povećati težinu brida za 1 ako brid postoji ili stvoriti brid s težinom 1 ako ne postoji.

#

# Kada pišete kod kao što je ovaj koji se više puta referira na isti usmjereni brid, pazite da budete u skladu sa smjerom brida.


# ## Analiza grafa


# ### Najviše retweetani korisnik


# Budući da su bridovi u smjeru protoka informacija, out-degree nam daje broj korisnika koji retweetaju određenog korisnika. Možemo dobiti korisnika s najvišim stupnjem izlaza pomoću ugrađene max funkcije (korisnika čije se objave najviše retweetaju):


# In[74]:


```
max(D.nodes, key=D.out_degree)
```


# ali možemo dobiti i više informacija za "najboljih N" korisnika:


# In[75]:


```
from operator import itemgetter

sorted(D.out_degree(), key=itemgetter(1), reverse=True)[:5]
```

# U ovom dijelu koda koristimo činjenicu da D.out_degree() vraća niz (ime, stupanj) tuplova; key=itemgetter(1) govori sortiranoj funkciji da sortira ove tuplove prema njihovoj vrijednosti na indeksu 1. Postavljanje reverse=True govori sortiranoj funkciji da to želimo u silaznom redoslijedu, a [:5] daje nam prvih 5 stavki s rezultirajuće liste.

#

# Međutim, ovo je težinski graf! Prema zadanim postavkama, out_degree() zanemaruje težine rubova. Možemo dobiti izlaznu težinu tako da kažemo funkciji out_degree() da uzme u obzir težinu bridova:

# In[76]:

```
sorted(D.out_degree(weight='weight'), key=itemgetter(1), reverse=True)[:5]
```

# U nekim će slučajevima ova dva rezultata biti ista, npr. ako niti jednog od ovih korisnika nije više puta retweetao isti korisnik. Ovisno o vašem slučaju upotrebe, možete ili ne morate uzeti težine u obzir.

# ### Detekcija anomalija

# Jedna vrsta manipulacije društvenih medija uključuje račune koji stvaraju vrlo malo originalnog sadržaja, umjesto toga "spammaju" retweetove svih sadržaja u određenom razgovoru. To su potencijalno korisnici koji puno više retweetaju od ostalih. Možemo li otkriti da neki korisnici znatno više retweetaju od ostalih? Pogledajmo N korisnika koji najčešće retweetaju:

# In[77]:

```
sorted(D.in_degree(weight='weight'), key=itemgetter(1), reverse=True)[:5]
```

# ### Povezanost

# Možemo se pitati predstavljaju li tweetovi jedan veliki razgovor ili mnogo malih razgovora; općenito govoreći, svaka slabo povezana komponenta predstavlja razgovor.

# In[78]:

```python
nx.is_weakly_connected(D)
```

# Tweetovi definitivno ne predstavljaju jedan veliki razgovor, no ono što možemo očekivati je da postoji velik broj malih razgovora. Pogledajmo koliko:

# In[79]:

```python
nx.number_weakly_connected_components(D)
```

# ### Crtanje grafa

# Možemo pokušati nacrtati ovaj graf s čvorovima veličine prema njihovoj izlaznoj snazi:

# In[80]:

```python
node_sizes = [D.out_degree(n, weight='weight') * 50 for n in D.nodes] # množimo s 50 da bi nam čvorovi na slici izgledali veće
```

```
get_ipython().run_line_magic('matplotlib', 'inline')

nx.draw(D, node_size=node_sizes)
```

# Imajte na umu da u ovom pojednostavljenom crtežu čvorovi s nultom vanjskom težinom nisu nacrtani na grafu jer je njihova veličina 0. To nam odgovara; ovdje su izvučeni samo korisnici koji su retweetani, ne i oni čije objave nitko nikad nije retweetao.

# # Twitter spominjanja

# Druga Twitter interakcija između korisnika događa se kada jedan korisnik spomene drugog u tweetu pod svojim @screen_name. Kao primjer, razmotrite sljedeći hipotetski tweet od @osome_iu:

#

# "Check out the new @IUSICE and @USC_ISI research https://..."

#

# Od ovog tweeta stvorili bismo dva brida:

#

# ('osome_iu', 'IUSICE')

# ('osome_iu', 'USC_ISI')

#

# Na nama je u kojem ćemo smjeru povući te rubove, ali moramo biti dosljedni. U ovom primjeru nacrtat ćemo rubove u smjeru toka pozornosti: @osome_iu posvećuje pozornost @IUSICE i @USC_ISI.

# ## Izrada DiGrafa

# Kao što smo na početku spomenuli, svaki tweet predstavljen je značajkom Tweet Object i svaki tweet ima Entitete (https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model/entities#entitiesobject) koji uvijek sadržavaju popis 'user_mentions' pa čak i kad je taj popis prazan. Zbog toga nije potrebno filtrirati tweetove koji sadrže spominjanja.

# In[81]:

```python
import networkx as nx

D = nx.DiGraph()

for tweet in search_tweets:
    tweet_sn = tweet['user']['screen_name']
    for user_mention in tweet['entities']['user_mentions']:
        mentioned_sn = user_mention['screen_name']

        edge = (tweet_sn, mentioned_sn)
        if D.has_edge(*edge):
            D.edges[edge]['weight'] += 1
        else:
            D.add_edge(*edge, weight=1)

D.edges
```

# ## Analiza grafa

# ### Najpopularniji korisnici

# Budući da su ti bridovi u smjeru protoka pažnje, in-degree nam daje broj drugih korisnika koji spominju određenog korisnika. Možemo dobiti korisnika s najvišim stupnjem pomoću ugrađene max funkcije - korisnika koji se najčešće spominje od strane drugih:

# In[82]:

```python
max(D.nodes, key=D.in_degree)
```

# ali opet možemo dobiti i više informacija za "najboljih N" korisnika - korisnika koji se najčešće spominju:

# In[83]:

```python
from operator import itemgetter

sorted(D.in_degree(), key=itemgetter(1), reverse=True)[:5]
```

# Korištenjem weight='weight' možemo dobiti prvih 5 korisnika prema ulaznoj težini umjesto prema ulaznom stupnju:

# In[84]:

```python
sorted(D.in_degree(weight='weight'), key=itemgetter(1), reverse=True)[:5]
```

# U nekim će slučajevima ova dva rezultata biti ista,npr. ako nijednog od ovih korisnika nije više puta spomenuo isti korisnik. Ovisno o vašem slučaju upotrebe, možete ili ne morate uzeti težine u obzir.

# ### Pokretaći razgovora - Conversation drivers

# Korisnik koji spominje mnoge druge u razgovoru možda "pokreće" razgovor i pokušava uključiti druge u dijalog. Takav korisnik može biti i spam. Da vidimo tko ovdje najviše spominje - ovdje gledamo vrijednost out degree:


# In[85]:


```
sorted(D.out_degree(weight='weight'), key=itemgetter(1), reverse=True)[:5]
```


# ### Povezanost


# Možemo pitati predstavljaju li tweetovi dobiveni pretraživanjem jedan veliki razgovor ili mnogo malih razgovora; općenito govoreći, svaka slabo povezana komponenta predstavlja razgovor.


# In[86]:


```
nx.is_weakly_connected(D)
```


# In[87]:


```
nx.number_weakly_connected_components(D)
```


# ### Crtanje grafa


# In[88]:

```
node_sizes= [D.in_degree(n, weight='weight') * 20 for n in D.nodes]
```

```
get_ipython().run_line_magic('matplotlib', 'inline')
```

```
nx.draw(D, node_size=node_sizes)
```

# Kao i u prethodnom primjeru, u ovom pojednostavljenom crtežu čvorovi s nultom vanjskom težinom nisu nacrtani na grafu jer je njihova veličina 0. To nam odgovara; ovdje su izvučeni samo korisnici koji su bili nekad spomenuti od strane drugih korisnika.

## # 2.2. PAGE RANK ALGORITAM

# PageRank je algoritam za izračunavanje mjere centralnosti koja ima za cilj uhvatiti važnost svakog čvora. Obično se koristi u usmjerenim grafovima (mrežama). Kada se primijeni na webu, algoritam svakoj stranici dodjeljuje PageRank vrijednost. Algoritam za rangiranje tražilice tada može koristiti ovu vrijednost, u kombinaciji s mnogim drugim čimbenicima — kao što je podudaranje između upita i teksta stranice — za sortiranje rezultata upita. Stranica s visokim PageRank-om smatra se važnom, a algoritam za rangiranje joj daje prednost: ako su ostale stvari iste, stranice s većim PageRank-om rangirane su više.

# Učitat ćemo novi dataset kao DiGraf: math Wikipedia dataset:

# In[89]:

```
D = nx.read_graphml('enwiki_math.graphml')
```

# In[90]:

```
len(D) # broj čvorova -> isto kao da pise len(D.nodes)
```

```
# In[91]:
```

```
sorted(D.degree, key=lambda x: x[1], reverse=True)[:5]
```

```
# Nad učitanim datasetom pokrenut ćemo PageRank algoritam i izračunati PageRank za svaki od članaka:
```

```
# In[92]:
```

```
pr = nx.pagerank(D, alpha=0.85)
```

```
# Zanima nas kojih je top 10 članaka po izračunatom PageRank-u:
```

```
# In[93]:
```

```
sorted(pr, key=itemgetter(1), reverse=True)[:10]
```

```
# Želimo usporediti top 10 članaka po PageRanku s top 10 članaka po in degree-u. Hoće li to biti isti
članci?
```

# In[94]:

```python
sorted(D.in_degree(weight='weight'), key=itemgetter(1), reverse=True)[:10]
```

# Distribucija PageRanka je prilično slična distribuciji in-degree-a na webu. Zašto onda jednostavno ne upotrijebite in-degree za rangiranje? Moramo uzeti u obzir da nisu sve staze jednake. Putevi sa stranica koje se često posjećuju daju veći doprinos. Drugim riječima, na važnost stranice utječe važnost stranica koje povezuju na nju.

# In[94]:

# Auditorne 4

```python
#!/usr/bin/env python
# coding: utf-8
```

# # LAB 4 - complex networks FER3 (community edition)

# In[43]:

```python
import networkx as nx
import random

get_ipython().run_line_magic('matplotlib', 'inline')
```

# ## 3.1 Synthetic example analysis (with modularity)

```
# In[44]:
```

```python
G = nx.barbell_graph(5, 0)
```

```
# In[45]:
```

```python
nx.draw(G, with_labels=True)
```

```
# In[46]:
```

```python
test_graph = nx.Graph()
```

```
# In[47]:
```

```python
nx.add_cycle(test_graph, [0, 1, 2, 3, 4, 5])
nx.add_cycle(test_graph, [6, 7, 8, 9])
nx.add_cycle(test_graph, [10, 11, 12, 13])
nx.add_cycle(test_graph, [14, 15, 16, 17])
```

```
# In[48]:
```

```
test_graph.add_edge(0, 7)
```

```
test_graph.add_edge(0, 11)
```

```
test_graph.add_edge(0, 14)
```

# In[49]:

```
test_graph.add_edge(13, 14)
```

# In[50]:

```
nx.draw(test_graph, with_labels=True)
```

# In[51]:

```
partition = [{0, 1, 2, 3, 4, 5}, {6, 7, 8, 9}, {10, 11, 12, 13}, {14, 15, 16, 17}]
```

# In[52]:

```
nx.community.is_partition(test_graph, partition)
```

# In[53]:

```python
partition_map = {}
for idx, cluster_nodes in enumerate(partition):
    for node in cluster_nodes:
        partition_map[node] = idx

partition_map
```

# In[54]:

```python
partition_map[0] == partition_map[15]
```

# In[55]:

```python
partition_map[0] == partition_map[1]
```

# In[56]:

```python
node_colors = [partition_map[n] for n in test_graph.nodes]
```

# In[57]:

```python
nx.draw(test_graph, node_color=node_colors, with_labels=True)
```

```python
# goal:  finding a partition that achieves good separation between the groups of nodes
#
# in general: get a partition on some objective function
#
# more in general: get some information using the connections/weights between nodes (distance) - adjacency matrix
#
# 
link:https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.community.quality.modularity.html
#
# Modularity is the fraction of the edges that fall within the given groups minus the expected fraction if edges were distributed at random. The value of the modularity for unweighted and undirected graphs lies in the range. It is positive if the number of edges within groups exceeds the number expected on the basis of chance.
```

# In[58]:

```python
partition_example =  [{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {10, 11, 12, 13, 14, 15, 16, 17}]
```

# In[59]:

```python
nx.community.quality.modularity(test_graph, partition_example)
```

# In[60]:

```python
partition_example2 =  [{0, 1, 2}, {3, 4, 5}, {6, 7, 8, 9}, {10, 11, 12, 13}, {14, 15, 16, 17}]
```

# In[61]:

```python
nx.community.quality.modularity(test_graph, partition_example2)
```

# In[62]:

```python
nx.community.quality.modularity(test_graph, partition)
```

# a random network does not have community structure -  modularity concept introduces a test to analyse fraction of the edges that fall within the given groups minus the expected fraction if edges were distributed at random.
#
# Analysed at the total network level goal is to maximize modularity score

```
# In[63]:
```

```python
random_alocation = random.sample(test_graph.nodes, 9)
random_alocation
```

```
# In[ ]:
```

```python
partition_rnd = [set(random_alocation), set(test_graph.nodes) - set(random_alocation)]
partition_rnd
```

```python
# how dose the score look on a random alocation vector - should be close to 0
```

```
# In[ ]:
```

```python
nx.community.quality.modularity(test_graph, partition_rnd)
```

```
# In[ ]:
```

```python
#visualize
random_node_color_map = ['red' if n in random_alocation else 'lightblue' for n in test_graph.nodes]
nx.draw(test_graph, node_color=random_node_color_map)
```

# ## 3.2 Karate Club example we can not avoid

# paper "An Information Flow Model for Conflict and Fission in Small Groups" by Wayne W. Zachary

#

#

# wiki: A social network of a karate club was studied by Wayne W. Zachary for a period of three years from 1970 to 1972.[2] The network captures 34 members of a karate club, documenting links between pairs of members who interacted outside the club.

#

# During the study a conflict arose between the administrator "John A" and instructor "Mr. Hi" (pseudonyms), which led to the split of the club into two. Half of the members formed a new club around Mr. Hi; members from the other part found a new instructor or gave up karate. Based on collected data Zachary correctly assigned all but one member of the club to the groups they actually joined after the split.

# In[ ]:

karate_graph = nx.karate_club_graph()

# In[ ]:

nx.draw(karate_graph, with_labels=True)

# In[ ]:

# there are properties

```python
karate_graph.nodes[0]
```

# In[ ]:

```python
karate_graph.nodes[1]
```

# In[ ]:

```python
karate_graph.nodes[2]
```

# In[ ]:

```python
karate_graph.nodes[20]
```

# In[ ]:

```python
#fancy collors
club_color = {'Mr. Hi': 'gray','Officer': 'red'}
node_colors = [club_color[karate_graph.nodes[n]['club']] for n in karate_graph.nodes]

nx.draw(karate_graph, node_color=node_colors, with_labels=True)
```

```python
# In[ ]:


groups = { 'Mr. Hi': set(), 'Officer': set()}


for n in karate_graph.nodes:
    club = karate_graph.nodes[n]['club']
    groups[club].add(n)


data_partition = list(groups.values())
data_partition


# In[ ]:


nx.community.is_partition(karate_graph, data_partition)


# In[ ]:


nx.community.quality.modularity(karate_graph, data_partition)


# can we automate the process? Find communities in G using greedy modularity maximization.
#
```

#

# paper: Clauset, A., Newman, M. E., & Moore, C. "Finding community structure in very large networks." Physical Review E 70(6), 2004.

#

# Greedy modularity maximization begins with each node in its own community and repeatedly joins the pair of communities that lead to the largest modularity until no futher increase in modularity is possible (a maximum).

# In[ ]:

```
from networkx.algorithms.community import greedy_modularity_communities
network_set = greedy_modularity_communities(karate_graph)
network_set
```

# In[ ]:

```
y = [list(x) for x in network_set]
y
```

# In[ ]:

```
nx.community.quality.modularity(karate_graph, y)
```

# In[ ]:

```python
partition_map = {}
for idx, cluster_nodes in enumerate(y):

    for node in cluster_nodes:

        partition_map[node] = idx


partition_map
```

# In[ ]:

```python
node_colors = [partition_map[n] for n in karate_graph.nodes]
nx.draw(karate_graph, node_color=node_colors, with_labels=True)
```

```python
# so the results find a subcommunity in mr. Hi that does not mingle with the 'traitors'. this is the 3rd
group
```

```python
# ## 3.3. k-clique communities  add-on
```

```python
# A k-clique community is the union of all cliques of size k that can be reached through adjacent (sharing
k-1 nodes) k-cliques.
#
# paper: Gergely Palla, Imre Derényi, Illés Farkas1, and Tamás Vicsek, Uncovering the overlapping
community structure of complex networks in nature and society Nature 435, 814-818, 2005,
doi:10.1038/nature03607
#
# Clique - subsets of vertices, all adjacent to each other, also called complete subgraphs
```

# In[ ]:

```python
sum(1 for c in nx.find_cliques(karate_graph))  # The number of maximal cliques in Karate graph
```

# In[ ]:

```python
max(nx.find_cliques(karate_graph), key=len)  # The largest maximal clique in Karate graph
```

# In[ ]:

```python
from networkx.algorithms.community import k_clique_communities
c = list(k_clique_communities(karate_graph, 4))
c
```

```python
# what do we get?
```

```python
# ## 3.4 Game of Thrones example (with Girvan-Newman clustering)
#
# These networks were created by connecting two characters whenever their names (or nicknames)
# appeared within 15 words of one another in one of the books in "A Song of Ice and Fire." The edge
# weight corresponds to the number of interactions.
#
```

```
# data at: https://github.com/mathbeveridge/asoiaf
```

```
# In[ ]:
```

```
import pandas as pd
GOT_books = pd.read_csv('C:/Users/demij/asoiaf-all-edges.csv')
```

```
# In[ ]:
```

```
GOT_books.head()
```

```
# In[ ]:
```

```
GOT_graph = nx.Graph()
```

```
# In[ ]:
```

```
for row in GOT_books.iterrows(): GOT_graph.add_edge(row[1]['Source'], row[1]['Target'],
weight=row[1]['weight'])
```

```
# In[ ]:
```

```python
list(GOT_graph.edges(data=True))[0]
```

# In[ ]:

```python
list(GOT_graph.edges(data=True))[100]
```

# ### Node characteristics

# In[ ]:

```python
GOT_graph.number_of_nodes()
```

# In[ ]:

```python
# get that degree stat - fraction of nodes it is connected to
degree_stats = nx.degree_centrality(GOT_graph)
```
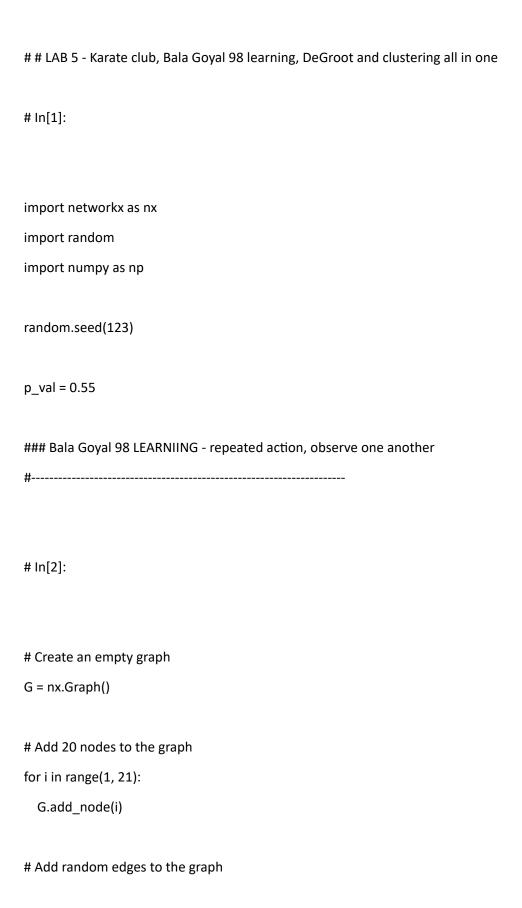
# In[ ]:

degree_stats

# In[ ]:

# get that degree stat - those most importaint plot characters
sorted(degree_stats.items(), key=lambda x:x[1], reverse=True)[0:10]

# In[ ]:

## betweensess centrality
betweenness_centrality_stats = nx.betweenness_centrality(GOT_graph)

# In[ ]:

# get that betweenness_centrality - those most important plot twisters characters - close to the action
sorted(betweenness_centrality_stats.items(), key=lambda x:x[1], reverse=True)[0:10]

# In[ ]:

degree_value = GOT_graph.degree()

# In[ ]:

```python
import numpy as np
np.mean([d for _, d in GOT_graph.degree()])
```

# In[ ]:

```python
#lets remove some marginal nodes to go easy on ourselves
remove_val = [node for node,degree in dict(GOT_graph.degree()).items() if degree < 2*7]
remove_val
```

# In[ ]:

```python
GOT_graph.remove_nodes_from(remove_val)
```

# In[ ]:

```python
GOT_graph.number_of_nodes()
```

# In[ ]:

```python
degree_stats2 = nx.degree_centrality(GOT_graph)

degree_stats2
```

# In[ ]:

```python
betweenness_centrality_stats2 = nx.betweenness_centrality(GOT_graph)

betweenness_centrality_stats2
```

# The Girvan-Newman algorithm for the detection and analysis of community structure relies on the iterative elimination of edges that have the highest number of shortest paths between nodes passing through them. By removing edges from the graph one-by-one, the network breaks down into smaller pieces, so-called communities. The algorithm was introduced by Michelle Girvan and Mark Newman.
#
# https://networkx.guide/algorithms/community-detection/girvan-newman/

# In[ ]:

```python
nx.draw(GOT_graph, with_labels=True)
```

# so if we remove the connection with the highest betweenness centrality it would be the central one leaving us wih the 2 groups
# note that this is an ideal case: groups are fully connected
#

```
# The Girvan-Newman algorithm can be divided into four main steps:

#

# 1. For every edge in a graph, calculate the edge betweenness centrality.

# 2. Remove the edge with the highest betweenness centrality.

# 3. Calculate the betweenness centrality for every remaining edge.

# 4. Repeat steps 2-4 until there are no more edges left.

#

# https://networkx.guide/algorithms/community-detection/girvan-newman/

#

# At the end - Evaluate each partition in the sequence and choose the one with the highest modularity


# In[ ]:



import matplotlib.pyplot as plt



# In[ ]:



# draw the subgraph using the spring layout
pos = nx.spring_layout(GOT_graph)


# color the nodes based on degree centrality
node_colors = [degree_stats2[node] for node in GOT_graph]
nx.draw(GOT_graph, pos, node_color=node_colors, cmap=plt.cm.Reds)


# show the plot
plt.savefig("ref_plot.png",dpi=500)
```

```
plt.show()
```

# In[ ]:

```
from networkx.algorithms.community.centrality import girvan_newman

partition_girvan_newman = girvan_newman(GOT_graph)
list(partition_girvan_newman)
```

# conclusion: central to plots and interactions in those plots

# In[ ]:

```
# k-cluique with a high k
from networkx.algorithms.community import k_clique_communities
c_got = list(k_clique_communities(GOT_graph, 10))
c_got
```

# conclusion - main plot (across all 5 books) and a big side plot (Night's Watch)

## Auditorne 5

```
#!/usr/bin/env python
# coding: utf-8
```

## # LAB 5 - Karate club, Bala Goyal 98 learning, DeGroot and clustering all in one

# In[1]:

```python
import networkx as nx
import random
import numpy as np

random.seed(123)

p_val = 0.55
```

### Bala Goyal 98 LEARNIING - repeated action, observe one another
```
#------------------------------------------------------------------
```

# In[2]:

```python
# Create an empty graph
G = nx.Graph()

# Add 20 nodes to the graph
for i in range(1, 21):
    G.add_node(i)

# Add random edges to the graph
```

```
for i in range(1, 21):

    for j in range(i+1, 21):

        if random.random() < 0.5:

            G.add_edge(i, j)


import matplotlib.pyplot as plt

nx.draw(G, with_labels = True)

plt.show()
```

# In[3]:

```
# Add random state "A" or "B" to each node

for node in G.nodes():

    if random.random() < 0.5:

        G.nodes[node]["state"] = "A"

        G.nodes[node]["value"] = 1

    else:

        G.nodes[node]["state"] = "B"

        G.nodes[node]["value"] = 2 if random.random() < p_val else 0
```

# In[4]:

```
#printing the node state and value

for node in G.nodes():

    print(f'{node},{G.nodes[node]["state"]},{G.nodes[node]["value"]}')
```

# In[5]:

# Create a dictionary to map each state to a color
state_colors = {"A": "blue", "B": "red"}

# Draw the graph, coloring each node by its state
pos = nx.spring_layout(G)
nx.draw(G, pos, node_color=[state_colors[G.nodes[node]["state"]] for node in G.nodes()], with_labels = True)
plt.show()

# In[6]:

# Create a dictionary to map each value to a color
value_colors = {1: "blue", 2: "green",0:"red"}

# Draw the graph, coloring each node by its value
pos = nx.spring_layout(G)
nx.draw(G, pos, node_color=[value_colors[G.nodes[node]["value"]] for node in G.nodes()], with_labels = True)
plt.show()

# In[7]:

```python
#20 iterations for loop

for i in range(20):

    new_states = {}

    new_values = {}

    for node in G.nodes():

        # calculate average value of state A

        A_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if
G.nodes[neighbor]["state"] == "A"]

        if len(A_neighbors) == 0:

            A_average = 0

        else:

            A_average = sum(A_neighbors) / len(A_neighbors)


        # calculate average value of state B

        B_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if
G.nodes[neighbor]["state"] == "B"]

        if len(B_neighbors) == 0:

            B_average = 0

        else:

            B_average = sum(B_neighbors) / len(B_neighbors)


        # change state of node to the one of higher average value

        if A_average > B_average:

            new_states[node] = "A"

            new_values[node] = 1

        elif B_average > A_average:

            new_states[node] = "B"
```

```python
        new_values[node] = 2 if random.random() < p_val else 0

    for node in new_states:

      G.nodes[node]["state"] = new_states[node]

      G.nodes[node]["value"] = new_values[node]
```

# In[8]:

```python
#printing the node state and value

for node in G.nodes():

    print(f'{node},{G.nodes[node]["state"]},{G.nodes[node]["value"]}')

#-------------------------------------------------------
```

# In[9]:

```python
# Create a dictionary to map each state to a color

state_colors = {"A": "blue", "B": "red"}


# Draw the graph, coloring each node by its state

pos = nx.spring_layout(G)

nx.draw(G, pos, node_color=[state_colors[G.nodes[node]["state"]] for node in G.nodes()], with_labels = True)

plt.show()
```

# In[10]:

```python
# Create a dictionary to map each value to a color
value_colors = {1: "blue", 2: "green",0:"red"}


# Draw the graph, coloring each node by its value
pos = nx.spring_layout(G)
nx.draw(G, pos, node_color=[value_colors[G.nodes[node]["value"]] for node in G.nodes()], with_labels =
True)
plt.show()



# In[11]:



##### what is the issue - random network
G = nx.karate_club_graph()
random.seed(12345)


# Add random state "A" or "B" to each node
for node in G.nodes():
    if random.random() < 0.75:
        G.nodes[node]["state"] = "A"
        G.nodes[node]["value"] = 1
    else:
        G.nodes[node]["state"] = "B"
        G.nodes[node]["value"] = 2 if random.random() < p_val else 0
```

# In[12]:

```python
import matplotlib.pyplot as plt
nx.draw(G, with_labels = True)
plt.show()
```

# In[13]:

```python
# Create a dictionary to map each state to a color
state_colors = {"A": "blue", "B": "red"}


# Draw the graph, coloring each node by its state
pos = nx.spring_layout(G)
nx.draw(G, pos, node_color=[state_colors[G.nodes[node]["state"]] for node in G.nodes()], with_labels = True)
plt.show()
```

# In[14]:

```python
# Create a dictionary to map each value to a color
value_colors = {1: "blue", 2: "green",0:"red"}


# Draw the graph, coloring each node by its value
```

```python
nx.draw(G, pos, node_color=[value_colors[G.nodes[node]["value"]] for node in G.nodes()], with_labels =
True)
plt.show()
```

# In[15]:

```python
average_values = []
```

# In[16]:

```python
new_states = {}
new_values = {}
for node in G.nodes():
    # calculate average value of state A
    A_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if
G.nodes[neighbor]["state"] == "A"]
    if len(A_neighbors) == 0:
        A_average = 0
    else:
        A_average = sum(A_neighbors) / len(A_neighbors)


    # calculate average value of state B
    B_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if
G.nodes[neighbor]["state"] == "B"]
    if len(B_neighbors) == 0:
        B_average = 0
```

```python
        else:
            B_average = sum(B_neighbors) / len(B_neighbors)


        # change state of node to the one of higher average value
        if A_average > B_average:
            new_states[node] = "A"
            new_values[node] = 1
        elif B_average > A_average:
            new_states[node] = "B"
            new_values[node] = 2 if random.random() < 0.5 else 0
for node in new_states:
    G.nodes[node]["state"] = new_states[node]
    G.nodes[node]["value"] = new_values[node]
average_values.append(sum([G.nodes[node]["value"] for node in G.nodes()])/len(G.nodes()))


    # Create a dictionary to map each state to a color
state_colors = {"A": "blue", "B": "red"}


# Draw the graph, coloring each node by its state
nx.draw(G, pos, node_color=[state_colors[G.nodes[node]["state"]] for node in G.nodes()], with_labels =
True)
plt.show()



# In[17]:



new_states = {}
new_values = {}
```

```python
for node in G.nodes():

    # calculate average value of state A

    A_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if
G.nodes[neighbor]["state"] == "A"]

    if len(A_neighbors) == 0:

        A_average = 0

    else:

        A_average = sum(A_neighbors) / len(A_neighbors)


    # calculate average value of state B

    B_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if
G.nodes[neighbor]["state"] == "B"]

    if len(B_neighbors) == 0:

        B_average = 0

    else:

        B_average = sum(B_neighbors) / len(B_neighbors)


    # change state of node to the one of higher average value

    if A_average > B_average:

        new_states[node] = "A"

        new_values[node] = 1

    elif B_average > A_average:

        new_states[node] = "B"

        new_values[node] = 2 if random.random() < 0.5 else 0

for node in new_states:

    G.nodes[node]["state"] = new_states[node]

    G.nodes[node]["value"] = new_values[node]

average_values.append(sum([G.nodes[node]["value"] for node in G.nodes()])/len(G.nodes()))
```

```python
    # Create a dictionary to map each state to a color
state_colors = {"A": "blue", "B": "red"}


# Draw the graph, coloring each node by its state
nx.draw(G, pos, node_color=[state_colors[G.nodes[node]["state"]] for node in G.nodes()], with_labels = True)
plt.show()




# In[18]:




new_states = {}
new_values = {}
for node in G.nodes():
    # calculate average value of state A
    A_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if G.nodes[neighbor]["state"] == "A"]
    if len(A_neighbors) == 0:
        A_average = 0
    else:
        A_average = sum(A_neighbors) / len(A_neighbors)


    # calculate average value of state B
    B_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if G.nodes[neighbor]["state"] == "B"]
    if len(B_neighbors) == 0:
        B_average = 0
    else:
        B_average = sum(B_neighbors) / len(B_neighbors)
```

```python
    # change state of node to the one of higher average value
    if A_average > B_average:
        new_states[node] = "A"
        new_values[node] = 1
    elif B_average > A_average:
        new_states[node] = "B"
        new_values[node] = 2 if random.random() < 0.5 else 0
for node in new_states:
    G.nodes[node]["state"] = new_states[node]
    G.nodes[node]["value"] = new_values[node]
average_values.append(sum([G.nodes[node]["value"] for node in G.nodes()])/len(G.nodes()))


# Create a dictionary to map each state to a color
state_colors = {"A": "blue", "B": "red"}


# Draw the graph, coloring each node by its state
nx.draw(G, pos, node_color=[state_colors[G.nodes[node]["state"]] for node in G.nodes()], with_labels =
True)
plt.show()



# In[19]:



new_states = {}
new_values = {}
for node in G.nodes():
    # calculate average value of state A
```

```python
    A_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if
G.nodes[neighbor]["state"] == "A"]

    if len(A_neighbors) == 0:

        A_average = 0

    else:

        A_average = sum(A_neighbors) / len(A_neighbors)


    # calculate average value of state B

    B_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if
G.nodes[neighbor]["state"] == "B"]

    if len(B_neighbors) == 0:

        B_average = 0

    else:

        B_average = sum(B_neighbors) / len(B_neighbors)


    # change state of node to the one of higher average value

    if A_average > B_average:

        new_states[node] = "A"

        new_values[node] = 1

    elif B_average > A_average:

        new_states[node] = "B"

        new_values[node] = 2 if random.random() < 0.5 else 0

for node in new_states:

    G.nodes[node]["state"] = new_states[node]

    G.nodes[node]["value"] = new_values[node]

average_values.append(sum([G.nodes[node]["value"] for node in G.nodes()])/len(G.nodes()))


    # Create a dictionary to map each state to a color

state_colors = {"A": "blue", "B": "red"}
```

```python
# Draw the graph, coloring each node by its state
nx.draw(G, pos, node_color=[state_colors[G.nodes[node]["state"]] for node in G.nodes()], with_labels = True)

plt.show()


# In[20]:


new_states = {}
new_values = {}
for node in G.nodes():
    # calculate average value of state A
    A_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if G.nodes[neighbor]["state"] == "A"]
    if len(A_neighbors) == 0:
        A_average = 0
    else:
        A_average = sum(A_neighbors) / len(A_neighbors)

    # calculate average value of state B
    B_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if G.nodes[neighbor]["state"] == "B"]
    if len(B_neighbors) == 0:
        B_average = 0
    else:
        B_average = sum(B_neighbors) / len(B_neighbors)

    # change state of node to the one of higher average value
```

```python
        if A_average > B_average:

            new_states[node] = "A"

            new_values[node] = 1

        elif B_average > A_average:

            new_states[node] = "B"

            new_values[node] = 2 if random.random() < 0.5 else 0

for node in new_states:

    G.nodes[node]["state"] = new_states[node]

    G.nodes[node]["value"] = new_values[node]

average_values.append(sum([G.nodes[node]["value"] for node in G.nodes()])/len(G.nodes()))


    # Create a dictionary to map each state to a color

state_colors = {"A": "blue", "B": "red"}


# Draw the graph, coloring each node by its state

nx.draw(G, pos, node_color=[state_colors[G.nodes[node]["state"]] for node in G.nodes()], with_labels =
True)

plt.show()




# In[21]:




new_states = {}

new_values = {}

for node in G.nodes():

    # calculate average value of state A

    A_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if
G.nodes[neighbor]["state"] == "A"]
```

```python
        if len(A_neighbors) == 0:

            A_average = 0

        else:

            A_average = sum(A_neighbors) / len(A_neighbors)


        # calculate average value of state B

        B_neighbors = [G.nodes[neighbor]["value"] for neighbor in G.neighbors(node) if
G.nodes[neighbor]["state"] == "B"]

        if len(B_neighbors) == 0:

            B_average = 0

        else:

            B_average = sum(B_neighbors) / len(B_neighbors)


        # change state of node to the one of higher average value

        if A_average > B_average:

            new_states[node] = "A"

            new_values[node] = 1

        elif B_average > A_average:

            new_states[node] = "B"

            new_values[node] = 2 if random.random() < 0.5 else 0

    for node in new_states:

        G.nodes[node]["state"] = new_states[node]

        G.nodes[node]["value"] = new_values[node]

    average_values.append(sum([G.nodes[node]["value"] for node in G.nodes()])/len(G.nodes()))


    # Create a dictionary to map each state to a color

    state_colors = {"A": "blue", "B": "red"}


# Draw the graph, coloring each node by its state
```

```python
nx.draw(G, pos, node_color=[state_colors[G.nodes[node]["state"]] for node in G.nodes()], with_labels =
True)

plt.show()
```

# In[22]:

```python
from sklearn.cluster import AgglomerativeClustering
```

# In[ ]:

```python
A = nx.floyd_warshall_numpy(G) # distance mx - istance[i, j] is the distance along a shortest path from i
to j
```

# In[ ]:

```python
A
```

# In[ ]:

```python
# Create an AgglomerativeClustering object
agg_clustering = AgglomerativeClustering(n_clusters=2, affinity = "precomputed",  linkage = 'complete')
```

```
# Fit the model to the data
agg_clustering.fit(A)


# Retrieve the labels for each node
cluster_labels = agg_clustering.labels_



# In[ ]:



# Define colors for the nodes
colors = ["gray" if cluster_labels[i] == 0 else "red" for i in range(len(cluster_labels))]


# Plot the graph with the colors
nx.draw(G, pos, node_color=colors, with_labels=True)
plt.show()



# In[ ]:



# Retrieve the labels for each node
# Get the original community labels
original_labels = nx.get_node_attributes(G,'club')


# Plot the graph with the colors
nx.draw(G, pos, node_color=colors, with_labels=False)
```

```python
# Add the original labels to the graph
nx.draw_networkx_labels(G, pos, original_labels)
plt.show()
```

```python
# In[ ]:
```

```python
propagating_state = nx.get_node_attributes(G,'state')
nx.draw(G, pos, node_color=colors, with_labels=False)
nx.draw_networkx_labels(G, pos, propagating_state)
plt.show()
```

```python
# In[ ]:
```

#### DeGroot learning

```python
import networkx as nx
import numpy as np

# Import the Karate Club graph
G = nx.karate_club_graph()

np.random.seed(123)
# Initialize the opinion vector with random values - probability of event X occuring
# Get the club labels of each node
club_labels = nx.get_node_attributes(G, "club")
```

```
# In[ ]:
```

club_labels

```
# In[ ]:
```

```python
# Create a dictionary to map each node to its club label
node_club = {}
for node, label in club_labels.items():
    node_club[node] = label
```

```
# In[ ]:
```

```python
# Initialize the opinion vector with random values
opinions = np.random.rand(G.number_of_nodes())

# Make the opinions 50% less for one group
for i in range(G.number_of_nodes()):
    if node_club[i] == 'Officer':
        opinions[i] = opinions[i]*0.5
```

# In[ ]:


opinions


# In[ ]:



# Draw the graph

pos = nx.spring_layout(G)

# Plot the graph with the colors

nx.draw(G, pos, node_color=colors, with_labels=False)

# Add the opinions to each node

for i in range(G.number_of_nodes()):

   plt.annotate(round(opinions[i], 2), xy=pos[i], fontsize=8)


plt.show()



# In[ ]:



# Initialize the dictionary to store the average opinion for each group

avg_opinions = {'Mr. Hi': 0, 'Officer': 0}

count = {'Mr. Hi': 0, 'Officer': 0}


# Calculate the average opinion for each group

for i in range(G.number_of_nodes()):

```python
        group = node_club[i]
        avg_opinions[group] += opinions[i]
        count[group] +=1
```

# In[ ]:

```python
# Divide the total opinion by the number of nodes in each group
for group in avg_opinions.keys():
    avg_opinions[group] = avg_opinions[group]/count[group]
```

```python
# Print the average opinion for each group
print(avg_opinions)
```

# In[ ]:

```python
# Set the number of iterations
num_iters = 20
```

```python
# Set the constant alpha also known as Tii - how much we are stubborn (confirmed by experiments)
alpha = 0.7
```

```python
# Simulate the DeGroot model - talking to neigbours gettinng average of their opinion
# and then update my belif in the event X
# note the unweighted graph vs full weights as in example - so this is different
for i in range(num_iters):
```

```python
    new_opinions = np.zeros(G.number_of_nodes())

    for j in range(G.number_of_nodes()):

        neighbors = list(G.neighbors(j))

        neighbors_opinions = opinions[neighbors]

        new_opinions[j] = alpha * opinions[j] + (1 - alpha) * np.mean(neighbors_opinions)

    opinions = new_opinions


# Print the final opinions

print(opinions)

###---------------------



# In[ ]:



# Get the club labels of each node

club_labels = nx.get_node_attributes(G, "club")


# Create a list of colors for each club label

colors = ["gray" if club_labels[i] == 'Officer' else "red" for i in range(len(club_labels))]


# Draw the graph

pos = nx.spring_layout(G)

# Plot the graph with the colors

nx.draw(G, pos, node_color=colors, with_labels=False)

# Add the opinions to each node

for i in range(G.number_of_nodes()):

    plt.annotate(round(opinions[i], 2), xy=pos[i], fontsize=8)
```

```
plt.show()
```

# In[ ]:

```
# Initialize the dictionary to store the average opinion for each group
avg_opinions = {'Mr. Hi': 0, 'Officer': 0}
count = {'Mr. Hi': 0, 'Officer': 0}
```

# In[ ]:

```
# Calculate the average opinion for each group
for i in range(G.number_of_nodes()):
    group = node_club[i]
    avg_opinions[group] += opinions[i]
    count[group] +=1

# Divide the total opinion by the number of nodes in each group
for group in avg_opinions.keys():
    avg_opinions[group] = avg_opinions[group]/count[group]

# Print the average opinion for each group
print(avg_opinions)
```

# In[ ]:

```python
# Set the number of iterations
num_iters = 100


# Set the constant alpha also known as Tii - how much we are stubborn (confirmed by experiments)
alpha = 0.7


# Simulate the DeGroot model - talking to neigbours gettinng average of their opinion
# and then update my belif in the event X
# note the unweighted graph vs full weights as in example - so this is different
for i in range(num_iters):
    new_opinions = np.zeros(G.number_of_nodes())
    for j in range(G.number_of_nodes()):
        neighbors = list(G.neighbors(j))
        neighbors_opinions = opinions[neighbors]
        new_opinions[j] = alpha * opinions[j] + (1 - alpha) * np.mean(neighbors_opinions)
    opinions = new_opinions


# Print the final opinions
print(opinions)
###----------------------
```

```python
# In[ ]:
```

```python
# Initialize the dictionary to store the average opinion for each group
avg_opinions = {'Mr. Hi': 0, 'Officer': 0}
```

```
count = {'Mr. Hi': 0, 'Officer': 0}
```

# In[ ]:

```
# Calculate the average opinion for each group
for i in range(G.number_of_nodes()):
    group = node_club[i]
    avg_opinions[group] += opinions[i]
    count[group] +=1

# Divide the total opinion by the number of nodes in each group
for group in avg_opinions.keys():
    avg_opinions[group] = avg_opinions[group]/count[group]

# Print the average opinion for each group
print(avg_opinions)
```

# In[ ]:

```
# the society is wise precisely when even the most influential individual's influence vanishes in the large
society limit
# if the society grows without bound, over time they will have a common and accurate belief on the
uncertain subject

#
```

#