Network Science Project 1

Sukmanova Elena

1. Network Summary

Let's import some useful libraries:

```
import vk_api
import requests
import numpy as np
import matplotlib.pyplot as plt
plt.xkcd()
import networkx as nx
%matplotlib inline
import scipy.io
import scipy.stats
import scipy.spatial as spt
from IPython.display import SVG
import pandas as pd
import igraph as ig
import matplotlib.image as mpimg
```

- Network source and preprocessing

For purposes of this project I chose my friend's project 'Software Culture' for analysis of its network.

'Software Culture' is an educational platform for the familiarization with modern digital design tools. It offers chamber approach to training through courses, workshops, lectures and film screenings. Main target audience - students and professionals of architectural and related areas (the majority of the audience is formed by Moscow Architectural Institute students).

'Software Culture' has vkontakte public group, where members can find course announcements and educational materials - https://vk.com/softculture. At this moment it has 1421 members. Vkontakte group was created in the end of 2014 at the same time of project creation. Curator of this group is Arseny Afonin, architect, alumnus of Moscow Architectural Institute.

Let's preprocess our data and create graph of 'Software Culture' network:

```
## download list of members ids
members_ids = []
for i in [1, 1000]:
    members_url = 'https://api.vk.com/method/groups.getMembers?group_id=80093690&offset={}'
    json_response = requests.get(members_url.format(i)).json()[u'response']['users']
    members_ids.extend(json_response)
```

```
## sign in to vkontakte to get expanded access
login, password = 'login', 'password'
vk_session = vk_api.VkApi(login, password)
vk_session.authorization()
vk = vk_session.get_api()
```

```
## create dictionary 'member_id: list of member's friends'
members_friends = {}
with vk_api.VkRequestsPool(vk_session) as pool:
    members_friends = pool.method_one_param('friends.get', key = 'user_id', values = members_ids)
```

```
## let's find out how many deactivated members we have
deactivated_members = [k for k, v in members_friends.items() if v == False]
print('From {} members {} are deactivated:\n'.format(len(members_ids), len(deactivated_members)))
x = [print(i) for i in deactivated_members]
```

```
From 1421 members 27 are deactivated:

3342139
176862226
235222045
319008952
10531923
54559847
326223198
22373789
182252299
263493095
93279772
277513957
980715
49485608
218437854
303155931
3460210
266885673
213822588
2583698
294923578
29392193
5724045
7433748
234807010
56164996
38256462
```

```
## make dictionary 'member_id: list of member's friends' without deactivated members
members_friends = {k: v for k, v in members_friends.items() if v != False}
```

```
## let's find out how many members hide information about their friends network
members_without_friends = [k for k, v in members_friends.items() if v['count'] == 0]
print('From {} active members {} hide their friends:\n'.format(len(members_friends),
                                                               len(members_without_friends)))
x = [print(i) for i in members_without_friends]
```

```
From 1393 active members 19 hide their friends:

267923569
266391820
246542776
235498234
359310037
225403835
29587140
145184077
163322466
27490976
49367741
222282869
```

```
224676625
270114044
306134677
254029101
207760911
83897801
225820454
```

```
## make dictionary 'member_id: list of member's friends' without members, who hide their friends network
members_friends = {k: v for k, v in members_friends.items() if v['count'] != 0}
```

```
## create graph of 'Software Culture' network
G = nx.Graph(directed = False)
for i in members_friends:
    G.add_node(i)
    for j in members_friends[i]['items']:
        if i != j and j in members_friends:
            G.add_edge(i, j)
```

- Node attributes

For further analysis let's tie some attributes to our members. Attributes selection depends on two parameters:

- high probability that user has this information
- high probability of assortativity mixing

Based on the above-mentioned requirements following attributes for each member were chosen: gender, city, university, number of friends (and also name for analysis simplification).

```
## make list with members info
members_ids1 = ', '.join(map(str, G.nodes()[:800]))
response1 = vk.users.get(user_ids = members_ids1, fields = 'sex, city, education', lang = 'en')
members_ids2 = ', '.join(map(str, G.nodes()[800:len(G.nodes())]))
response2 = vk.users.get(user_ids = members_ids2, fields = 'sex, city, education', lang = 'en')
response = response1 + response2
```

```
## set attributes to nodes
# name
member_name = [i['first_name'] + ' ' + i['last_name'] for i in response]
member_name = dict(zip(G.nodes(), member_name))
nx.set_node_attributes(G, 'name', member_name)

# gender
member_gender = [i['sex'] for i in response]
member_gender = dict(zip(G.nodes(), member_gender))
nx.set_node_attributes(G, 'gender', member_gender)

# city title
member_city = [i['city']['title'] if 'city' in i else '-' for i in response]
member_city = dict(zip(G.nodes(), member_city))
nx.set_node_attributes(G, 'city', member_city)

# university id
member_university = [i['university'] if 'university' in i else 0 for i in response]
member_university = dict(zip(G.nodes(), member_university))
nx.set_node_attributes(G, 'university', member_university)

# number of friends (popularity)
member_friends_count = [members_friends[i]['count'] for i in G.nodes()]
member_friends_count = dict(zip(G.nodes(), member_friends_count))
nx.set_node_attributes(G, 'friends', member_friends_count)
```

So finally we can create 'Software_Culture.gml' file with 'Software Culture' network:

```
nx.write_gml(G, 'Software_Culture.gml')
```

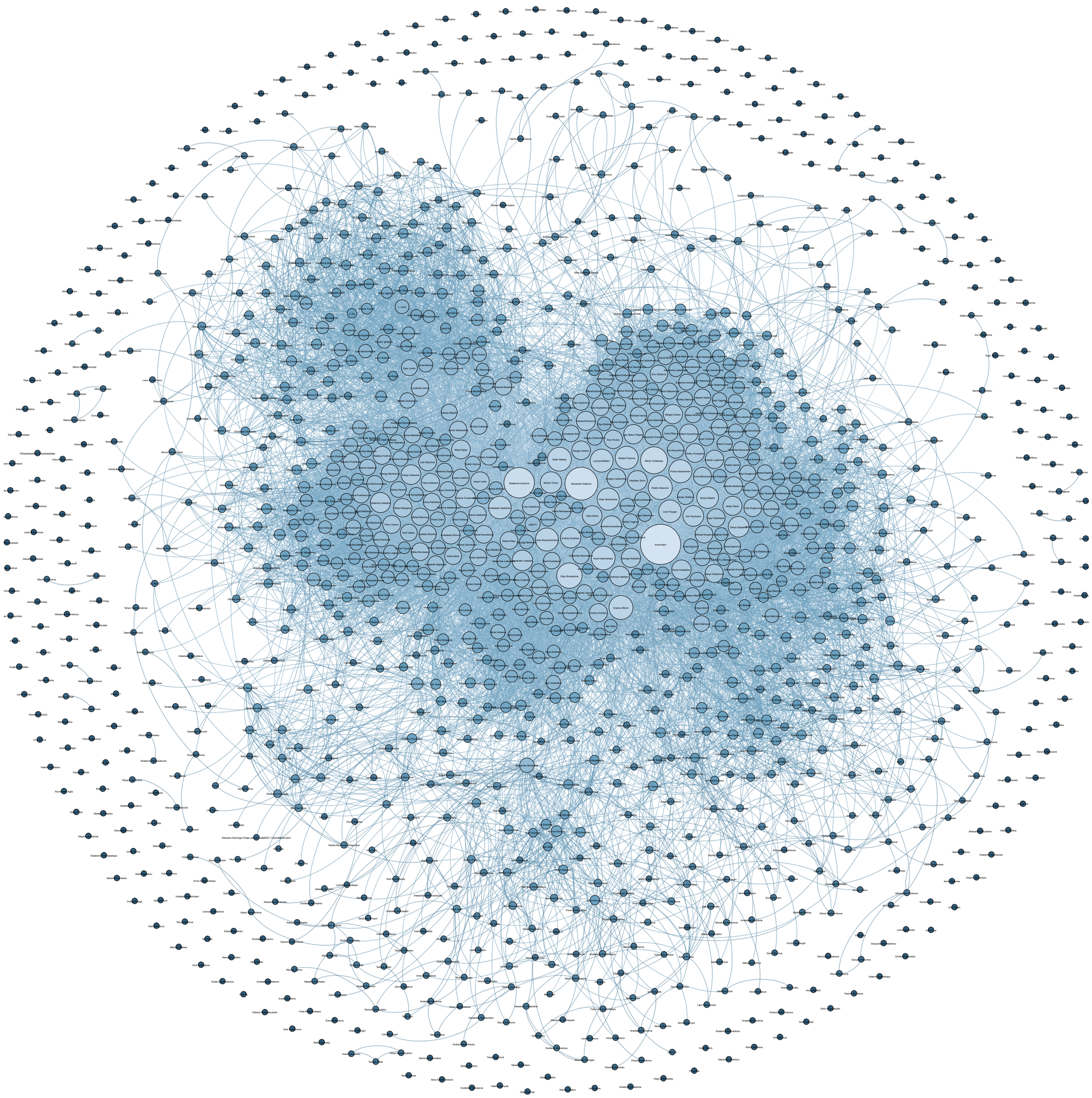
- Size of network and its visualization

```
G = nx.read_gml('Software_Culture.gml')
```

```
print("'Software Culture' network has {} active members with {} connections between each other."\
      .format(G.number_of_nodes(), G.number_of_edges()))
print('Number of connected components = {}'.format(nx.number_connected_components(G)))
```

```
'Software Culture' network has 1374 active members with 16008 connections between each other.
Number of connected components = 290
```

```
SVG(filename='initial_graph.svg')
```



Plot above represents graph visualization of 'Software Culture' network. We can see that our network has gigantic connected component and some isolated nodes around. Size and color of node represent its degree (number of nearest neighbours) - here we present degree centrality. We can easily distinguish top-3 members with big amount of links:

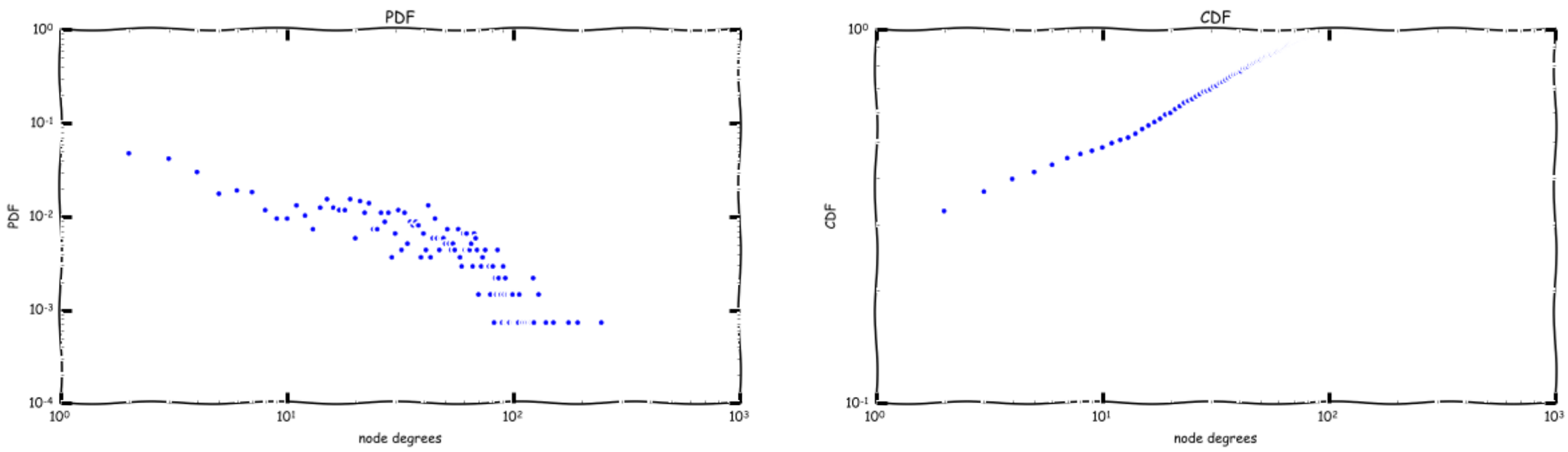
- **Anna Bern** - undergraduate of Moscow Architectural Institute, worked with 'Software Culture' project for some time, highly sociable person
- **Alexander Safonov** - undergraduate of Moscow Architectural Institute, attended different 'Software Culture' courses
- **Ekaterina Ivoylova** - undergraduate of Moscow Architectural Institute, didn't attend 'Software Culture' courses

We can also notice that our network has some exuding structure - we will study it later.

- Degree distribution, Diameter, Clustering Coefficient

```
## let's plot degree distribution of our network
# make list with node degrees
nodes_degree = list(G.degree().values())
# find number of nodes with k links
number_of_nodes_with_k_links = np.bincount(nodes_degree)
# find pdf
pdf = number_of_nodes_with_k_links.astype(float) / G.number_of_nodes()
# find cdf
cdf = np.cumsum(pdf)
# plotting
plt.figure(figsize=(20,5))

plt.subplot(1,2,1)
plt.loglog(np.arange(0, len(pdf)), pdf, 'b.')
plt.title('PDF')
plt.xlabel('node degrees')
plt.ylabel('PDF')

plt.subplot(1,2,2)
plt.loglog(np.arange(0, len(cdf)), cdf, 'b.')
plt.title('CDF')
plt.xlabel('node degrees')
plt.ylabel('CDF')
```

```
<matplotlib.text.Text at 0x112545cf8>
```



From plot above we can see that 'Software Culture' network has heavy-tailed distribution, that in general corresponds to power law.

```
print('Max value of node degree = {}'.format(np.max(nodes_degree)))
print('Mean value of node degree = {}'.format(np.mean(nodes_degree)))
```

```
Max value of node degree = 244
Mean value of node degree = 23.30131004366812
```

From Gephi we calculated (as networkx doesn't calculate these parameters for not-connected graph):

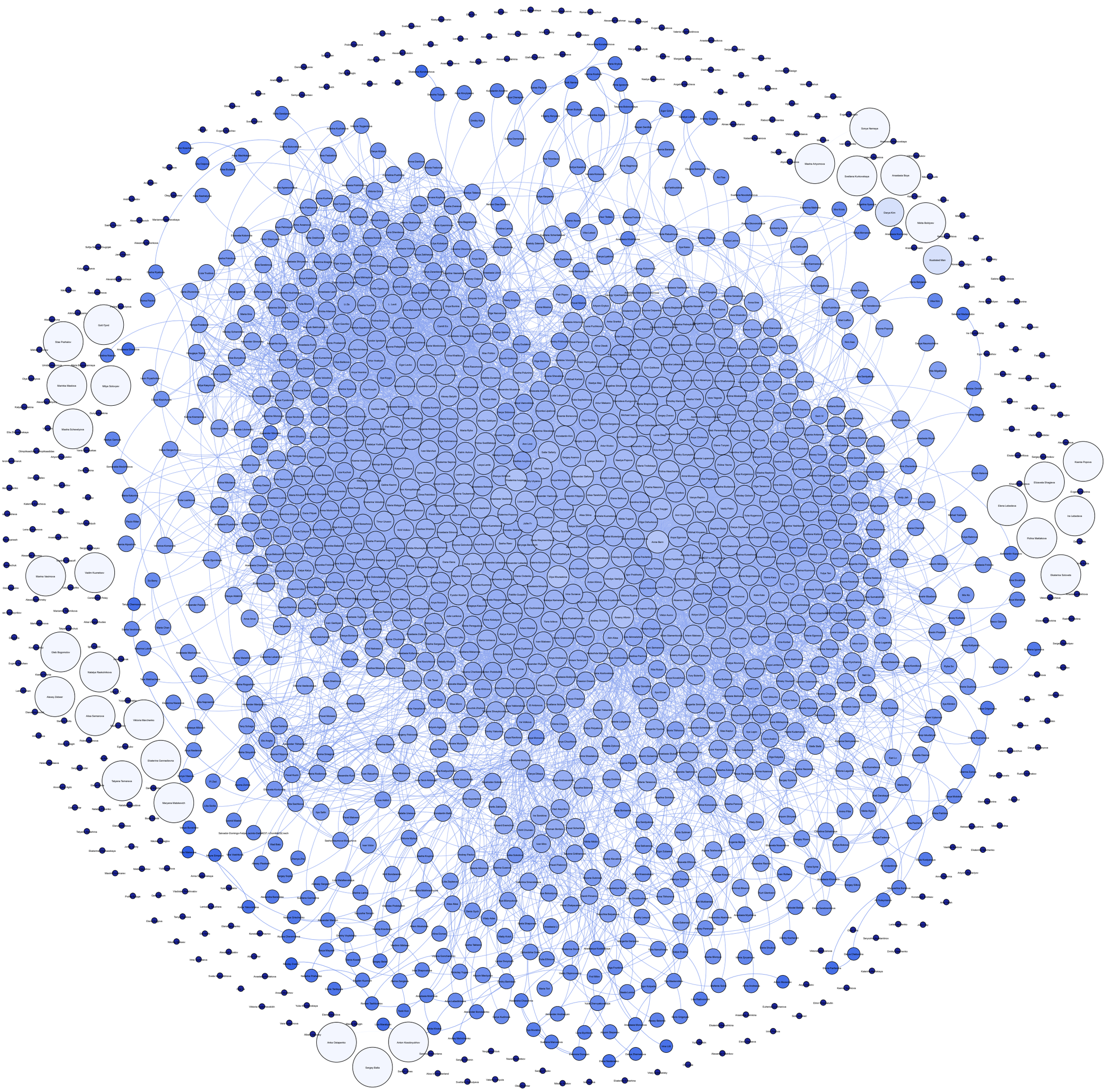
- **graph diameter** (the largest number of nodes which must be traversed in order to travel from one node to another) = **8**
- **average path length** (average number of steps along the shortest paths for all possible pairs of network nodes) = **2.972**
- **average clustering coefficient** (average measure of the degree to which nodes in a graph tend to cluster together) = **0,368** - not so high

2. Structural Analysis

- Closeness/Betweenness/Eigenvector centralities. Top nodes interpretation

We have already present degree centrality. Let's also consider closeness centrality, which reflects how close member to all the other members in network:

```
SVG(filename='graph_closeness_centrality.svg')
```



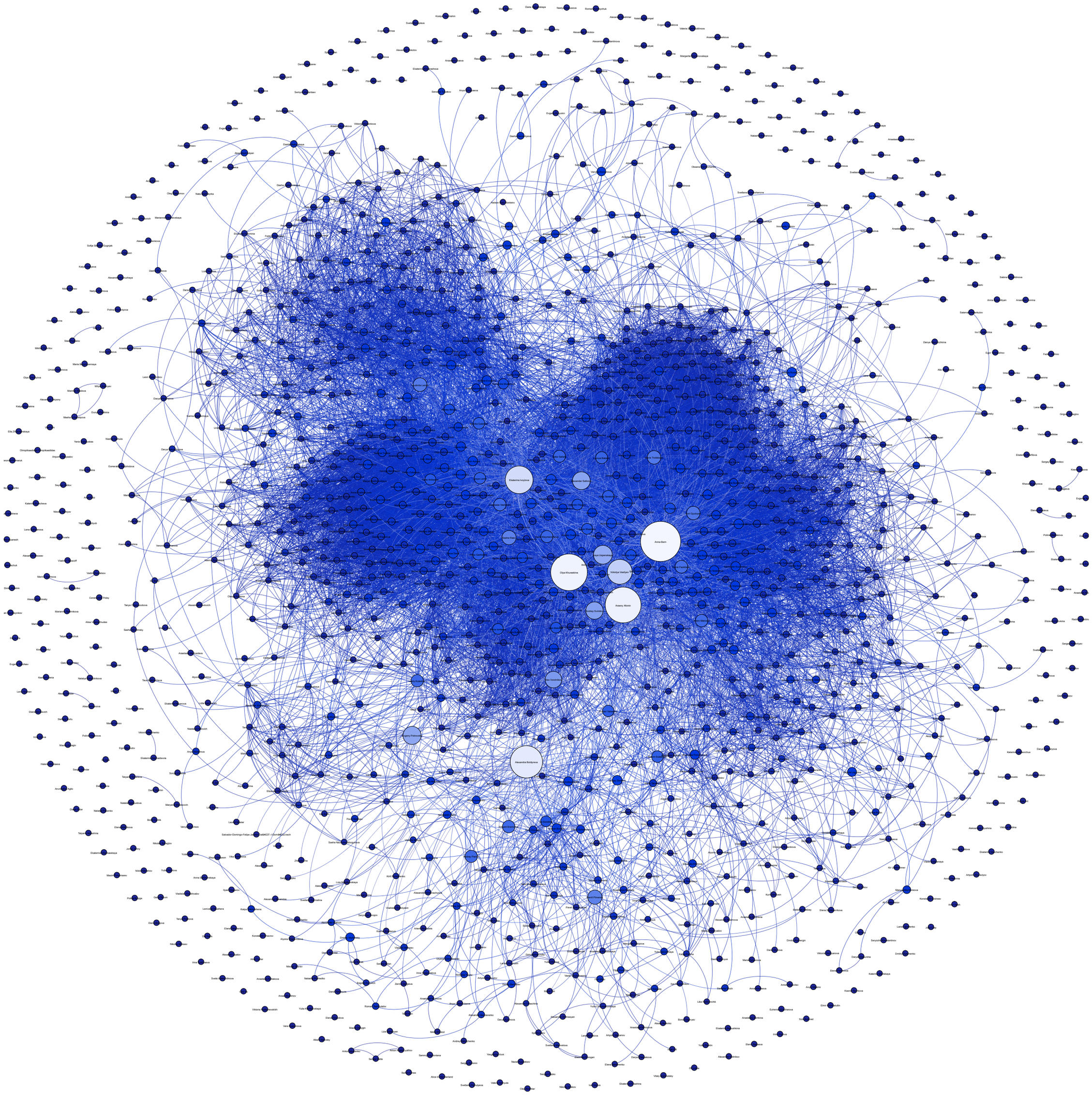
As we see from graph above in Gephi calculation of closeness centrality measures is affected by size of connected component - we can see that in small communities we have nodes with high closeness centrality. Networkx module normalizes closeness centrality measures by the size of connected component, but unfortunatelly we can't use networkx for

visualization as our network is rather big. Despite disadvantages of Gephi calculations we still can see some distinguished members in gigantic connected component:

- **Anna Bern** - same top member from degree centrality
- **Arseny Afonin** - 'Software Culture' project creator and curator, so there is nothing surprising that he has high closeness centrality

Let's also consider betweenness centrality, which shows number of shortest paths going through the member:

```
SVG(filename='graph_betweenness_centrality.svg')
```



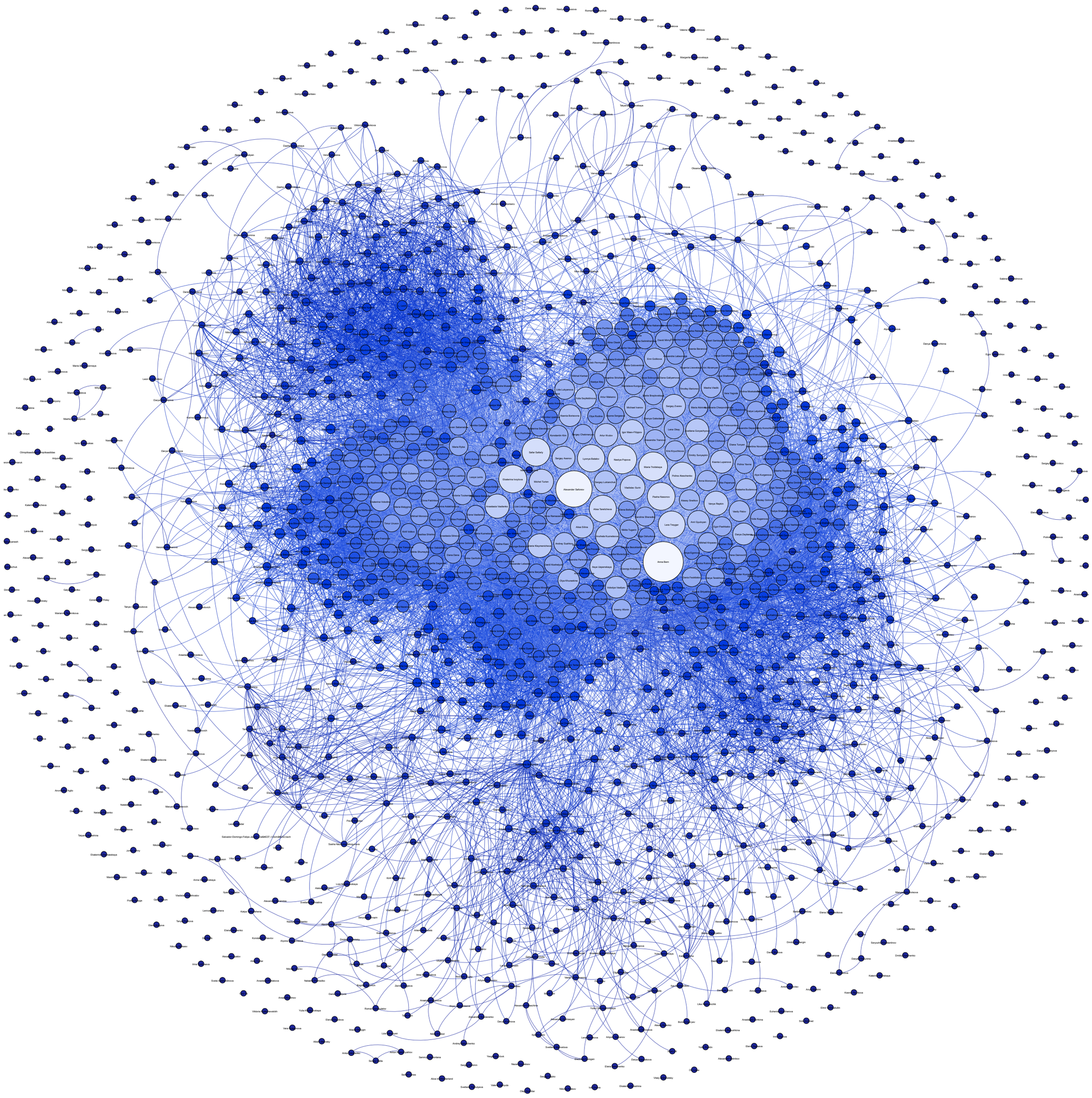
Through betweenness centrality visualization we can easily distinguish top-3 members:

- **Anna Bern** - same top member from degree and closeness centralities
- **Olya Khuraskina** - undergraduate of Moscow Architectural Institute, attended different 'Software Culture' courses, highly sociable person
- **Arseny Afonin** - same top member from closeness centrality, 'Software Culture' project creator and curator

Let's also consider eigenvector centrality, in which importance of a member depends on the importance of its neighbours

(calculated with 100 iterations):

```
SVG(filename='graph_eigenvector_centrality.svg')
```



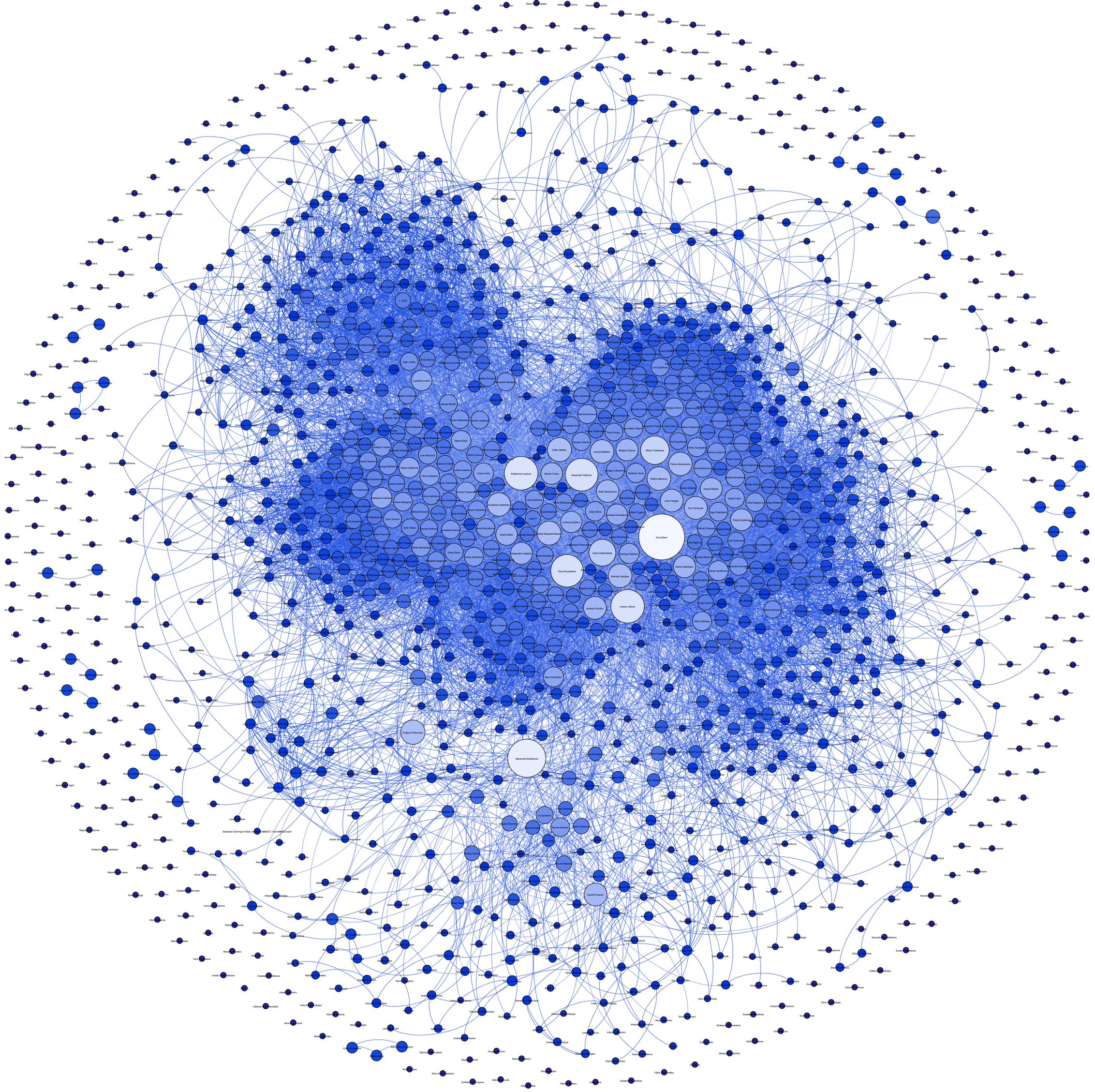
Through eigenvector centrality visualization we can easily distinguish top-2 members:

- **Anna Bern** - same top member from degree, closeness and betweenness centralities
- **Alexander Safonov** - same top member from degree centrality

- Pagerank. Comparison with centralities

Pagerank works by counting the number and quality of links to a node to determine a rough estimate of how important the node is. Let's find out pagerank of our network:

```
SVG(filename='graph_pagerank.svg')
```



We can pick out top-2 members from pagerank results:

- **Anna Bern** - same top member from degree, closeness, betweenness and eigenvector centralities
- **Alexandra Boldyreva** - coordinator of competitive courses in Architectural School MARSH

At the moment we calculated 5 different types of node centralities. From results above we can remark that member importance depends on logic of specific centrality measure. Let's find correlation between node rankings based on different centrality measures. To make better comparison we can use such measures of rank correlation as Spearman rank correlation coefficient and Kendall rank correlation coefficient.

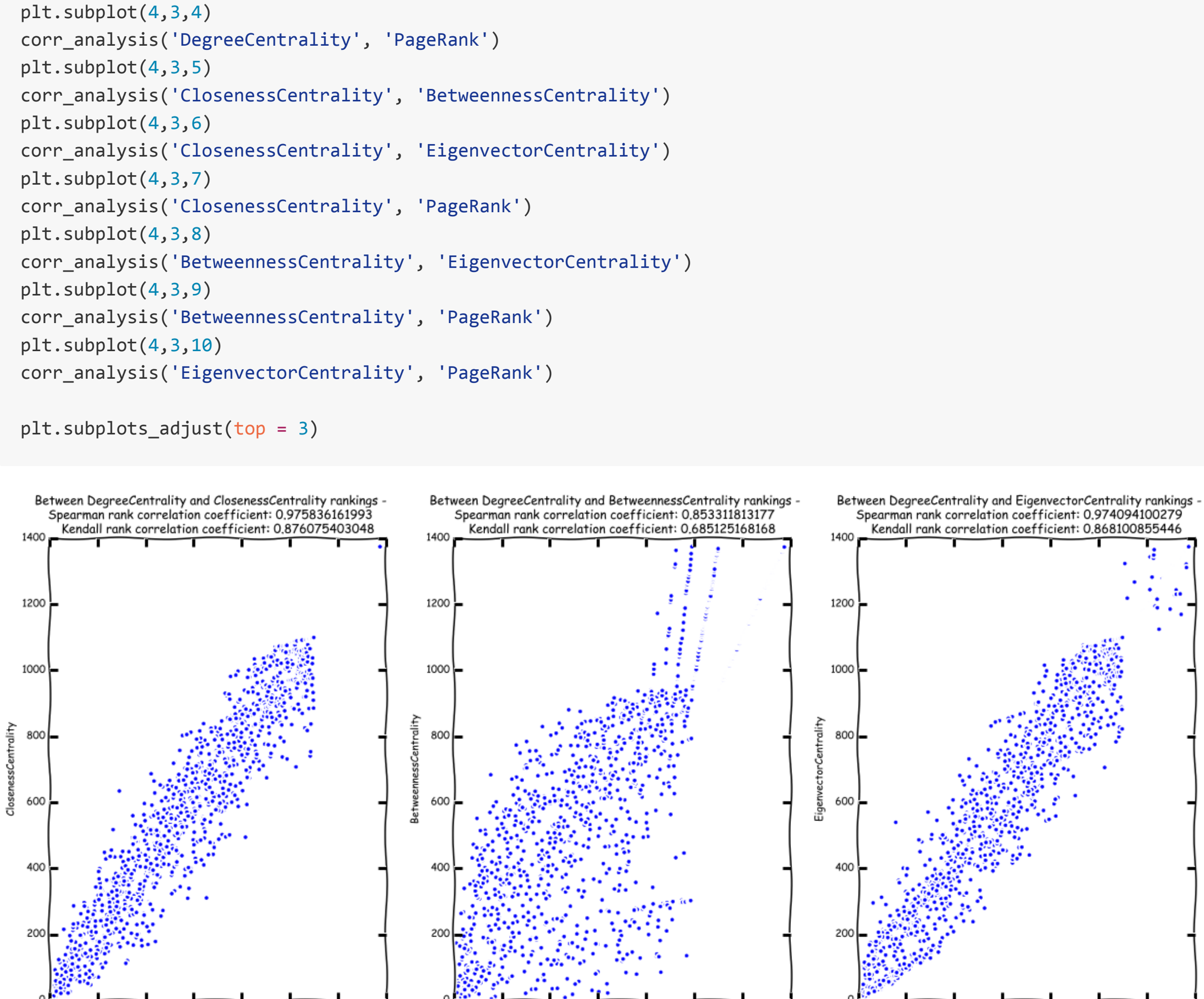
Spearman rank correlation coefficient assesses how well the relationship between two variables can be described using a monotonic function.

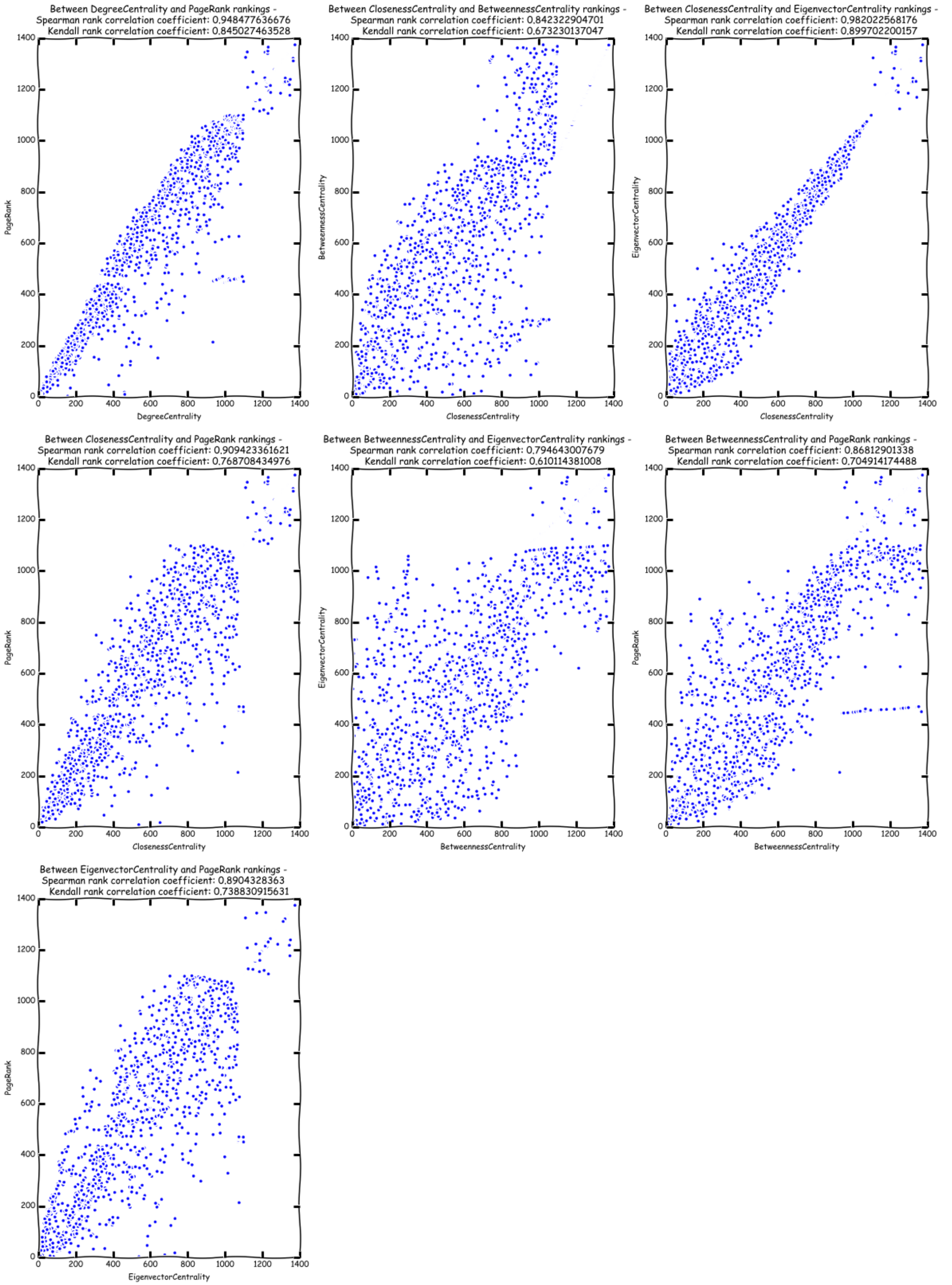
Kendall rank correlation coefficient uses metric that counts the number of pairwise disagreements between two ranking lists.

```
## find degree, closeness, betweenness and eigenvector centralities and pagerank
degree_centrality = nx.degree_centrality(G)
closeness_centrality = nx.closeness_centrality(G)
betweenness_centrality = nx.betweenness_centrality(G)
eigenvector_centrality = nx.eigenvector_centrality(G)
pagerank = nx.pagerank(G)
```

```
## rank nodes depending on scores and set it to nodes attributes
scores = {'DegreeCentrality': degree_centrality, 'ClosenessCentrality': closeness_centrality,
          'BetweennessCentrality': betweenness_centrality, 'EigenvectorCentrality': eigenvector_centrality,
         'PageRank': pagerank}
for i, j in scores.items():
    score_ranking_dict = {key: rank for rank, key in enumerate(sorted(j, key = j.get, reverse=True), 1)}
    nx.set_node_attributes(G, i, j)
    nx.set_node_attributes(G, i+'Rank', score_ranking_dict)

## make function for plotting and computing correlation between two different centrality measures
def corr_analysis(first_score, second_score):
    first_score_ranking = list(nx.get_node_attributes(G, first_score+'Rank').values())
    second_score_ranking = list(nx.get_node_attributes(G, second_score+'Rank').values())

    plt.plot(first_score_ranking, second_score_ranking, 'b.')
    plt.title('Between {} and {} rankings -\
    \nSpearman rank correlation coefficient: {}\
    \nKendall rank correlation coefficient: {}'.format(first_score, second_score,
                                                       str(scipy.stats.stats.spearmanr(first_score_ranking,
                                                                                       second_score_ranking)[0]),
                                                       str(scipy.stats.stats.kendalltau(first_score_ranking,
                                                                                        second_score_ranking)[0]))
    plt.xlabel(first_score)
    plt.ylabel(second_score)
```

```
plt.figure(figsize=(20,10))

plt.subplot(4,3,1)
corr_analysis('DegreeCentrality', 'ClosenessCentrality')
plt.subplot(4,3,2)
corr_analysis('DegreeCentrality', 'BetweennessCentrality')
plt.subplot(4,3,3)
corr_analysis('DegreeCentrality', 'EigenvectorCentrality')
plt.subplot(4,3,4)
corr_analysis('DegreeCentrality', 'PageRank')
plt.subplot(4,3,5)
corr_analysis('ClosenessCentrality', 'BetweennessCentrality')
plt.subplot(4,3,6)
corr_analysis('ClosenessCentrality', 'EigenvectorCentrality')
plt.subplot(4,3,7)
corr_analysis('ClosenessCentrality', 'PageRank')
plt.subplot(4,3,8)
corr_analysis('BetweennessCentrality', 'EigenvectorCentrality')
plt.subplot(4,3,9)
corr_analysis('BetweennessCentrality', 'PageRank')
plt.subplot(4,3,10)
corr_analysis('EigenvectorCentrality', 'PageRank')

plt.subplots_adjust(top = 3)
```





From calculations above we can see that there is strong direct monotonic association between ranking combinations. The weakest correlation - between betweenness centrality and eigenvector centrality rankings; the strongest one - between closeness centrality and eigenvector centrality rankings.

- Assortative Mixing according to node attributes

Assortativity is a preference for a network's nodes to attach to others that are similar in some way. When r = 1, the network

is said to have perfect assortative mixing patterns, when r = 0 the network is non-assortative, while at r = −1 the network is completely disassortative.

First, let's look at mixing by node degree:

```
degree_assort = nx.degree_assortativity_coefficient(G)
print('Node degree assortativity: {}'.format(degree_assort))
```

```
Node degree assortativity: 0.17171671919238055
```

Node degree assortativity of our network is close to zero, which means that network is non-assortative and has no clear structure (smth between core with periphery and star).

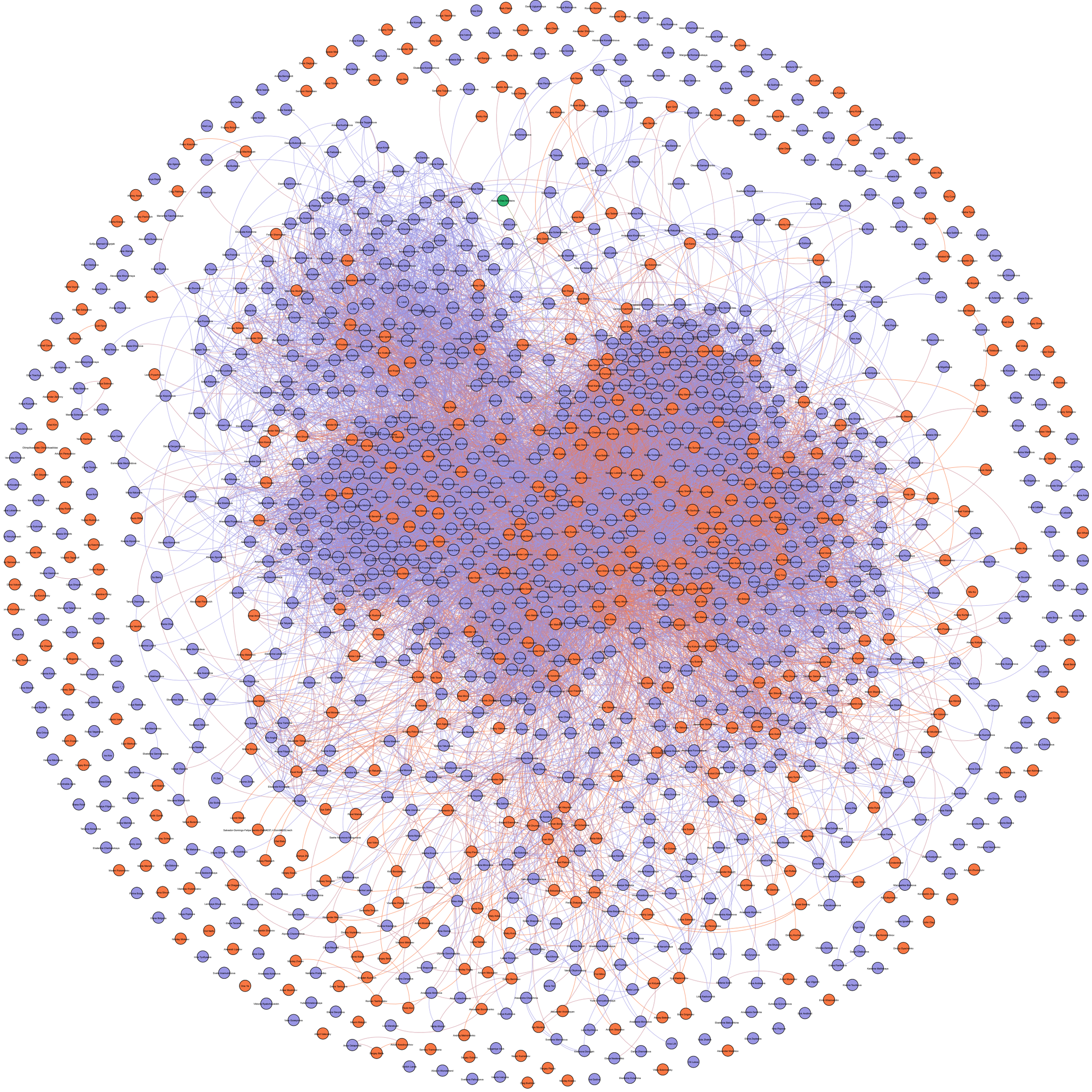
Let's check, whether our members mix on node attributes:

Gender:

```
modularity_gender = nx.attribute_assortativity_coefficient(G, 'gender')
print('Assortativity coefficient (modularity) for {}: {}'.format('gender', modularity_gender))
```

```
Assortativity coefficient (modularity) for gender: 0.053842676415054094
```

```
SVG(filename='graph_gender.svg')
```



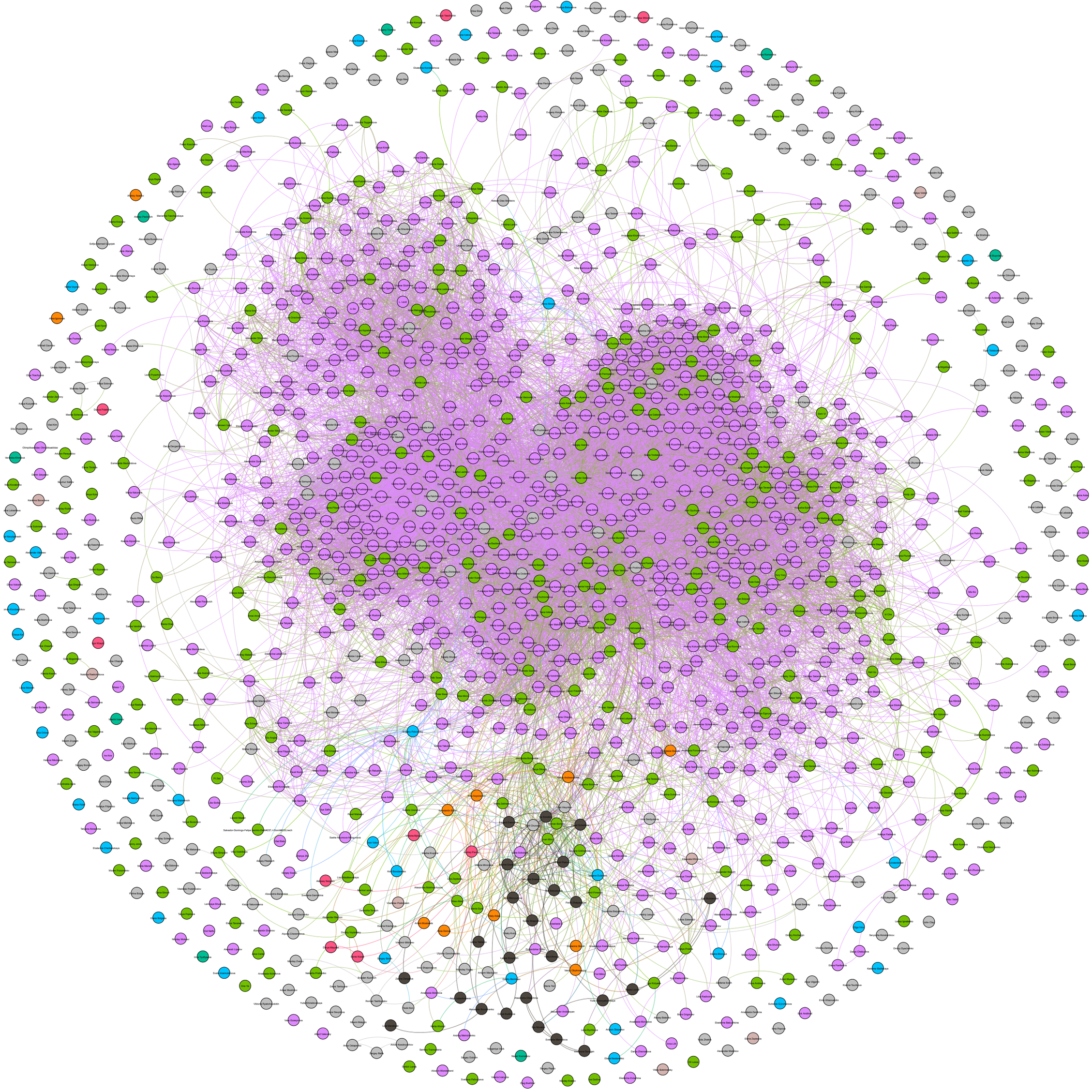
From assortativity coefficient for gender and graph above we can see that our network is non-assortative by gender. But it becomes obvious that female gender (about 70% of all members) prevailes in our network.

City:

```
modularity_city = nx.attribute_assortativity_coefficient(G, 'city')
print('Assortativity coefficient (modularity) for {}: {}'.format('city', modularity_city))
```

```
Assortativity coefficient (modularity) for city: 0.03941453367246818
```

```
SVG(filename='graph_city.svg')
```



From assortativity coefficient for city we can observe that our network is non-assortative by city. But graph above and ordinary logic indicate that our members are mixed by their location - Moscow citizens are about 55% of network and form some kind of communities. This situation corresponds to reality - 'Software Culture' project organizes only off-line courses in Moscow and in general didn't communicate with members from regions.

Low modularity can be explained by the fact that about 22% of network didn't specify their city location.

University:

```
members_university = nx.get_node_attributes(G, 'university')
print("Proportion of members who didn't specify their university: {}".format(
        len([k for k, v in members_university.items() if v == 0])/G.number_of_nodes()))
```

```
Proportion of members who didn't specify their university: 0.7590975254730713
```

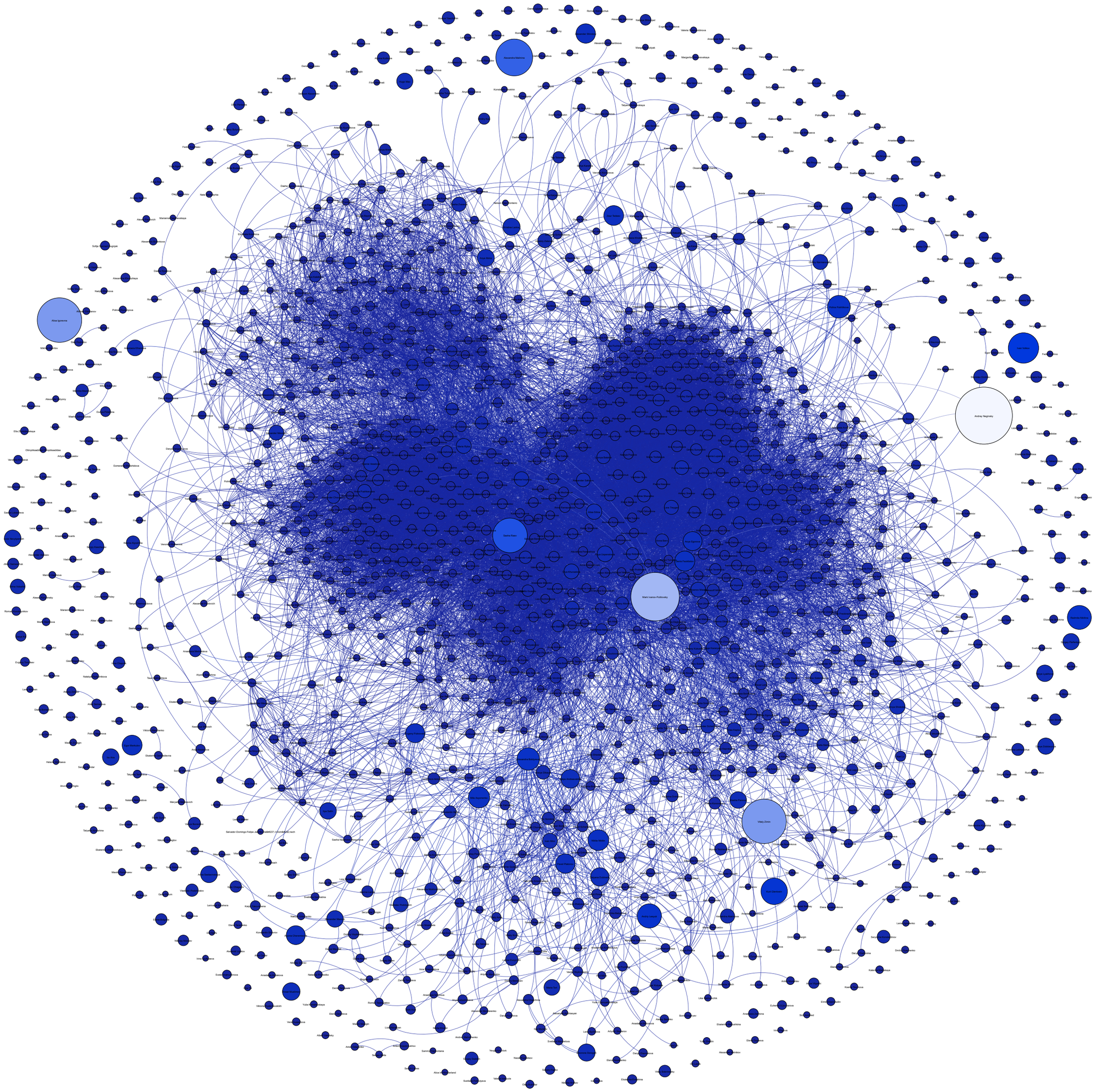
Through analysis it was figured out that we can't take information about members university for assortativity mixing analysis as about 76% of members didn't specify their university. But on the logical assumption university attribute should have high modularity as main target audience of 'Software Culture' is students of highly specialized professions, familiar with each other.

Number of friends (popularity):

```
modularity_friends = nx.attribute_assortativity_coefficient(G, 'friends')
print('Assortativity coefficient (modularity) for {}: {}'.format('friends', modularity_friends))
```

```
Assortativity coefficient (modularity) for friends: -0.0016134560864976373
```

```
SVG(filename='graph_friends.svg')
```



From assortativity coefficient for popularity attribute and graph above we can take notice that our network is even disassortative by number of friends. It is obvious as popular members in the majority connects to ordinary ones (because there are a limited number of popular users).

- Node structural equivalence/similarity

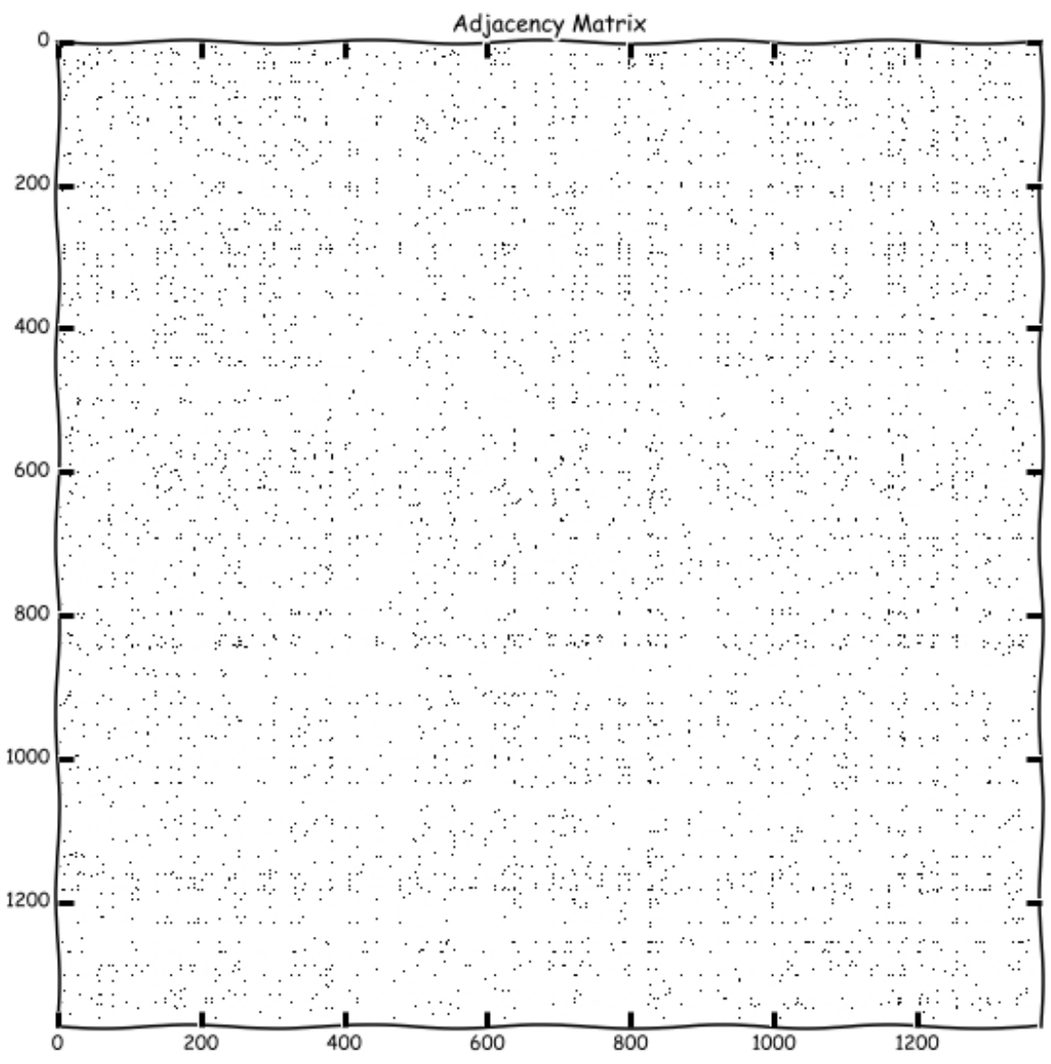
Similarity in network analysis occurs when two nodes fall in the same equivalence class. We will consider structural equivalence. Two nodes of a network are structurally equivalent if they share many of the same neighbors.

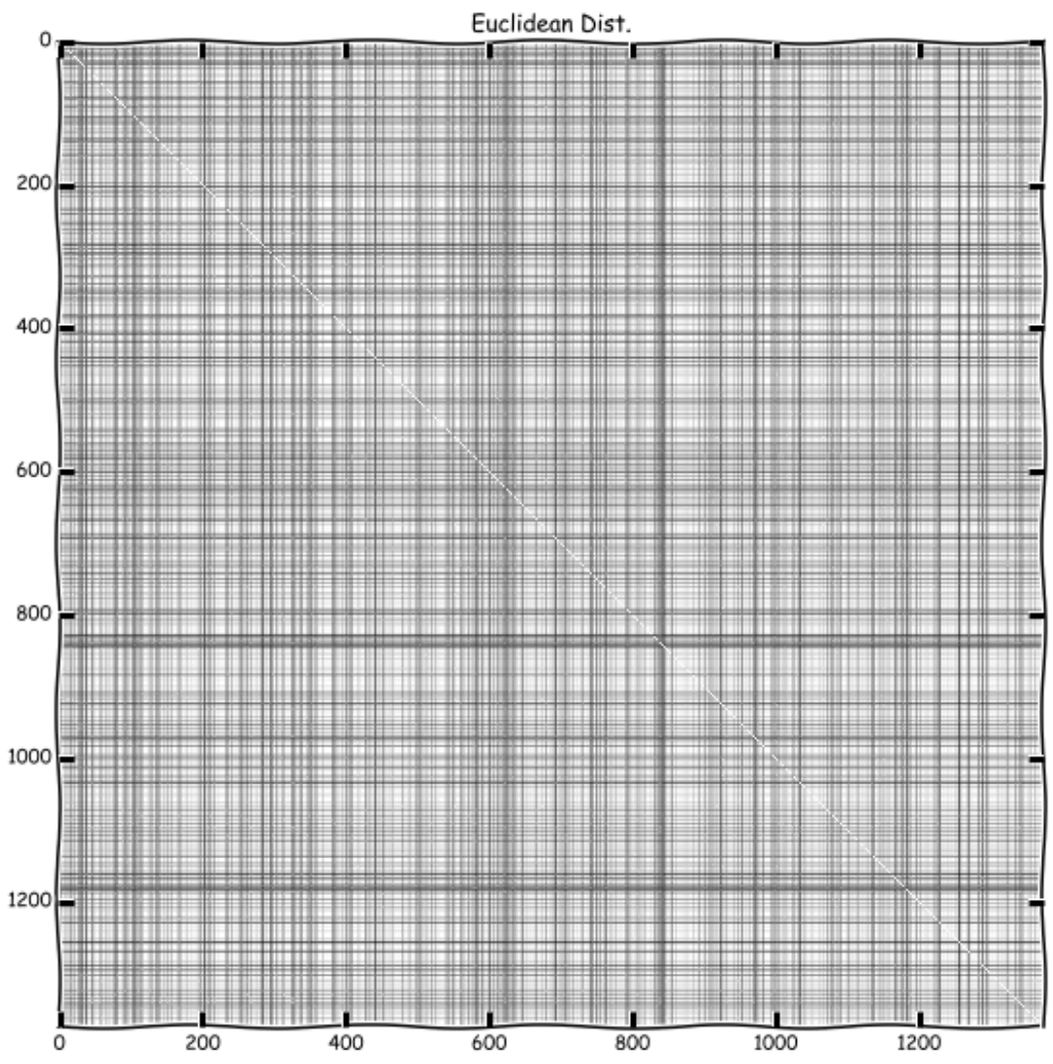
For node structural equivalence calculation let's use one of algorithm - Euclidean Distance. Euclidean distance is equal to

the number of neighbors that differ between two vertices. It is rather a dissimilarity measure, since it is larger for vertices which differ more.

```
A = nx.to_numpy_matrix(G)
A = np.asarray(A)

## plotting
plt.figure(figsize=(20,10))

# plot adjacency matrix
plt.subplot(1,2,1)
plt.imshow(A, cmap = 'Greys', interpolation = 'None')
plt.title('Adjacency Matrix')

# plot similarity matrix with Euclidean Distance
plt.subplot(1,2,2)
dVec = spt.distance.pdist(A, metric = 'euclidean')
D = spt.distance.squareform(dVec)
plt.imshow(D, cmap = 'Greys', interpolation = 'None')
plt.title('Euclidean Dist.')
```

```
<matplotlib.text.Text at 0x1124bae10>
```





Let's find nodes that have structural equivalence on real neighbors:

```
similar_nodes = np.where(D == 0)
pairs = []
neighbors = []
for i in range(len(similar_nodes[0])):
    if similar_nodes[0][i] != similar_nodes[1][i]:
        if G.neighbors(G.nodes()[similar_nodes[0][i]]) != [] and
        G.neighbors(G.nodes()[similar_nodes[1][i]]) != []:
            if '{}_{}'.format(G.nodes()[similar_nodes[0][i]], G.nodes()[similar_nodes[1][i]]) not in pairs:
                pairs.append('{}_{}'.format(G.nodes()[similar_nodes[1][i]], G.nodes()[similar_nodes[0][i]]))
                print('Members {} and {} have structural equivalence with neigbors: {}'
                      .format(G.nodes()[similar_nodes[0][i]],
                              G.nodes()[similar_nodes[1][i]],
                              G.neighbors(G.nodes()[similar_nodes[0][i]])))
                neighbors.append(G.neighbors(G.nodes()[similar_nodes[0][i]])[0])
```

```
Members 16883945 and 12240400 have structural equivalence with neigbors: [166624273]
Members 16883945 and 24005858 have structural equivalence with neigbors: [166624273]
Members 29470965 and 233722385 have structural equivalence with neigbors: [190099226]
```

```
Members 29470965 and 23794426 have structural equivalence with neigbors: [190099226]
Members 29470965 and 33920129 have structural equivalence with neigbors: [190099226]
Members 29470965 and 2103884 have structural equivalence with neigbors: [190099226]
Members 29470965 and 338403157 have structural equivalence with neigbors: [190099226]
Members 29470965 and 260935282 have structural equivalence with neigbors: [190099226]
Members 29470965 and 45813616 have structural equivalence with neigbors: [190099226]
Members 6189575 and 50932237 have structural equivalence with neigbors: [2472970]
Members 6189575 and 146946598 have structural equivalence with neigbors: [2472970]
Members 6189575 and 41298588 have structural equivalence with neigbors: [2472970]
Members 233722385 and 23794426 have structural equivalence with neigbors: [190099226]
Members 233722385 and 33920129 have structural equivalence with neigbors: [190099226]
Members 233722385 and 2103884 have structural equivalence with neigbors: [190099226]
Members 233722385 and 338403157 have structural equivalence with neigbors: [190099226]
Members 233722385 and 260935282 have structural equivalence with neigbors: [190099226]
Members 233722385 and 45813616 have structural equivalence with neigbors: [190099226]
Members 23794426 and 33920129 have structural equivalence with neigbors: [190099226]
Members 23794426 and 2103884 have structural equivalence with neigbors: [190099226]
Members 23794426 and 338403157 have structural equivalence with neigbors: [190099226]
Members 23794426 and 260935282 have structural equivalence with neigbors: [190099226]
Members 23794426 and 45813616 have structural equivalence with neigbors: [190099226]
Members 240788595 and 9526180 have structural equivalence with neigbors: [144216864]
Members 33920129 and 2103884 have structural equivalence with neigbors: [190099226]
Members 33920129 and 338403157 have structural equivalence with neigbors: [190099226]
Members 33920129 and 260935282 have structural equivalence with neigbors: [190099226]
Members 33920129 and 45813616 have structural equivalence with neigbors: [190099226]
Members 23515348 and 40631222 have structural equivalence with neigbors: [134031992, 1976178]
Members 12240400 and 24005858 have structural equivalence with neigbors: [166624273]
Members 19100699 and 1686619 have structural equivalence with neigbors: [826244]
Members 205653791 and 39466979 have structural equivalence with neigbors: [19029823]
Members 23795640 and 32303534 have structural equivalence with neigbors: [49783137]
Members 23795640 and 20261998 have structural equivalence with neigbors: [49783137]
Members 32303534 and 20261998 have structural equivalence with neigbors: [49783137]
Members 50932237 and 146946598 have structural equivalence with neigbors: [2472970]
Members 50932237 and 41298588 have structural equivalence with neigbors: [2472970]
Members 146946598 and 41298588 have structural equivalence with neigbors: [2472970]
Members 2103884 and 338403157 have structural equivalence with neigbors: [190099226]
Members 2103884 and 260935282 have structural equivalence with neigbors: [190099226]
Members 2103884 and 45813616 have structural equivalence with neigbors: [190099226]
Members 338403157 and 260935282 have structural equivalence with neigbors: [190099226]
Members 338403157 and 45813616 have structural equivalence with neigbors: [190099226]
Members 14135867 and 28192403 have structural equivalence with neigbors: [11452697]
Members 260935282 and 45813616 have structural equivalence with neigbors: [190099226]
```

We can see that in our network mostly nodes with one neighbor have structural equivalence with similar ones. If we take a closer look, we can notice that there are several members to which many singletons are attached. It is very important to determine such central members when working with network because through these central members we can influence our singletons.

Let's find these central members:

```
for i, j in {x: neighbors.count(x) for x in neighbors}.items():
    if j > 1:
        print('Member {} with ID {} is exclusive neighbor for {} members'.format(G.node[i]['name'], i, j))
```

```
Member Daniil Pimenov with ID 166624273 is exclusive neighbor for 3 members
Member Maxim Sulaeman with ID 49783137 is exclusive neighbor for 3 members
Member Evgeny Pokrovsky with ID 190099226 is exclusive neighbor for 28 members
Member Alexandra Boldyreva with ID 2472970 is exclusive neighbor for 6 members
```

- The closest random graph model similar to our social network

Let's build familiar to us theoretical random graph models to find the closest one to 'Software Culture' network:

```
## 'Software Culture' network attributes
n = nx.number_of_nodes(G)
```

```
m = nx.number_of_edges(G)
k = np.mean(list(G.degree().values()))

## Erdos-Renyi graph model
erdos = nx.erdos_renyi_graph(n, m / float(n*(n-1)/2))
## Barabasi-Albert graph model
barbarasi = nx.barabasi_albert_graph(n, int(m/n))
## Watts-Strogatz graph model
watts = nx.watts_strogatz_graph(n, int(k), 0.5)

## Kolmogorov-Smirnov test
print('Kolmogorov-Smirnov test results between Software Culture network and Erdos-Renyi model -\n {}'
      .format(scipy.stats.ks_2samp(list(G.degree().values()), list(erdos.degree().values()))))
print('Kolmogorov-Smirnov test results between Software Culture network and Barabasi-Albert model -\n {}'
      .format(scipy.stats.ks_2samp(list(G.degree().values()), list(barbarasi.degree().values()))))
print('Kolmogorov-Smirnov test results between Software Culture network and Watts-Strogatz model -\n {}'
      .format(scipy.stats.ks_2samp(list(G.degree().values()), list(watts.degree().values()))))
```

```
## Basic graph statistics
data = [[m, '0,368', '2.972', '8'],
        [nx.number_of_edges(erdos), nx.average_clustering(erdos),
         nx.average_shortest_path_length(erdos), nx.diameter(erdos)],
        [nx.number_of_edges(barbarasi), nx.average_clustering(barbarasi),
         nx.average_shortest_path_length(barbarasi), nx.diameter(barbarasi)],
        [nx.number_of_edges(watts), nx.average_clustering(watts),
         nx.average_shortest_path_length(watts), nx.diameter(watts)]]
rows = ['Software Culture network', 'Erdos-Renyi model',
        'Barabasi-Albert model', 'Watts-Strogatz model']
columns = ['# edges', 'Clustering coeff', 'Path length', 'Diameter']
pd.DataFrame(data, rows, columns)
```

```
Kolmogorov-Smirnov test results between Software Culture network and Erdos-Renyi model -
 Ks_2sampResult(statistic=0.50436681222707413, pvalue=1.1528377465204282e-153)
Kolmogorov-Smirnov test results between Software Culture network and Barabasi-Albert model -
 Ks_2sampResult(statistic=0.48326055312954874, pvalue=4.1556507377466767e-141)
Kolmogorov-Smirnov test results between Software Culture network and Watts-Strogatz model -
 Ks_2sampResult(statistic=0.53420669577874824, pvalue=2.4727644274828598e-172)
```

	# edges	Clustering coeff	Path length	Diameter
Software Culture network	16008	0,368	2.972	8
Erdos-Renyi model	16192	0.0172695	2.64008	4
Barabasi-Albert model	14993	0.051203	2.59031	4
Watts-Strogatz model	15114	0.099778	2.73195	4

```
print("From information above we can see that our theoretical models are not so close to our network.\n\
But based largely on p-value of Kolmogorov-Smirnov test we can say that the closest model\nto\
'Software Culture' network is Barabasi-Albert model with {} nodes and {} edges to add on every step."
      .format(n, int(m/n)))
```

```
From information above we can see that our theoretical models are not so close to our network.
But based largely on p-value of Kolmogorov-Smirnov test we can say that the closest model
to'Software Culture' network is Barabasi-Albert model with 1374 nodes and 11 edges to add on every step.
```

3. Community Detection

At this time there is no rigorous and universally accepted definition of community in field of network analysis. But we can bring following definition:

Network communities are groups of vertices, such, that vertices inside the group connected with many more edges than

between groups, with some desired properties:

- Low overlapping
- Density
- Low distance (diameter)
- Connectivity

We have big number of connected components in 'Software Culture' network. Let's first look at its structure:

```
print("In 'Software Culture' network are {} connected components.\n\
Let's make dictionary 'volume of connected component: number of connected components with such volume':"
     .format(nx.number_connected_components(G)))
volume = [len(i) for i in sorted(nx.connected_components(G), key = len, reverse = True)]
volume_dict = dict([(i, volume.count(i)) for i in set(volume)])
print(volume_dict)
```

```
In 'Software Culture' network are 290 connected components.
Let's make dictionary 'volume of connected component: number of connected components with such volume':
{1: 275, 2: 11, 3: 3, 1068: 1}
```

We see that there is only one gigantic connected component in our network. For convenience of further clique search and community detection let's work only with this gigantic connected component:

```
## Graph of Gigantic Connected Component in networkx - GCC
GCC = nx.Graph(G.subgraph(list(max(nx.connected_components(G)))), directed = False)
## Graph of Gigantic Connected Component in igraph - iGCC
nx.write_edgelist(GCC, 'list_of_edges_GCC')
iGCC = ig.Graph.Read_Ncol('list_of_edges_GCC', directed = False)
```

- Clique search

In this part we will combine two modules - networkx and igraph - to improve efficiency.

A clique is a complete (fully connected) subgraph. A maximal clique is a clique that cannot be extended by including one more adjacent vertex.

```
print("Number of maximal cliques of 'Software Culture' network is: {}"
      .format(nx.graph_number_of_cliques(GCC)))
```

```
Number of maximal cliques of 'Software Culture' network is: 34085
```

Graph clique number is the size of the maximum clique, where maximum clique is a clique of the largest possible size in a given graph.

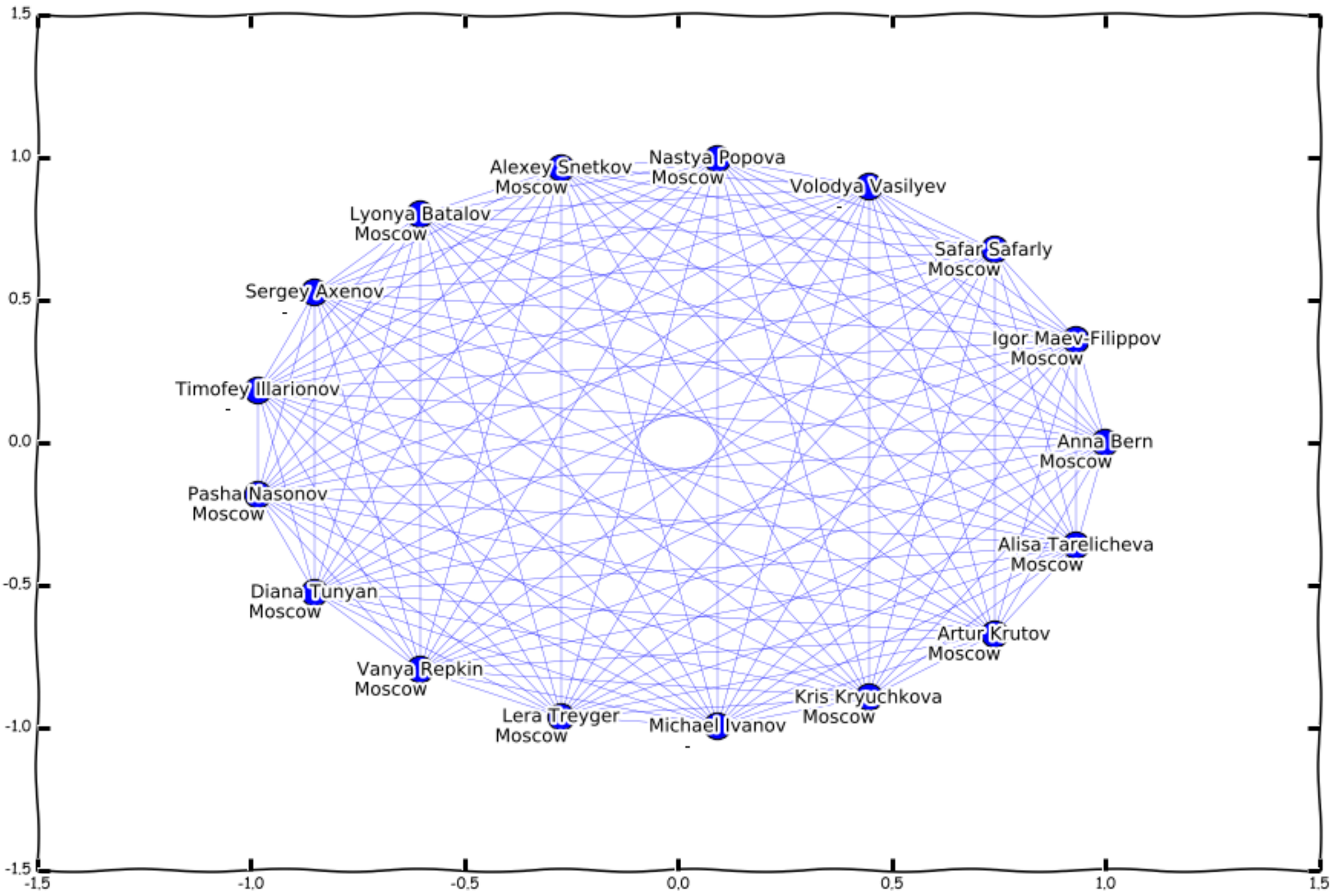
```
print("Graph clique number of 'Software Culture' network is: {}"
     .format(nx.graph_clique_number(GCC)))
```

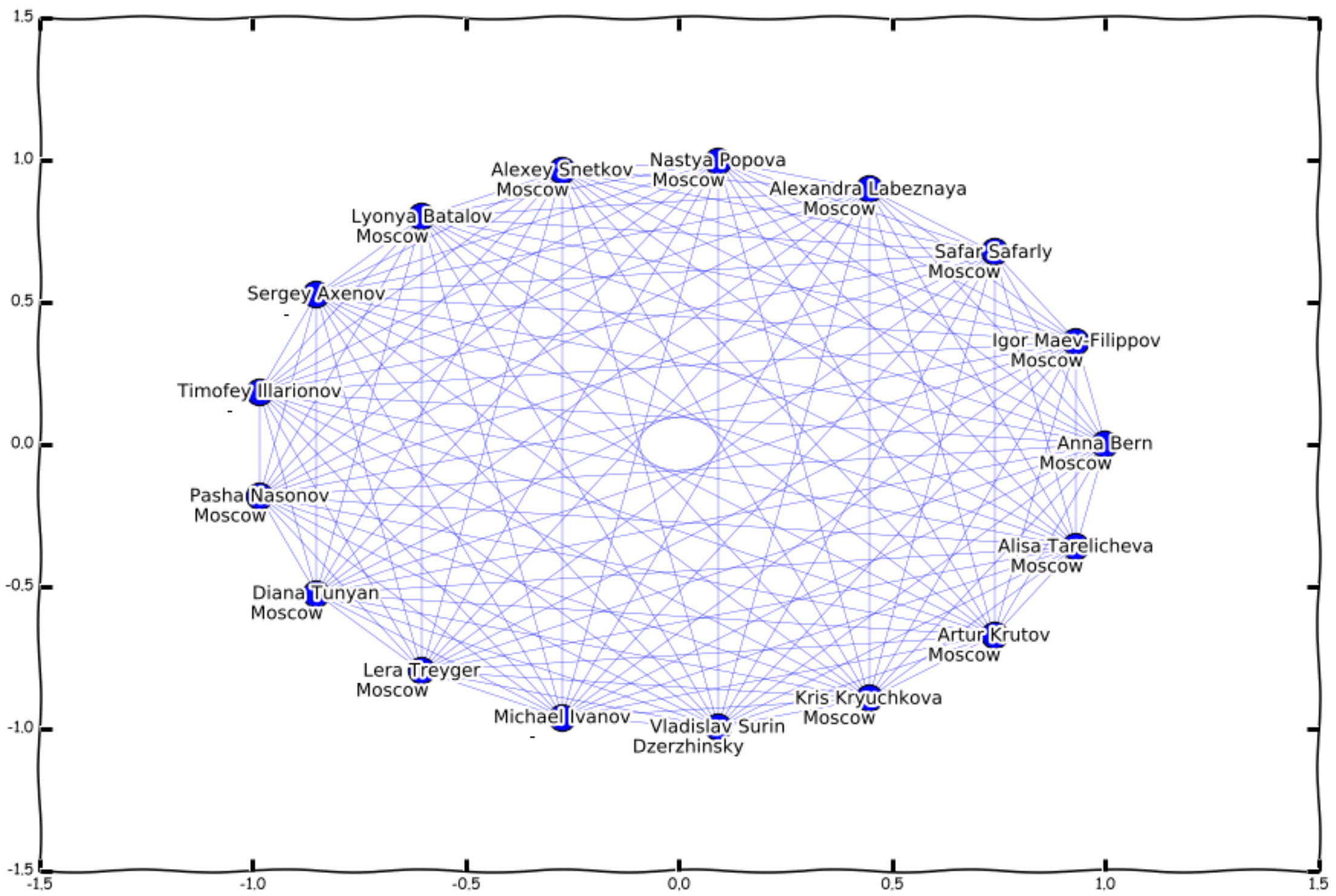
```
Graph clique number of 'Software Culture' network is: 17
```

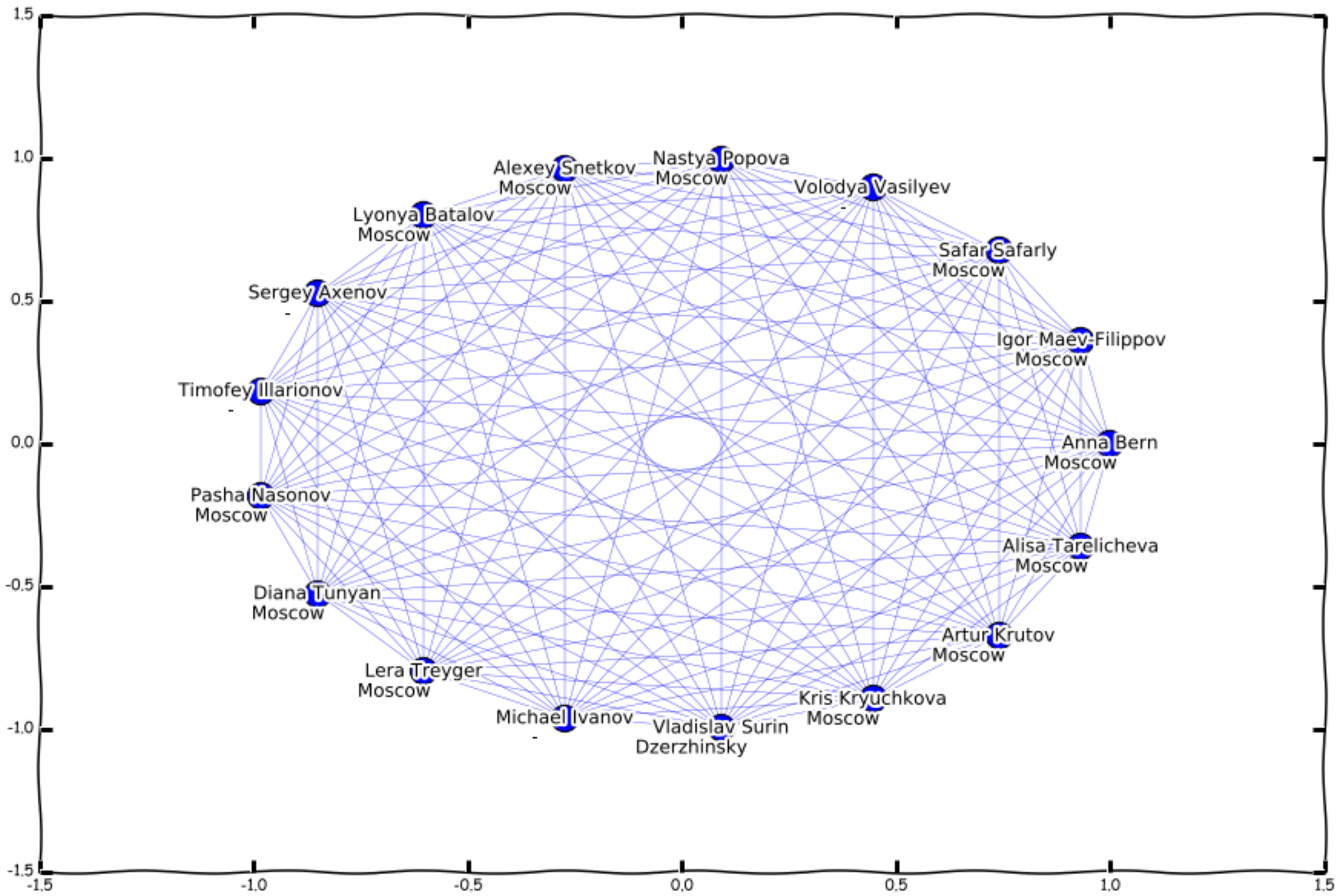
Let's visualize all maximum cliques of 'Software Culture' network:

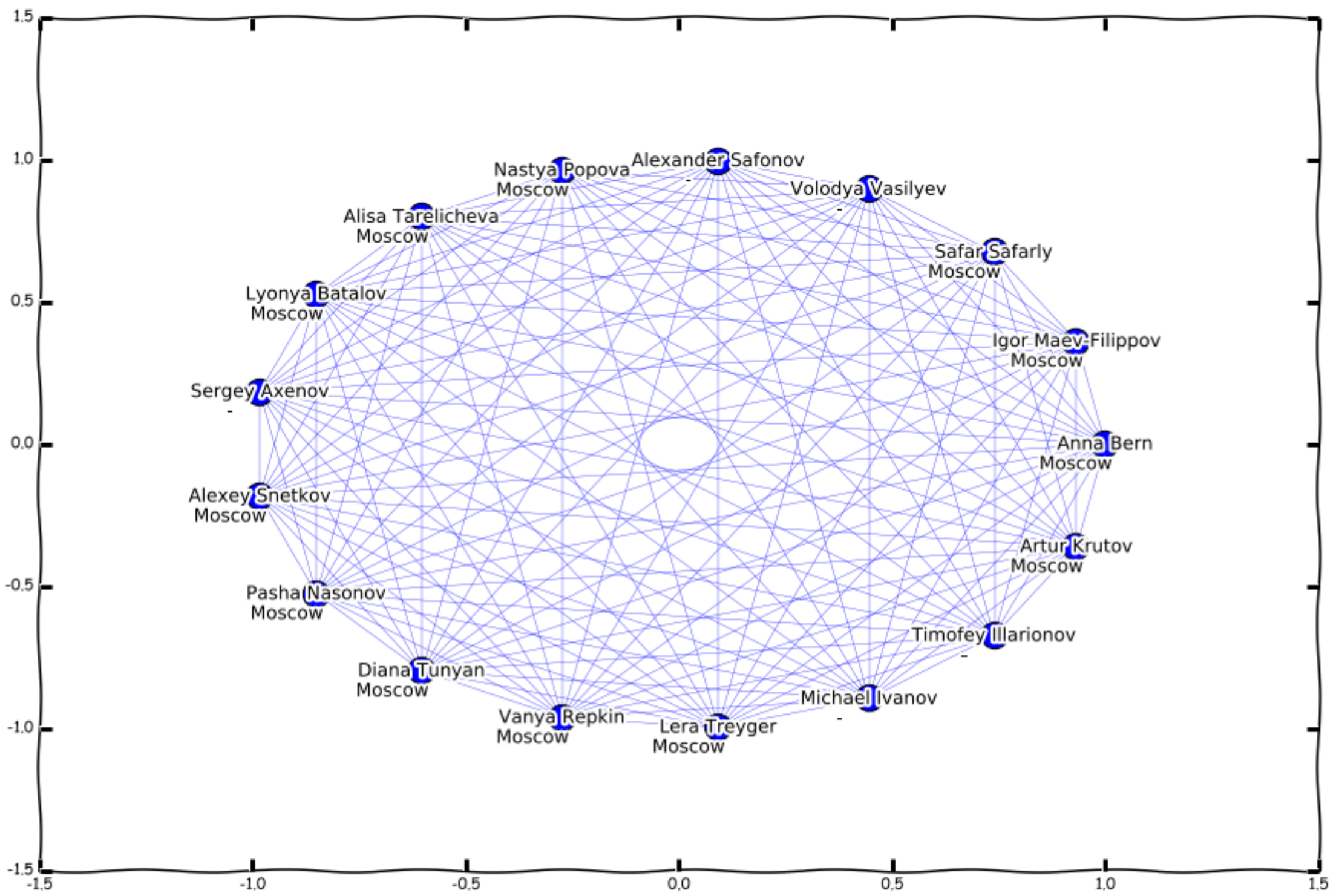
```
for i in iGCC.largest_cliques():
    maximum_clique = []
    for j in i:
        maximum_clique.append(int(iGCC.vs(j)['name'][0]))
```

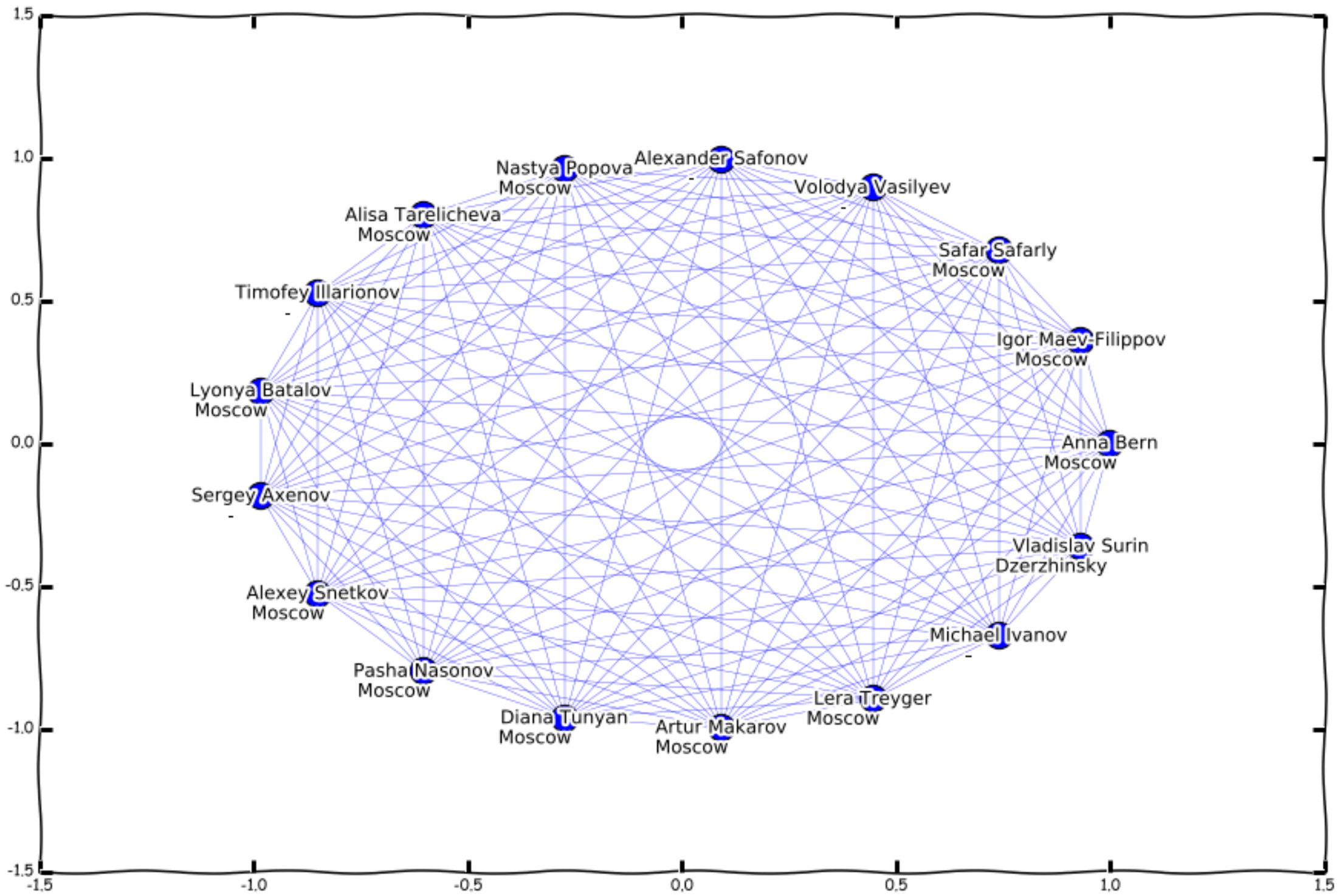
```
maximum_clique_subgraph = GCC.subgraph(maximum_clique)
plt.figure(figsize=(15, 10))
pos1 = nx.circular_layout(maximum_clique_subgraph)
labels1 = nx.get_node_attributes(maximum_clique_subgraph, 'name')
nx.draw_networkx(maximum_clique_subgraph, pos1, with_labels = False, node_color = 'blue',
                 edge_color = 'blue', width = 0.3, font_size = 13)
nx.draw_networkx_labels(maximum_clique_subgraph, pos1, labels1)
pos2 = {k: v-0.07 for k, v in pos1.items()}
labels2 = nx.get_node_attributes(maximum_clique_subgraph, 'city')
nx.draw_networkx_labels(maximum_clique_subgraph, pos2, labels2)
```

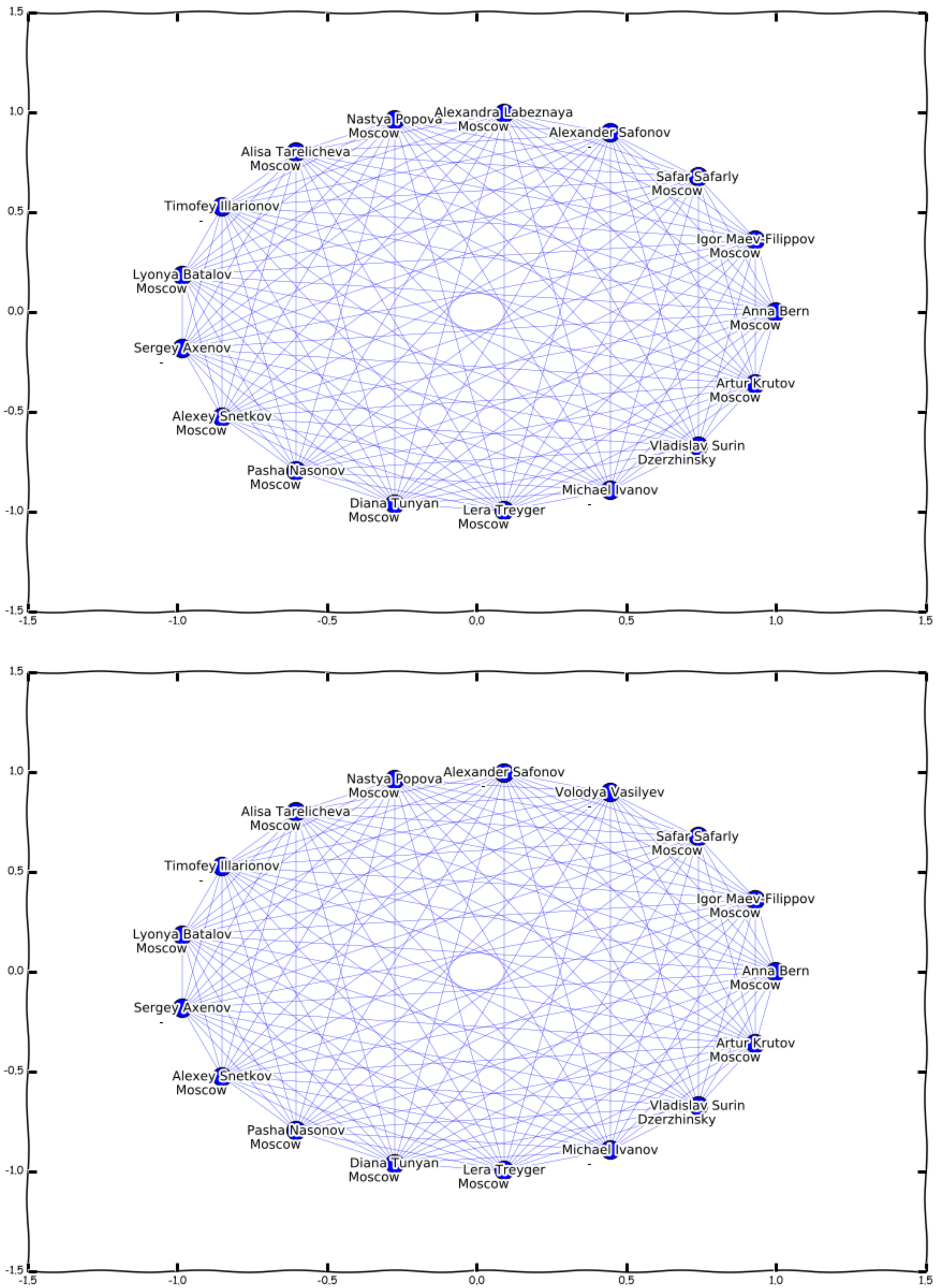



Alexey Snetkov
Moscow
Nastya Popova
Moscow
Lyonya Batalov
Moscow
Alexandra Labeznaya
Moscow
Safar Safarly
Moscow
Sergey Axenov
-
Igor Maev-Filippov
Moscow
Timofey Illarionov
-
Anna Bern
Moscow
Pasha Nasonov
Moscow
Alisa Tarelicheva
Moscow
Diana Tunyan
Moscow
Artur Krutov
Moscow
Lera Treyger
Moscow
Kris Kryuchkova
Moscow
Michael Ivanov
-
Vladislav Surin
Dzerzhinsky
1.5
1.0
0.5
0.0
-0.5
-1.0
-1.5
-1.5
-1.0
-0.5
0,0
0,5
1,0
1.5


Alexey Snetkov
Moscow
Nastya Popova
Moscow
Volodya Vasilyev
-
Lyonya Batalov
Moscow
Safar Safarly
Moscow
Sergey Axenov
-
Igor Maev-Filippov
Moscow
Timofey Illarionov
-
Anna Bern
Moscow
Pasha Nasonov
Moscow
Alisa Tarelicheva
Moscow
Diana Tunyan
Moscow
Artur Krutov
Moscow
Lera Treyger
Moscow
Kris Kryuchkova
Moscow
Michael Ivanov
-
Vladislav Surin
Dzerzhinsky
1.5
1.0
0.5
0.0
-0.5
-1.0
-1.5
-1.5
-1.0
-0.5
0,0
0,5
1,0
1.5


Nastya Popova
Moscow
Alexander Safonov
-
Volodya Vasilyev
-
Alisa Tarelicheva
Moscow
Safar Safarly
Moscow
Lyonya Batalov
Moscow
Igor Maev-Filippov
Moscow
Sergey Axenov
-
Anna Bern
Moscow
Alexey Snetkov
Moscow
Artur Krutov
Moscow
Pasha Nasonov
Moscow
Timofey Illarionov
-
Diana Tunyan
Moscow
Michael Ivanov
-
Vanya Repkin
Moscow
Lera Treyger
Moscow
1.5
1.0
0.5
0.0
-0.5
-1.0
-1.5
-1.5
-1.0
-0.5
0,0
0,5
1,0
1.5


Nastya Popova
Moscow
Alexander Safonov
-
Volodya Vasilyev
-
Alisa Tarelicheva
Moscow
Safar Safarly
Moscow
Timofey Illarionov
-
Igor Maev-Filippov
Moscow
Lyonya Batalov
Moscow
Anna Bern
Moscow
Sergey Axenov
-
Vladislav Surin
Dzerzhinsky
Alexey Snetkov
Moscow
Michael Ivanov
-
Pasha Nasonov
Moscow
Lera Treyger
Moscow
Diana Tunyan
Moscow
Artur Makarov
Moscow
1.5
1.0
0.5
0.0
-0.5
-1.0
-1.5
-1.5
-1.0
-0.5
0,0
0,5
1,0
1.5



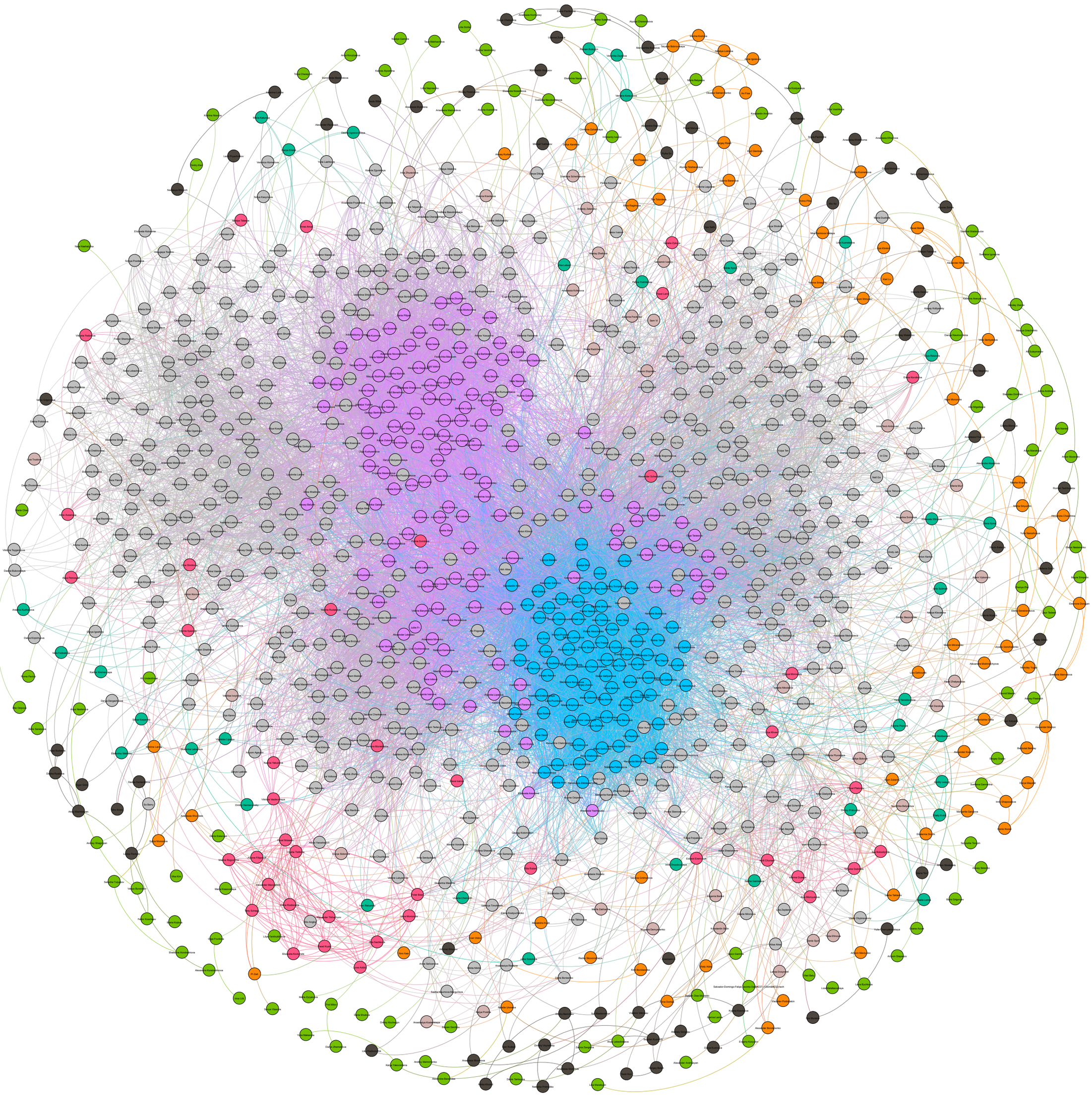
Unfortunately, these users didn't specify their university, but we determined that all these cliques are formed from Moscow Architectural Institute students of 5th course, many of whom also worked in 'Software Culture' project.

As we saw clique concept is very strict. For real analysis can be better to use some relaxations of the clique concept. Let's focus on **k-core** decomposition. A k-core is a maximal subgraph that contains nodes of degree k or more. The core number of a node is the largest value k of a k-core containing that node.

Let's draw k-core decomposition for 'Software Culture' network:

```
core_dict = nx.core_number(GCC)
nx.set_node_attributes(GCC, 'Core', core_dict)
nx.write_gml(GCC, 'GCC.gml')
```

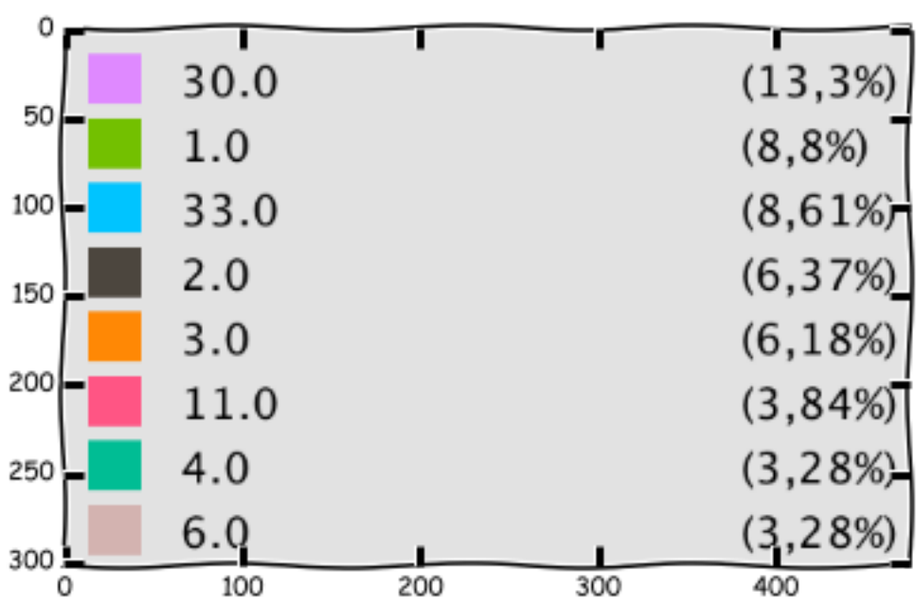
```
SVG(filename='graph_k_core.svg')
```



This is palette, that shows correspondence between color and most common core number:

```
palette = mpimg.imread('graph_k_core_palette.png')
plt.imshow(palette)
```

```
<matplotlib.image.AxesImage at 0x11b57fc50>
```



The k-core decomposition allows the progressive pruning of the networks and the identification of subgraphs of increasing centrality. These subgraphs have the property of being more and more densely connected, and therefore of presenting more and more robust routing capabilities. The study of the obtained subgraphs uncovers the main hierarchical layers of the network and allows for their statistical characterization.

- Community detection

For community detection of 'Software Culture' network we will use module igraph.

Igraph provides 10 powerful community detection algorithms. Each of them works in a different way. Let's go for several algorithms and compare their performances with the help of modularity. This metrics measures the strength of division of a network into modules. Networks with high modularity have dense connections between the nodes within modules but sparse connections between nodes in different modules. Modularity is basically the fraction of the edges that fall within the given groups minus the expected such fraction if edges were distributed at random. So the higher the better.

Let's choose the best community detection algorithm for 'Software Culture' network from several igraph algorithms:

- **edge.betweenness.community algorithm:** It is a hierarchical decomposition process where edges are removed in the decreasing order of their edge betweenness scores (i.e. the number of shortest paths that pass through a given edge). This is motivated by the fact that edges connecting different groups are more likely to be contained in multiple shortest paths simply because in many cases they are the only option to go from one group to another. This method yields good results but is very slow because of the computational complexity of edge betweenness calculations and because the betweenness scores have to be re-calculated after every edge removal. We won't implement this algorithm because of its slow speed.

- **fastgreedy.community algorithm:** It is another hierarchical approach, but it is bottom-up instead of top-down. It tries to optimize a quality function of modularity in a greedy manner. Initially, every vertex belongs to a separate community, and communities are merged iteratively such that each merge is locally optimal (i.e. yields the largest increase in the current value of modularity). The algorithm stops when it is not possible to increase the modularity any more. The method is fast and it is the method that is usually tried as a first approximation because it has no parameters to tune. However, it is known to suffer from a resolution limit, i.e. communities below a given size threshold will always be merged with neighboring communities.

- **walktrap.community algorithm:** It is an approach based on random walks. The general idea is that if you perform random walks on the graph, then the walks are more likely to stay within the same community because there are only a few edges that lead outside a given community. Walktrap runs short random walks of 3-4-5 steps (depending on one of its parameters) and uses the results of these random walks to merge separate communities in a bottom-up manner like fastgreedy.community. It is a bit slower than the fast greedy approach but also a bit more accurate (according to the original publication).

- **spinglass.community algorithm:** It is an approach from statistical physics, based on the so-called Potts model. In this model, each particle (i.e. vertex) can be in one of c spin states, and the interactions between the particles (i.e. the edges of the graph) specify which pairs of vertices would prefer to stay in the same spin state and which ones prefer to have different spin states. The model is then simulated for a given number of steps, and the spin states of the particles in the end define the communities. The method is not particularly fast and not deterministic (because of the simulation itself), but has a tunable resolution parameter that determines the cluster sizes. We won't implement this algorithm because of its slow speed.

- **leading.eigenvector.community algorithm:** It is a top-down hierarchical approach that optimizes the modularity function again. In each step, the graph is split into two parts in a way that the separation itself yields a significant increase in the modularity. The split is determined by evaluating the leading eigenvector of the so-called modularity matrix, and there is also a stopping condition which prevents tightly connected groups to be split further. Due to the eigenvector calculations involved, it might not work on degenerate graphs where the ARPACK eigenvector solver is unstable. Unfortunately, it doesn't work on our network.

- **label.propagation.community algorithm:** It is a simple approach in which every node is assigned one of k labels. The method then proceeds iteratively and re-assigns labels to nodes in a way that each node takes the most frequent label of its neighbors in a synchronous manner. The method stops when the label of each node is one of the most frequent labels in its neighborhood. It is very fast but yields different results based on the initial configuration (which is decided randomly).

- **multilevel.community algorithm:** This is a bottom-up algorithm: initially every vertex belongs to a separate community, and vertices are moved between communities iteratively in a way that maximizes the vertices' local contribution to the overall modularity score. When a consensus is reached (i.e. no single move would increase the modularity score), every community in the original graph is shrank to a single vertex (while keeping the total weight of the adjacent edges) and the process continues on the next level. The algorithm stops when it is not possible to increase the modularity any more after shrinking the communities to vertices. This algorithm is said to run almost in linear time on sparse graphs.

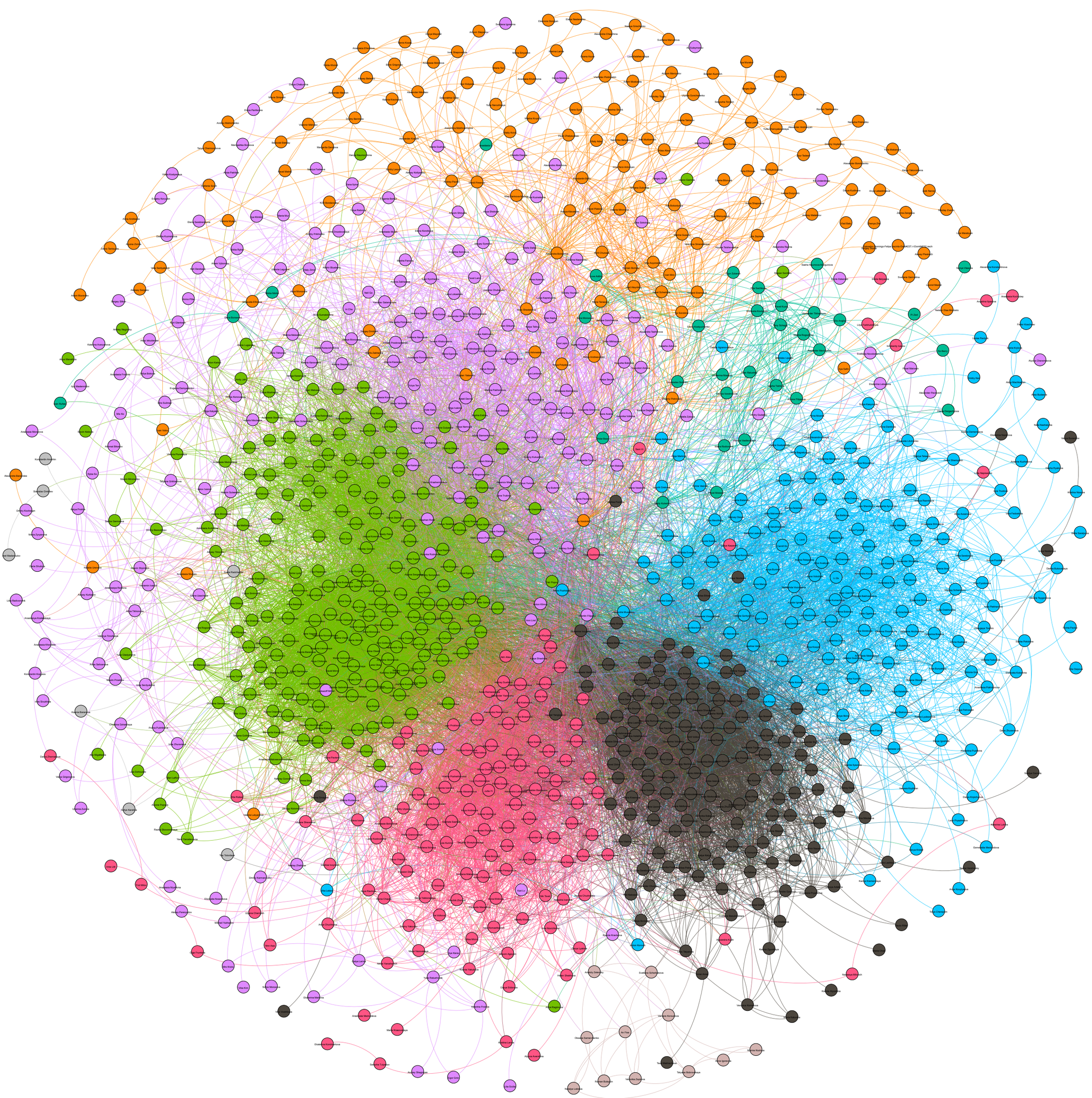
```
## fastgreedy.community algorithm
fg_dendrogram = iGCC.community_fastgreedy()
fg = fg_dendrogram.as_clustering()
print('Modularity for fastgreedy.community algorithm is: {}'.format(fg.modularity))

## walktrap.community algorithm
wt_dendrogram = iGCC.community_walktrap()
wt = wt_dendrogram.as_clustering()
print('Modularity for walktrap.community algorithm is: {}'.format(wt.modularity))

## label.propagation.community algorithm
lp = iGCC.community_label_propagation()
print('Modularity for label.propagation.community algorithm is: {}'.format(lp.modularity))

## multilevel.community algorithm
ml = iGCC.community_multilevel()
print('Modularity for multilevel.community algorithm is: {}'.format(ml.modularity))
```

```
Modularity for fastgreedy.community algorithm is: 0.4783245150469931
Modularity for walktrap.community algorithm is: 0.5101460475823336
Modularity for label.propagation.community algorithm is: 0.24152594886468964
Modularity for multilevel.community algorithm is: 0.5288502257366483
```

So we choose multilevel.community algorithm because of its high modularity:

```
node_dict = {}
for i in range(GCC.number_of_nodes()):
    node_dict[i] = int(iGCC.vs(i)['name'][0])
community_dict = dict(zip(list(node_dict.values()), ml.membership))
nx.set_node_attributes(GCC, 'Community', community_dict)
nx.write_gml(GCC, 'GCC_community.gml')
```

```
SVG(filename='graph_community.svg')
```



Using multilevel.community algorithm we distinguished 9 communities. Let's explain 7 biggest of them:

- **purple community:** mostly graduates and undergraduates from 6 (last) course from Moscow Architectural Institute. In this community we can also find older guys from other creative universities and people, who work/worked in 'Software Culture' project (as creators of 'Software Culture' project are in this community).
- **green community:** mostly undergraduates from Moscow Architectural Institute of 5-6 courses.
- **blue community:** mostly undergraduates from Moscow Architectural Institute of 2 course.
- **black community:** mostly undergraduates from Moscow Architectural Institute of 3 course.
- **orange community:** users from other Moscow universities and regions.
- **pink community:** mostly undergraduates from Moscow Architectural Institute of 4-5 courses.
- **emerald community:** students from Moscow State University of Geodesy and Cartography, where 'Software Culture' organised courses once.

We can see that in our network main splitting is between Moscow Architectural Institute students and users from other cities and universities. But Moscow Architectural Institute students also split on different communities depending on the following factors: age, year of graduation, sociability and so on.

4. Brief analysis of competitors social networks

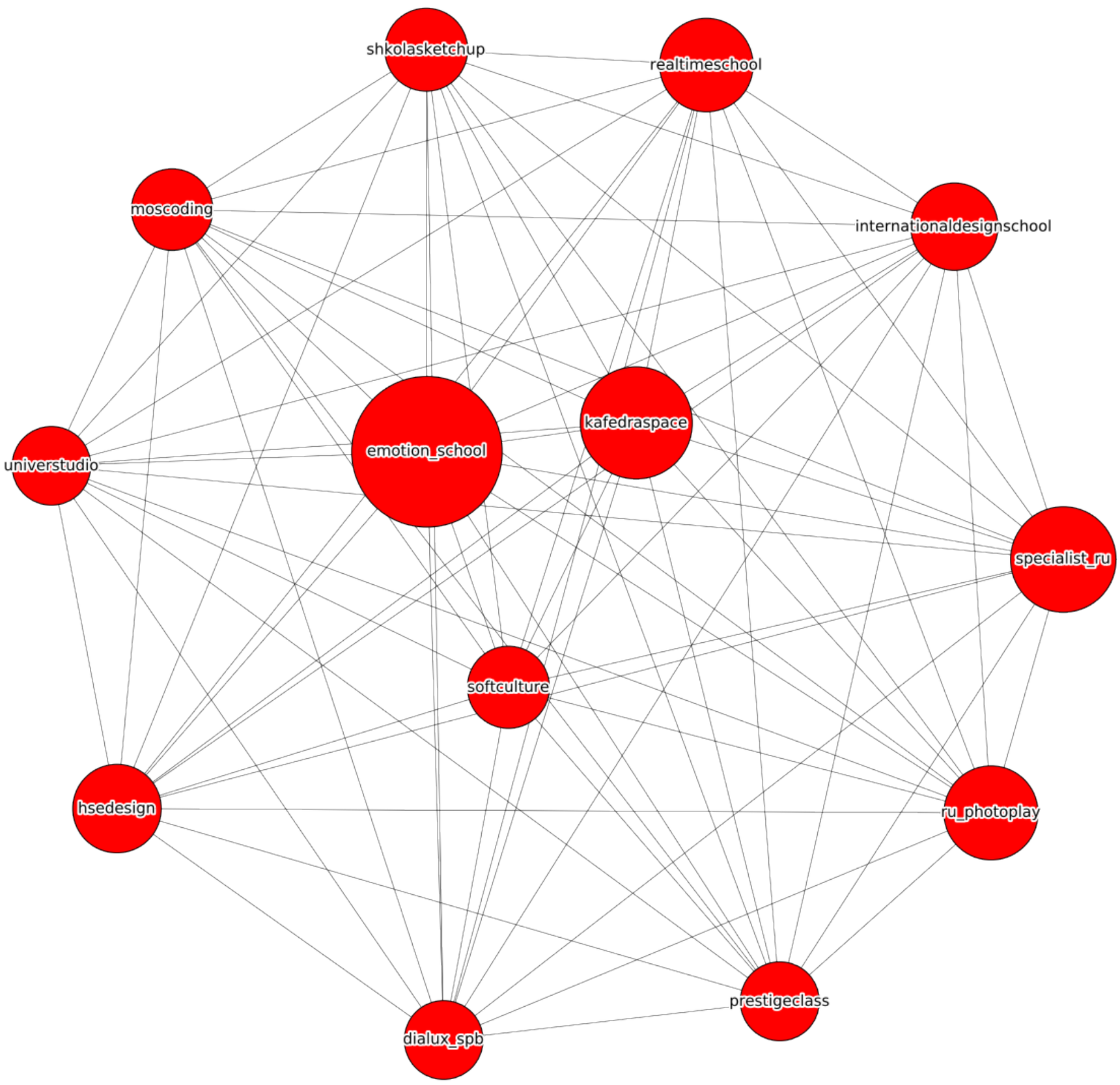
Let's plot such graph where:

- nodes - 'Software Culture' project itself and its competitors
- edges - existence of common subscribers
- the more subscribers of this group, the larger the node
- the more common subscribers between groups, the closer together are situated the nodes

```
## code is taken from https://habrahabr.ru/company/hh/blog/263313/
def getVKMembers(group_id, count=1000, offset=0):
    host = 'http://api.vk.com/method'
    if count > 1000:
        raise Exception('Bad params: max of count = 1000')
    response = requests.get('{host}/groups.getMembers?group_id={group_id}&count={count}&offset={offset}'
                            .format(host=host, group_id=group_id, count=count, offset=offset))
    if not response.ok:
        raise Exception('Bad response code')
    return response.json()

def allCountOffset(func, func_id):
    set_members_id = set()
    count_members = -1
    offset = 0
    while count_members != len(set_members_id):
        response = func(func_id, offset=offset)['response']
        if count_members != response['count']:
            count_members = response['count']
        new_members_id = response['users']
        offset += len(new_members_id)
        if set_members_id | set(new_members_id) == set_members_id != set():
            print('WARNING: break loop', count_members, len(set_members_id))
            break
        set_members_id = set_members_id.union(new_members_id)

    return set_members_id

groups = ['http://vk.com/ru_photoplay',
'http://vk.com/specialist_ru',
'http://vk.com/moscoding',
'http://vk.com/universtudio',
'http://vk.com/softculture',
'http://vk.com/kafedraspace',
'http://vk.com/internationaldesignschool',
'http://vk.com/realtimeschool',
'http://vk.com/dialux_spb',
'http://vk.com/prestigeclass',
'http://vk.com/hsedesign',
'http://vk.com/emotion_school',
'http://vk.com/shkolasketchup']

members = {}
for g in groups:
    name = g.split('http://vk.com/')[1]
    members[name] = allCountOffset(getVKMembers, name)

matrix = {}

for i in members:
    for j in members:
        if i != j:
            matrix[i+j] = len(members[i] & members[j]) * 1.0/ min(len(members[i]), len(members[j]))

max_matrix = max(matrix.values())
min_matrix = min(matrix.values())
```

```
for i in matrix:
    matrix[i] = (matrix[i] - min_matrix) / (max_matrix - min_matrix)

g = nx.Graph(directed=False)
for i in members:
    for j in members:
        if i != j:
            g.add_edge(i, j, weight=matrix[i+j])

members_count = {x:len(members[x]) for x in members}

max_value = max(members_count.values()) * 1.0
size = []
max_size = 1600
min_size = 600
for node in g.nodes():
    size.append(((members_count[node]/max_value)*max_size + min_size)*10)
```

```
pos = nx.spring_layout(g)
plt.figure(figsize=(25, 25))
nx.draw_networkx(g, pos, node_size=size, width=0.5, font_size=15)
plt.axis('off')
plt.show()
```



From graph above we can see that 'Software Culture' has average number of subscribers among its competitors, but it has

room to grow and develop. For this purpose, perhaps, it makes sense to look at the most distant groups to 'sneak' their unique subscribers to gain a new audience.