Social Network Analysis

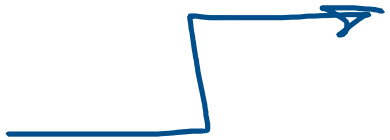
Friends graph of Falchikova Veronika

Constructing the graph using vk api

```
def get_friends_ids(user_id):
    friends_url = 'https://api.vk.com/method/friends.get?user_id={}'
    json_response = requests.get(friends_url.format(user_id)).json()
    if json_response.get('error'):
        print(json_response.get('error'))
        return list()
    return json_response['response']['items']

friends = requests.get('https://api.vk.com/method/friends.get?v=5.52&access_token=
friend_ids = friends['response']['items']
graph = {}
for friend_id in friend_ids:
    graph[friend_id] = get_friends_ids(friend_id)

g = networkx.Graph(directed=False)
for i in graph:
    g.add_node(i)
    for j in graph[i]:
        if i != j and i in friend_ids and j in friend_ids:
            g.add_edge(i, j)
```

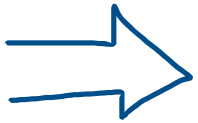


Request for a list of friends for a certain user id

Initial request for my own list of friends

Saving lists of friends of every friend

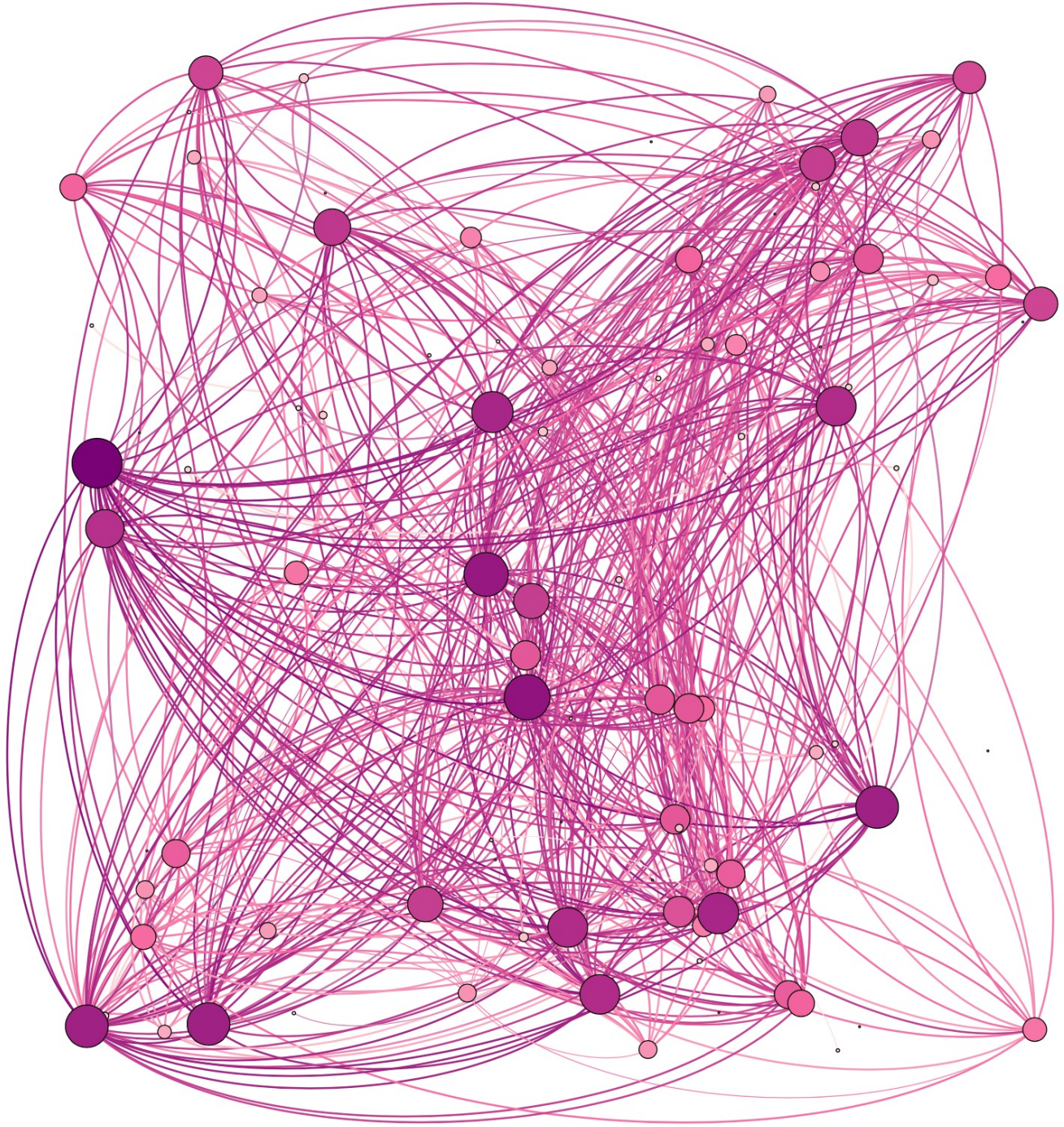
Adding an edge if two of my friends are also each other's friends



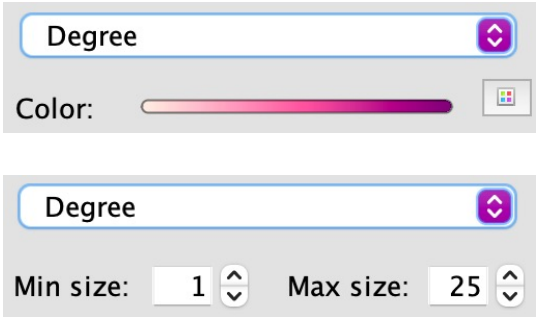
g – constructed graph of friends

Nodes = 102
Edges = 537

Attributes: id, first_name, last_name, screen_name, sex, bdate

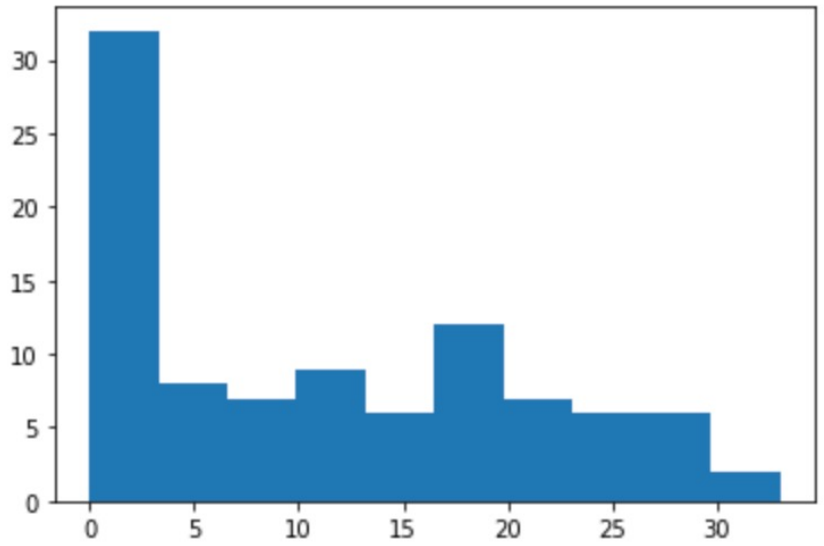


Size and depth of color both depend on degree of the node



Degree distribution

```
degrees = [g.degree(v) for v in g.nodes()]
plt.hist(degrees)
plt.show()
```

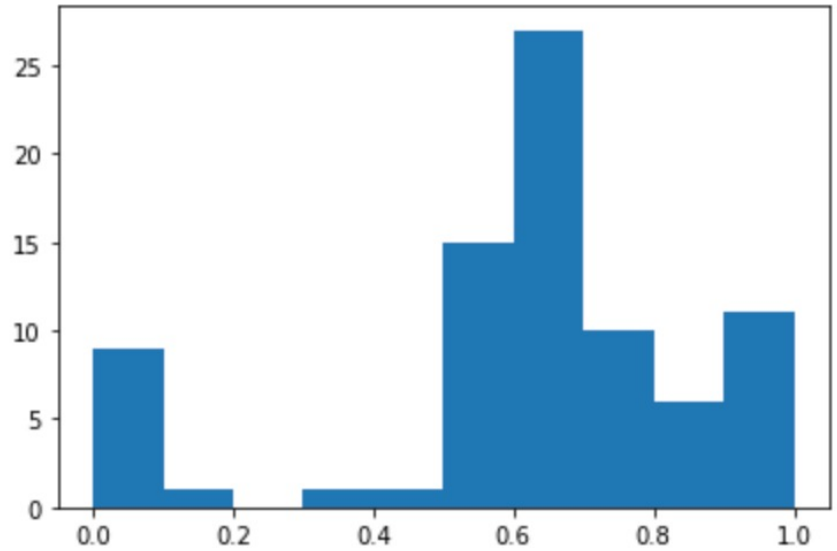


Node degree is the number of edges adjacent to the node.

Clustering coefficient distribution

$$c_u = \frac{2T(u)}{deg(u)(deg(u)-1)},$$

```
[41] clust_coef = [nx.clustering(g, v) for v in g.nodes()]
     plt.hist(clust_coef)
     plt.show()
```



Average Clustering coefficient

```
[37] nx.average_clustering(g)

     0.6175517397332378
```

$$C = \frac{1}{n}\sum_{v \in G} c_v,$$

```
[48] pathlengths = []

    for v in g.nodes():
        spl = dict(nx.single_source_shortest_path_length(g, v))
        for p in spl:
            pathlengths.append(spl[p])
    print(f"average shortest path length: {sum(pathlengths) / len(pathlengths)}")

    # histogram of path lengths
    dist = {}
    for p in pathlengths:
        if p in dist:
            dist[p] += 1
        else:
            dist[p] = 1

    print()
    print("length-paths")
    verts = dist.keys()
    for d in sorted(verts):
        print(f"{d} {dist[d]}")
```

```
average shortest path length: 2.43639080889221

length-paths
0 81
1 1074
2 1814
3 1388
4 810
5 180
6 6
```

Average shortest path length

Shortest path distribution

```
print(f"eccentricity: {nx.eccentricity(S)}")
print(f"radius: {nx.radius(S)}")
print(f"diameter: {nx.diameter(S)}")
```

```
eccentricity: {'8867228': 4, '31144969': 5, '68474465': 4,
radius: 3
diameter: 6
```

The **eccentricity** of a graph vertex v in a connected graph G is the maximum graph distance between v and any other vertex u of G. For a disconnected graph, all vertices are defined to have infinite eccentricity.

Maximum graph eccentricity is the graph **diameter**.

Minimum graph eccentricity is the graph **radius**.

Degree centrality for a node v is the fraction of nodes it is connected to.

```
centr = nx.degree_centrality(g)
print("id =", max(centr, key=centr.get))
print("max degree centrality =", centr[max(centr, key=centr.get)])
```

```
id = 334940037
max degree centrality = 0.351063829787234 05
```

Betweenness centrality of a node v is the sum of the fraction of all-pairs shortest paths that pass through v

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

```
between = nx.betweenness_centrality(g)
print("id =", max(between, key=between.get))
print("max between centrality =", between[max(between, key=between.get)])
```

```
id = 289797558
max between centrality = 0.27584895809631854
```

Closeness centrality of a node u is the reciprocal of the sum of the shortest path distances from u to all n – 1 other nodes.

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)},$$

```
close = nx.closeness_centrality(g)
print("id =", max(close, key=close.get))
print("max closeness centrality =", close[max(close, key=close.get)])
```

```
id = 289797558
max closeness centrality = 0.46736386584926076
```

has max degree centrality

```
pr = nx.pagerank(g)
pr_sort = sorted(pr, key=pr.get, reverse=True)
for i in pr_sort:
  print(i, pr[i])
```

```
289797558 0.02288649243987907 2
334940037 0.02177770413271671 5
99727622 0.02050067890886505 4
```

has max betweenness and closeness centrality

Top 3 ranked nodes

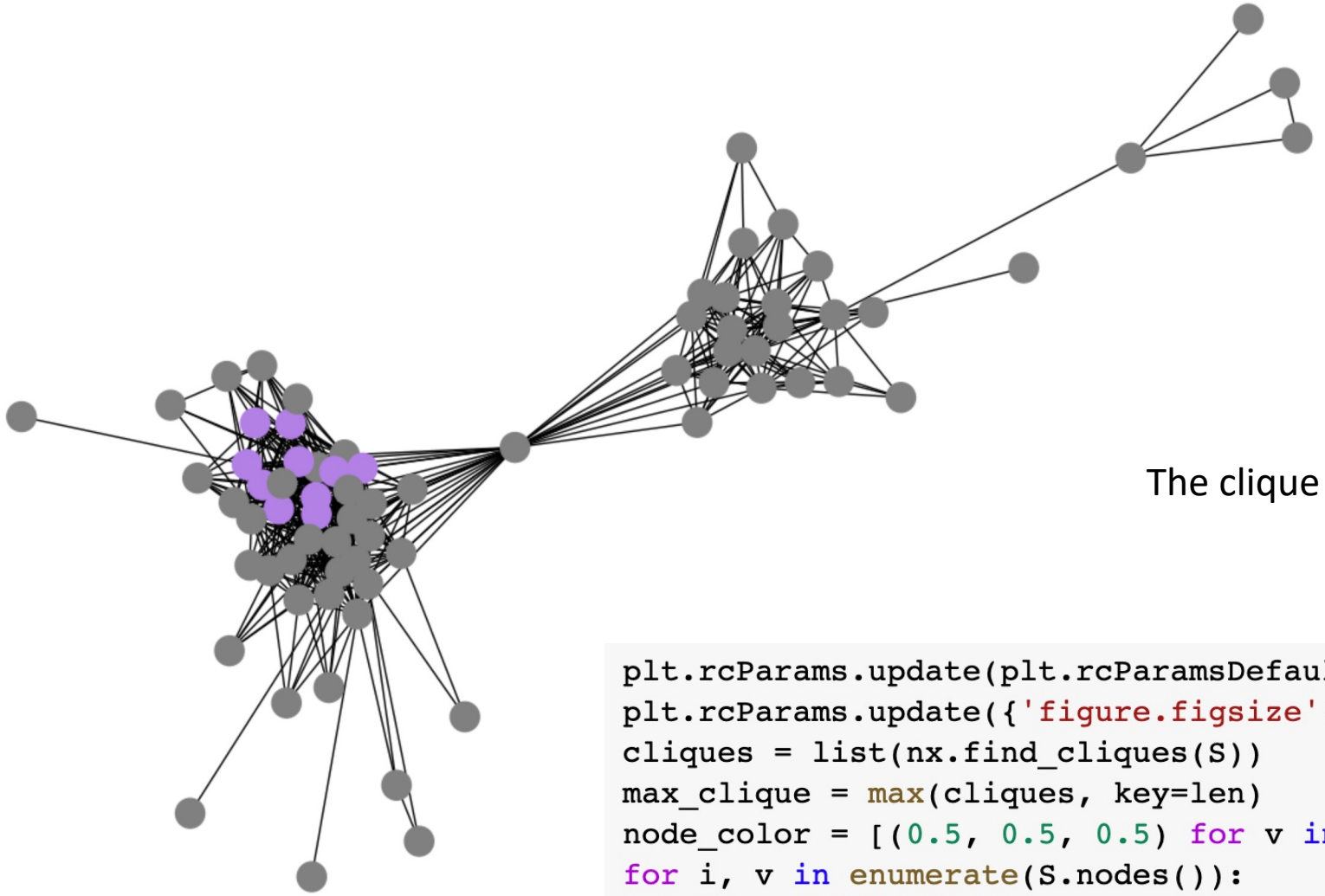
The **PageRank** algorithm measures the importance of each node within the graph, based on the number of incoming relationships and the importance of the corresponding source nodes. It was originally designed as an algorithm to rank web pages.

The underlying assumption is that a page is only as important as the pages that link to it.

PageRank is introduced in the original Google paper as a function that solves the following equation:

$$PR(A) = (1-d) + d(\frac{PR(T_1)}{C(T_1)} + ... + \frac{PR(T_n)}{C(T_n)})$$

A **clique** in a graph is a subset of vertices, such that every two distinct vertices are adjacent.



```
clique_sizes = [len(i) for i in cliques]
print(max(clique_sizes))
```

```
10
```

Maximal size of a clique in the graph is 10.

The clique of maximal size is colored purple

```
plt.rcParams.update(plt.rcParamsDefault)
plt.rcParams.update({'figure.figsize': (15, 10)})
cliques = list(nx.find_cliques(S))
max_clique = max(cliques, key=len)
node_color = [(0.5, 0.5, 0.5) for v in S.nodes()]
for i, v in enumerate(S.nodes()):
    if v in max_clique:
        node_color[i] = (0.7, 0.5, 0.9)
nx.draw_networkx(S, node_color=node_color, pos=karate_pos, with_labels=False)
```

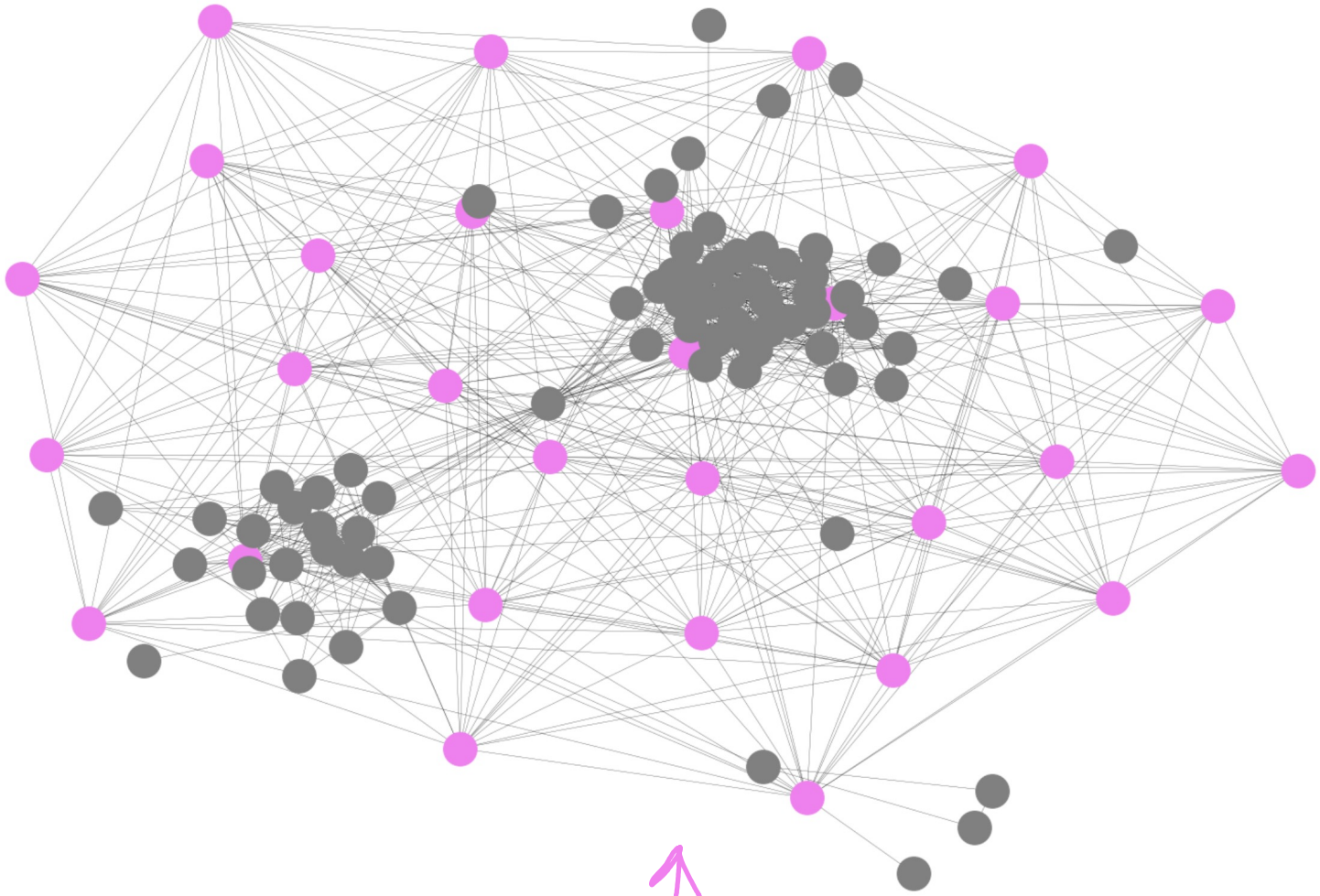
A **k-core** of a graph *G* is the largest subgraph of *G* such that every vertex has k neighbors or more.

The k-core is found by recursively pruning nodes with degrees less than k.

```
k_core_g = nx.k_core(g)
len(list(k_core_g))
```

```
29
```

The largest k that satisfies the mentioned conditions



Here is the k-core

A **k-plex** is a maximal subgraph, in which each vertex is connected to at least n-k other vertices.

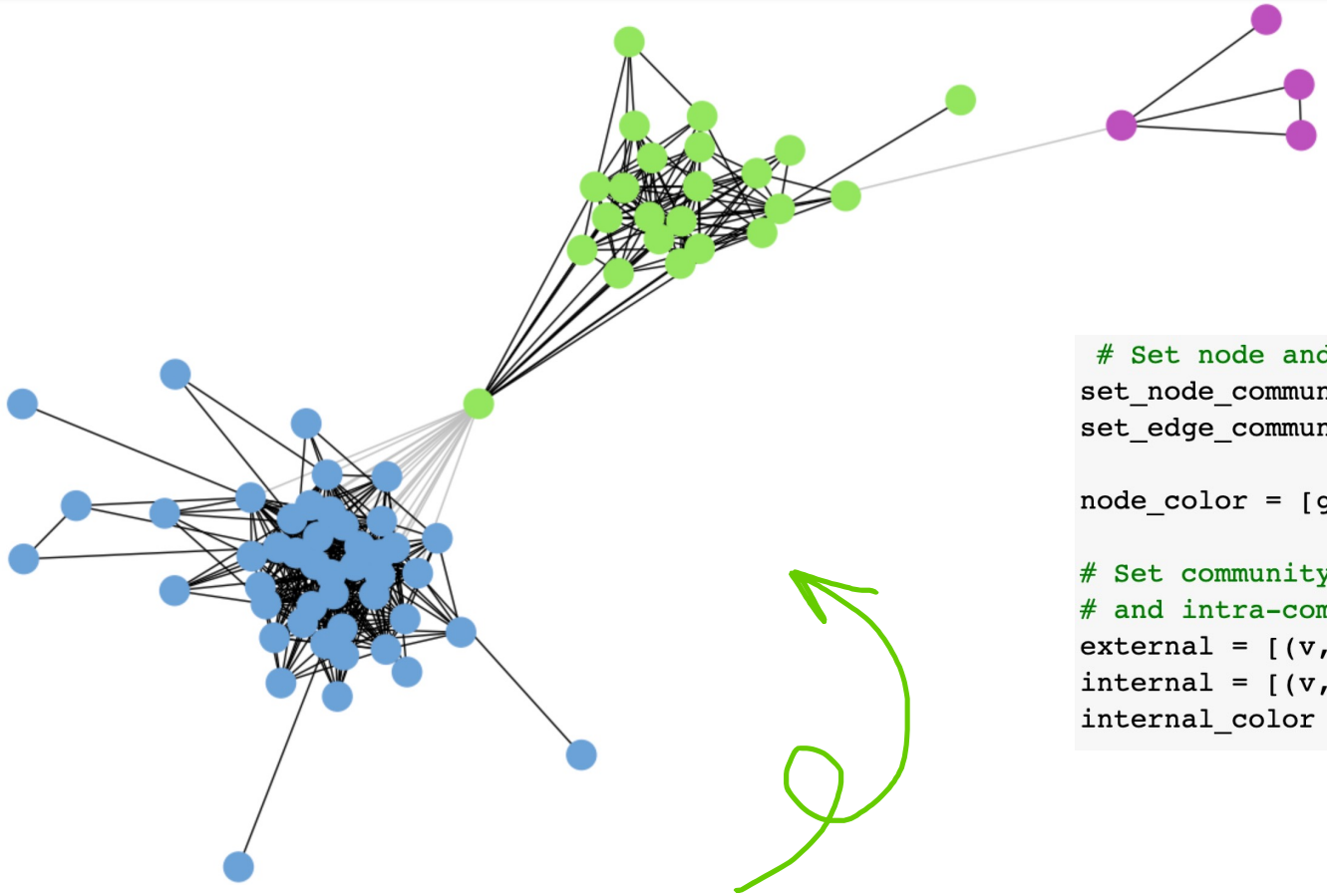
If k = 29, there are no k-plexes in the graph.
Therefore largest k would be 102 – k = 29 => 73.

Community detection
using **Louvain method**

```
#compute the best partition
partition = community.best_partition(S)

size = float(len(set(partition.values())))
pos = nx.spring_layout(S)
count = 0.
plt.figure(figsize=(15, 10))
for com in set(partition.values()) :
    count = count + 1.
    list_nodes = [nodes for nodes in partition.keys()
                                if partition[nodes] == com]
    nx.draw_networkx_nodes(S, pos, list_nodes, node_size = 500,
                                node_color = str(count / size))
```

The nodes are colored
according to their partition



Community detection using Clauset-Newman-Mooore **greedy modularity maximization method**

```
 # Set node and edge communities
set_node_community(S, communities)
set_edge_community(S)

node_color = [get_color(S.nodes[v]['community']) for v in S.nodes]

# Set community color for edges between members of the same community (internal)
# and intra-community edges (external)
external = [(v, w) for v, w in S.edges if S.edges[v, w]['community'] == 0]
internal = [(v, w) for v, w in S.edges if S.edges[v, w]['community'] > 0]
internal_color = ['black' for e in internal]
```

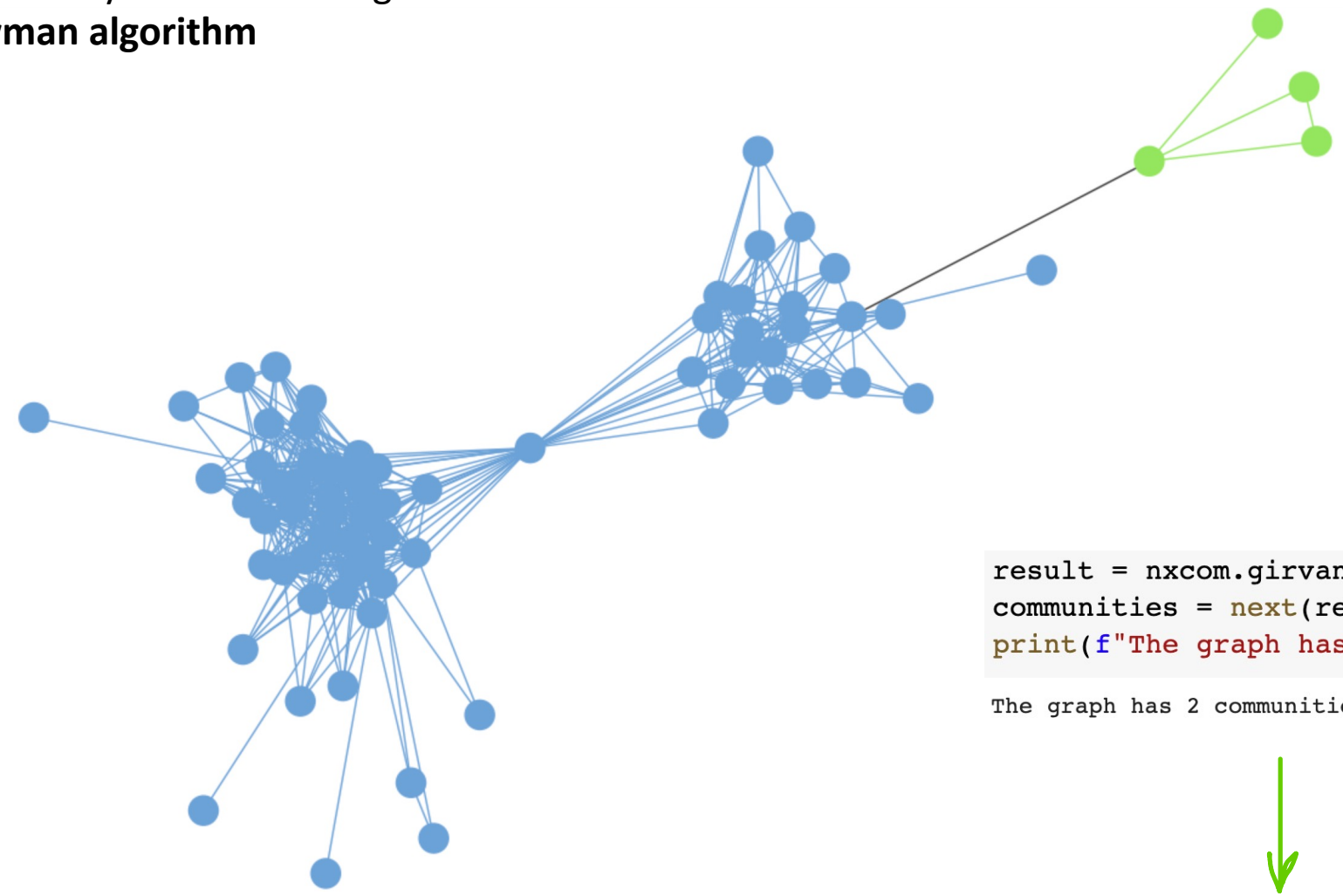
Edges between vertices of the same community are colored **black**, edges between different communities are grey

```
# Find the communities
communities = sorted(nxcom.greedy_modularity_communities(S), key=len, reverse=True)
# Count the communities
print(f"The graph has {len(communities)} communities.")
```

```
The graph has 3 communities.
```

This method detected 3 communities

Community detection using the **Girvan-Newman algorithm**



```
result = nxcom.girvan_newman(S)
communities = next(result)
print(f"The graph has {len(communities)} communities.")
```

```
The graph has 2 communities.
```


This method detected only 2 communities.