documentation

April 28, 2019

[21]:
```
import itertools
import random
import os
import sys

import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
import pandas as pd

import networkx as nx


import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings('ignore')
```

[ ]:
```
# This class represents a directed graph using adjacency list representation.
class Graph:
    def __init__(self, vertices=None, oriented=False):
        self.adjacency_list = defaultdict(list)
        self.oriented = oriented

        if vertices is not None:
            for kvertex, vlist in vertices.items():
                self.adjacency_list[kvertex] = vlist
                #print(kvertex, vlist)
                if oriented:
                    for vertex in vlist:
                        if kvertex not in self.adjacency_list[vertex]:
                            self.adjacency_list[vertex].append(kvertex)

        #print(self.adjacency_list["230612164"])

    def __contains__(self, vertex):
        return vertex in self.adjacency_list.keys()
```

```python
def __getitem__(self, vertex):

    result = set(self.adjacency_list.get(vertex))
    for v, adj_vlist in self.adjacency_list.items():
        if vertex in adj_vlist:
            result.add(v)
    return result

def get_edges(self):
    result = []
    for u in self.adjacency_list.keys():
        l = [(u, v) for v in self[u]
             if (u, v) not in result and
                 (v, u) not in result]
        result.extend(l)
    return result

def get_degrees(self):
    result = {v: len(self[v]) for v in self.adjacency_list.keys()}
    return result

def degree(self, vertex):
    if vertex not in vertex in self:
        raise ValueError("Vertex is not in the graph.")

    return len(self[vertex])

# Function to add an edge to graph.
def add_edge(self, u, v):
    self.adjacency_list[u].append(v)
    if v not in self:
        self.adjacency_list[v] = []
    if self.oriented:
        self.adjacency_list[v].append(u)

# Function that returns reverse (or transpose) of this graph.
def get_transpose(self):
    g = Graph(oriented=self.oriented)
    # Recur for all the vertices adjacent to this vertex.
    for i, i_list in self.adjacency_list.items():
        for j in i_list:
            g.add_edge(j, i)
    return g

# A function used to perform DFS.
def dfs_util(self, v, visited, stack=None, output=False):
```

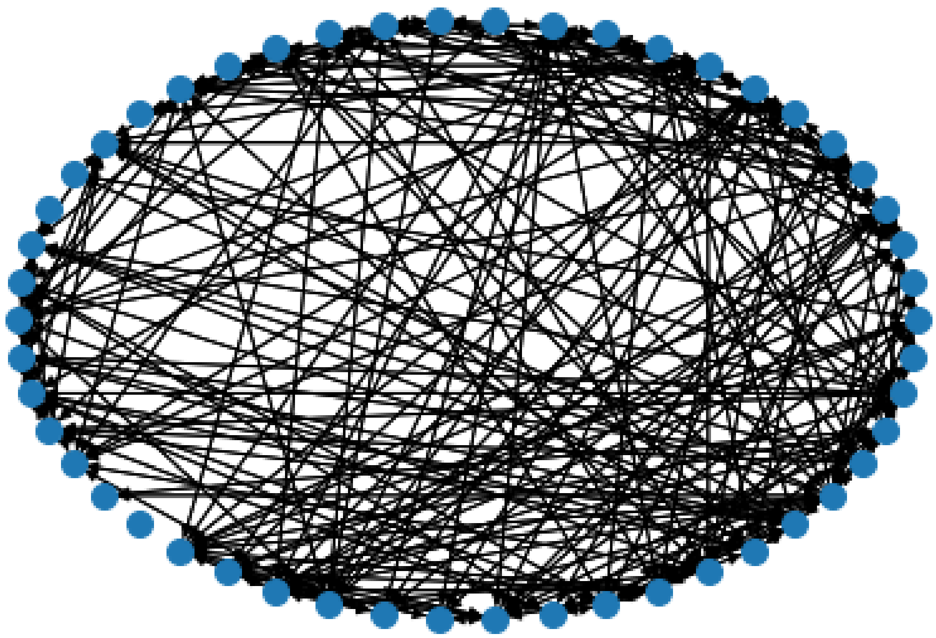
```
        # Mark the current node as visited.
        visited[v] = True
        if output:
            print(v, end=" ")

        # Recur for all the vertices adjacent to this vertex.
        for i in self.adjacency_list[v]:
            if not visited[i]:
                self.dfs_util(i, visited, stack, output)
        if stack is not None:
            stack.append(v)

    # Function that returns true if graph is strongly connected.
    def is_strongly_connected(self):
        return (len(self.get_strongly_connected_components()[0]) ==
                len(self.adjacency_list))

    # Function that finds and prints all strongly connected components.
    def get_strongly_connected_components(self, output=False):
        stack = []
        # Mark all the vertices as not visited (For first DFS).
        visited = {key: False for key in self.adjacency_list.keys()}
        visited = {}
        for key in self.adjacency_list.keys() :
            visited[key] = False
        # Fill vertices in stack according to their finishing times.
        for i in self.adjacency_list.keys():
            if not visited[i]:
                self.dfs_util(i, visited, stack=stack)

        # Create a reversed graph.
        reversed_graph = self.get_transpose()

        # Mark all the vertices as not visited (For second DFS).
        visited = {key: False for key in self.adjacency_list.keys()}

        sc_components = defaultdict(list)
        counter = 0
        # Now process all vertices in order defined by Stack.
        while stack:
            i = stack.pop()
            if not visited[i]:
                i_stack = []
                reversed_graph.dfs_util(i, visited, stack=i_stack,
                                        output=output)
                sc_components[counter].extend(i_stack)
                counter += 1
```

```python
                if output:
                    print()
        return sc_components

    # Function that returns true if graph is strongly connected.
    def is_weakly_connected(self):
        return (len(self.get_weakly_connected_components()[0]) ==
                len(self.adjacency_list))

    # Method to retrieve connected components in graph.
    def get_weakly_connected_components(self, output=False):
        visited = {key: False for key in self.adjacency_list.keys()}
        wc_components = defaultdict(list)
        counter = 0
        for v in self.adjacency_list.keys():
            if not visited[v]:
                v_stack = []
                self.dfs_util(v, visited, stack=v_stack, output=output)
                wc_components[counter].extend(v_stack)
                counter += 1
                if output:
                    print()
        return wc_components

    def get_subgraph(self, vertex_labels):
        subgraph = Graph()
        for vertex_label in vertex_labels:
            vertex_list = self.adjacency_list.get(vertex_label)
            if vertex_list is not None:
                subgraph.adjacency_list[vertex_label].extend(vertex_list)
        return subgraph


def check_graph_connectivity(graph, output=True):
    print(f"Strongly connected: {graph.is_strongly_connected()}")
    print(f"Weakly connected: {graph.is_weakly_connected()}")
    graph.get_strongly_connected_components(output=output)
    if output:
        print()
    graph.get_weakly_connected_components(output=output)
    if output:
        print()
```

[396]:
```python
G = nx.read_gexf('./../vk-friends-137252115.gexf')
def save_from_plt(path, title="", axis=False, xlabel="", ylabel=""):
    if axis:
        plt.xlabel(xlabel)
```

```
        plt.ylabel(ylabel)
        plt.axis("auto")
    else:
        plt.axis("off")
    plt.title(title)
    plt.savefig(path)


nx.draw_circular(G, with_labels=False, node_size=100,
                     label="Graph")
save_from_plt("../task1/original_graph.png", title=" ")
```

Исходный граф



[52]:
```
nx.write_adjlist(G,"./../list_adjacency.csv")
```

[59]:
```
adjacency_matrix = nx.to_numpy_matrix(G)
type(adjacency_matrix), len(adjacency_matrix)
```

[59]: (numpy.matrix, 50)

[237]:
```
adjacency_list = nx.to_dict_of_lists(G)
type(adjacency_list), len(adjacency_list)
```

[237]: (dict, 50)

[39]:
```
from collections import defaultdict
```

[234]:
```
social_graph = Graph(adjacency_list)
sc_components_graph = social_graph.get_strongly_connected_components()
wc_components_graph = social_graph.get_weakly_connected_components()
print(" :", len(social_graph.adjacency_list))
print("   :", len(sc_components_graph))
print("Set of strongly connected component lengths",
      [len(component) for component in sc_components_graph.values()])
print("   :", len(wc_components_graph))
print("     ",
      [len(component) for component in wc_components_graph.values()])
```

```
 : 50
   : 3
Set of strongly connected component lengths [1, 1, 48]
   : 3
      [48, 1, 1]
  :
['56678018', '65714558', '205014908', '108367089', '78700195', '74200453',
'57742251', '43434750', '58383215', '42705793', '35797958', '33966177',
'22248394', '87088313', '319970905', '203437876', '154723906', '151019033',
'93530797', '38202546', '216953513', '177123098', '157062074', '68783210',
'225812577', '247405142', '42041847', '78602687', '78543018', '76411897',
'67446082', '64004147', '36175307', '50390508', '32879395', '373044930',
'4455750', '64059258', '244623361', '145683668', '97287755', '93248647',
'49687517', '40737361', '35472542', '15047022', '19536574', '1386039']
['88988639']
['418567501']
```

[235]:
```
max_sc_component = max(sc_components_graph.values(), key=len)
max_wc_component = max(wc_components_graph.values(), key=len)
print("      :",
      len(max_sc_component))
print("      :",
      len(max_wc_component))
```

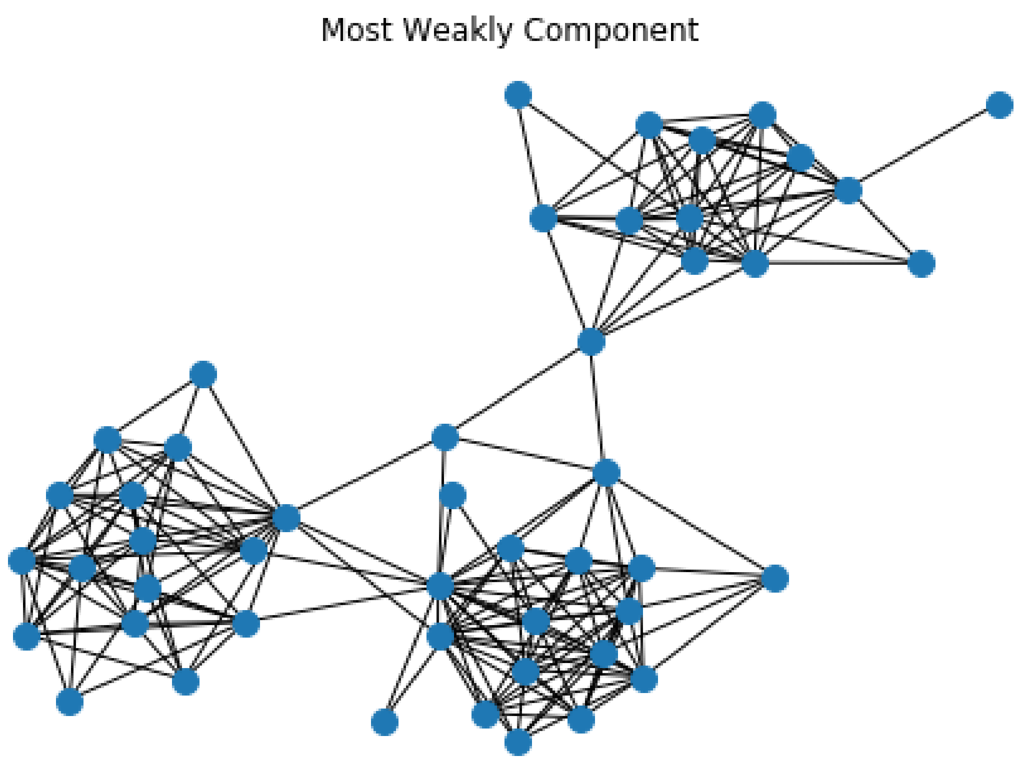
```
      : 48
      : 48
```

1 Task 2

.

[397]:
```
social_subgraph = social_graph.get_subgraph(max_wc_component)

subgraph_nx = nx.Graph(social_subgraph.adjacency_list)
nx.draw_kamada_kawai(subgraph_nx, with_labels=False, node_size=100,
                     label="Most Weakly Component")
```

```
save_from_plt("../task2/social_sub_graph.png","Most Weakly Component")
```



.

[388]:
```
from statistics import mean
from collections import Counter
adjacency_list_main =social_subgraph.adjacency_list
v_degrees = defaultdict(int, ((n, len(v)) for n, v in
                                   adjacency_list_main.items()))
v_degrees_statistic = Counter(v_degrees.values())
average = mean(v_degrees_statistic.keys())
print("  :",average)
fig, ax = plt.subplots(1, 1, figsize=(5, 5))
ax.hist(v_degrees.values(), bins=max(v_degrees.values())+1, density=True)
ax.set_title('     ')
ax.set_xticks(range(max(v_degrees.values())+1))
plt.savefig('../task2/histogram.png')
plt.show()
```

```
 : 8.333333333333334
```

1.0.1

[329]:
```
max_weekly_adjacency_matrix = nx.Graph(social_subgraph.adjacency_list)
matrix_path = nx.to_numpy_matrix(max_weekly_adjacency_matrix, nonedge=np.inf)
print(matrix_path)
```

```
[[inf inf inf ... inf inf inf]
 [inf inf inf ... inf inf inf]
 [inf inf inf ... inf inf inf]
 ...
 [inf inf inf ... inf  1. inf]
 [inf inf inf ...  1. inf  1.]
 [inf inf inf ... inf  1. inf]]
```

- $O(|V|^3)$

[330]:
```
n = matrix_path.shape[0]
for k in range(n):
    for i in range(n):
        for j in range(n):
            matrix_path[i, j] = min(matrix_path[i, k]
                                  + matrix_path[k, j],
                                    matrix_path[i, j])
print(matrix_path)
```

```
[[2. 2. 2. ... 5. 5. 4.]
 [2. 2. 2. ... 6. 6. 5.]
 [2. 2. 2. ... 5. 5. 4.]
 ...
 [5. 6. 5. ... 2. 1. 2.]
 [5. 6. 5. ... 1. 2. 1.]
 [4. 5. 4. ... 2. 1. 2.]]
```

1.0.2

-

[331]:
```python
eccentricity = matrix_path.max(axis=1).astype(int)
print(eccentricity)
```

```
[[6]
 [7]
 [6]
 [5]
 [6]
 [6]
 [5]
 [6]
 [6]
 [5]
 [5]
 [6]
 [5]
 [6]
 [6]
 [5]
 [5]
 [5]
 [4]
 [4]
 [7]
 [6]
 [6]
 [7]
 [7]
 [6]
 [6]
 [6]
 [6]
 [6]
 [6]
 [6]
 [6]
 [6]
```

```
 [6]
 [5]
 [4]
 [6]
 [5]
 [6]
 [6]
 [6]
 [6]
 [6]
 [5]
 [6]
 [6]
 [5]]
```

1.0.3

-

[332]:
```python
radius = np.min(eccentricity)
print(radius)
```

```
4
```

1.0.4

-

[333]:
```python
diameter = int (np.max(eccentricity))
print(diameter)
```

```
7
```

1.0.5

- .

[334]:
```python
array_peref,_ = np.where(eccentricity==diameter)
print(array_peref)
```

```
[ 1 20 23 24]
```

1.0.6

- .

[259]:
```python
array_center, _ = np.where(eccentricity==radius)

print(array_center)
```

```
[18 19 36]
```

1.0.7

[335]:
```
mean_distance =matrix_path.mean()
print('    : {0:.3f}'.format( mean_distance))
```

```
    : 2.982
```

2 Task 3

2.0.1 Common Neighbors

$N(x)$ - x.

Common Neighbors $(x,y) = |N(x) \cap N(y)|$

[231]:
```
social_subgraph = social_graph.get_subgraph(max_wc_component)
max_weekly_adjacency_matrix = nx.Graph(social_subgraph.adjacency_list)
adjacency_matrix_main = nx.
 ↪to_numpy_matrix(max_weekly_adjacency_matrix,dtype=int)
```

[265]:
```
common_neighbors = np.zeros(shape=(n, n), dtype=int)
for i in range(n):
    for j in itertools.chain(range(i), range(i+1, n)):
        common_neighbors[i, j] = common_neighbors[j, i] = \
            np.sum(adjacency_matrix_main[i] & adjacency_matrix_main[j])
```

[265]:
```
numpy.matrix
```

2.0.2 Jaccard's Coefficient ( )

Jaccard's Coefficient $(x,y) = \frac{|N(x) \cap N(y)|}{|N(x) \cup N(y)|}$

[326]:
```
jaccards_coefficient = np.zeros(shape=(n, n), dtype=float)

for i in range(n):
    for j in itertools.chain(range(i), range(i+1, n)):
        intersection = np.sum(adjacency_matrix_main[i] &␣
 ↪adjacency_matrix_main[j])
        union = np.sum(adjacency_matrix_main[i] | adjacency_matrix_main[j])
        jaccards_coefficient[i, j] = jaccards_coefficient[j, i] = np.
 ↪round(intersection/union, 2)
```

2.0.3 Adamic/Adar (Frequency-Weighted Common Neighbors)

Frequency-Weighted Common Neighbors $(x,y) = \sum_{z \in N(x) \cap N(y)} \frac{1}{log(N(z))}$

[300]:
```
fw_common_neighbors = np.zeros(shape=(n, n), dtype=float)

for i in range(n):
    for j in itertools.chain(range(i), range(i+1, n)):
```

```
        intersection = np.
 ↪intersect1d(adjacency_matrix_main[i],adjacency_matrix_main[j])
        intersection = np.squeeze(np.asarray(intersection))
        fw_common_neighbors[i, j] = fw_common_neighbors[j, i] = np.round(\
            np.sum(1 / np.log([np.sum(adjacency_matrix[vertex]) for vertex,␣
 ↪flug in
                    enumerate(intersection) if flug])), 2)
print(adjacency_matrix_main.shape)
```

```
(48, 48)
```

2.0.4 Preferential Attachment

Preferential Attachment $(x, y) = |N(x)| \times |N(y)|$

[160]:
```
preferential_attachment  = np.zeros(shape=(n, n), dtype=int)

for i in range(n):
    for j in itertools.chain(range(i), range(i+1, n)):
        preferential_attachment[i, j] = preferential_attachment[j, i] = \
            np.sum(adjacency_matrix_main[i]) * np.sum(adjacency_matrix_main[j])
```

[ ]:
```
with open('./../report/task 3/preferential attachment.csv', 'w') as f:
    f.write(',' + ','.join(str(x) for x in range(len(preferential_attachment)))␣
 ↪+ '\n')
    for vertex in range(len(preferential_attachment)):
        f.write(str(vertex) + ',' + ','.join(str(x) for x in␣
 ↪preferential_attachment[vertex]) + '\n')
```

2.0.5

[328]:
```
np.savetxt("../task3/common_neighbors.csv",common_neighbors, delimiter=",")
np.savetxt("../task3/jaccards_coefficient.csv",jaccards_coefficient,␣
 ↪delimiter=",")
np.savetxt("../task3/fw_common_neighbors.csv",fw_common_neighbors,␣
 ↪delimiter=",")
np.savetxt("../task3/preferential attachment.csv",preferential_attachment,␣
 ↪delimiter=",")
#pd.DataFrame(preferential_attachment).to_csv("../task1/preferential attachment.
 ↪csv")
```

3 Task 4

[355]:
```
nx_graph = nx.Graph(social_subgraph.adjacency_list)
def draw_graph_with_centrality(G,centrality) :
    n = len(social_subgraph.adjacency_list)
    max_value = max(centrality.values())
```

```
    min_value = min(centrality.values())
    rgba_colors = np.zeros((n,3))
    rgba_colors[:,2] = (np.array([x for x in centrality.values()])-min_value)/
↪(max_value-min_value)
    nx_graph = nx.Graph(social_subgraph.adjacency_list)
    nx.draw_kamada_kawai(nx_graph, cmap=plt.cm.Reds, node_color=rgba_colors,
                         node_size=100, with_labels=False, label="Graph")
```

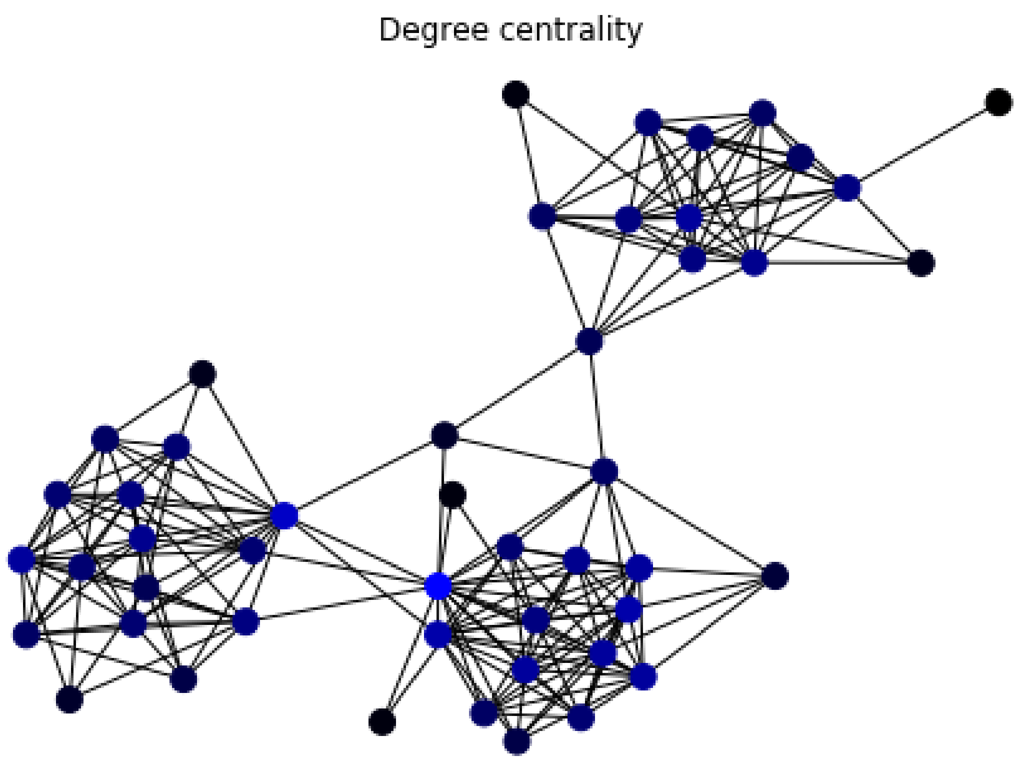
3.0.1 Degree centrality

- .

normal g(v) = $\frac{g(x)-min g(x)}{max g(x)-min g(x)}$,

$d(u,v)$ - u v.

[398]:
```
degree_centrality = {vertex: np.sum(adjacency_matrix_main[vertex]) / (len(G)-1)␣
↪for vertex in range(adjacency_matrix_main.shape[0])}
draw_graph_with_centrality(nx_graph,degree_centrality)
save_from_plt("../task4/degree_centrality.png","Degree centrality")
```



3.0.2 Closeness centrality

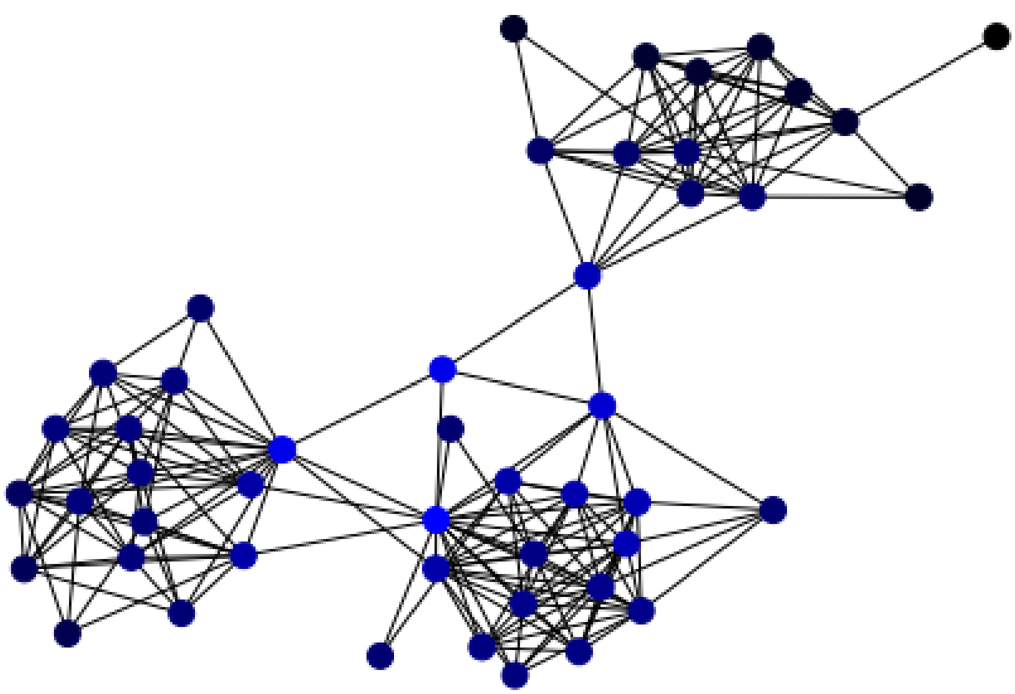
- - 1 .

closeness centrality(v) = $\frac{|V|-1}{\sum_{u \in V(G)} d(u,v)}$,

$d(u,v)$ - u v.

[386]:
```
spmd =matrix_path.copy()
np.fill_diagonal(spmd,0)
closeness_centrality = {vertex: (adjacency_matrix_main.shape[0]-1) / np.
 ↪sum(paths) for vertex, paths in enumerate(spmd)}
draw_graph_with_centrality(nx_graph,closeness_centrality)
save_from_plt("../task4/closeness_centrality.png","Closeness centrality")
```

```
<class 'dict'>
```



3.0.3 Betweenness centrality (nodes)

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

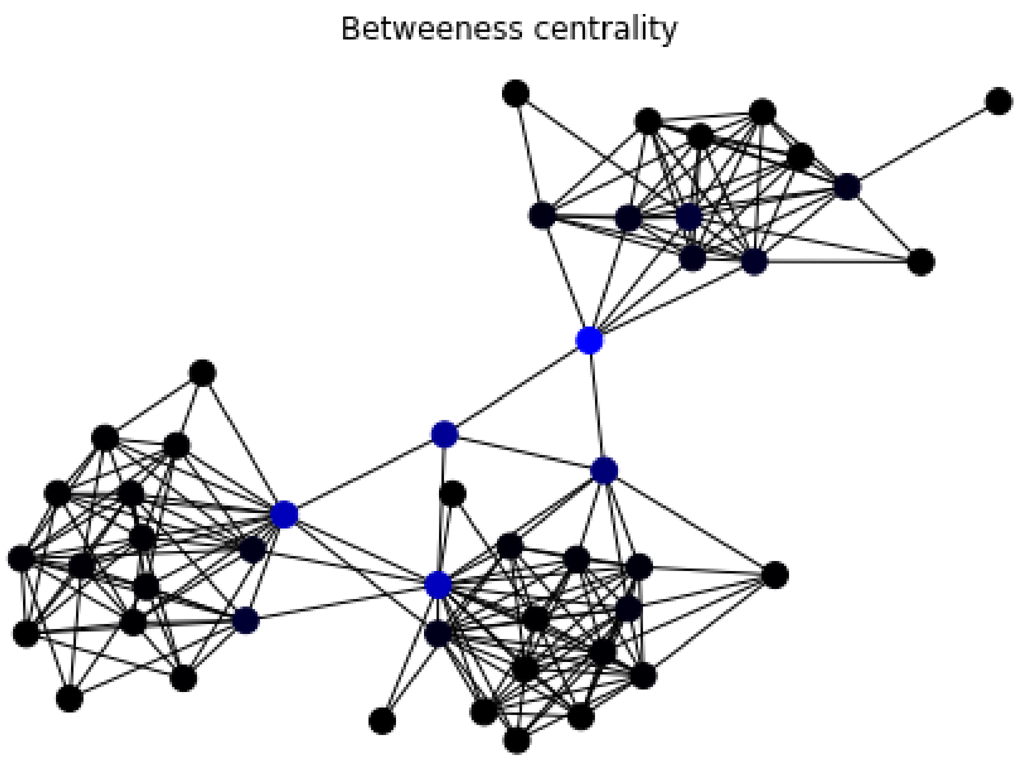
$\sigma(s,t)$ - (s,t) - .

$\sigma(s,t|v)$ - (s,t) - , v.

[400]:
```
def shortest_path(graph, source):
    order = []
    # predecessors of every vertex
    predeccessors = {}
    for v in graph.adjacency_list.keys():
        predeccessors[v] = []
```

```
    # sigma[v] = 0 for v in G
    shortest_lenth = dict.fromkeys(graph.adjacency_list.keys(), 0.0)
    # depth of BFS
    D = {}
    shortest_lenth[source] = 1.0
    D[source] = 0
    Q = [source]
    #  find shortest paths by BFS
    while Q:
        v = Q.pop(0)
        order.append(v)
        Dv = D[v]
        sigmav = shortest_lenth[v]
        for w in graph[v]: # not visited
            if w not in D:
                Q.append(w)
                #save shortest paht to vertex
                D[w] = Dv + 1
            if D[w] == Dv + 1:  # this is a shortest path
                shortest_lenth[w] += sigmav # number of shortest path to w
                # Predecessors.
                predeccessors[w].append(v)
    return order, predeccessors, shortest_lenth

def accumulate_basic(betweenness, order, predeccessors, shortest_path, source):
    delta = dict.fromkeys(order, 0)
    while order:
        w = order.pop()
        coeff = (1 + delta[w]) / shortest_path[w]
        for v in predeccessors[w]:
            delta[v] += shortest_path[v] * coeff
        if w != source:
            betweenness[w] += delta[w]
    return betweenness

def betweenness_centrality(graph,normalized=False):

    n = len(graph.adjacency_list)
    betweenness = dict.fromkeys(graph.adjacency_list.keys(), 0.0)
    for s in graph.adjacency_list:
        # Single source shortest paths using BFS.
        order, predeccessors, shortest_lenth = shortest_path(graph, s)
        # Accumulation.
        betweenness = accumulate_basic(betweenness, order, predeccessors,␣
 ↪shortest_lenth, s)
    # Rescale by 2 for undirected graphs.
    if normalized:              # normalization
```

```
        for v in betweenness.keys():
            betweenness[v] /= (n-1)*(n-2)
    return betweenness

betweenness_centrality = betweenness_centrality(social_subgraph)
draw_graph_with_centrality(nx_graph, betweenness_centrality)
save_from_plt("../task4/betweenness_centrality.png","Betweeness centrality")
```



3.0.4 Edge Betweenness Centrality

betweenness centrality $(v) = \sum_{s,t \in V(G)} \frac{\sigma(s,t|e)}{\sigma(s,t)}$

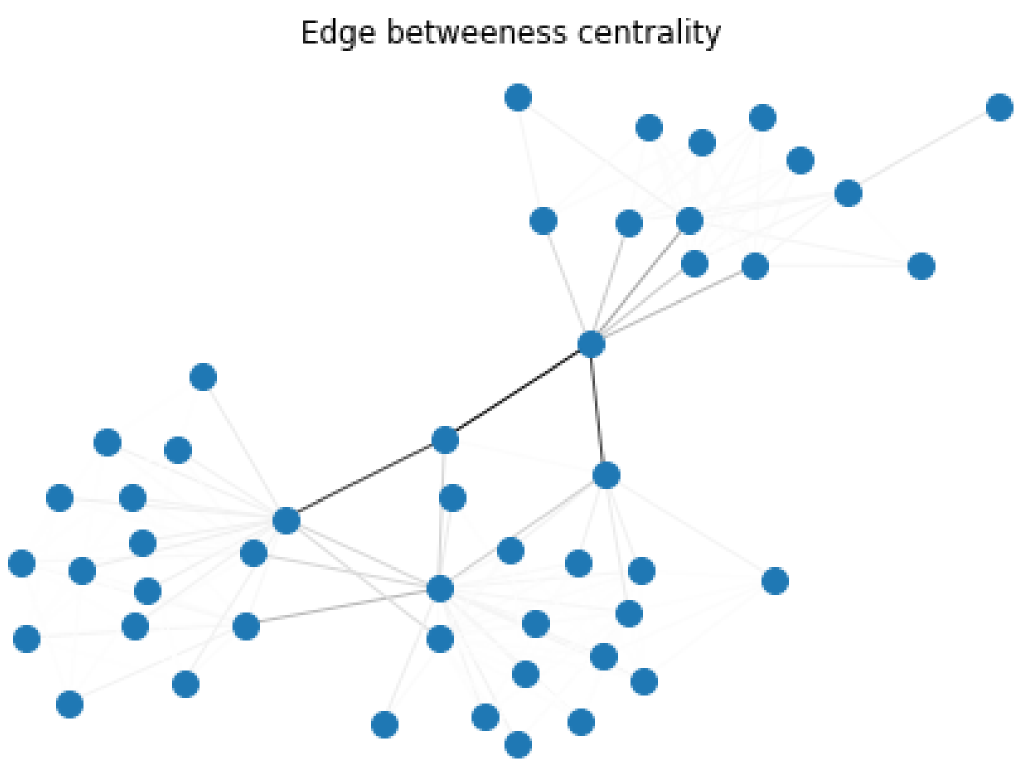
$\sigma(s,t)$ - (s,t) - .

$\sigma(s,t|v)$ - (s,t) - , e.

$n(n-1)$, n - .

[399]:
```
def accumulate_edges(betweenness, S, P, sigma, s):
    delta = dict.fromkeys(S, 0)
    while S:
        w = S.pop()
        coeff = (1 + delta[w]) / sigma[w]
        for v in P[w]:
            c = sigma[v] * coeff
```

```
            if (v, w) not in betweenness:
                betweenness[(w, v)] += c
            else:
                betweenness[(v, w)] += c
            delta[v] += c
        if w != s:
            betweenness[w] += delta[w]
    return betweenness


def edge_betweenness_centrality(graph):
    # b[v] = 0 for v in G.
    betweenness = dict.fromkeys(graph.adjacency_list.keys(), 0.0)
    # b[e] = 0 for e in Edges.
    betweenness.update(dict.fromkeys(graph.get_edges(), 0.0))
    for s in graph.adjacency_list.keys():
        # Single source shortest paths using BFS.
        S, P, sigma = shortest_path(graph, s)
        # Accumulation.
        betweenness = accumulate_edges(betweenness, S, P, sigma, s)
    # Remove nodes to only return edges.
    for n in graph.adjacency_list.keys():
        del betweenness[n]
    # Rescale by 2 for undirected graphs.
    scale = 0.5
    for v in betweenness.keys():
        betweenness[v] *= scale
    return betweenness

edge_betweennes = edge_betweenness_centrality(social_subgraph)
values = [edge_betweennes.get(edge, 0.0) for edge in nx_graph.edges()]
nx.draw_kamada_kawai(nx_graph, edge_cmap =plt.cm.Greys,
                         edge_color=values, node_size=100, with_labels=False,
                         label="Graph")
save_from_plt("../task4/edge_betweeness_centrality.png","Edge betweeness␣
 ↪centrality")
```



3.0.5 Eigenvector centrality

Eigenvector centrality - x.

$$Ax = \lambda x$$

A -

.

[401]:
```
def eigenvector_centrality(graph, max_iter=100, tol=1e-06):
    # Start with the all-ones vector.
    nstart = {v: 1 for v in graph.adjacency_list.keys()}
    # Normalize the initial vector so that each entry is in [0, 1]. This is
    # guaranteed to never have a divide-by-zero error by the previous line.
    nstart_sum = sum(nstart.values())
    x = {k: v / nstart_sum for k, v in nstart.items()}
    nnodes = len(graph.adjacency_list.keys())
    # make up to max_iter iterations
    for i in range(max_iter):
        xlast = x
        # Start with xlast times I to iterate with (A+I).
        x = xlast.copy()
        # Do the multiplication y^T = x^T A (left eigenvector).
```

```python
        for n in x.keys():
            for nbr in graph[n]:
                w = 1
                x[nbr] += xlast[n] * w
        # Normalize the vector. The normalization denominator 'norm'
        # should never be zero by the Perron-Frobenius
        # theorem. However, in case it is due to numerical error, we
        # assume the norm to be one instead.
        norm = np.sqrt(sum(z ** 2 for z in x.values())) or 1
        x = {k: v / norm for k, v in x.items()}
        # Check for convergence (in the L_1 norm).
        if sum(abs(x[n] - xlast[n]) for n in x) < nnodes * tol:
            return x
    raise ValueError(f"Maximum number of iterations reached: {max_iter}.")

eigenvector_centrali = eigenvector_centrality(social_subgraph)
draw_graph_with_centrality(nx_graph,eigenvector_centrali)
save_from_plt("../task4/eigenvector_centrality.png","Eigenvector centrality")
```