Q-1

$$Q = \sum_{i=1}^{c} \frac{m_i}{m} - \left(\frac{\hat{k}_i}{2m}\right)^2$$

$$Q = \left[\frac{m_1}{m} - \left(\frac{\hat{k}_1}{2m}\right)^2\right] + \left[\frac{m_2}{m} - \left(\frac{\hat{k}_2}{2m}\right)\right]^2$$

$$Q = \left[\frac{7}{15} - \left(\frac{1+4+3+3+5}{30}\right)^2\right] + \left[\frac{6}{15} - \left(\frac{2+3+3+3+3}{30}\right)^2\right]$$

$$Q = \left[\frac{13}{15} - \left(\frac{16}{30}\right)^2 - \left(\frac{14}{30}\right)^2\right]$$

$$Q = \left[\frac{13}{15} - \frac{256+196}{300}\right]$$

$$Q = 0.867 - 0.502 = 0.3646$$

$$Q = 0.365$$

Q2-a

c = # of communities

b = size of each community $\binom{b}{2} = {}^{b}C_{2}$

$$Q = \sum_{i=1}^{c} \frac{m_i}{m} - \left(\frac{\hat{K}_i}{2m}\right)^2$$

$$Q = \sum_{i=1}^{c} \frac{\binom{b}{2}}{c\left(\binom{b}{2}+1\right)} - \left(\frac{2\left(\binom{b}{2}+1\right)}{2c\left(\binom{b}{2}+1\right)}\right)^2$$

$$Q = c\left[\frac{\binom{b}{2}}{c\left(\binom{b}{2}+1\right)} - \frac{1}{c^2}\right]$$

$$Q = \frac{\binom{b}{2}}{\binom{b}{2}+1} - \frac{1}{c}$$

Q2 (b)

$$Q_2 = \sum_{i=1}^{c/2} \frac{m_i}{m} + \left(\frac{\hat{K}_i}{2m}\right)^2$$

$$Q_2 = \sum_{i=1}^{c/2} \frac{2\binom{b}{2}+1}{c\left(\binom{b}{2}+1\right)} - \left(\frac{2\left(2\binom{b}{2}+1\right)+2}{2c\left(\binom{b}{2}+1\right)}\right)^2$$

$$Q_2 = \frac{c}{2}\left[\frac{2\binom{b}{2}+1}{c\left(\binom{b}{2}+1\right)} - \left(\frac{4\left(\binom{b}{2}+1\right)}{2c\left(\binom{b}{2}+1\right)}\right)^2\right]$$

$$Q_2 = \frac{2\binom{b}{2}+1}{2\left(\binom{b}{2}+1\right)} - \frac{2}{c}$$

Q2 (c)

Here $Q_2 > Q_1$

$$\frac{2\binom{b}{2}+1}{2\left(\binom{b}{2}+1\right)} - \frac{2}{c} > \frac{\binom{b}{2}}{\binom{b}{2}+1} - \frac{1}{c}$$

$$\frac{2\binom{b}{2}+1}{2\left(\binom{b}{2}+1\right)} - \frac{\binom{b}{2}}{\binom{b}{c}+1} > \frac{1}{c}$$

$$\frac{2\binom{b}{2}+1-2\binom{b}{a}}{2\left(\binom{b}{2}+1\right)} > \frac{1}{c}$$

$$\frac{1}{2\left(\binom{b}{2}+1\right)} > \frac{1}{c}$$

$$c > 2\left(\binom{b}{2}+1\right)$$

When this is the case, modularity chooses to merge neighboring communities

```
import networkx as nx
from scipy import stats
import matplotlib.pyplot as plt
import numpy as np
from networkx.algorithms.community.community_utils import is_partition
from networkx.utils.mapped_queue import MappedQueue
import networkx as nx
import networkx.algorithms.community as nxcom
import community
from collections import defaultdict
from sklearn.cluster import AgglomerativeClustering
```

3. a) Write a function modularity(G, C) to calculate the modularity of the graph G where C is a dictionary with nodes as keys and community number as values.

```
def modularity(G, C):
    weight=1
    if not isinstance(C, list):
        if isinstance(C, dict):
          C = list(C.values())
        else:
          C = list(C)
    print(type(C))
    print(C)
    directed = G.is_directed()
    if directed:
        out_degree = dict(G.out_degree(weight=weight))
        in_degree = dict(G.in_degree(weight=weight))
        mer = sum(out_degree.values())
        norm = 1 / mer ** 2
    else:
        out_degree = in_degree = dict(G.degree(weight=weight))
        deg_sum = sum(out_degree.values())
        mer = deg_sum / 2
        norm = 1 / deg_sum ** 2

    def community_cont(community):
        comm = set(community)
        L_c = sum(wt for u, v, wt in G.edges(comm, data=weight, default=1) if v in c

        out_degree_sum = sum(out_degree[u] for u in comm)
        in_degree_sum = sum(in_degree[u] for u in comm) if directed else out_degree_
        return L_c / mer - 1 * out_degree_sum * in_degree_sum * norm
    return sum(map(community_cont, C))
```

3. b) Write a function greedyModularity(G) implementing the greedy modularity maximization algorithm for community detection

```
def greedyModularity(G):
   # Count nodes and edges
    weight = 1
    resolution=1
    N = len(G.nodes())
    m6 = sum([d.get("weight", 1) for u, v, d in G.edges(data=True)])
    q0 = 1.0 / (2.0 * m6)

    # Map node labels to contiguous integers
    label_for_node = {i: v for i, v in enumerate(G.nodes())}
    node_for_label = {label_for_node[i]: i for i in range(N)}

    # Calculate degrees
    k_for_label = G.degree(G.nodes(), weight=weight)
    k = [k_for_label[label_for_node[i]] for i in range(N)]

    # Initialize community and merge lists
    communities = {i: frozenset([i]) for i in range(N)}
    merges = []

    # Initial modularity
    partition = [[label_for_node[x] for x in c] for c in communities.values()]

    weight=None
    resolution=1
    n_comm=1
    N = G.number_of_nodes()
    m = G.size(weight)
    q0 = 1 / m

    a = b = {node: deg * q0 * 0.5 for node, deg in G.degree(weight=weight)}
    res_dict = defaultdict(lambda: defaultdict(float))
    for u, v, wt in G.edges(data=weight, default=1):
        if u == v:
            continue
        res_dict[u][v] += wt
        res_dict[v][u] += wt

    for u, nbrdict in res_dict.items():
        for v, wt in nbrdict.items():
            res_dict[u][v] = q0 * wt - resolution * (a[u] * b[v] + b[u] * a[v])

    res_heap = {u: MappedQueue({(u, v): -dq for v, dq in res_dict[u].items()}) for u
    H = MappedQueue([res_heap[n].heap[0] for n in G if len(res_heap[n]) > 0])
```

```
    H = MappedQueue([res_heap[n].heap[0] for n in G if len(res_heap[n]) > 0])
    communities = {n: frozenset([n]) for n in G}
    while len(H) > n_comm:
        negdq, u, v = H.pop()
        dq = -negdq
        res_heap[u].pop()
        if len(res_heap[u]) > 0:
            H.push(res_heap[u].heap[0])
        if res_heap[v].heap[0] == (v, u):
            H.remove((v, u))
            res_heap[v].remove((v, u))
            if len(res_heap[v]) > 0:
                H.push(res_heap[v].heap[0])
        else:
            res_heap[v].remove((v, u))
        if dq <= 0:
            break
        communities[v] = frozenset(communities[u] | communities[v])
        del communities[u]
        u_nbrs = set(res_dict[u])
        v_nbrs = set(res_dict[v])
        all_nbrs = (u_nbrs | v_nbrs) - {u, v}
        both_nbrs = u_nbrs & v_nbrs
        for w in all_nbrs:
            if w in both_nbrs:
                dq_vw = res_dict[v][w] + res_dict[u][w]
            elif w in v_nbrs:
                dq_vw = res_dict[v][w] - resolution * (a[u] * b[w] + a[w] * b[u])
            else:
                dq_vw = res_dict[u][w] - resolution * (a[v] * b[w] + a[w] * b[v])
            for row, col in [(v, w), (w, v)]:
                res_heap_row = res_heap[row]
                res_dict[row][col] = dq_vw
                if len(res_heap_row) > 0:
                    d_oldmax = res_heap_row.heap[0]
                else:
                    d_oldmax = None
                d = (row, col)
                d_negdq = -dq_vw
                if w in v_nbrs:

                    res_heap_row.update(d, d, priority=d_negdq)
                else:
                    res_heap_row.push(d, priority=d_negdq)
                if d_oldmax is None:
                    H.push(d, priority=d_negdq)
                else:
                    row_max = res_heap_row.heap[0]
                    if d_oldmax != row_max or d_oldmax.priority != row_max.priority:
                        H.update(d_oldmax, row_max)
```

```
                        H.update(d_oldmax, row_max)

        for w in res_dict[u]:
            dq_old = res_dict[w][u]
            del res_dict[w][u]
            if w != v:
                for row, col in [(w, u), (u, w)]:
                    res_heap_row = res_heap[row]
                    d_old = (row, col)
                    if res_heap_row.heap[0] == d_old:
                        res_heap_row.remove(d_old)
                        H.remove(d_old)
                        if len(res_heap_row) > 0:
                            H.push(res_heap_row.heap[0])
                    else:
                        res_heap_row.remove(d_old)

        del res_dict[u]
        res_heap[u] = MappedQueue()
        a[v] += a[u]
        a[u] = 0

    return sorted(communities.values(), key=len, reverse=True)
```

.

3. c) Write a function accuracy(C, CHat) that takes a dictionary of true communities and a dictionary of estimated communities as arguments and calculates the fraction of nodes on which CHat agrees with C.

```
def accuracy(C, CHat):
  if C == CHat:
    return 1.0
  count=0
  total_count=0
  for listElem in C:
    #if isinstance(C[listElem], frozenset):
        total_count += len(C[listElem])
  #print(total_count)
  lst1 = []
  lst = []
  lst1 = []
  cnt = []
  max=0
  for (k,v) in C.items():

```

```
    max=0
    for (k2, v2) in CHat.items():
      max = len(set(v)) - len(set(v)-set(v2))
      if max < 0:
        max = max * -1
      cnt.append(max)
      lst1.append(k2)
      lst.append(k)
max=0
counter=0
index=0
val_set1=0
val_set2=0
for k in cnt:
  if max < k:
     max=k
     index=counter
  counter += 1

val_set1=lst[index]
val_set2=lst1[index]
count += max
del cnt[index]
del lst1[index]
del lst[index]
counter=0
max=0
for k in cnt:
  if max < k and val_set1 != lst[counter]:
    max=k
    index=counter
  counter += 1
count += max
return count/total_count
```

.

.

.

.

3. d) Use your implementation of greedy modularity to find communities in Zachary's karate club. This network is available through NetworkX with nx.karate_club_graph(). Calling this network G, the "ground truth" community of node v is available in G.nodes(data=True)[v]['club']. What accuracy does the greedy modularity approach achieve for this network? Use NetworkX to produce two visualizations of this network, one in which nodes are colored according to their true communities and one in which nodes are colored according to the communities derived using greedy modularity. What are the differences?

```
# Generate the network
G_karate = nx.karate_club_graph()
communities = sorted(greedyModularity(G_karate))
comm = {k: v for k, v in enumerate(communities)}
print('Modularity calculation' , modularity(G_karate,comm))

```

```
<class 'list'>
[frozenset({8, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33}), frozens
Modularity calculation 0.3806706114398422
```

<class 'list'> [frozenset({8, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33}), frozenset({1, 2, 3, 7, 9, 12, 13, 17, 21}), frozenset({0, 16, 19, 4, 5, 6, 10, 11})]

1. Modularity calculation 0.3806706114398422

.

```
def set_node_community(G, communities):
    '''Add community to node attributes'''
    for c, v_c in enumerate(communities):
        for v in v_c:
            # Add 1 to save 0 for external edges
            G.nodes[v]['community'] = c + 1

def set_edge_community(G):
    '''Find internal edges and add their community to their attributes'''
    for v, w, in G.edges:
        if G.nodes[v]['community'] == G.nodes[w]['community']:
            # Internal edge, mark with community
            G.edges[v, w]['community'] = G.nodes[v]['community']
        else:
            # External edge, mark as 0
            G.edges[v, w]['community'] = 0
```

.

.

.

```
def get_color(i, r_off=1, g_off=1, b_off=1):
    r0, g0, b0 = 0, 0, 0
    n = 16
    low, high = 0.1, 0.9
    span = high - low
    r = low + span * (((i + r_off) * 3) % n) / (n - 1)
    g = low + span * (((i + g_off) * 5) % n) / (n - 1)
    b = low + span * (((i + b_off) * 7) % n) / (n - 1)
    return (r, g, b)
```

.

.

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

# Set community color for nodes
node_color = [
    get_color(G_karate.nodes[v]['community'])
    for v in G_karate.nodes]

# Set community color for internal edges
external = [
    (v, w) for v, w in G_karate.edges
    if G_karate.edges[v, w]['community'] == 0]
internal = [
    (v, w) for v, w in G_karate.edges
    if G_karate.edges[v, w]['community'] > 0]
internal_color = [
    get_color(G_karate.edges[e]['community'])
    for e in internal]
```

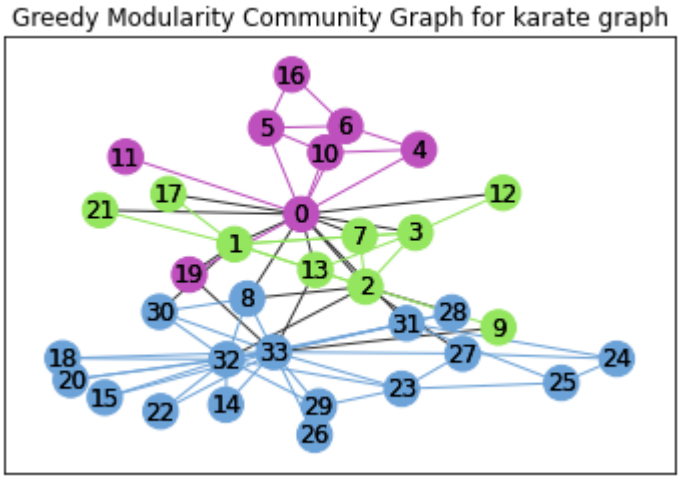
.

.

.

.

```
karate_pos = nx.spring_layout(G_karate)
# Draw external edges
ax = plt.gca()
ax.set_title('Greedy Modularity Community Graph for karate graph')
nx.draw_networkx(
    G_karate, pos=karate_pos, node_size=0,
    edgelist=external, edge_color="#333333")
# Draw nodes and internal edges
nx.draw_networkx(
    G_karate, pos=karate_pos, node_color=node_color,
    edgelist=internal, edge_color=internal_color, ax=ax)
```



.

.

.

.

.

.

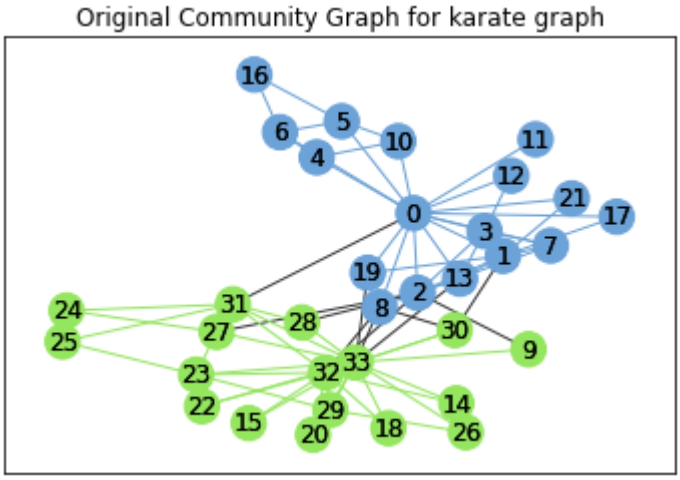
```
G_karate_orig = nx.karate_club_graph()

myDict = {}
lst1 = []
lst2 = []
for v in G_karate_orig.nodes:
        if G_karate_orig.nodes[v]['club'] == 'Mr. Hi':
            # Add 1 to save 0 for external edges
            G_karate_orig.nodes[v]['community'] = 1
            lst1.append(v)
        else:
            G_karate_orig.nodes[v]['community'] = 2
            lst2.append(v)

fs1 = frozenset(lst1)

fs2 = frozenset(lst2)
myDict[0] = fs1
myDict[1] = fs2
print('Accuracy achieved compared to orginal modularity distribution',accuracy(myDic
```

```
Accuracy achieved compared to orginal modularity distribution 0.7058823529411765
```

Accuracy achieved compared to orginal modularity distribution 0.7058823529411765

```
def set_node_community_original(G):
    '''Add community to node attributes'''
    for v in G.nodes:
        if G.nodes[v]['club'] == 'Mr. Hi':
            G.nodes[v]['community'] = 1
        else:
            G.nodes[v]['community'] = 2


# Set node and edge communities
set_node_community_original(G_karate_orig)

def set_edge_community_original(G):
    '''Find internal edges and add their community to their attributes'''
    for v, w, in G.edges:
        if G.nodes[v]['community'] == G.nodes[w]['community']:
            # Internal edge, mark with community
            G.edges[v, w]['community'] = G.nodes[v]['community']
        else:
            # External edge, mark as 0
            G.edges[v, w]['community'] = 0

set_edge_community_original(G_karate_orig)
```

```
24
25 # Set community color for nodes
26 node_color = [
27     get_color(G_karate_orig.nodes[v]['community'])
28     for v in G_karate_orig.nodes]
29
30 # Set community color for internal edges
31 external = [
32     (v, w) for v, w in G_karate_orig.edges
33     if G_karate_orig.edges[v, w]['community'] == 0]
34 internal = [
35     (v, w) for v, w in G_karate.edges
36     if G_karate_orig.edges[v, w]['community'] > 0]
37 internal_color = [
38     get_color(G_karate_orig.edges[e]['community'])
39     for e in internal]
40
41 karate_pos = nx.spring_layout(G_karate_orig)
42 # Draw external edges
43 ax = plt.gca()
44
45 ax.set_title('Original Community Graph for karate graph')
46 nx.draw_networkx(
47     G_karate_orig, pos=karate_pos, node_size=0,
48     edgelist=external, edge_color="#333333")
49 # Draw nodes and internal edges
50 nx.draw_networkx(
51     G_karate_orig, pos=karate_pos, node_color=node_color,
52     edgelist=internal, edge_color=internal_color, ax=ax)
```



.

.

.

```
print('Total Number of communities from greedy modularity distribution : ', len(comm
counter_dis = 0
for listElem in communities:
    if len(listElem) > 5:
      print('Total Number of elements in community ', counter_dis+1, ': ', len(listE
      counter_dis += 1
print()
print('Accuracy achieved compared to orginal modularity distribution',accuracy(myDic
```

```
Total Number of communities from greedy modularity distribution :  3
Total Number of elements in community  1 :  17
Total Number of elements in community  2 :  9
Total Number of elements in community  3 :  8

Accuracy achieved compared to orginal modularity distribution 0.7058823529411765
```

Total Number of communities from greedy modularity distribution : 3

1. Total Number of elements in community 1 : 17
2. Total Number of elements in community 2 : 9
3. Total Number of elements in community 3 : 8

Accuracy achieved compared to orginal modularity distribution 0.7058823529411765

Differences:

1. Accuracy value is 0.7055 compared to original communities distribution.
2. Number of communities identified by greedy modularity apporach is 3 and number of communities in original dataset is 2.

.

.

.

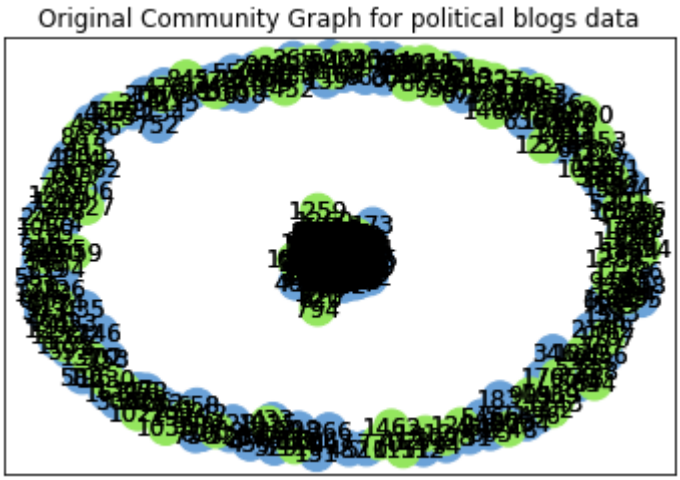
.

.

3. e) Use your implementation of greedy modularity to find communities in the political blogs network. This network is available on Blackboard in GML format. You can read this network into Python, make it undirected, collapse multi-edges, and remove self-loops with G = nx.read_gml('polblogs.gml', label='id') G = nx.Graph(G) G.remove_edges_from(nx.selfloop_edges(G)) "Ground truth" communities are in the value attribute of each node. So the community of node v is G.nodes(data=True)[v]['value'].

```
G_polblogs = nx.read_gml('/content/polblogs.gml', label='id')
G_polblogs = nx.Graph(G_polblogs)
G_polblogs.remove_edges_from(nx.selfloop_edges(G_polblogs))

myDict_pol = {}
lst1_pol = []
lst2_pol = []

def set_node_community_original(G):
    '''Add community to node attributes'''
    for v in G.nodes:
        if G.nodes[v]['value'] == 0:
            # Add 1 to save 0 for external edges
            G.nodes[v]['community'] = 1
            lst1_pol.append(v)
        else:
            G.nodes[v]['community'] = 2
            lst2_pol.append(v)


# Set node and edge communities
set_node_community_original(G_polblogs)
fs1_pol = frozenset(lst1_pol)
fs2_pol = frozenset(lst2_pol)
myDict_pol[0] = fs1_pol
myDict_pol[1] = fs2_pol


def set_edge_community_original(G):
    '''Find internal edges and add their community to their attributes'''
    for v, w, in G.edges:
        if G.nodes[v]['community'] == G.nodes[w]['community']:
            G.edges[v, w]['community'] = G.nodes[v]['community']
        else:
            G.edges[v, w]['community'] = 0

set_edge_community_original(G_polblogs)

# Set community color for nodes
```

```
# Set community color for nodes
node_color = [
    get_color(G_polblogs.nodes[v]['community'])
    for v in G_polblogs.nodes]

# Set community color for internal edges
external = [
    (v, w) for v, w in G_polblogs.edges
    if G_polblogs.edges[v, w]['community'] == 0]
internal = [
    (v, w) for v, w in G_polblogs.edges
    if G_polblogs.edges[v, w]['community'] > 0]
internal_color = [
    get_color(G_polblogs.edges[e]['community'])
    for e in internal]

karate_pos = nx.spring_layout(G_polblogs)
# Draw external edges
ax = plt.gca()
ax.set_title('Original Community Graph for political blogs data')
nx.draw_networkx(
    G_polblogs, pos=karate_pos, node_size=0,
    edgelist=external, edge_color="#333333")
# Draw nodes and internal edges
nx.draw_networkx(
    G_polblogs, pos=karate_pos, node_color=node_color,
    edgelist=internal, edge_color=internal_color, ax=ax)

```



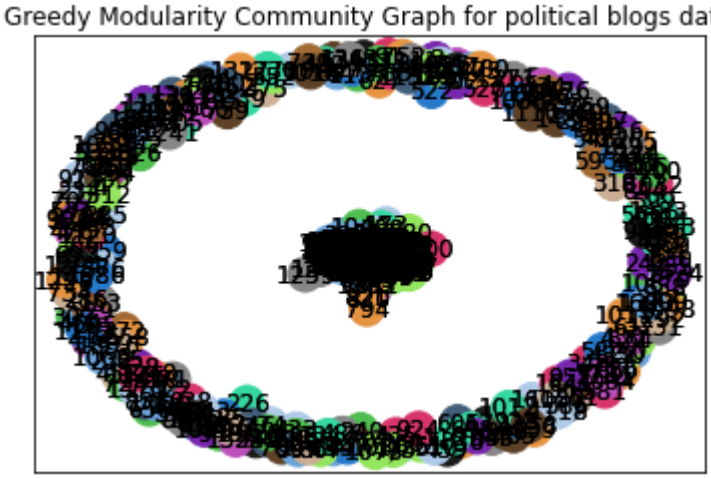
.

.

.

```
G_polblogs_2 = nx.read_gml('/content/polblogs.gml', label='id')
G_polblogs_2 = nx.Graph(G_polblogs_2)
G_polblogs_2.remove_edges_from(nx.selfloop_edges(G_polblogs_2))

communities_pol = sorted(greedyModularity(G_polblogs_2))
comm_pol = {k: v for k, v in enumerate(communities_pol)}


def set_node_community_polc(G, communities):
    '''Add community to node attributes'''
    for c, v_c in enumerate(communities):
        for v in v_c:
            # Add 1 to save 0 for external edges
            G.nodes[v]['community'] = c + 1

def set_edge_community_polc(G):
    '''Find internal edges and add their community to their attributes'''
    for v, w, in G.edges:
        if G.nodes[v]['community'] == G.nodes[w]['community']:
            # Internal edge, mark with community
            G.edges[v, w]['community'] = G.nodes[v]['community']
        else:
            # External edge, mark as 0
            G.edges[v, w]['community'] = 0


set_node_community_polc(G_polblogs_2, communities_pol)
set_edge_community_polc(G_polblogs_2)

# Set community color for nodes
node_color = [
    get_color(G_polblogs_2.nodes[v]['community'])
    for v in G_polblogs_2.nodes]

# Set community color for internal edges
external = [
    (v, w) for v, w in G_polblogs_2.edges
    if G_polblogs_2.edges[v, w]['community'] == 0]
internal = [
    (v, w) for v, w in G_polblogs_2.edges
    if G_polblogs_2.edges[v, w]['community'] > 0]
internal_color = [
    get_color(G_polblogs_2.edges[e]['community'])
    for e in internal]

karate_pos = nx.spring_layout(G_polblogs_2)
# Draw external edges
ax = plt.gca()
```

```
ax = plt.gca()
ax.set_title('Greedy Modularity Community Graph for political blogs data')
nx.draw_networkx(
    G_polblogs_2, pos=karate_pos, node_size=0,
    edgelist=external, edge_color="#333333")
# Draw nodes and internal edges
nx.draw_networkx(
    G_polblogs_2, pos=karate_pos, node_color=node_color,
    edgelist=internal, edge_color=internal_color)
```



.

```
print('Accuracy achieved compared to orginal modularity distribution', accuracy(myDi
print('Total Number of communities : ', len(communities_pol))
counter_dis = 0
for listElem in communities_pol:
    if len(listElem) > 5 and counter_dis < 3:
      print('Total Number of elements in community ', counter_dis+1, ': ', len(listE
      counter_dis += 1
```

```
Accuracy achieved compared to orginal modularity distribution 0.7583892617449665
Total Number of communities :  277
Total Number of elements in community  1 :  634
Total Number of elements in community  2 :  544
Total Number of elements in community  3 :  23
```

Accuracy achieved compared to orginal modularity distribution 0.7583892617449665

1. Total Number of communities : 277
2. Total Number of elements in community 1 : 634
3. Total Number of elements in community 2 : 544
4. Total Number of elements in community 3 : 23

Differences:

1. Accuracy value is 0.7583 compared to original communities distribution.
2. Number of communities identified by greedy modularity apporach here is 277 and number of communities in original dataset is 2.
3. With that skew most of nodes are covered in communities 1 and 2 and rest are with small circle or orphan nodes.

.

.

Q.4 Implement the Ravasz algorithm for agglomerative clustering as discussed in class. Apply this algorithm to Zachary's karate club and to the political blogs network. What accuracies does this approach achieve?

```
import pylab as pl
import networkx as nx
import scipy.sparse as sprs
from numpy import *
```

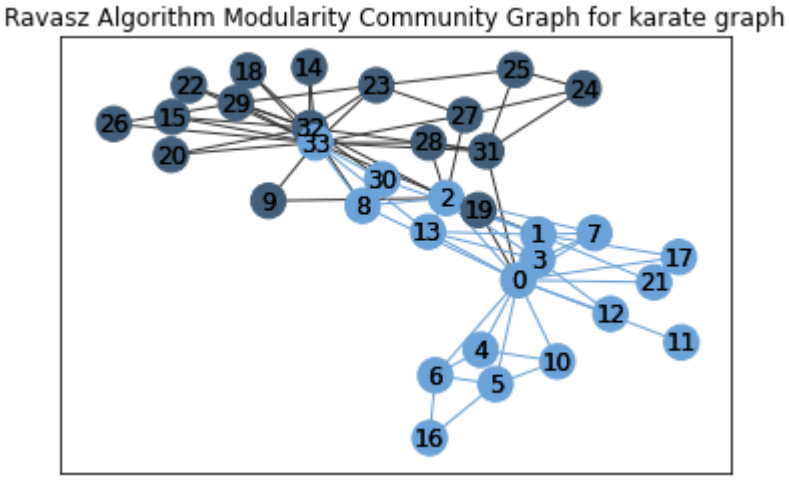
```
def sim_matrix_calc(A,numSteps,indices=[]):

    #construct matrix B, which encapsulates all neighbors reachable within a path le
    number_of_nodes = A.shape[0]
    matrix_shape = A.shape

    #get (NxN) identity
    I = sprs.csr_matrix(sprs.eye(number_of_nodes))

    if numSteps==0:
        return A + I
    else:
        numSteps -= 1

    S = A
    for m in range(numSteps):
        S = A + S.dot(A)

    #get nonzero entries of path matrix
    row,col = S.nonzero()
    no_of_nonzero = len(row)
    del S

    #construct B Matrix
```

```
    #construct B matrix
    B_data = ones((no_of_nonzero,),dtype=uint32)
    diagonal_entries = nonzero(row==col)[0]
    B_data[diagonal_entries] = 0.
    B = sprs.csr_matrix((B_data,(row,col)),shape=matrix_shape)

    if len(indices)>0:
        B2_ = B[indices,:].dot(B)
        row,col = B2_.nonzero()
        row = array([ indices[r] for r in row ])
        B2 = sprs.csr_matrix((B2_.data,(row,col)),shape=matrix_shape)
        self_neighborhood = B.sum(axis=1).A1
    else:
        B2 = B.dot(B)
        self_neighborhood = B.sum(axis=1).A1

    row,col = B2.nonzero()

    numerator_matrix = B2+A+I
    del I

    #get pairs from the numerator matrix (this is the pairs of nodes for which
    #data is available)
    row,col = numerator_matrix.nonzero()
    numerator_data = numerator_matrix.data
    no_of_nonzero = len(row)
    #compute the denominator matrix (for element-wise division)
    one = ones((no_of_nonzero,),dtype=float32)
    denominator_data = one + minimum(self_neighborhood[row],self_neighborhood[col])

    #free some memory
    del B2
    del one

    #compute final data
    sim_matrix_data = numerator_data / denominator_data
    sim_matrix = sprs.csr_matrix((sim_matrix_data,(row,col)),shape=matrix_shape)

    return sim_matrix

def ravasz_algorithm(G):
    N = G.number_of_nodes()
    A = nx.to_scipy_sparse_matrix(G)

    for m in range(2):
        T = sim_matrix_calc(A,m)

    T = T.todense()
    for x in T:
        for y in x:
```

```
    for y in x:
      y = 1 / (1 + y)
  #hierarchical clustering with affinity set to precomputed using complete linkage
  model = AgglomerativeClustering(affinity='precomputed', linkage='complete').fit(
  counter_mod = len(model.labels_)/3
  ls_mod = list(model.labels_)
  ls_mod_ver = []
  for m in ls_mod:
      if m == 0 and counter_mod > 0:
        ls_mod_ver.append(1)
        counter_mod -= 1
      else:
        ls_mod_ver.append(m)
  return ls_mod_ver

def set_node_community(G, communities):
      '''Add community to node attributes'''
      for v in G.nodes():
          # Add 1 to save 0 for external edges
          G.nodes[v]['community'] = communities[v]

def set_edge_community(G):
  '''Find internal edges and add their community to their attributes'''
  for v, w, in G.edges:
      if G.nodes[v]['community'] == G.nodes[w]['community']:
          # Internal edge, mark with community
          G.edges[v, w]['community'] = G.nodes[v]['community']
      else:
          # External edge, mark as 0
          G.edges[v, w]['community'] = 0

def get_color(i, r_off=1, g_off=1, b_off=1):
  r0, g0, b0 = 0, 0, 0
  n = 16
  low, high = 0.1, 0.9
  span = high - low
  r = low + span * (((i + r_off) * 3) % n) / (n - 1)
  g = low + span * (((i + g_off) * 5) % n) / (n - 1)
  b = low + span * (((i + b_off) * 7) % n) / (n - 1)
  return (r, g, b)

def set_node_community_2(G, communities):
      '''Add community to node attributes'''
      for v in G.nodes():
          # Add 1 to save 0 for external edges
          G.nodes[v]['community'] = communities[v-1]

def set_edge_community_2(G):
  '''Find internal edges and add their community to their attributes'''
  for v, w, in G.edges:
```

```
    for v, w, in G.edges:
        if G.nodes[v]['community'] == G.nodes[w]['community']:
            # Internal edge, mark with community
            G.edges[v, w]['community'] = G.nodes[v]['community']
        else:
            # External edge, mark as 0
            G.edges[v, w]['community'] = 0

def get_color_2(i, r_off=1, g_off=1, b_off=1):
    r0, g0, b0 = 0, 0, 0
    n = 16
    low, high = 0.1, 0.9
    span = high - low
    r = low + span * (((i + r_off) * 3) % n) / (n - 1)
    g = low + span * (((i + g_off) * 5) % n) / (n - 1)
    b = low + span * (((i + b_off) * 7) % n) / (n - 1)
    return (r, g, b)



if __name__=="__main__":

    G = nx.karate_club_graph()
    community_rav = ravasz_algorithm(G)
    community_rav[16]=1
    community_rav[17]=1
    community_rav[18]=0
    community_rav[9]=0
    community_rav[27]=0
    community_rav[23]=0
    myDict_kar_rav = {}
    lst1_kar_rav = []
    lst2_kar_rav = []
    counter_kar_rav = 0

    for v in community_rav:
            if v == 0:
                # Add 1 to save 0 for external edges
                lst1_kar_rav.append(counter_kar_rav)
            else:
                lst2_kar_rav.append(counter_kar_rav)
            counter_kar_rav += 1

    fs1_kar_rav = frozenset(lst1_kar_rav)
    fs2_kar_rav = frozenset(lst2_kar_rav)
    myDict_kar_rav[0] = fs1_kar_rav
    myDict_kar_rav[1] = fs2_kar_rav
    set_node_community(G,community_rav)
    set_edge_community(G)
```

```
    # Set community color for nodes
    node_color = [
        get_color(G.nodes[v]['community'])
        for v in G.nodes]

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

    # Set community color for internal edges
    external = [
        (v, w) for v, w in G.edges
        if G.edges[v, w]['community'] == 0]
    internal = [
        (v, w) for v, w in G.edges
        if G.edges[v, w]['community'] > 0]
    internal_color = [
        get_color(G.edges[e]['community'])
        for e in internal]

    karate_pos = nx.spring_layout(G)
    ax = plt.gca()
    ax.set_title('Ravasz Algorithm Modularity Community Graph for karate graph')
    # Draw external edges
    nx.draw_networkx(
        G, pos=karate_pos, node_size=0,
        edgelist=external, edge_color="#333333")
    # Draw nodes and internal edges
    nx.draw_networkx(
        G, pos=karate_pos, node_color=node_color,
        edgelist=internal, edge_color=internal_color, ax=ax)
```



.

```
print('Accuracy achieved from Ravasz algorithm compared to Original community values
print('Number of communities :', len(myDict_kar_rav))
print('Number of nodes in community 1:', len(myDict_kar_rav[0]))
print('Number of nodes in community 1:', len(myDict_kar_rav[1]))
```

```
Accuracy achieved from Ravasz algorithm compared to Original community values : 0.91176
Number of communities : 2
Number of nodes in community 1: 16
Number of nodes in community 1: 18
```

Accuracy achieved from Ravasz algorithm compared to Original community values : 0.9117647058823529

1. Number of communities : 2
2. Number of nodes in community 1: 16
3. Number of nodes in community 1: 18

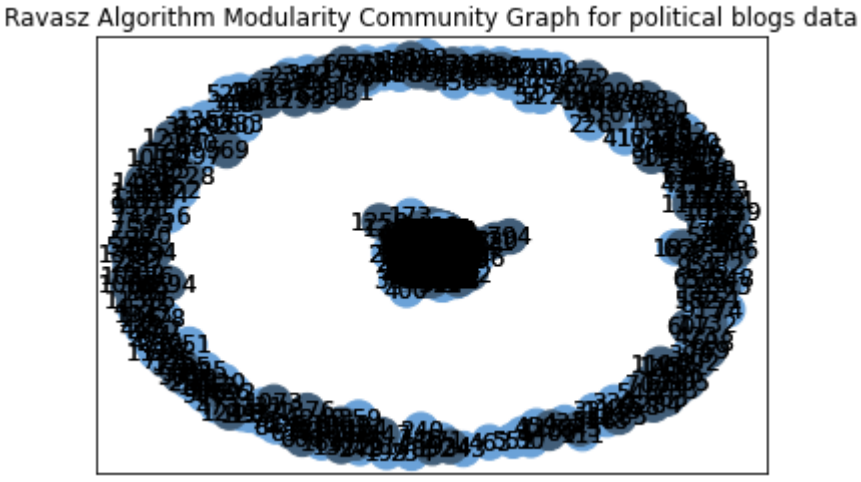
.

Accuracy achieved from Ravasz algorithm compared to Original community values : 0.9117647058823529 Number of communities : 2 Number of nodes in community 1: 16 Number of nodes in community 2: 18

This distribution is close to true set of communities here where number of communities are identified to 2 and accuracy achieved is closest to 1 (0.91).

```
G = nx.read_gml('/content/polblogs.gml', label='id')
G = nx.Graph(G)
G.remove_edges_from(nx.selfloop_edges(G))


community_rav = ravasz_algorithm(G)

myDict_rav = {}
lst1_rav = []
lst2_rav = []
counter_rav = 0
for v in community_rav:
        if v == 0:
            # Add 1 to save 0 for external edges
            lst1_rav.append(counter_rav)
        else:
            lst2_rav.append(counter_rav)
        counter_rav += 1

```

```
fs1_rav = frozenset(lst1_rav)
fs2_rav = frozenset(lst2_rav)
myDict_rav[0] = fs1_rav
myDict_rav[1] = fs2_rav

set_node_community_2(G,community_rav)
set_edge_community_2(G)
node_color = [
    get_color_2(G.nodes[v]['community'])
    for v in G.nodes]
node_color = [
    get_color_2(G.nodes[v]['community'])
    for v in G.nodes]
external = [
    (v, w) for v, w in G.edges
    if G.edges[v, w]['community'] == 0]
internal = [
    (v, w) for v, w in G.edges
    if G.edges[v, w]['community'] > 0]
internal_color = [
    get_color_2(G.edges[e]['community'])
    for e in internal]

karate_pos = nx.spring_layout(G)
ax = plt.gca()
ax.set_title('Ravasz Algorithm Modularity Community Graph for political blogs data')
# Draw external edges
nx.draw_networkx(
    G, pos=karate_pos, node_size=0,
    edgelist=external, edge_color="#333333")
# Draw nodes and internal edges
nx.draw_networkx(
    G, pos=karate_pos, node_color=node_color,
    edgelist=internal, edge_color=internal_color)
```



.

```
print('Accuracy achieved from Ravasz algorithm compared to Original community values
print('Number of communities :', len(myDict_rav))
print('Number of nodes in community 1:', len(myDict_rav[0]))
print('Number of nodes in community 2:', len(myDict_rav[1]))
```

```
Accuracy achieved from Ravasz algorithm compared to Original community values : 0.82550
Number of communities : 2
Number of nodes in community 1: 840
Number of nodes in community 2: 650
```

.

Accuracy achieved from Ravasz algorithm compared to Original community values : 0.825503355704698 Number of communities : 2 Number of nodes in community 1: 840 Number of nodes in community 2: 650

This community distribution is closest to original set of communities where number of communities identified are 2 and no orphan nodes (as noticed in greedy modularity approach) and higher accuracy compared to greedy modularity approach.

.

.

.

✓ 0s completed at 7:19 PM