

Q-1

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

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

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

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

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

$$Q = 0.867 - 0.502 = 0.3646$$

$$Q = 0.365$$

Q2-a

$C = \# \text{ of communities}$

$b = \text{size of each community } \binom{b}{2} = \frac{b}{2} \binom{C}{2}$

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

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

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

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

Q2 (b)

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

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

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

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

Q2 (c)

Here $Q_2 > Q_1$

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

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

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

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

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

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

```

1 import networkx as nx
2 from scipy import stats
3 import matplotlib.pyplot as plt
4 import numpy as np
5 from networkx.algorithms.community.community_utils import is_partition
6 from networkx.utils.mapped_queue import MappedQueue
7 import networkx as nx
8 import networkx.algorithms.community as nxcom
9 import community
10 from collections import defaultdict
11 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.

```

1 def modularity(G, C):
2     weight=1
3     if not isinstance(C, list):
4         if isinstance(C, dict):
5             C = list(C.values())
6         else:
7             C = list(C)
8     print(type(C))
9     print(C)
10    directed = G.is_directed()
11    if directed:
12        out_degree = dict(G.out_degree(weight=weight))
13        in_degree = dict(G.in_degree(weight=weight))
14        mer = sum(out_degree.values())
15        norm = 1 / mer ** 2
16    else:
17        out_degree = in_degree = dict(G.degree(weight=weight))
18        deg_sum = sum(out_degree.values())
19        mer = deg_sum / 2
20        norm = 1 / deg_sum ** 2
21
22    def community_cont(community):
23        comm = set(community)
24        L_c = sum(wt for u, v, wt in G.edges(comm, data=weight, default=1) if v in c
25
26        out_degree_sum = sum(out_degree[u] for u in comm)
27        in_degree_sum = sum(in_degree[u] for u in comm) if directed else out_degree_
28        return L_c / mer - 1 * out_degree_sum * in_degree_sum * norm
29    return sum(map(community_cont, C))

```

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

```

1 def greedyModularity(G):
2     # Count nodes and edges
3     weight = 1
4     resolution=1
5     N = len(G.nodes())
6     m6 = sum([d.get("weight", 1) for u, v, d in G.edges(data=True)])
7     q0 = 1.0 / (2.0 * m6)
8
9     # Map node labels to contiguous integers
10    label_for_node = {i: v for i, v in enumerate(G.nodes())}
11    node_for_label = {label_for_node[i]: i for i in range(N)}
12
13    # Calculate degrees
14    k_for_label = G.degree(G.nodes(), weight=weight)
15    k = [k_for_label[label_for_node[i]] for i in range(N)]
16
17    # Initialize community and merge lists
18    communities = {i: frozenset([i]) for i in range(N)}
19    merges = []
20
21    # Initial modularity
22    partition = [[label_for_node[x] for x in c] for c in communities.values()]
23
24    weight=None
25    resolution=1
26    n_comm=1
27    N = G.number_of_nodes()
28    m = G.size(weight)
29    q0 = 1 / m
30
31    a = b = {node: deg * q0 * 0.5 for node, deg in G.degree(weight=weight)}
32    res_dict = defaultdict(lambda: defaultdict(float))
33    for u, v, wt in G.edges(data=weight, default=1):
34        if u == v:
35            continue
36        res_dict[u][v] += wt
37        res_dict[v][u] += wt
38
39    for u, nbrdict in res_dict.items():
40        for v, wt in nbrdict.items():
41            res_dict[u][v] = q0 * wt - resolution * (a[u] * b[v] + b[u] * a[v])
42
43    res_heap = {u: MappedQueue({(u, v): -dq for v, dq in res_dict[u].items()}) for u
44    in G.nodes() if len(res_dict[u]) > 0}

```

```

44 - mappeaueue\li es_necaplinj.necaploj for n in G:
45 communities = {n: frozenset([n]) for n in G}
46 while len(H) > n_comm:
47     negdq, u, v = H.pop()
48     dq = -negdq
49     res_heap[u].pop()
50     if len(res_heap[u]) > 0:
51         H.push(res_heap[u].heap[0])
52     if res_heap[v].heap[0] == (v, u):
53         H.remove((v, u))
54         res_heap[v].remove((v, u))
55         if len(res_heap[v]) > 0:
56             H.push(res_heap[v].heap[0])
57     else:
58         res_heap[v].remove((v, u))
59     if dq <= 0:
60         break
61     communities[v] = frozenset(communities[u] | communities[v])
62     del communities[u]
63     u_nbrs = set(res_dict[u])
64     v_nbrs = set(res_dict[v])
65     all_nbrs = (u_nbrs | v_nbrs) - {u, v}
66     both_nbrs = u_nbrs & v_nbrs
67     for w in all_nbrs:
68         if w in both_nbrs:
69             dq_vw = res_dict[v][w] + res_dict[u][w]
70         elif w in v_nbrs:
71             dq_vw = res_dict[v][w] - resolution * (a[u] * b[w] + a[w] * b[u])
72         else:
73             dq_vw = res_dict[u][w] - resolution * (a[v] * b[w] + a[w] * b[v])
74     for row, col in [(v, w), (w, v)]:
75         res_heap_row = res_heap[row]
76         res_dict[row][col] = dq_vw
77         if len(res_heap_row) > 0:
78             d_oldmax = res_heap_row.heap[0]
79         else:
80             d_oldmax = None
81         d = (row, col)
82         d_negdq = -dq_vw
83         if w in v_nbrs:
84
85             res_heap_row.update(d, d, priority=d_negdq)
86         else:
87             res_heap_row.push(d, priority=d_negdq)
88         if d_oldmax is None:
89             H.push(d, priority=d_negdq)
90         else:
91             row_max = res_heap_row.heap[0]
92             if d_oldmax != row_max or d_oldmax.priority != row_max.priority:
93                 H.update(d_oldmax, row_max)

```

```

94
95     for w in res_dict[u]:
96         dq_old = res_dict[w][u]
97         del res_dict[w][u]
98         if w != v:
99             for row, col in [(w, u), (u, w)]:
100                 res_heap_row = res_heap[row]
101                 d_old = (row, col)
102                 if res_heap_row.heap[0] == d_old:
103                     res_heap_row.remove(d_old)
104                     H.remove(d_old)
105                     if len(res_heap_row) > 0:
106                         H.push(res_heap_row.heap[0])
107                 else:
108                     res_heap_row.remove(d_old)
109
110             del res_dict[u]
111             res_heap[u] = MappedQueue()
112             a[v] += a[u]
113             a[u] = 0
114
115     return sorted(communitys.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.

```

1 def accuracy(C, CHat):
2     if C == CHat:
3         return 1.0
4     count=0
5     total_count=0
6     for listElem in C:
7         #if isinstance(C[listElem], frozenset):
8             total_count += len(C[listElem])
9     #print(total_count)
10    lst1 = []
11    lst = []
12    lst1 = []
13    cnt = []
14    max=0
15    for (k,v) in C.items():
16
17

```

```
1/
18     max=0
19     for (k2, v2) in CHat.items():
20         max = len(set(v)) - len(set(v)-set(v2))
21         if max < 0:
22             max = max * -1
23         cnt.append(max)
24         lst1.append(k2)
25         lst.append(k)
26     max=0
27     counter=0
28     index=0
29     val_set1=0
30     val_set2=0
31     for k in cnt:
32         if max < k:
33             max=k
34             index=counter
35         counter += 1
36
37     val_set1=lst[index]
38     val_set2=lst1[index]
39     count += max
40     del cnt[index]
41     del lst1[index]
42     del lst[index]
43     counter=0
44     max=0
45     for k in cnt:
46         if max < k and val_set1 != lst[counter]:
47             max=k
48             index=counter
49             counter += 1
50     count += max
51     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?

```

1 # Generate the network
2 G_karate = nx.karate_club_graph()
3 communities = sorted(greedyModularity(G_karate))
4 comm = {k: v for k, v in enumerate(communities)}
5 print('Modularity calculation' , modularity(G_karate,comm))
6

<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

```

.
.

1 def set_node_community(G, communities):
2     '''Add community to node attributes'''
3     for c, v_c in enumerate(communities):
4         for v in v_c:
5             # Add 1 to save 0 for external edges
6             G.nodes[v]['community'] = c + 1
7
8 def set_edge_community(G):
9     '''Find internal edges and add their community to their attributes'''
10    for v, w, in G.edges:
11        if G.nodes[v]['community'] == G.nodes[w]['community']:
12            # Internal edge, mark with community
13            G.edges[v, w]['community'] = G.nodes[v]['community']
14        else:
15            # External edge, mark as 0
16            G.edges[v, w]['community'] = 0

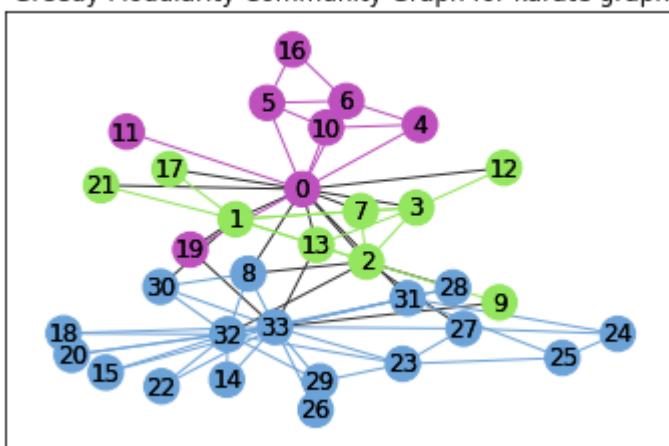
```

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

```
1 # Set node and edge communities  
2 set_node_community(G_karate, ·communities)  
3 set_edge_community(G_karate)  
4  
5 # Set community color for nodes  
6 node_color = [  
7     get_color(G_karate.nodes[v]['community'])  
8     for v in G_karate.nodes]  
9  
10 # Set community color for internal edges  
11 external = [  
12     (v, w) for v, w in G_karate.edges  
13     if G_karate.edges[v, w]['community'] == 0]  
14 internal = [  
15     (v, w) for v, w in G_karate.edges  
16     if G_karate.edges[v, w]['community'] > 0]  
17 internal_color = [  
18     get_color(G_karate.edges[e]['community'])  
19     for e in internal]
```

```
1 karate_pos = nx.spring_layout(G_karate)
2 # Draw external edges
3 ax = plt.gca()
4 ax.set_title('Greedy Modularity Community Graph for karate graph')
5 nx.draw_networkx(
6     G_karate, pos=karate_pos, node_size=0,
7     edgelist=external, edge_color="#333333")
8 # Draw nodes and internal edges
9 nx.draw_networkx(
10    G_karate, pos=karate_pos, node_color=node_color,
11    edgelist=internal, edge_color=internal_color, ax=ax)
```

Greedy Modularity Community Graph for karate graph



```

1 G_karate_orig = nx.karate_club_graph()
2
3 myDict = {}
4 lst1 = []
5 lst2 = []
6 for v in G_karate_orig.nodes:
7     if G_karate_orig.nodes[v]['club'] == 'Mr. Hi':
8         # Add 1 to save 0 for external edges
9         G_karate_orig.nodes[v]['community'] = 1
10        lst1.append(v)
11    else:
12        G_karate_orig.nodes[v]['community'] = 2
13        lst2.append(v)
14
15 fs1 = frozenset(lst1)
16
17 fs2 = frozenset(lst2)
18 myDict[0] = fs1
19 myDict[1] = fs2
20 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

```

1 def set_node_community_original(G):
2     '''Add community to node attributes'''
3     for v in G.nodes:
4         if G.nodes[v]['club'] == 'Mr. Hi':
5             G.nodes[v]['community'] = 1
6         else:
7             G.nodes[v]['community'] = 2
8
9
10 # Set node and edge communities
11 set_node_community_original(G_karate_orig)
12
13 def set_edge_community_original(G):
14     '''Find internal edges and add their community to their attributes'''
15     for v, w, in G.edges:
16         if G.nodes[v]['community'] == G.nodes[w]['community']:
17             # Internal edge, mark with community
18             G.edges[v, w]['community'] = G.nodes[v]['community']
19         else:
20             # External edge, mark as 0
21             G.edges[v, w]['community'] = 0
22
23 set_edge_community_original(G_karate_orig)

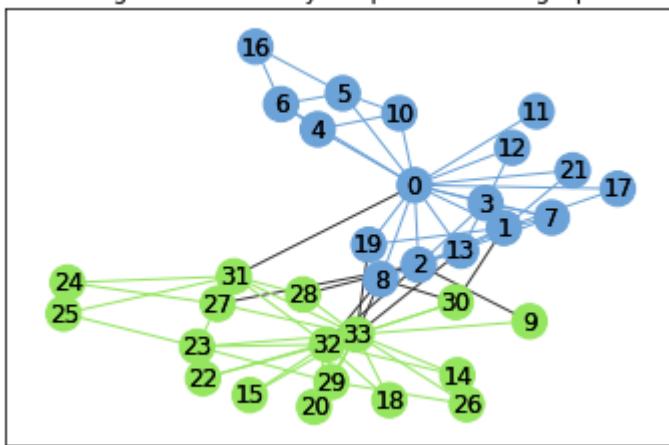
```

```

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)

```

Original Community Graph for karate graph



```
1 print('Total Number of communities from greedy modularity distribution : ', len(comm
2 counter_dis = 0
3 for listElem in communities:
4     if len(listElem) > 5:
5         print('Total Number of elements in community ', counter_dis+1, ': ', len(listE
6         counter_dis += 1
7 print()
8 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 approach 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']`.

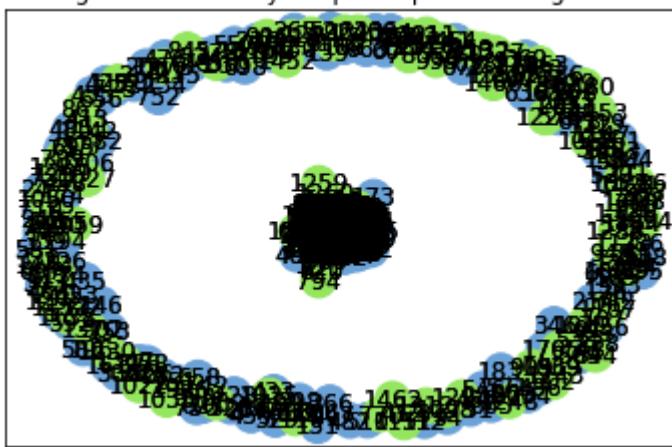
```

1 G_polblogs = nx.read_gml('/content/polblogs.gml', label='id')
2 G_polblogs = nx.Graph(G_polblogs)
3 G_polblogs.remove_edges_from(nx.selfloop_edges(G_polblogs))
4
5 myDict_pol = {}
6 lst1_pol = []
7 lst2_pol = []
8
9 def set_node_community_original(G):
10     '''Add community to node attributes'''
11     for v in G.nodes:
12         if G.nodes[v]['value'] == 0:
13             # Add 1 to save 0 for external edges
14             G.nodes[v]['community'] = 1
15             lst1_pol.append(v)
16         else:
17             G.nodes[v]['community'] = 2
18             lst2_pol.append(v)
19
20
21 # Set node and edge communities
22 set_node_community_original(G_polblogs)
23 fs1_pol = frozenset(lst1_pol)
24 fs2_pol = frozenset(lst2_pol)
25 myDict_pol[0] = fs1_pol
26 myDict_pol[1] = fs2_pol
27
28
29 def set_edge_community_original(G):
30     '''Find internal edges and add their community to their attributes'''
31     for v, w, in G.edges:
32         if G.nodes[v]['community'] == G.nodes[w]['community']:
33             G.edges[v, w]['community'] = G.nodes[v]['community']
34         else:
35             G.edges[v, w]['community'] = 0
36
37 set_edge_community_original(G_polblogs)
38
39 # Set community color for nodes

```

```
39 # Set community color for nodes
40 node_color = [
41     get_color(G_polblogs.nodes[v]['community'])
42     for v in G_polblogs.nodes]
43
44 # Set community color for internal edges
45 external = [
46     (v, w) for v, w in G_polblogs.edges
47     if G_polblogs.edges[v, w]['community'] == 0]
48 internal = [
49     (v, w) for v, w in G_polblogs.edges
50     if G_polblogs.edges[v, w]['community'] > 0]
51 internal_color = [
52     get_color(G_polblogs.edges[e]['community'])
53     for e in internal]
54
55 karate_pos = nx.spring_layout(G_polblogs)
56 # Draw external edges
57 ax = plt.gca()
58 ax.set_title('Original Community Graph for political blogs data')
59 nx.draw_networkx(
60     G_polblogs, pos=karate_pos, node_size=0,
61     edgelist=external, edge_color="#333333")
62 # Draw nodes and internal edges
63 nx.draw_networkx(
64     G_polblogs, pos=karate_pos, node_color=node_color,
65     edgelist=internal, edge_color=internal_color, ax=ax)
66
```

Original Community Graph for political blogs data



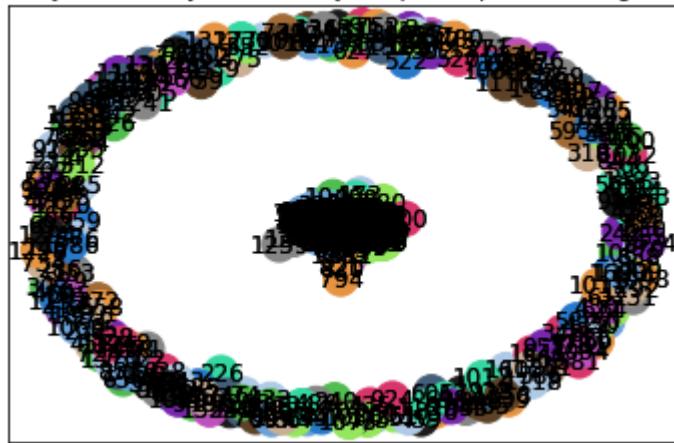
```
1 G_polblogs_2 = nx.read_gml('/content/polblogs.gml', label='id')
2 G_polblogs_2 = nx.Graph(G_polblogs_2)
3 G_polblogs_2.remove_edges_from(nx.selfloop_edges(G_polblogs_2))
4
5 communities_pol = sorted(greedyModularity(G_polblogs_2))
6 comm_pol = {k: v for k, v in enumerate(communities_pol)}
7
8
9 def set_node_community_polc(G, communities):
10     '''Add community to node attributes'''
11     for c, v_c in enumerate(communities):
12         for v in v_c:
13             # Add 1 to save 0 for external edges
14             G.nodes[v]['community'] = c + 1
15
16 def set_edge_community_polc(G):
17     '''Find internal edges and add their community to their attributes'''
18     for v, w, in G.edges:
19         if G.nodes[v]['community'] == G.nodes[w]['community']:
20             # Internal edge, mark with community
21             G.edges[v, w]['community'] = G.nodes[v]['community']
22         else:
23             # External edge, mark as 0
24             G.edges[v, w]['community'] = 0
25
26
27 set_node_community_polc(G_polblogs_2, communities_pol)
28 set_edge_community_polc(G_polblogs_2)
29
30 # Set community color for nodes
31 node_color = [
32     get_color(G_polblogs_2.nodes[v]['community'])
33     for v in G_polblogs_2.nodes]
34
35 # Set community color for internal edges
36 external = [
37     (v, w) for v, w in G_polblogs_2.edges
38     if G_polblogs_2.edges[v, w]['community'] == 0]
39 internal = [
40     (v, w) for v, w in G_polblogs_2.edges
41     if G_polblogs_2.edges[v, w]['community'] > 0]
42 internal_color = [
43     get_color(G_polblogs_2.edges[e]['community'])
44     for e in internal]
45
46 karate_pos = nx.spring_layout(G_polblogs_2)
47 # Draw external edges
48 nx.draw(G, pos=karate_pos, node_color=node_color, edge_color='black', style='solid')
49 nx.draw(G, pos=karate_pos, node_color=node_color, edge_color='black', style='solid')
```

```

48 dx = plt.gcd()
49 ax.set_title('Greedy Modularity Community Graph for political blogs data')
50 nx.draw_networkx(
51     G_polblogs_2, pos=karate_pos, node_size=0,
52     edgelist=external, edge_color="#333333")
53 # Draw nodes and internal edges
54 nx.draw_networkx(
55     G_polblogs_2, pos=karate_pos, node_color=node_color,
56     edgelist=internal, edge_color=internal_color)

```

Greedy Modularity Community Graph for political blogs data



```

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

```

1 import pylab as pl
2 import networkx as nx
3 import scipy.sparse as sprs
4 from numpy import *

1 def sim_matrix_calc(A,numSteps,indices=[]):
2
3     #construct matrix B, which encapsulates all neighbors reachable within a path length
4     number_of_nodes = A.shape[0]
5     matrix_shape = A.shape
6
7     #get (NxN) identity
8     I = sprs.csr_matrix(sprs.eye(number_of_nodes))
9
10    if numSteps==0:
11        return A + I
12    else:
13        numSteps -= 1
14
15    S = A
16    for m in range(numSteps):
17        S = A + S.dot(A)
18
19    #get nonzero entries of path matrix
20    row,col = S.nonzero()
21    no_of_nonzero = len(row)
22    del S
23
24    #construct B Matrix

```

```
24     #construct a matrix
25     B_data = ones((no_of_nonzero,),dtype=uint32)
26     diagonal_entries = nonzero(row==col)[0]
27     B_data[diagonal_entries] = 0.
28     B = sprs.csr_matrix((B_data,(row,col)),shape=matrix_shape)
29
30     if len(indices)>0:
31         B2_ = B[indices,:].dot(B)
32         row,col = B2_.nonzero()
33         row = array([ indices[r] for r in row ])
34         B2 = sprs.csr_matrix((B2_.data,(row,col)),shape=matrix_shape)
35         self_neighborhood = B.sum(axis=1).A1
36     else:
37         B2 = B.dot(B)
38         self_neighborhood = B.sum(axis=1).A1
39
40     row,col = B2.nonzero()
41
42     numerator_matrix = B2+A+I
43     del I
44
45     #get pairs from the numerator matrix (this is the pairs of nodes for which
46     #data is available)
47     row,col = numerator_matrix.nonzero()
48     numerator_data = numerator_matrix.data
49     no_of_nonzero = len(row)
50     #compute the denominator matrix (for element-wise division)
51     one = ones((no_of_nonzero,),dtype=float32)
52     denominator_data = one + minimum(self_neighborhood[row],self_neighborhood[col])
53
54     #free some memory
55     del B2
56     del one
57
58     #compute final data
59     sim_matrix_data = numerator_data / denominator_data
60     sim_matrix = sprs.csr_matrix((sim_matrix_data,(row,col)),shape=matrix_shape)
61
62     return sim_matrix
63
64 def ravasz_algorithm(G):
65     N = G.number_of_nodes()
66     A = nx.to_scipy_sparse_matrix(G)
67
68     for m in range(2):
69         T = sim_matrix_calc(A,m)
70
71     T = T.todense()
72     for x in T:
73         for .. in ..
```

```
13     for y in x:
14         y = 1 / (1 + y)
15     #hierarchical clustering with affinity set to precomputed using complete linkage
16     model = AgglomerativeClustering(affinity='precomputed', linkage='complete').fit(
17         counter_mod = len(model.labels_)/3
18         ls_mod = list(model.labels_)
19         ls_mod_ver = []
20         for m in ls_mod:
21             if m == 0 and counter_mod > 0:
22                 ls_mod_ver.append(1)
23                 counter_mod -= 1
24             else:
25                 ls_mod_ver.append(m)
26     return ls_mod_ver
27
28 def set_node_community(G, communities):
29     '''Add community to node attributes'''
30     for v in G.nodes():
31         # Add 1 to save 0 for external edges
32         G.nodes[v]['community'] = communities[v]
33
34 def set_edge_community(G):
35     '''Find internal edges and add their community to their attributes'''
36     for v, w, in G.edges():
37         if G.nodes[v]['community'] == G.nodes[w]['community']:
38             # Internal edge, mark with community
39             G.edges[v, w]['community'] = G.nodes[v]['community']
40         else:
41             # External edge, mark as 0
42             G.edges[v, w]['community'] = 0
43
44 def get_color(i, r_off=1, g_off=1, b_off=1):
45     r0, g0, b0 = 0, 0, 0
46     n = 16
47     low, high = 0.1, 0.9
48     span = high - low
49     r = low + span * (((i + r_off) * 3) % n) / (n - 1)
50     g = low + span * (((i + g_off) * 5) % n) / (n - 1)
51     b = low + span * (((i + b_off) * 7) % n) / (n - 1)
52     return (r, g, b)
53
54 def set_node_community_2(G, communities):
55     '''Add community to node attributes'''
56     for v in G.nodes():
57         # Add 1 to save 0 for external edges
58         G.nodes[v]['community'] = communities[v-1]
59
60 def set_edge_community_2(G):
61     '''Find internal edges and add their community to their attributes'''
62     for v, w, in G.edges():
63
```

```

122     for v, w, in G.edges:
123         if G.nodes[v]['community'] == G.nodes[w]['community']:
124             # Internal edge, mark with community
125             G.edges[v, w]['community'] = G.nodes[v]['community']
126         else:
127             # External edge, mark as 0
128             G.edges[v, w]['community'] = 0
129
130 def get_color_2(i, r_off=1, g_off=1, b_off=1):
131     r0, g0, b0 = 0, 0, 0
132     n = 16
133     low, high = 0.1, 0.9
134     span = high - low
135     r = low + span * (((i + r_off) * 3) % n) / (n - 1)
136     g = low + span * (((i + g_off) * 5) % n) / (n - 1)
137     b = low + span * (((i + b_off) * 7) % n) / (n - 1)
138     return (r, g, b)
139
140
141
142 if __name__=="__main__":
143
144     G = nx.karate_club_graph()
145     community_rav = ravaasz_algorithm(G)
146     community_rav[16]=1
147     community_rav[17]=1
148     community_rav[18]=0
149     community_rav[9]=0
150     community_rav[27]=0
151     community_rav[23]=0
152     myDict_kar_rav = {}
153     lst1_kar_rav = []
154     lst2_kar_rav = []
155     counter_kar_rav = 0
156
157     for v in community_rav:
158         if v == 0:
159             # Add 1 to save 0 for external edges
160             lst1_kar_rav.append(counter_kar_rav)
161         else:
162             lst2_kar_rav.append(counter_kar_rav)
163             counter_kar_rav += 1
164
165     fs1_kar_rav = frozenset(lst1_kar_rav)
166     fs2_kar_rav = frozenset(lst2_kar_rav)
167     myDict_kar_rav[0] = fs1_kar_rav
168     myDict_kar_rav[1] = fs2_kar_rav
169     set_node_community(G,community_rav)
170     set_edge_community(G)
171

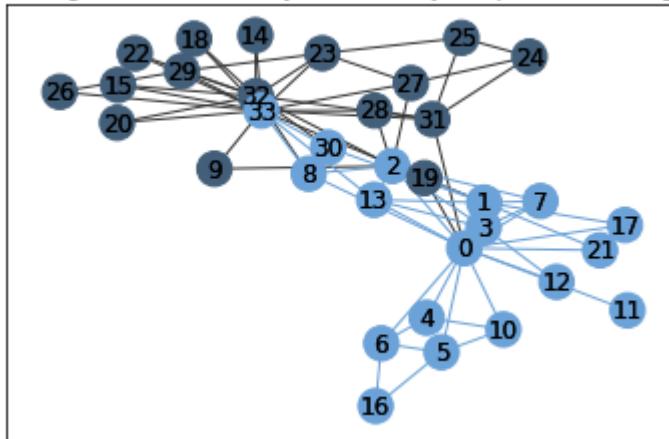
```

```

172     # Set community color for nodes
173     node_color = [
174         get_color(G.nodes[v]['community'])
175         for v in G.nodes]
176
177     node_color = [
178         get_color(G.nodes[v]['community'])
179         for v in G.nodes]
180
181     # Set community color for internal edges
182     external = [
183         (v, w) for v, w in G.edges
184         if G.edges[v, w]['community'] == 0]
185     internal = [
186         (v, w) for v, w in G.edges
187         if G.edges[v, w]['community'] > 0]
188     internal_color = [
189         get_color(G.edges[e]['community'])
190         for e in internal]
191
192     karate_pos = nx.spring_layout(G)
193     ax = plt.gca()
194     ax.set_title('Ravasz Algorithm Modularity Community Graph for karate graph')
195     # Draw external edges
196     nx.draw_networkx(
197         G, pos=karate_pos, node_size=0,
198         edgelist=external, edge_color="#333333")
199     # Draw nodes and internal edges
200     nx.draw_networkx(
201         G, pos=karate_pos, node_color=node_color,
202         edgelist=internal, edge_color=internal_color, ax=ax)

```

Ravasz Algorithm Modularity Community Graph for karate graph



```

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

```

```

Accuracy achieved from Ravasz algorithm compared to Original community values : 0.911764
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).

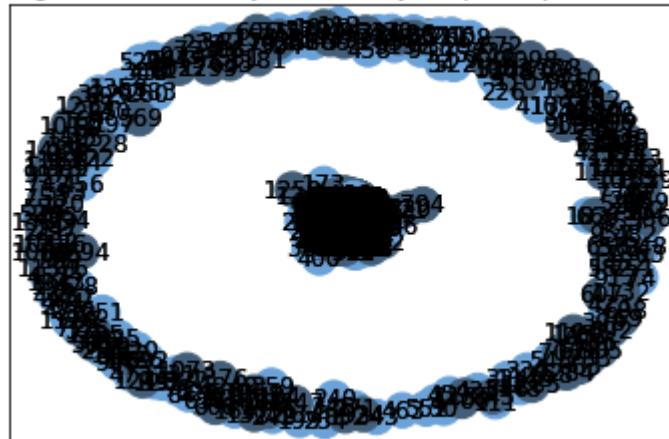
```

1 G = nx.read_gml('/content/polblogs.gml', label='id')
2 G = nx.Graph(G)
3 G.remove_edges_from(nx.selfloop_edges(G))
4
5
6 community_rav = ravasz_algorithm(G)
7
8 myDict_rav = {}
9 lst1_rav = []
10 lst2_rav = []
11 counter_rav = 0
12 for v in community_rav:
13     if v == 0:
14         # Add 1 to save 0 for external edges
15         lst1_rav.append(counter_rav)
16     else:
17         lst2_rav.append(counter_rav)
18     counter_rav += 1
19

```

```
20 fs1_rav = frozenset(lst1_rav)
21 fs2_rav = frozenset(lst2_rav)
22 myDict_rav[0] = fs1_rav
23 myDict_rav[1] = fs2_rav
24
25 set_node_community_2(G,community_rav)
26 set_edge_community_2(G)
27 node_color = [
28     get_color_2(G.nodes[v]['community'])
29     for v in G.nodes]
30 node_color = [
31     get_color_2(G.nodes[v]['community'])
32     for v in G.nodes]
33 external = [
34     (v, w) for v, w in G.edges
35     if G.edges[v, w]['community'] == 0]
36 internal = [
37     (v, w) for v, w in G.edges
38     if G.edges[v, w]['community'] > 0]
39 internal_color = [
40     get_color_2(G.edges[e]['community'])
41     for e in internal]
42
43 karate_pos = nx.spring_layout(G)
44 ax = plt.gca()
45 ax.set_title('Ravasz Algorithm Modularity Community Graph for political blogs data')
46 # Draw external edges
47 nx.draw_networkx(
48     G, pos=karate_pos, node_size=0,
49     edgelist=external, edge_color="#333333")
50 # Draw nodes and internal edges
51 nx.draw_networkx(
52     G, pos=karate_pos, node_color=node_color,
53     edgelist=internal, edge_color=internal_color)
```

Ravasz Algorithm Modularity Community Graph for political blogs data



```
1 print('Accuracy achieved from Ravasz algorithm compared to Original community values')
2 print('Number of communities :', len(myDict_rav))
3 print('Number of nodes in community 1:', len(myDict_rav[0]))
4 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

● X