In [3]:

```
'''
(30 points) Part 5: Community Detection

User-hashtag relations have been extracted and saved in the file s3://us-congress-t
weets/user_hashtags.csv. If a user uses a hashtag there will be a record with the u
serid and the hashtag.
Use the Trawling algorithm discussed in class to find potential user communities in
the dataset. (Hint: use FPGrowth in the Spark ML package). Explore different values
for the support parameter.
'''
```

```
An error was encountered:
Invalid status code '404' from https://172.31.6.195:18888/sessions/1 wi
th error payload: {"msg":"Session '1' not found."}
```

In [2]:

```
user_hashtags = spark.read.csv("s3://us-congress-tweets/user_hashtags.csv", header=
True)

reply_network = spark.read.csv("s3://us-congress-tweets/reply_network.csv", header=
True)
```

```
'''
  Implements the Louvain method.
  Input: a weighted undirected graph
  Ouput: a (partition, modularity) pair where modularity is maximum
'''
class PyLouvain:

  '''
    Builds a graph from _path.
    _path: a path to a file containing "node_from node_to" edges (one per line)
  '''
  @classmethod
  def from_file(cls, path):
    f = open(path, 'r')
    lines = f.readlines()
    f.close()
    nodes = {}
    edges = []
    for line in lines:
      n = line.split()
      if not n:
```

```
            break
        nodes[n[0]] = 1
        nodes[n[1]] = 1
        w = 1
        if len(n) == 3:
            w = int(n[2])
        edges.append(((n[0], n[1]), w))
    # rebuild graph with successive identifiers
    nodes_, edges_ = in_order(nodes, edges)
    print("%d nodes, %d edges" % (len(nodes_), len(edges_)))
    return cls(nodes_, edges_)

'''
    Builds a graph from _path.
    _path: a path to a file following the Graph Modeling Language specification
'''
@classmethod
def from_gml_file(cls, path):
    f = open(path, 'r')
    lines = f.readlines()
    f.close()
    nodes = {}
    edges = []
    current_edge = (-1, -1, 1)
    in_edge = 0
    for line in lines:
        words = line.split()
        if not words:
            break
        if words[0] == 'id':
            nodes[int(words[1])] = 1
        elif words[0] == 'source':
            in_edge = 1
            current_edge = (int(words[1]), current_edge[1], current_edge[2])
        elif words[0] == 'target' and in_edge:
            current_edge = (current_edge[0], int(words[1]), current_edge[2])
        elif words[0] == 'value' and in_edge:
            current_edge = (current_edge[0], current_edge[1], int(words[1]))
        elif words[0] == ']' and in_edge:
            edges.append(((current_edge[0], current_edge[1]), 1))
            current_edge = (-1, -1, 1)
            in_edge = 0
    nodes, edges = in_order(nodes, edges)
    print("%d nodes, %d edges" % (len(nodes), len(edges)))
```

```
    return cls(nodes, edges)

'''
    Initializes the method.
    _nodes: a list of ints
    _edges: a list of ((int, int), weight) pairs
'''
def __init__(self, nodes, edges):
    self.nodes = nodes
    self.edges = edges
    # precompute m (sum of the weights of all links in network)
    #            k_i (sum of the weights of the links incident to node i)
    self.m = 0
    self.k_i = [0 for n in nodes]
    self.edges_of_node = {}
    self.w = [0 for n in nodes]
    for e in edges:
        self.m += e[1]
        self.k_i[e[0][0]] += e[1]
        self.k_i[e[0][1]] += e[1] # there's no self-loop initially
        # save edges by node
        if e[0][0] not in self.edges_of_node:
            self.edges_of_node[e[0][0]] = [e]
        else:
            self.edges_of_node[e[0][0]].append(e)
        if e[0][1] not in self.edges_of_node:
            self.edges_of_node[e[0][1]] = [e]
        elif e[0][0] != e[0][1]:
            self.edges_of_node[e[0][1]].append(e)
    # access community of a node in O(1) time
    self.communities = [n for n in nodes]
    self.actual_partition = []


'''
    Applies the Louvain method.
'''
def apply_method(self):
    network = (self.nodes, self.edges)
    best_partition = [[node] for node in network[0]]
    best_q = -1
    i = 1
    while 1:
        #print("pass #%d" % i)
```

```
        i += 1
        partition = self.first_phase(network)
        q = self.compute_modularity(partition)
        partition = [c for c in partition if c]
        #print("%s (%.8f)" % (partition, q))
        # clustering initial nodes with partition
        if self.actual_partition:
            actual = []
            for p in partition:
                part = []
                for n in p:
                    part.extend(self.actual_partition[n])
                actual.append(part)
            self.actual_partition = actual
        else:
            self.actual_partition = partition
        if q == best_q:
            break
        network = self.second_phase(network, partition)
        best_partition = partition
        best_q = q
    return (self.actual_partition, best_q)

'''
    Computes the modularity of the current network.
    _partition: a list of lists of nodes
'''
def compute_modularity(self, partition):
    q = 0
    m2 = self.m * 2
    for i in range(len(partition)):
        q += self.s_in[i] / m2 - (self.s_tot[i] / m2) ** 2
    return q

'''
    Computes the modularity gain of having node in community _c.
    _node: an int
    _c: an int
    _k_i_in: the sum of the weights of the links from _node to nodes in _c
'''
def compute_modularity_gain(self, node, c, k_i_in):
    return 2 * k_i_in - self.s_tot[c] * self.k_i[node] / self.m

'''
```

```
        Performs the first phase of the method.
        _network: a (nodes, edges) pair
    '''
    def first_phase(self, network):
        # make initial partition
        best_partition = self.make_initial_partition(network)
        while 1:
            improvement = 0
            for node in network[0]:
                node_community = self.communities[node]
                # default best community is its own
                best_community = node_community
                best_gain = 0
                # remove _node from its community
                best_partition[node_community].remove(node)
                best_shared_links = 0
                for e in self.edges_of_node[node]:
                    if e[0][0] == e[0][1]:
                        continue
                    if e[0][0] == node and self.communities[e[0][1]] == node_community or e[0][1] ==
node and self.communities[e[0][0]] == node_community:
                        best_shared_links += e[1]
                self.s_in[node_community] -= 2 * (best_shared_links + self.w[node])
                self.s_tot[node_community] -= self.k_i[node]
                self.communities[node] = -1
                communities = {} # only consider neighbors of different communities
                for neighbor in self.get_neighbors(node):
                    community = self.communities[neighbor]
                    if community in communities:
                        continue
                    communities[community] = 1
                    shared_links = 0
                    for e in self.edges_of_node[node]:
                        if e[0][0] == e[0][1]:
                            continue
                        if e[0][0] == node and self.communities[e[0][1]] == community or e[0][1] == node
and self.communities[e[0][0]] == community:
                            shared_links += e[1]
                    # compute modularity gain obtained by moving _node to the community of
_neighbor
                    gain = self.compute_modularity_gain(node, community, shared_links)
                    if gain > best_gain:
                        best_community = community
                        best_gain = gain
```

```
                best_shared_links = shared_links
            # insert _node into the community maximizing the modularity gain
            best_partition[best_community].append(node)
            self.communities[node] = best_community
            self.s_in[best_community] += 2 * (best_shared_links + self.w[node])
            self.s_tot[best_community] += self.k_i[node]
            if node_community != best_community:
                improvement = 1
        if not improvement:
            break
    return best_partition

'''
    Yields the nodes adjacent to _node.
    _node: an int
'''
def get_neighbors(self, node):
    for e in self.edges_of_node[node]:
        if e[0][0] == e[0][1]: # a node is not neighbor with itself
            continue
        if e[0][0] == node:
            yield e[0][1]
        if e[0][1] == node:
            yield e[0][0]

'''
    Builds the initial partition from _network.
    _network: a (nodes, edges) pair
'''
def make_initial_partition(self, network):
    partition = [[node] for node in network[0]]
    self.s_in = [0 for node in network[0]]
    self.s_tot = [self.k_i[node] for node in network[0]]
    for e in network[1]:
        if e[0][0] == e[0][1]: # only self-loops
            self.s_in[e[0][0]] += e[1]
            self.s_in[e[0][1]] += e[1]
    return partition

'''
    Performs the second phase of the method.
    _network: a (nodes, edges) pair
    _partition: a list of lists of nodes
'''
```

```
def second_phase(self, network, partition):
    nodes_ = [i for i in range(len(partition))]
    # relabelling communities
    communities_ = []
    d = {}
    i = 0
    for community in self.communities:
        if community in d:
            communities_.append(d[community])
        else:
            d[community] = i
            communities_.append(i)
            i += 1
    self.communities = communities_
    # building relabelled edges
    edges_ = {}
    for e in network[1]:
        ci = self.communities[e[0][0]]
        cj = self.communities[e[0][1]]
        try:
            edges_[(ci, cj)] += e[1]
        except KeyError:
            edges_[(ci, cj)] = e[1]
    edges_ = [(k, v) for k, v in edges_.items()]
    # recomputing k_i vector and storing edges by node
    self.k_i = [0 for n in nodes_]
    self.edges_of_node = {}
    self.w = [0 for n in nodes_]
    for e in edges_:
        self.k_i[e[0][0]] += e[1]
        self.k_i[e[0][1]] += e[1]
        if e[0][0] == e[0][1]:
            self.w[e[0][0]] += e[1]
        if e[0][0] not in self.edges_of_node:
            self.edges_of_node[e[0][0]] = [e]
        else:
            self.edges_of_node[e[0][0]].append(e)
        if e[0][1] not in self.edges_of_node:
            self.edges_of_node[e[0][1]] = [e]
        elif e[0][0] != e[0][1]:
            self.edges_of_node[e[0][1]].append(e)
    # resetting communities
    self.communities = [n for n in nodes_]
    return (nodes_, edges_)
```

```
'''
    Rebuilds a graph with successive nodes' ids.
    _nodes: a dict of int
    _edges: a list of ((int, int), weight) pairs
'''
def in_order(nodes, edges):
        # rebuild graph with successive identifiers
        nodes = list(nodes.keys())
        nodes.sort()
        i = 0
        nodes_ = []
        d = {}
        for n in nodes:
            nodes_.append(i)
            d[n] = i
            i += 1
        edges_ = []
        for e in edges:
            edges_.append(((d[e[0][0]], d[e[0][1]]), e[1]))
        return (nodes_, edges_)
```