Emilia Bourgeois and Nick Zumpano
CSCI 347
3/23/2022
Project 2

Problem 1: Think about the data

We chose the Twitch Gamers Social Network dataset from SNAP for our project. It is a social network of users on the website Twitch, which was formerly a video game streaming platform that has now become a general one. The data was collected from the public API in 2018. We chose the dataset because we are both avid users of the platform and wanted to work with data with terminology we could understand. It is far easier to perform analysis on a dataset that we understand the content of. Also, the file size was quite small given the almost 7 million edges, and download speed was a limiting factor. We chose not to pre-process the data, we wanted to process all of it and our desktop PCs are both quite powerful. We believe the nodes with the highest centrality will have a balance of most views, will not be a dead account, and will all be affiliates. The language is likely to be predominately English. Twitch affiliate is a low bar to achieve for a streamer, as in it is a requirement to being even remotely successful so the central nodes must have affiliate. Views are a necessity for the most central node because each view is a user who has viewed their stream. However, high view count doesn't make the channel stand out as the one with the highest centrality since interaction also matters. In the same vein life time hours stream is a requirement but not an indication of the most central node. A dead account cannot possibly hold enough interaction to compete with the active ones so it would be doubtful to see a high centrality on one unless it has just died at the time of data collection.

Problem 2: Write python code for graph analysis

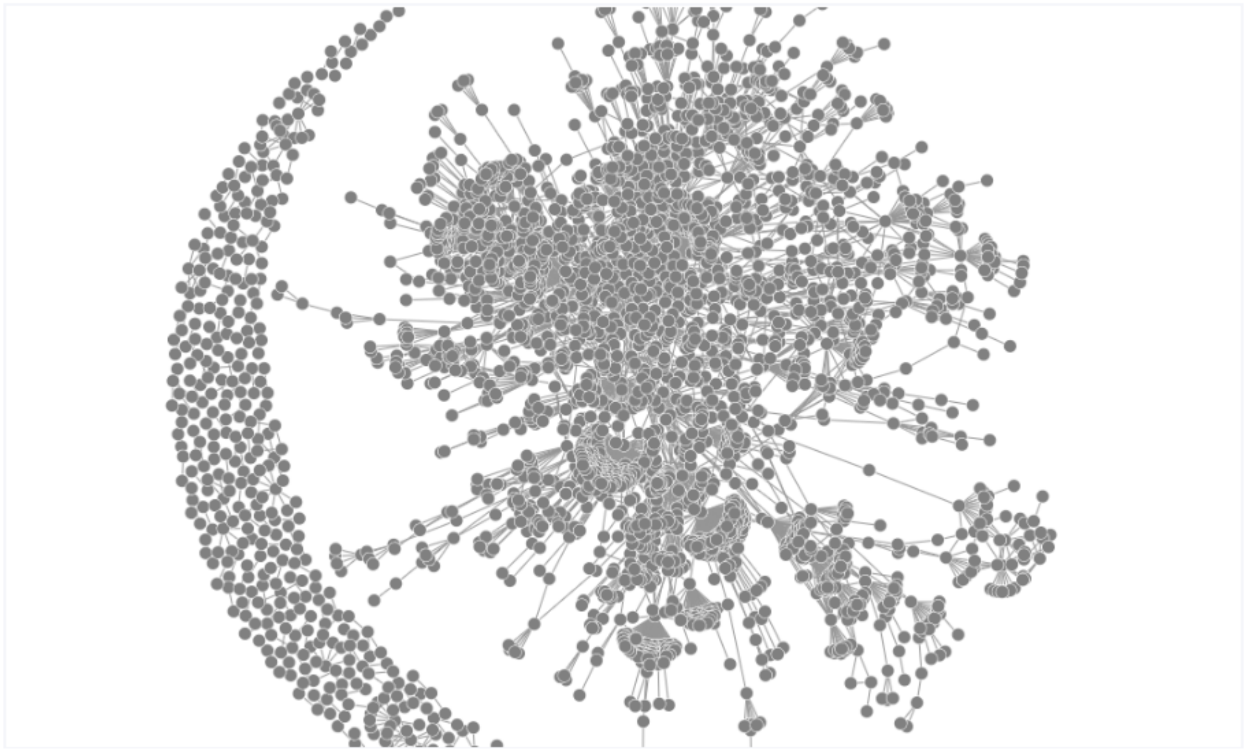
1-6.

```
import copy

import networkx as nx
import sys
import numpy as np
from webweb import Web


def DFS(g,v,visited): #DFS to count nodes
    visited.append(v)
    for edge in g.edges(v):
        if edge[1] not in visited:
                DFS(g,edge[1],visited)
def get_neighbors(g,id):
    neighbors = []
    for edge in g.edges(id):
        neighbors.append(edge[1])
    return neighbors
def num_verts(g):
```

```python
    visited = []
    nodes = list(g.nodes())
    DFS(g,nodes[0],visited)
    visited.sort()
    return len(visited)
def calcdegree(g, id):
    sum = 0
    for edge in g.edges(id):
        sum+=1
    return sum
def clustering(g,id):
    c = 0
    neighbors = get_neighbors(g,id)
    counted = []
    for n in neighbors:
        for edge in g.edges(n):
            if edge[0] in neighbors and edge[1] in neighbors:
                edge = sorted(edge)
                if edge not in counted:
                    c+=1
                    counted.append(edge)
    val = copy.deepcopy(calcdegree(g, id))
    bottom = ((val-1)*(val))
    if bottom == 0:
        coeff = 0
    else:
        coeff = 2*(float(c))/bottom
    return coeff

def betweenness(g,id):
    sum = 0
    top = 0
    bot = 0
    for i in g:
        for j in g:
            if i != j and i != id:
                paths = nx.all_shortest_paths(g, source=i, target=j)
                for p in paths:
                    bot+=1
                    if id in p:
                        top +=1
            if bot != 0:
                sum+=(top/bot)
        print(i)
    return sum

def adjacency(g):
    #returns a sorted matrix with newly mapped vals
    vals = []
    for i in range(len(g.nodes())):
        vals.append(i)
    labels = dict(zip(sorted(g.nodes()), vals))
    newg = nx.relabel_nodes(g,mapping=labels)
    m = []
    n = num_verts(g)
    for i in range(n):
        m.append([0 for i in range(n)])
    for edge in newg.edges():
        m[edge[0]][edge[1]] = 1
        m[edge[1]][edge[0]] = 1

    return m, labels
def prestige(m):
    vector = np.ones(len(m))
    lastv = np.ones(len(m))
    i = 0
    while i < 1000:
        lastv = vector
        vector = np.dot(m,lastv)
        i+=1
        print(vector)
    return vector
def main():
    sys.setrecursionlimit(10000)
    f = open("oregon1_010331.txt", 'r')
    g = nx.Graph()
    f.readline()
    f.readline()
    f.readline()
    f.readline()
    for line in f:
        arr = line.split("\t")
        arr[1] = arr[1].strip("\n")
        g.add_edge(int(arr[0].strip()),int(arr[1].strip()))
    edge_list = []
    for edge in g.edges:
        edge_list.append([edge[0],edge[1]])
    degrees = []
    edge_list = edge_list[-4000:]
    web = Web(edge_list)
    web.show()
    for node in g.nodes:
        degrees.append(calcdegree(g,node))
    x = dict(zip(g.nodes,degrees))
    x = {k: v for k, v in sorted(x.items(), key=lambda item: item[1])}
    print("Top 10 nodes by highest degree in reverse order, nodes come
first")
    print(list(x.items())[-10:])
    x = {k: v for k, v in sorted((nx.betweenness_centrality(g, 100).items()),
key=lambda item: item[1])}
    print("Top 10 nodes by betweeness in reverse order, nodes come first")
    print(list(x.items())[-10:])
    print("Top 10 nodes by clustering in reverse order, nodes come first")
    cls = []
    for node in g.nodes:
        cls.append(clustering(g, node))
    x = dict(zip(g.nodes, cls))
    x = {k: v for k, v in sorted(x.items(), key=lambda item: item[1])}
    print(list(x.items())[-10:])
    print("Top 10 nodes by prestige in reverse order, nodes come first")
    x = {k: v for k, v in sorted(nx.eigenvector_centrality(g).items(),
key=lambda item: item[1])}
    print(list(x.items())[-10:])
    print("Top 10 nodes by pagerank in reverse order, nodes come first")
    x = {k: v for k, v in sorted(nx.pagerank(g).items(), key=lambda item:
item[1])}
    print(list(x.items())[-10:])

main()
```

Problem 3: Analyze the Graph

1.



2.
Top 10 nodes by highest degree in reverse order, nodes come first
[(3257, 333), (3356, 335), (2914, 416), (3549, 431), (1, 565), (209, 581), (3561, 871), (7018, 936), (1239, 1259), (701, 2312)]

3.
Top 10 nodes by betweenness in reverse order, nodes come first
[(174, 0.04283316493337106), (702, 0.05337495961200043), (3549, 0.05537729156211078), (6453, 0.05662069497085098), (3257, 0.06396879131191655), (209, 0.08259667696603175), (7018, 0.08735296747493057), (1239, 0.10646159457350601), (3561, 0.1820552785886929), (701, 0.34015245954784956)]
4.
Top 10 nodes by clustering in reverse order, nodes come first
[(5419, 1.0), (13, 1.0), (6365, 1.0), (7015, 1.0), (1967, 1.0), (2592, 1.0), (8831, 1.0), (8474, 1.0), (8505, 1.0), (8707, 1.0)]

5.
Top 10 nodes by prestige in reverse order, nodes come first
[(3356, 0.09887946492274846), (2548, 0.1111562683277448), (3549, 0.11407307358974339), (2914, 0.11629780294415192), (209, 0.12012126491848947), (1, 0.14725879134015898), (3561, 0.19100671395455188), (7018, 0.19135329951423252), (1239, 0.2720271387684948), (701, 0.5026151117439306)]

6.
Top 10 nodes by pagerank in reverse order, nodes come first
[(702, 0.006269199995996183), (3356, 0.006433841545201565), (2914, 0.007789164622599538), (3549, 0.0080814249059467), (1, 0.010757453779243553), (209, 0.011712187329726476), (3561, 0.017046491424731483), (7018, 0.019160208841147105), (1239, 0.025500045346126935), (701, 0.04838051737177157)]

7.
Unfortunately, in order to actually get the program to run due to the 160,000 nodes and 7 million edges in the Twitch data set, we had to use a separate dataset linked here: https://snap.stanford.edu/data/Oregon-1.html. There isn't much information on this network at all, so the explanation will be limited. We can see quite a few nodes appear multiple times in our top 1239, 209, 7018, 701, and 1 are there quite often, in fact the top 10 that differs the most would be the clustering list because there was problem too many nodes with a clustering coefficient of one. Otherwise, we can clearly see a pattern of the central nodes of the graph in the other lists.