**Twitter community detection**

**Overview:** Get my Twitter friend egonet and find the community of my friends.

**Data:** My Twitter friend list and my Twitter friends' friend list. I got access to them via twitter4j.api. It's 66 vertex and 365 edges in total.

**Question: 1:** How to define community in a directed graph like Twitter?

**2.** What are the communities in my Twitter friend’s egonet.

**Data Structures:**

vertex: represented as integers

**Edges:** Includes a HashSet that stores all neighbor vertexes of a center vertex and a hashtable that stores all neighbor vertexes as key and betweenness as value. (I know it's redundant but I'm just too lazy to refine it).

**CapGraph:** a HashMap with all vertexes as key and corresponding Edges as value.

**TwitterFriendInfo:** a data structure that is used to get the corresponding user name and user Id(by a user name list and user id list) of a given vertex. It also stores a HashSet of friend Id to rule out irrelevant users and build my egonet.

**Algorithms:**

**Question 1 algorithm:** the SCC(strongly connected component) algorithm from warm-up assignment is used. It uses DFS to traverse the graph once, and then traverse a reversed graph once.

**Complexity:** O(|V| + |E|) for DFS and build a reversed graph, so overall complexity is the same.

**Result:** for a relatively small graph like mine, SCC itself gives very nice community information. I can see that all Japanese drawers and voice actors are in one group. Which makes good sense because they are all related to Japanese animes. Elon musk and Cathie Wood are in the same group. Most of the politicians are in another group. My personal friends in one group, etc. Therefore, SCC definitely serves as a good definition of community in a directed graph. Actually it's too good that I feel unnecessary to build another algorithm for question 2.

**Correctness:** SCC correctness is verified by the warmup assignment. So there's no need to worry about it.

**Question 2 algorithm:**

Input My Twitter egonet graph **tg** (with myself removed, because I'm for sure following everyone in my egonet), target group number **n**.

List<graph> **scc** = **tg**.getSCC()

while **scc**.length < **n**:

for each **subGraph** in **scc**, do:

initialize betweeness for all Edges in the subGraph.

initialize a hash map **visited** that stores the vertexes visited as key and the vertex which is the previous vertex to reach

the newly visited vertex as value.

initialize a **queue** for BFS.

for each vertex**v** in subGraph, do:

put (**v**, -1) into visited, -1 is used to indicate that we reach the start point when backtracing the route.

put **v** into **queue**

while **queue** is not empty:

**from** = queue.pop()

for all neighbors '**to**' of v1:

if 'to' is not in visited, push 'to' in **queue**, put (**to**, **from**) in **visited**

**cu**r = **to**, **prev** =**from**

while **cur** != -1:

betweeness of edge(**cur**,**prev**) += 1

**cur** = **prev**, **prev** = **value of visited(prev)**

loop over all edges in all **subGraph**, record the **maximum betweeness** and the corresponding **edge(from, to)**

remove the **edge(from, to)** from tg.

**scc** = **tg**.getSCC()

for each**scc** print out the username of all vertex.

**Complexity:**

maximum remove edge number = O(|E|)

for each removal, get SCC cost O(|V| +|E|)

loop over all edges to get maximum betweenness edge cost O(|V| + |E|)

calculate for betweenness:

Loop over all vertexes is O(|V|)

for each loop, Do BFS, which cost O(|V|+|E|) to visit all vertexes

for each visit, backtrace the shortest path can cost up to O(|V|)

print out cost O(|V|)

Therefore, the overall complexity is O(|E| \* (|V| + |E| + |V| + |E| + |V|\*(|V|+|E|)\*|V|) ) = O(|E| \* |V|^2 \* |V+E|). Since we are calculating the betweenness of each subGraph in SCC, the actual complexity should be much smaller than this value.

**Correctness:** I built some small sample data to verify the correctness. I counted the betweenness of these samples manually and check if the program gives the same result. I checked the removed edge to make sure the edge with the largest betweenness is removed. I further checked whether they are divided into multiple groups as I expected.

After the test cases, I apply the algorithm to my twitter graph, the voice actors and drawers for animes are separated into 2 groups, which makes a lot of sense to me. With a larger n number(required community number) the community is further divided into smaller parts. Each part is very densely connected. Which further verifies the correctness of this algorithm.

**Reflection:** at first I was trying to make an undirected graph from Twitter. Because the course video uses an undirected graph to illustrate betweenness and community. I simply make all edges undirected from my directed twitter graph. However, the result looks terrible. For example one of my personal friends is grouped with big guys like Elon Musk, Bill Gates and cannot be separated. Just because my friend follows these big guys doesn't mean he is one of them. Therefore, I decided to use a directed graph and try SCC to get community. Luckily, it works quite well.

I figured out that most parts of my friends are already grouped well, and maybe too well(about 18 of my friends stay in a group with only themselves.) I know it's because my input egonet is small. Some of my following just can't find the connection to others in my small egonet. But there's still a large community that contains all the anime guys: voice actors and drawers. There are still 30 people in that community and I know they can be further separated. I developed the betweenness algorithm and am glad to see it works.

I know my algorithm is not strictly following the Girvan-Newman method. That method uses a more complicated algorithm to calculate for average value when there are multiple shortest paths to an edge. However, I just use BFS to get the first shortest path which creates some bias. However, for the test cases, my algorithm works fine. Therefore I think it's good enough to use.

  I'm also very glad that I choose to use my own data. Therefore I can deal with vertexes which I can get user names I'm familiar with rather than just deal with meaningless integers. The hardest part of this project is actually to get my Twitter friend info in a text doc, which takes me weeks. I learned to use the twitter4j API and finally got all the data I need for my graph. Then I need to build a TwitterFriendInfo data structure that can load the txt file and store the vertex value and corresponding user name and id so that I can print out meaningful results.