
Social Network Analysis - Proof of Concept

Algorithms and Data Structures

Khanh Nguyen

University of the Cumberland

knguyen19181@ucumberland.edu

1 Overview

1.1 Problem Statements

A social network is incredibly useful for maintaining friendships, reaching out, and following your favorite celebrity. In particular, Twitter is one social media that allows people to share and exchange ideas freely and easily. Twitter offers many features that allow interactions between end-users and followers, and one of them would be Retweet. Retweet is an important concept as it determines how trends form and how tweets go viral. Distributing content is as simple as a click of a button, and social network analysis can help us understand the feature's behavior behind the scenes at a larger scale.

1.2 Solution

Social network analysis is the process of investigating social structures using networks and graph theories. It combines various techniques for analyzing the structure of social networks and theories that aim to explain the underlying dynamics and patterns observed in these structures. It is an inherently interdisciplinary field originally from social psychology, statistics, and graph theory. Today, we will investigate, analyze, and learn more about the Social Network's algorithm to find influential users in a social network and how influential users grow daily.

2 Implementation

Twitter has a natural model of end-users following their favorite celebrity, which contradicts Facebook (Meta), where people can connect to each other bidirectionally. Therefore, to extend the features, we can build based on the following algorithms:

- Degree centrality
- Shortest path
- Betweenness centrality

we will use an un-directed graph with Facebook datasets:

```
23 def downloadDataset() -> list[str]:
24     facebook = 'https://assets.datacamp.com/production/repositories/580/datasets/69ada08d5cce7f35f38ffef8f2291
25     twitter = 'https://assets.datacamp.com/production/repositories/580/datasets/64cf6963a7e8005e3771ef3b256817
26     dataDir = Path('data/')
27     datasets = [twitter, facebook]
28     dataPaths = list()
29     for data in datasets:
30         fileName = data.split('/')[-1].replace('?raw=true', '')
31         dataPath = dataDir / fileName
32         createDirSaveFile(dataPath, data)
33         dataPaths.append(dataPath)
34
35     return dataPaths
```

Figure 1: Download Facebook Dataset

Since there is a restricted limitation on the number of nodes and edges when drawing with Matplotlib, we will use the resampling method [1] to reduce the size and only consider the nodes that have the highest degree to simulate Facebook's network. In this case, the 200 nodes with the highest degree will be considered:

```
def createSocialNet() -> SocialNetwork:
    dataPath = downloadDataset()
    with open(dataPath[1], 'rb') as f:
        fbGraph = pickle.load(f)

    # Since plotlib has a problem with performance for over than 2000 nodes with the corresponding edges
    # We will only take the 200 nodes which have the highest degree (nodes that have most edges)
    # and simulate a subgraph of fb
    fbNodeDegree = dict(fbGraph.degree())
    sortedNodes = sorted(fbNodeDegree.items(), key=lambda x: x[1], reverse=True)
    topDegreeNodes = [node for node, _ in sortedNodes[:200]]
    fbSubGraph = fbGraph.subgraph(topDegreeNodes)
    print("Number of nodes", len(fbSubGraph.nodes()))
    print("Number of edges", len(fbSubGraph.edges()))
    print("Number of nodes", fbSubGraph.nodes())
    sn = SocialNetwork(fbSubGraph)
    # sn.drawGraph()
    return sn
```

Figure 2: Create Social Network in Python

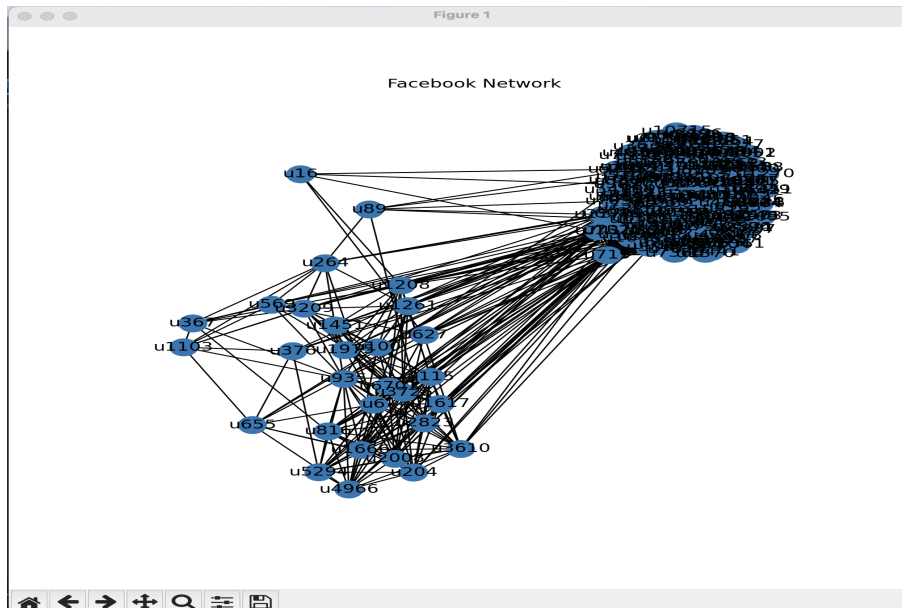


Figure 3: Facebook Network

The following key operations include four sections:

- Find important people
- Find largest communities
- Connect communities
- Recommend friends

2.1 Find important people

Degree centrality[2] is used to find the "popularity" of each node in the graph database based on the number of connections it has to other nodes. On Facebook, it indicates the number of friends or a number of followers an end-user can have before recommending that user to others in a similar community.

$$DC(v_i) = \frac{1}{N-1} \sum_{j=1}^N \alpha_{i,j}$$

Therefore, we will use the same algorithm to find the most important people in the social network:

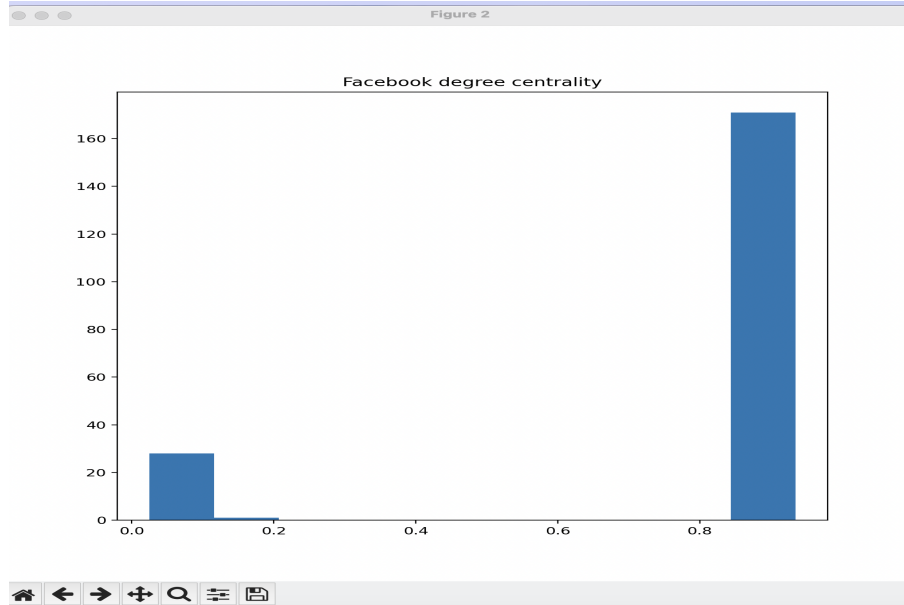


Figure 4: Facebook Histogram Distribution for degree centrality

As we can see in the histogram distribution graph, most of the nodes have a degree of centrality in the range of (0.8, 0.95). However, for simplicity, we will determine the most important people by finding the person with a maximum degree of centrality.

```

23
24 # will use degree centrality to determine the most important Facebook person
25 # that have many friends or being followed by others facebook user
26 def findImportantPeople(self) -> list[tuple]:
27     degCent = nx.degree_centrality(self.fbGraph)
28     degCent = {k: v for k, v in sorted(degCent.items(), key=lambda item: item[1], reverse=True)}
29     maxDegCent = max(list(degCent.values()))
30     importantPeople = [(n, dc) for n, dc in degCent.items() if dc == maxDegCent]
31     return importantPeople
32

```

Figure 5: Find important people in Python

As a result, the person who has the maximum degree of centrality would be user 719, with a degree centrality of 0.9346733668341709

```

2024-09-22 17:02:42.620 Python[18474:265457] WARNING: Secure coding is not enabled for restorable state! Enable secure coding by implementing NSApplicationDelegate.applicationSupportsSecureRestorableState: and returning YES.
The most prolific people [(u'719', 0.9346733668341709)]
To help quiet these messages, please see:

```

Figure 6: Find important people's output

2.2 Find largest communities

A clique [3] is a subset of nodes in a graph such that every two distinct nodes are adjacent; in other words, a clique is a complete subgraph. It is often used with a degree of centrality to locate maximal cliques and find clusters for the purpose of identifying a target demographic[4]. Therefore, we will use the same clique to identify the largest communities that share a similar portion of interest to each other.

```
# findImportantPeople will use degree centrality to determine the most important Facebook person
# that have many friends or being followed by others facebook user
def findImportantPeople(self) -> list[tuple]:
    degCent = nx.degree_centrality(self.fbGraph)
    degCent = {k: v for k, v in sorted(degCent.items(), key=lambda item: item[1], reverse=True)}
    maxDegCent = max(list(degCent.values()))
    importantPeople = [(n, dc) for n, dc in degCent.items() if dc == maxDegCent]
    return importantPeople
```

Figure 7: Find largest communities in Python

In this case, the largest clique is

```
Facebook largest clique ['u9745', 'u2594', 'u8692', 'u3428', 'u10261', 'u9046', 'u3777', 'u8871', 'u10503', 'u3242',
'u3669', 'u10404', 'u7368', 'u6980', 'u7377', 'u5395', 'u1571', 'u719', 'u4951', 'u3798', 'u3636', 'u587', 'u2170',
'u10813', 'u10742', 'u8125', 'u10623', 'u6196', 'u7302', 'u5470', 'u5132', 'u1851', 'u7356', 'u5205', 'u6245', 'u67
39', 'u7034', 'u6837', 'u8643', 'u7459', 'u10274', 'u6361', 'u75', 'u6565', 'u3219', 'u8093', 'u7025', 'u6664', 'u58
17', 'u172', 'u3468', 'u10032', 'u8726', 'u10559', 'u9610', 'u1204', 'u9489', 'u2994', 'u4296', 'u3561', 'u1080', 'u
639', 'u5457', 'u9123', 'u2925', 'u1370', 'u4824', 'u9770', 'u10000', 'u5298', 'u9096', 'u7607', 'u3355', 'u1011', 'u
u1039', 'u982', 'u4606', 'u5032', 'u8924', 'u8315', 'u698', 'u5269', 'u1414', 'u3683', 'u7473', 'u5882', 'u8523', 'u
397', 'u8494', 'u3283', 'u5184', 'u1330', 'u6335', 'u4209', 'u255', 'u724', 'u4604', 'u4658', 'u7952', 'u6552', 'u32
48', 'u3082', 'u8101', 'u157', 'u1270', 'u9551', 'u6726', 'u17', 'u8625', 'u8057', 'u4588', 'u7525', 'u6158', 'u8520'
'u2218', 'u4029', 'u7471', 'u3075', 'u2480', 'u1481', 'u5108', 'u10537', 'u1588', 'u167', 'u1073', 'u4754', 'u164
8', 'u5331', 'u2520', 'u1469', 'u1695', 'u10143', 'u5602', 'u9437', 'u10715', 'u1687', 'u6441', 'u2806', 'u9965', 'u
1327', 'u741', 'u3092', 'u1351', 'u2285', 'u8900', 'u3841', 'u9369', 'u817', 'u7624', 'u10642', 'u3346', 'u4819', 'u
855', 'u6438', 'u2476', 'u6949', 'u1262', 'u8365', 'u8175', 'u4312', 'u1387', 'u4077', 'u201', 'u10862', 'u607', 'u3
233', 'u664', 'u7608', 'u186', 'u527', 'u530']
```

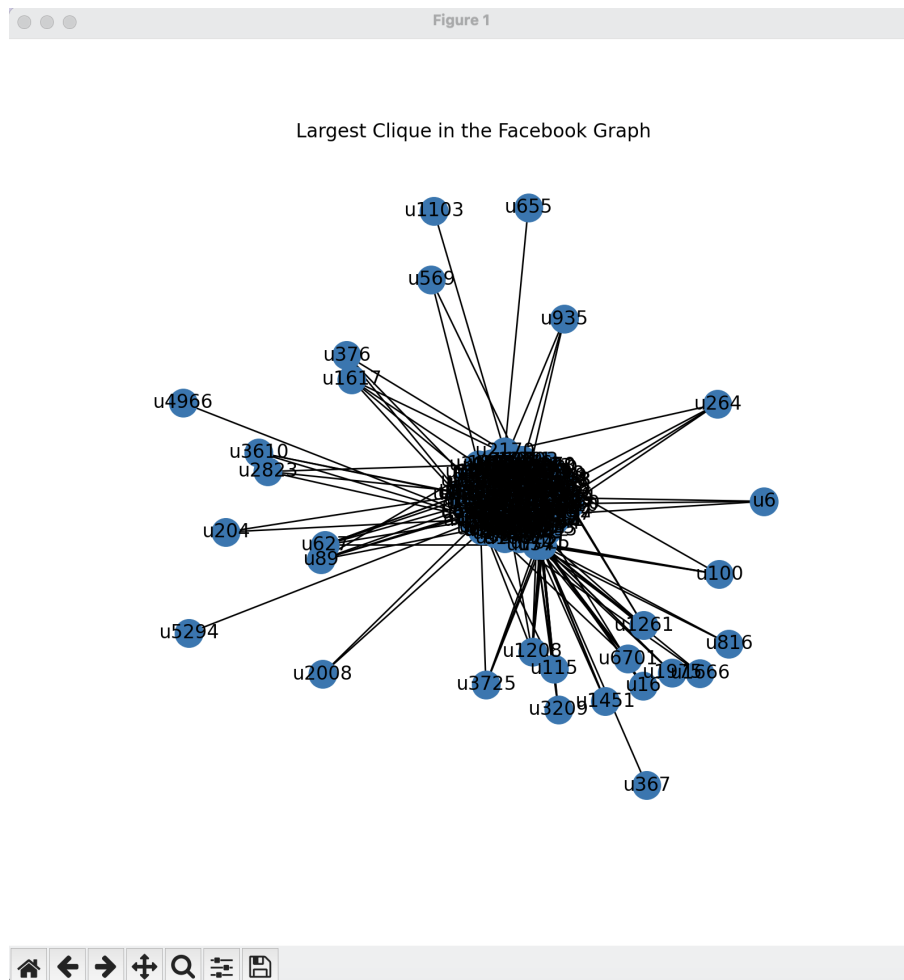


Figure 8: Find largest communities's output

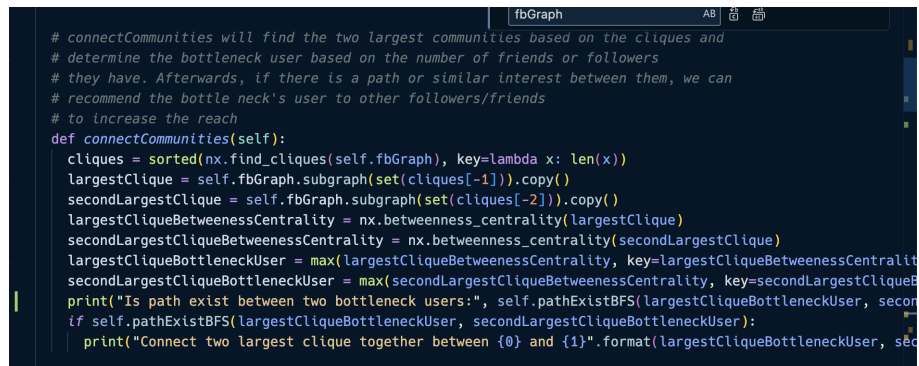
2.3 Connect communities

The shortest path[5] is the problem of finding a path between two articles in a graph such that the sum of the weights of its constituent edges is minimized. It is also a famous use case within Facebook that can be solved for many features:

- **Influencer Outreach:** When a brand tries to reach a specific influencer with a large following, the shortest path can determine the shortest route, minimize the intermediate users, and increase the likelihood for the brand to reach out to the influencer.
- **Content Promotion:** When a content creator wants to advertise their product via news, The shortest path algorithm can suggest the most effective sequence of users to engage with, maximizing the new's reach and visibility while minimizing the number of intermediate users.

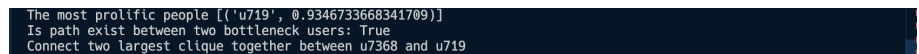
Similarly, we can use the shortest path with cliques to determine if we can connect the two communities together via the "bottleneck" user. Before that, we would need to find the two largest communities with cliques, and we will use betweenness centrality[6], which is used to find bottleneck nodes in a graph, in order to reach larger communities later on and increase the reach between people in different communities. To determine if these two bottlenecks users can connect to each other, we will shorten the least communication by determining the shortest path with BFS [7].

$$B(u) = \sum_{u \neq v \neq w} \frac{\sigma_{v,w}(u)}{\sigma_{v,w}}$$



```
# connectCommunities will find the two largest communities based on the cliques and
# determine the bottleneck user based on the number of friends or followers
# they have. Afterwards, if there is a path or similar interest between them, we can
# recommend the bottle neck's user to other followers/friends
# to increase the reach
def connectCommunities(self):
    cliques = sorted(nx.find_cliques(self.fbGraph), key=lambda x: len(x))
    largestClique = self.fbGraph.subgraph(set(cliques[-1])).copy()
    secondLargestClique = self.fbGraph.subgraph(set(cliques[-2])).copy()
    largestCliqueBetweennessCentrality = nx.betweenness_centrality(largestClique)
    secondLargestCliqueBetweennessCentrality = nx.betweenness_centrality(secondLargestClique)
    largestCliqueBottleneckUser = max(largestCliqueBetweennessCentrality, key=largestCliqueBetweennessCentrality)
    secondLargestCliqueBottleneckUser = max(secondLargestCliqueBetweennessCentrality, key=secondLargestCliqueBetweennessCentrality)
    print("Is path exist between two bottleneck users:", self.pathExistBFS(largestCliqueBottleneckUser, secondLargestCliqueBottleneckUser))
    if self.pathExistBFS(largestCliqueBottleneckUser, secondLargestCliqueBottleneckUser):
        print("Connect two largest clique together between {0} and {1}".format(largestCliqueBottleneckUser, secondLargestCliqueBottleneckUser))
```

Figure 9: Connect communities in Python



```
The most prolific people [('u719', 0.9346733668341709)]
Is path exist between two bottleneck users: True
Connect two largest clique together between u7368 and u719
```

Figure 10: Connect communities's output

2.4 Recommended Friends

After connecting the two communities together, we will recommend users to each other to connect larger communities. we can determine a varies of factors such as hobby, characteristics. For simplicity, we will determine the recommendation between two users if they have similar friends by using the shortest path with BFS.

```

main.py 1, U x
main.py > pathExistBFS
2 # pathExistBFS will use breath first search
3 # https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/#
4 # to determine if the path between two vertexes are connected in the shortest way possible
5 # Step 1: Begins with a node, traverses all its adjacent
6 # Step 2: Once all adjacent are visited, then their adjacent are traversed
7 # Time Complexity: O(v + e) (since its traverse through all vertex and edge in worst case)
8 def pathExistBFS(G: nx.DiGraph, startVertex: int, endVertex: int) -> bool:
9     visited_nodes = set()
10    # Begin traverse from the starting node
11    queue = [startVertex]
12
13    # Traverse all its adjacent node
14    for node in queue:
15        # Once all adjacent are visited, then their adjacent are traversed
16        neighbors = G.neighbors(node)
17        if endVertex in neighbors:
18            print('Path exists between nodes {0} and {1}'.format(startVertex, endVertex))
19            return True
20        else:
21            visited_nodes.add(node)
22            queue.extend([n for n in neighbors if n not in visited_nodes])
23
24    # Check to see if there are any vertex that can be reached from the current node adjacent
25    # in order to know if there are any path that BFS can traverse
26    if node == queue[-1]:
27        print('Path does not exist between nodes {0} and {1}'.format(startVertex, endVertex))
28        return False
29

```

Figure 11: Bread First Search Shortest Path's algorithm

```

social_net.py > ...
9 class SocialNetwork:
69 # recommendedFriends will recommend friends who shared
70 # but hasn't been friends yet by creating a combination
71 # If there is no connection, we will recommend them
72 def recommendedFriends(self) -> list[tuple]:
73     recommendedFriends = defaultdict(int)
74     for node, _ in self.fbGraph.nodes(data=True):
75         # Create a combination of 2 with all the neighbor nodes that the node has
76         # and determine if there is a path between them. If not,
77         # add them
78         for firstNeighborFriend, secondNeighborFriend in combinations(self.fbGraph.neighbors(node), 2):
79             if not self.pathExistBFS(firstNeighborFriend, secondNeighborFriend):
80                 recommendedFriends[(firstNeighborFriend, secondNeighborFriend)] += 1
81
82     # Identify the top 10 pairs of users
83     sortedRecommendedFriends = sorted(recommendedFriends.values())
84     top10Pairs = [pair for pair, count in recommendedFriends.items() if count > sortedRecommendedFriends[-10]]
85     return top10Pairs
86
87 # pathExistBFS will use breath first search
88 # https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/#

```

Figure 12: Recommend Friends in Python

```

The most prolific people [(u'719', 0.9346733668341709)]
Is path exist between two bottleneck users: False
Top 10 recommended friends: [(u'741', 'u935'), (u'100', 'u719'), (u'719', 'u1261'), (u'6', 'u741')]

```

Figure 13: Recommend Friends's output

3 Encountered challenges

On the road to implementation, there are many challenges that we have encountered.

The first issue would be finding the datasets that fit with the Networkx package in order to create the Facebook un-directed graph. However, there is a course related to network analysis in the data camp that helps me by providing me with the necessary data.

```

facebook = 'https://assets.datacamp.com/production/repositories/580/datasets/69ada08d5cce7f35f38ffef8f2291b7cfd6000/github_users.p
twitter = 'https://assets.datacamp.com/production/repositories/580/datasets/64cf6963a7e8005e3771ef3b256812a579732f0/ego-twitter.p'
dataDir = Path('data/')
datasets = [twitter, facebook]

```

Figure 14: Dataset Problem

The second issue would be the Matplotlib limitation in showing the corresponding Networkx graph's node and edge. However, to resolve that issue, I have to use a sampling method[2] to reduce the size of nodes/edges down to 200 nodes with the highest degree.

```

# Since pilotlib has a problem with performance for over than 2000 nodes with the corresponding edges
# We will only take the 200 nodes which have the highest degree (nodes that have most edges)
# and simulate a subgraph of fb
fbNodeDegree = dict(fbGraph.degree())
sortedNodes = sorted(fbNodeDegree.items(), key=lambda x: x[1], reverse=True)
topDegreeNodes = [node for node, _ in sortedNodes[:200]]
fbSubGraph = fbGraph.subgraph(topDegreeNodes)
print("Number of nodes", len(fbSubGraph.nodes()))
print("Number of edges", len(fbSubGraph.edges()))
print("Number of nodes", fbSubGraph.nodes())

```

Figure 15: Sampling Facebook data

Since the package Networkx provides built-in functions that are suitable for network analysis use cases; therefore, reduce the friction for implementation and displaying the corresponding result in Matplotlib.

References

- [1] Nupur Jain (2024).
- [2] Jayant Bisht (2022). *Degree Centrality Centrality Measure*. GeeksForGeeks.
- [3] Karthik Reddy (2022). *Cliques in Graph*. GeeksForGeeks.
- [4] Craigory Coppola, Heba Elgazzar (2020). *Novel Machine Learning Algorithms for Centrality and Cliques Detection in YouTube Social Networks*. Cornell University.
- [5] Wikipedia. *Shortest Path Problem*.
- [6] Alec Kirkley, Hugo Barbosa, Marc Barthelemy, Gourab Ghoshal (2018). *From the betweenness centrality in street networks to structural invariants in random planar graphs*. Nature Communications.
- [7] GeekForGeeks (2023). *Applications, Advantages and Disadvantages of Breadth First Search (BFS)*
- [8] Dmitri Goldenberg(2019). *Social Network Analysis: From Graph Theory to Applications with Python*. ResearchGate.
- [9] William Campbell, C.K. Dagli, C.J. Weinstein (2013). *Social Network Analysis with Content and Graphs*. Research Gate.
- [10] Adhe Rizky Anugerah, Prafajar Suksessanno Muttaqin, Wahyu Trinarningsih (2022). *Social network analysis in business and management research: A bibliometric analysis of the research trend and performance from 2001 to 2020*. National Library of Medicine.
- [11] Nguyen The Duy Khanh. *Final Assignment's Github Repository*