Social Network Analysis - Final

Algorithms and Data Structures

Khanh Nguyen
University of the Cumberlands
knguyen19181@ucumberlands.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 Algorithms

The section includes three parts:

- Degree centrality
- Shortest path
- Betweenness centrality

2.1 Degree centrality

Degree centrality[1] 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 Twitter, it indicates the number of followers an end-user has to determine the end user's popularity before recommending that user to others in a similar community. Twitter has a natural model of end-users following their favorite celebrity, which contradicts Facebook, where people can connect to each other bidirectionally. Therefore, we only consider the total amount of inwards edge for a particular node when calculating degree centrality [2] (e.g Doug has a degree centrality of 1 while Michael has 0.2)

Preprint. Under review.

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

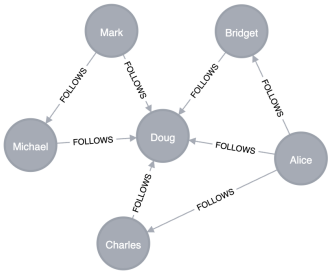

Figure 1: Twitter Social Network

With the networkx package, we can use the in-built function to calculate the centrality degree of each vertex accordingly:

```python
import networkx as nx

if __name__ =="__main__":
  # Initialize an empty graph with the corresponding vertex
  G = nx.Graph()
  G.add_nodes_from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
  print("Graph vertex ", G.nodes())

  # Add the corresponding edge between vertex before calculating degree centrality
  for nearestVertex in range(2, len(G.nodes()) + 1 ):
    G.add_edge(1, nearestVertex)

  print("Graph egdes", G.edges())
  print("Vertex 1's neighbor", nx.degree_centrality(G))
```

Figure 2: Degree centrality's implementation in Python

2.2 Shortest path

The shortest path[3] 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 Twitter 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 tweet, The shortest path algorithm can suggest the most effective sequence of users to engage with, maximizing the tweet's reach and visibility while minimizing the number of intermediate users.

Similarly, the recommendation system can effectively connect users who share common characteristics (e.g., ethnicity, favorite singers, etc.) or belong to the same community to follow the targeted celebrities, which will significantly boost their influence daily. To solve the problem, there are several well-known algorithms[4] that exist:

- Dijkstra's algorithm
- Bellman–Ford algorithm
- Floyd–Warshall algorithm
- BFS algorithm

Each algorithm has its use case, including its advantages and disadvantages, which have been listed in the following table:

	BFS	Dijkstra's	Bellman-Ford	Floyd Warshall
Time Complexity	$O(V + E)$	$O(V + E)$ log V	$O(V + E)$	$O(V^3)$
Recommended Graph Size	Large	Large/Medium	Medium/ Small	Small
All pairs shortest path	Only unweighted graphs	Ok	Bad	Yes
Negative cycles	No	No	Yes	Yes
Shortest path with weighted edges	Bad	Best algorithm	Works	Bad in general
Shortest path with unweighted edges	Best algorithm	Ok	Bad	Bad in general

Figure 3: Shortest path algorithm's analysis

As of now, Twitter has 368 million monthly active users [5]. If we are considering similar characteristics before recommending the targeted celebrities, Dijkstra's algorithm[6] would be the best fit for the current use case. However, since the data we aim to get does not have any characteristics, we will use Bread First Search (BFS)[7] as an algorithm for the Shortest Path Problem. Dijkstra's algorithm will also be used as an algorithm for comparing performance as a base with BFS and other algorithms.

With the networkx package, we can develop a shortest-path BFS version to identify if two vertex are connected:

Figure 4: Bread First Search Shortest Path's algorithm

2.3 Betweenness centrality

Betweenness centrality[8] is used to find bottleneck nodes in a graph or the extent to which a certain vertex lies on the shortest paths between other vertices. On Twitter, it helps identify individuals who play a "bridge-spanning role" in the network. Therefore, helping to recommend the targeted celebrities to other communities.

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

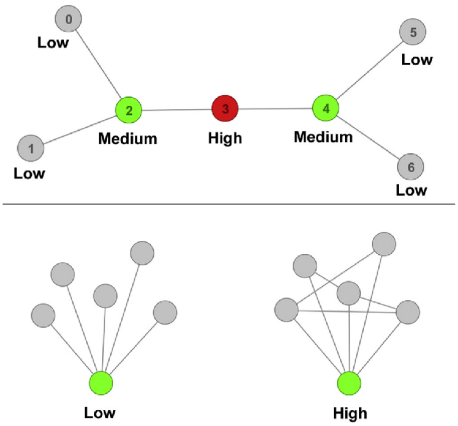

Figure 5: Betweeness centrality

With the networkx package, we can utilize the in-house version version to identify if two vertex are connected:

```
33  if __name__ =="__main__":
34      # Initialize an empty graph with the corresponding vertex
35      G = nx.barbell_graph(m1=5, m2=1)
36      print("Between centrality",nx.betweenness_centrality(G))
37      nx.draw(G)
38      plt.show()      You, 3 seconds ago • Uncommitted changes
```

Figure 6: Betweenness centrality's implementation in Python

3 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/69ada08d5cce7f35f38ffefe8f22
25      twitter = 'https://assets.datacamp.com/production/repositories/580/datasets/64cf6963a7e8005e3771ef3b25681
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 7: 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 [9] 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:

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

  # 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())
  sn = SocialNetwork(fbSubGraph)
  # sn.drawGraph()
  return sn
```

Figure 8: Create Social Network in Python

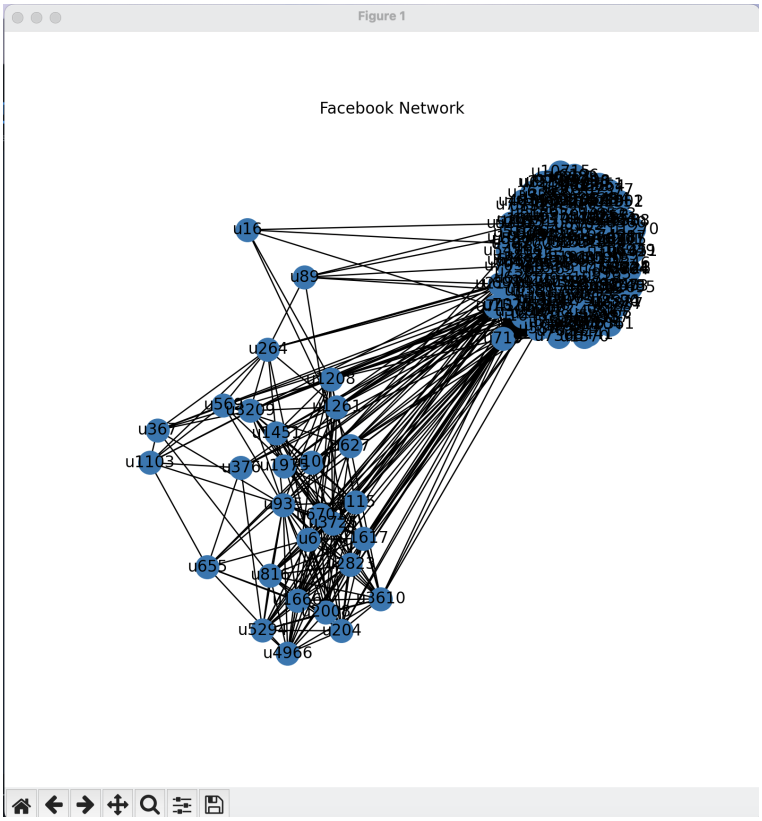

Figure 9: Facebook Network

The following key operations include four sections:

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

3.1 Find important people

Degree centrality is used to find the "popularity" of each node in the graph database based on the number of connections it has to other nodes. Therefore, we will use the same algorithm to find the most important people in the social network:

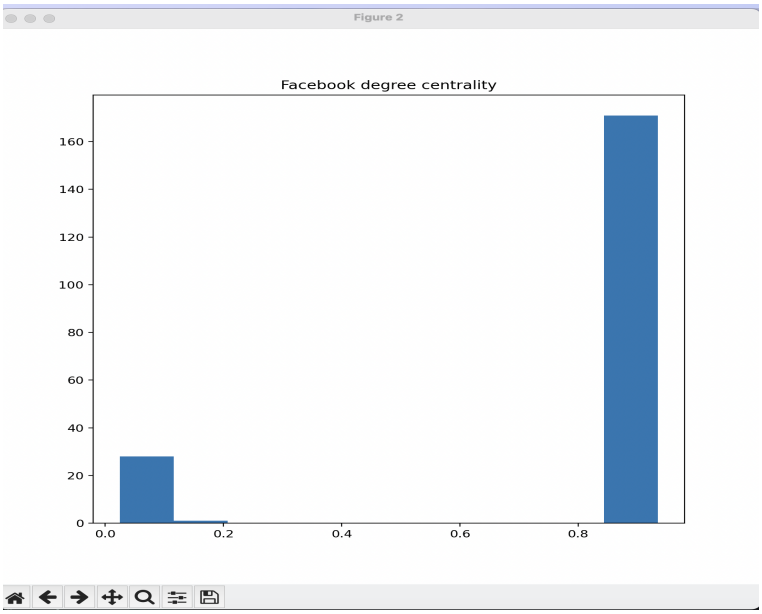

Figure 10: Facebook Histogram Distribution for degree centrality

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  # findImportantPeople 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 11: 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 [('u719', 0.9346733668341709)]
```

Figure 12: Find important people's output

3.2 Find largest communities

A clique [10] 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[12]. Therefore, we will use the same clique to identify the largest communities that share a similar portion of interest to each other.

```python
# 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 13: 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', '
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']
```

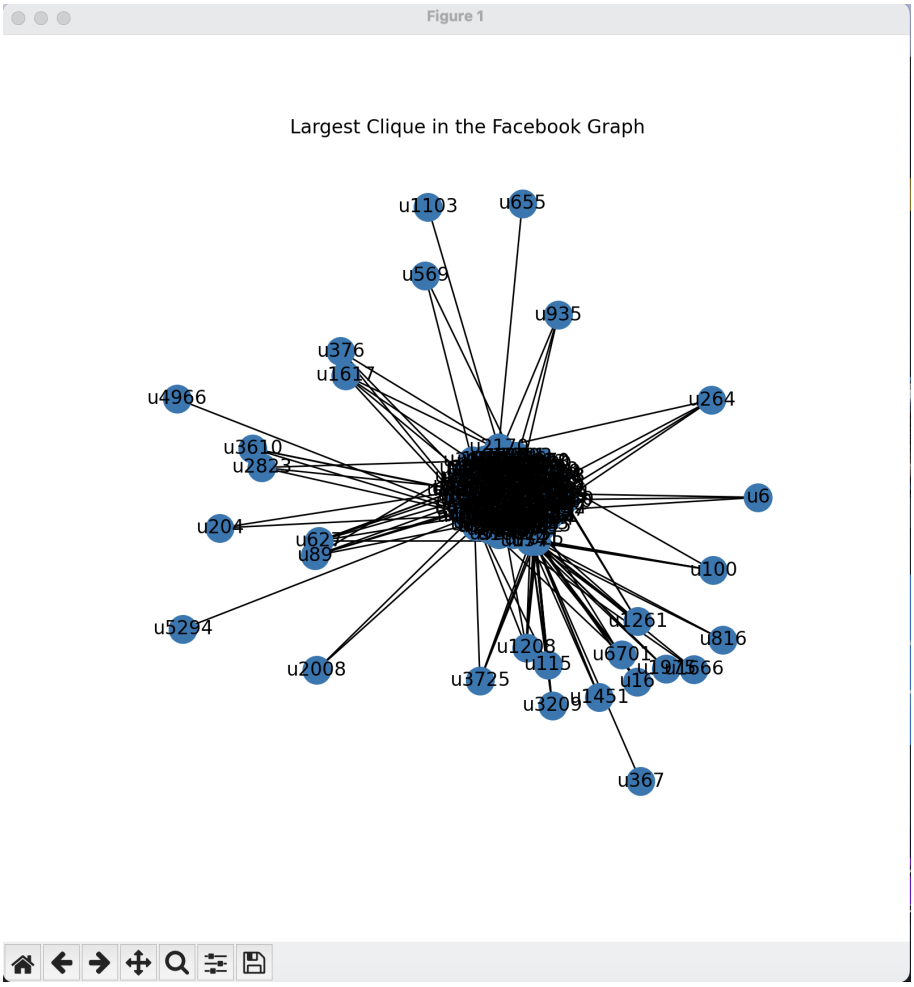

Figure 14: Find largest communities's output

3.3 Connect communities

The shortest path 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, 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.

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

To summarize, the feature has 3 steps:

- **Step 1:** Find the cliques of the social network
- **Step 2:** Find the bottleneck user which serves as a "bridge-spanning role" in the network using betweenness centrality
- **Step 3:** Calculate the minimum distance between the two bottleneck users to see if there are any connections.

```
fbGraph
# 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()
  largestCliqueBetweenessCentrality = nx.betweenness_centrality(largestClique)
  secondLargestCliqueBetweenessCentrality = nx.betweenness_centrality(secondLargestClique)
  largestCliqueBottleneckUser = max(largestCliqueBetweenessCentrality, key=largestCliqueBetweenessCentralit
  secondLargestCliqueBottleneckUser = max(secondLargestCliqueBetweenessCentrality, key=secondLargestCliqueB
  print("Is path exist between two bottleneck users:", self.pathExistBFS(largestCliqueBottleneckUser, secon
  if self.pathExistBFS(largestCliqueBottleneckUser, secondLargestCliqueBottleneckUser):
    print("Connect two largest clique together between {0} and {1}".format(largestCliqueBottleneckUser, sec
```

Figure 15: 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 16: Connect communities's output

3.4 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. The feature has 3 steps:

- **Step 1:** Find all the combinations of all the node's neighbors
- **Step 2:** If there is no connection between two neighbor nodes, we would recommend them and increase further if there are more intermediate node
- **Step 3:** Prioritize the top 10 pairs of nodes that have the highest intermediate nodes (or bypass connections instead of direct connections)

Figure 17: Bread First Search Shortest Path's algorithm

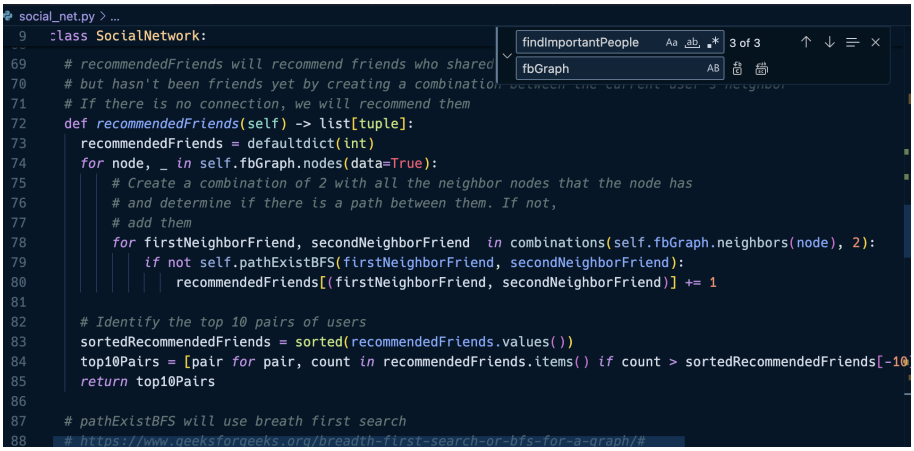

Figure 18: Recommend Friends in Python

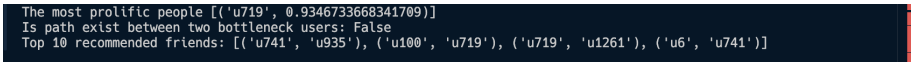

Figure 19: Recommend Friends's output

4 Optimization of Data Structures

4.1 Recommended friends

The time complexity of the algorithm for recommending friends is expected to be O(V x C(V, 2) x (V + E)) in total and O(V + E) for BFS, where V is the number of vertices and E is the number of edges. For unsorted nodes with random degrees, the edges are small enough to be negligible. However, with sorting nodes based on their degree (the number of edges connecting to the node), the edge has increased from 20 to 3824 (a total of 190.2% increasing.

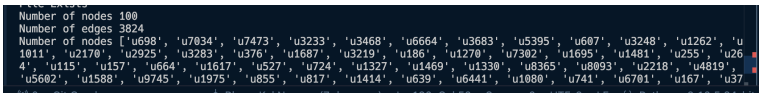

Figure 20: Sorted nodes with the highest degree (highest edge)

As a result, we encounter the scaling issue: running the recommended friends feature takes from 0.076 to 18.63s (a total of 24457% increasing) with sorting nodes based on their degree. Fortunately, Python offers a Profiler [13] (including CPU Profile and Memory Profiler) that helps us identify the feature's bottleneck in the execution time. A total of 4400152 calls, and each takes 20.857s.

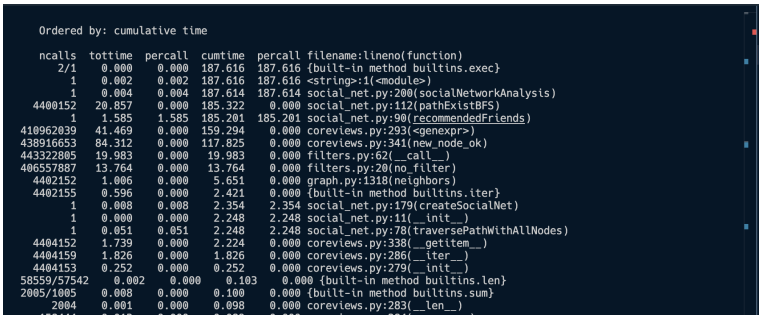

Figure 21: Profiler for recommended friends

To optimize the algorithm, instead of finding whether there aren't any connections between the node's neighbors, we will traverse the path during the initialization, recording the path between two nodes via cache if there is a connection. Afterward, we will only use the recorded path as a baseline for the recommended friends feature by referring the friends via the current node if there are no connections between them.

```python
def traversePathWithAllNodes(self) -> set:      You, 5 hours ago • Add optimization for traversing
    traversePath = set()
    for node, _ in self.fbGraph.nodes(data=True):
        for neighbour in self.fbGraph.neighbors(node):
            if (node, neighbour) not in traversePath and self.pathExistBFS(node, neighbour):
                traversePath.add((node, neighbour))

    return traversePath
```

```python
# recommendedFriends will recommend friends who shared
# but hasn't been friends yet by creating a combination between the current user's neighbor
# If there is no connection, we will recommend them
def recommendedFriends(self) -> list[tuple]:
    recommendedFriends = defaultdict(int)
    for node, _ in self.fbGraph.nodes(data=True):
        # Create a combination of 2 with all the neighbor nodes that the node has
        # and determine if there is a path between them. If not,
        # add them
        for firstNeighborFriend, secondNeighborFriend  in combinations(self.fbGraph.neighbors(node), 2):
            if (firstNeighborFriend, secondNeighborFriend) not in self.traversePath:
            #if not self.pathExistBFS(firstNeighborFriend, secondNeighborFriend):
                recommendedFriends[(firstNeighborFriend, secondNeighborFriend)] += 1

    # Identify the top 10 pairs of users
    sortedRecommendedFriends = sorted(recommendedFriends.values())
    top10Pairs = [pair for pair, count in recommendedFriends.items() if count > sortedRecommendedFriends[-10
    return top10Pairs
```

Figure 22: Optimization for recommended friends

As a result of the optimization, the time complexity has been improved to $O(V^2x(V+E))$; however, in exchange for the space complexity of O(V * P) where V is the number of vertices and P is the maximum path a vertice can have, which leads to the following graph:

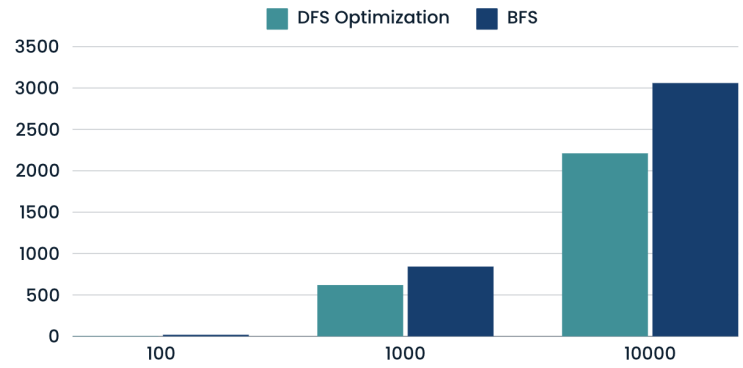

Figure 23: BFS Optimization for sorted degree Execution Time

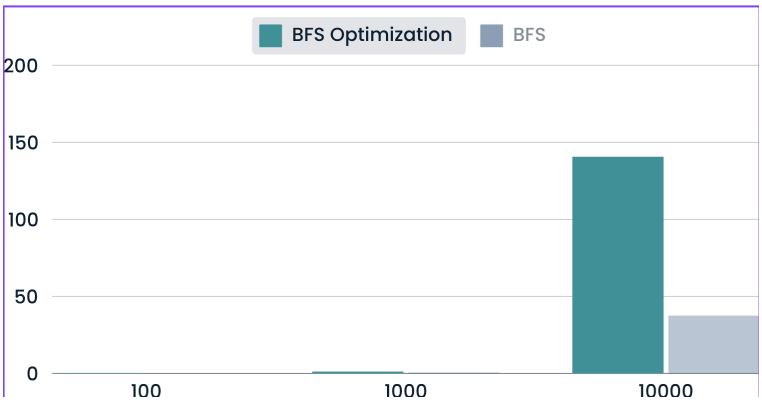

Figure 24: BFS Optimization for sorted degree Memory Usage (MiB)

However, for unsorted nodes with random degrees, the edges are small enough to be negligible, which results in similar execution time for both optimized BFS with cache and original BFS.

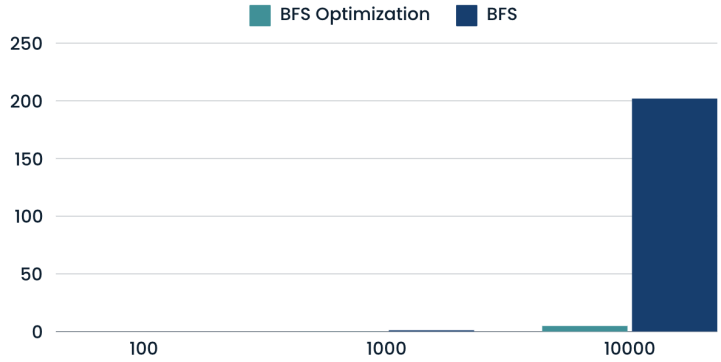

Figure 25: BFS Optimization for unsorted degree Execution Time (second)

4.2 Connect communities

Since Facebook's social network is an indirect graph, it would work best with BFS instead of other algorithms such as Floyd–Warshall algorithm, Bellman-Ford algorithm, or even Dijkstra's algorithm.

The time complexity of the algorithm is expected to be $O(3^{\frac{v}{3}} + v + e)$ [4] in total and O(v + e) for BFS. We can use the same aforementioned algorithm to cache all the traverse paths of all the nodes

inside the social network and determine if there is a connection between two nodes, and we can also know the minimum distance between the two.

```
def connectCommunities(self)-> int:
    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()
    largestCliqueBetweenessCentrality = nx.betweenness_centrality(largestClique)
    secondLargestCliqueBetweenessCentrality = nx.betweenness_centrality(secondLargestClique)
    largestCliqueBottleneckUser = max(largestCliqueBetweenessCentrality, key=largestCliqueBetweenessCentrali
    secondLargestCliqueBottleneckUser = max(secondLargestCliqueBetweenessCentrality, key=secondLargestClique
    pathExist = self.pathExistBFS(largestCliqueBottleneckUser, secondLargestCliqueBottleneckUser)
    #distanceBetweenTwoUsers = self.getDistanceWithCurrentNode(largestCliqueBottleneckUser, secondLargestCli
    #print("Distance between two bottleneck users:", distanceBetweenTwoUsers)
    #print("There is path between the two largest clique", pathExist)
    if pathExist:      You, 1 second ago • Uncommitted changes
        print("Connect two largest clique together between {0} and {1}".format(largestCliqueBottleneckUser, se
    return distanceBetweenTwoUsers

# findLargestCommunities can find the largest cliques
```

Figure 26: Optimization for connecting communities

As the cache is using BFS to traverse the path behind the scenes, we will increase the execution time when compared to the original BFS from 0.845s to 4.6s(a total of 444% increasing) However, with Dijkstra O(V + E) log V, there will be slower execution time but negligible for the addition log V from 0.845s to 0.877s (a total of 3.6% increasing. The result has been shown below:

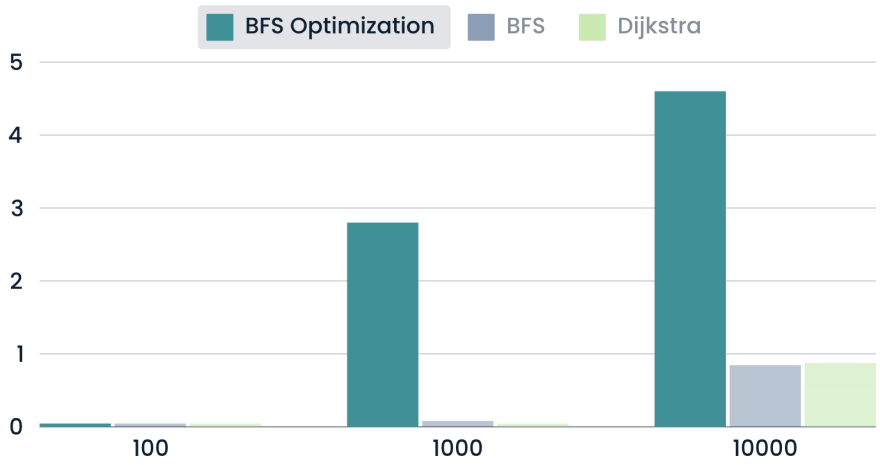

Figure 27: Execution time for Optimized BFS, BFS and Dijkstra

Therefore, we can say BFS with the unweighted graph works best for our current use case when determining the connection between the two largest communities.

5 Advanced Testing and Validation

To maintain backward compatibility and documentation for future use, we will develop a set of test cases. There are 5 features complex enough to develop a test case for it:

- **Shortest path with BFS:**

```
def test_ShortestPathBFS(self):
    dataset = downloadDataset()
    with open(dataset[1], 'rb') as f:
        fbGraph = pickle.load(f)
    nodes = [node for node in list(fbGraph.nodes())[:1000]]
    fbSubGraph = fbGraph.subgraph(nodes)
    sn = SocialNetwork(fbSubGraph)
    for node, _ in fbSubGraph.nodes(data=True):
        for neighbour  in fbSubGraph.neighbors(node):
            self.assertEqual(fbSubGraph.has_edge(node, neighbour), sn.pathExistBFS(node, neighbour))
```

Figure 28: Test finding path between two nodes with BFS

- **Shortest path with BFS Optimization:**

```
def test_TraversePathWillAllNodes(self):
  dataset = downloadDataset()
  with open(dataset[1], 'rb') as f:
    fbGraph = pickle.load(f)
  nodes = [node for node in list(fbGraph.nodes())[:1000]]
  fbSubGraph = fbGraph.subgraph(nodes)
  sn = SocialNetwork(fbSubGraph)
  for node, _ in fbSubGraph.nodes(data=True):
    for neighbour  in fbSubGraph.neighbors(node):
      if fbSubGraph.has_edge(node, neighbour):
        self.assertTrue((node, neighbour) in sn.traversePath)
  self.assertGreater(len(sn.traversePath), 0)
```

Figure 29: Test recording all traversal paths with BFS Optimization

- **Find the largest community:**

```
def test_FindLargestCommunity(self):      You, 2 hours ago • Add corresponding unit test
  dataset = downloadDataset()
  with open(dataset[1], 'rb') as f:
    fbGraph = pickle.load(f)
  nodes = [node for node in list(fbGraph.nodes())[:1000]]
  fbSubGraph = fbGraph.subgraph(nodes)
  sn = SocialNetwork(fbSubGraph)
  largestClique = set(sorted(nx.find_cliques(fbSubGraph), key=lambda x: len(x))[-1])
  expectedFacebookLargestClique = fbSubGraph.subgraph(largestClique).copy()
  # Go out 1 degree of separation
  for node in list(expectedFacebookLargestClique.nodes()):
      expectedFacebookLargestClique.add_nodes_from(fbSubGraph.neighbors(node))
      expectedFacebookLargestClique.add_edges_from(zip([node]*len(list(fbSubGraph.neighbors(node))), fbSu

  self.assertIsNotNone(expectedFacebookLargestClique)
  self.assertEqual(expectedFacebookLargestClique.nodes(), sn.findLargestCommunities().nodes())

def test_FindLargestCommunityWithEmptyGraph(self):
  fbGraph = nx.Graph()
  sn = SocialNetwork(fbGraph)
  self.assertIsNone(sn.findLargestCommunities())
```

Figure 30: Test for finding largest community

- **Find important people:**

```
def test_FindImportantPeople(self):
  dataset = downloadDataset()
  with open(dataset[1], 'rb') as f:
    fbGraph = pickle.load(f)
  nodes = [node for node in list(fbGraph.nodes())[:1000]]
  fbSubGraph = fbGraph.subgraph(nodes)
  sn = SocialNetwork(fbSubGraph)
  importantPeople = sn.findImportantPeople()

  degCent = nx.degree_centrality(fbSubGraph)
  degCent = {k: v for k, v in sorted(degCent.items(), key=lambda item: item[1], reverse=True)}
  maxDegCent = max(list(degCent.values()))
  expectedImportantPeople = [(n, dc) for n, dc in degCent.items() if dc == maxDegCent]
  self.assertEqual([expectedImportantPeople], importantPeople)

def test_FindImportantPeopleWithEmptyGraph(self):
  fbGraph = nx.Graph()
  sn = SocialNetwork(fbGraph)
  importantPeople = sn.findImportantPeople()
  self.assertEqual([], importantPeople)
```

Figure 31: Test for finding important users

- **Connect communities:**

```python
def test_ConnectCommunities(self):
    dataset = downloadDataset()
    with open(dataset[1], 'rb') as f:
        fbGraph = pickle.load(f)
    nodes = [node for node in list(fbGraph.nodes())[:1000]]
    fbSubGraph = fbGraph.subgraph(nodes)
    sn = SocialNetwork(fbSubGraph)
    cliques = sorted(nx.find_cliques(fbSubGraph), key=lambda x: len(x))
    largestClique = fbSubGraph.subgraph(set(cliques[-1])).copy()
    secondLargestClique = fbSubGraph.subgraph(set(cliques[-2])).copy()
    largestCliqueBetweenessCentrality = nx.betweenness_centrality(largestClique)
    secondLargestCliqueBetweenessCentrality = nx.betweenness_centrality(secondLargestClique)
    largestCliqueBottleneckUser = max(largestCliqueBetweenessCentrality, key=largestCliqueBetweenessCentrali
    secondLargestCliqueBottleneckUser = max(secondLargestCliqueBetweenessCentrality, key=secondLargestCliqueB
    expectedDistanceBetweenTwoUsers = sn.getDistanceWithCurrentNode(largestCliqueBottleneckUser, secondLarges

    self.assertEqual(expectedDistanceBetweenTwoUsers, sn.connectCommunities())

def test_ConnectCommunitiesWithEmptyGraph(self):
    fbGraph = nx.Graph()
    sn = SocialNetwork(fbGraph)
    self.assertEqual(-1, sn.connectCommunities())
```

Figure 32: Test for connecting communities and the shortest distance

- **Find recommended friends:**

```python
def test_FindRecommendedFriends(self):
    dataset = downloadDataset()
    with open(dataset[1], 'rb') as f:
        fbGraph = pickle.load(f)
    nodes = [node for node in list(fbGraph.nodes())[:1000]]
    fbSubGraph = fbGraph.subgraph(nodes)
    sn = SocialNetwork(fbSubGraph)
    recommendedFriends = sn.recommendedFriends()
    expectedRecommendedFriends = set()

    expectedRecommendedFriends = defaultdict(int)
    for node, _ in fbSubGraph.nodes(data=True):
        for firstNeighborFriend, secondNeighborFriend  in combinations(fbSubGraph.neighbors(node), 2):
            if not fbSubGraph.has_edge(firstNeighborFriend, secondNeighborFriend):
                expectedRecommendedFriends[(firstNeighborFriend, secondNeighborFriend)] += 1

    sortedExpectedRecommendedFriends = sorted(expectedRecommendedFriends.values())
    expectedRecommendedFriends = [pair for pair, count in expectedRecommendedFriends.items() if count > sort
    # Since it is unsorted node, there will be more guarantee to have friends instead
    self.assertGreater(len(recommendedFriends), 0)
    self.assertEqual(expectedRecommendedFriends, recommendedFriends)

def test_FindRecommendedFriendsWithEmptyGraph(self):
    fbGraph = nx.Graph()
    sn = SocialNetwork(fbGraph)
    self.assertEqual([], sn.recommendedFriends())
```

Figure 33: Test for finding recommend friends

When running the test in parallel, we will have the following result:

```python
unitTests = [
    'test_FindRecommendedFriends',
    'test_ShortestPathWithEmptyGraph',
    'test_FindImportantPeople',
    'test_ConnectCommunities',
    'test_FindLargestCommunity',
    'test_ShortestPathBFS',
    'test_TraversePathWillAllNodes'
]

with ThreadPoolExecutor() as executor:
    # Execute only the specified tests in parallel
    futures = [executor.submit(runTests, TestSocialNetworkAnalysis, test) for test in unitTests]
    for future in futures:
        future.result()
```

Figure 34: Run unit test in parallel

```
.
----------------------------------------------------------------------
Ran 1 test in 0.604s

Distance between two bottleneck users: 2
.Connect two largest clique together between u397 and u204
OK

.
Facebook largest clique ['u397', 'u587', 'u527', 'u855', 'u982', 'u201', 'u639', 'u741', 'u186', 'u157', 'u724', 'u5
30', 'u719', 'u664', 'u817', 'u698', 'u17', 'u75', 'u167', 'u607', 'u255', 'u172']
----------------------------------------------------------------------
Ran 1 test in 0.693s
.
----------------------------------------------------------------------
.
OK
Ran 1 test in 0.828s

----------------------------------------------------------------------

Ran 1 test in 0.970s

OK
----------------------------------------------------------------------
.OK
Ran 1 test in 1.024s
```

Figure 35: Unit test result

As a result of these tests, if there are any features being optimized even more or any set of bug fixes/improvements, the test will provide a set of bulletproof when releasing a new version. Moreover, it also unit tests edge cases (e.g., paths with empty nodes or empty edges) to provide more robust functionalities.

However, we also need to perform a corresponding stress test [5] (a performance test in the future to give a user transparency of how the algorithm uses their allocation resource) in order to identify the breaking point of a system and improve it under extreme load conditions.

- **Find the largest community:**

```python
def test_StressTestingFrom1000To20000NodesForFindingLargestCommunities(self):
    dataset = downloadDataset()
    with open(dataset[1], 'rb') as f:
        fbGraph = pickle.load(f)
    for numberOfNodes in range(1000, 20000, 1000):
        nodes = [node for node in list(fbGraph.nodes())[:numberOfNodes]]
        fbSubGraph = fbGraph.subgraph(nodes)
        sn = SocialNetwork(fbSubGraph)
        sn.findLargestCommunities()
```

Figure 36: Stress Test for finding largest community

- **Find important people:**

```python
def test_StressTestingFrom1000To20000NodesForFindingImportantPeople(self):
    dataset = downloadDataset()
    with open(dataset[1], 'rb') as f:
        fbGraph = pickle.load(f)
    for numberOfNodes in range(1000, 20000, 1000):
        nodes = [node for node in list(fbGraph.nodes())[:numberOfNodes]]
        fbSubGraph = fbGraph.subgraph(nodes)
        sn = SocialNetwork(fbSubGraph)
        sn.findImportantPeople()
```

Figure 37: Stress test for finding important users

- **Connect communities:**

```python
def test_StressTestingFrom1000To20000NodesForConnectingComunities(self):
    dataset = downloadDataset()
    with open(dataset[1], 'rb') as f:
        fbGraph = pickle.load(f)
    for numberOfNodes in range(1000, 20000, 1000):
        nodes = [node for node in list(fbGraph.nodes())[:numberOfNodes]]
        fbSubGraph = fbGraph.subgraph(nodes)
        sn = SocialNetwork(fbSubGraph)
        sn.connectCommunities()
```

Figure 38: Stress test for connecting communities

- **Find recommended friends:**

```
def test_StressTestingFrom1000To20000NodesForRecommendedFriendsWithUnsortedData(self):
    dataset = downloadDataset()
    with open(dataset[1], 'rb') as f:
      fbGraph = pickle.load(f)
    for numberOfNodes in range(1000, 20000, 1000):
      nodes = [node for node in list(fbGraph.nodes())[:numberOfNodes]]
      fbSubGraph = fbGraph.subgraph(nodes)
      sn = SocialNetwork(fbSubGraph)
      sn.recommendedFriends()

def test_StressTestingFrom1000To20000NodesForRecommendedFriendsWithSortedDegreeData(self):
  dataset = downloadDataset()
  with open(dataset[1], 'rb') as f:
    fbGraph = pickle.load(f)
  for numberOfNodes in range(1000, 20000, 1000):
    nodes = [node for node in list(fbGraph.nodes())[:numberOfNodes]]
    fbSubGraph = fbGraph.subgraph(nodes)
    sn = SocialNetwork(fbSubGraph)
    sn.recommendedFriends()
```

Figure 39: Stress test for finding recommended friends

6 Final Evaluation and Performance Analysis

For the optimization of both shortest path problems with recommended friends and connect communities, we have noticed a increase in execution time (from 0.076 to 18.63s - a total of 24457% increasing). However, increasing memory for storing all the traversal paths beforehand (from 37.5 MiB to 140 MiB - a total of 3.75% increasing)

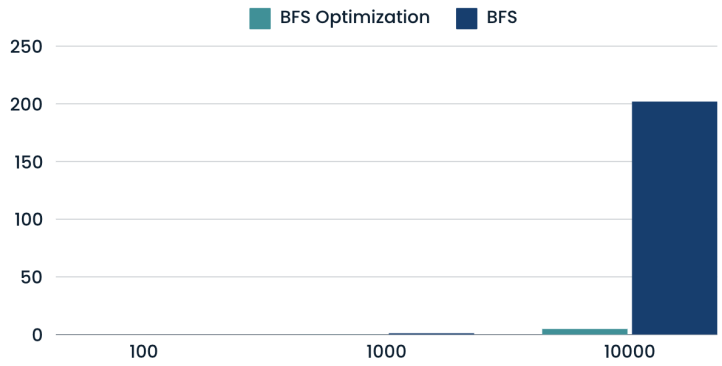

Figure 40: Reduction in execution time for Optimized BFS for sorted data (second)

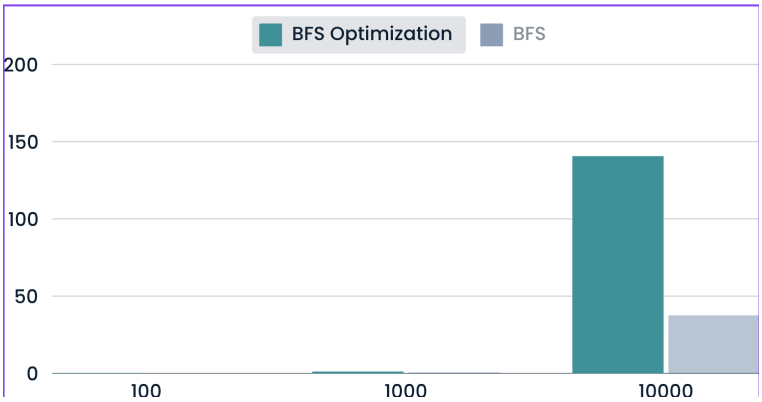

Figure 41: Increasing in memory usage for Optimized BFS for sorted data (MiB)

However, even though we have optimized the data structures, there are some areas we can improve on:

- The dataset from social networks is static; therefore, we would need to find a way to always append the new traversal path when there is a new user joining it.
- The solution only optimizes the features that include traversing to all the paths or share similar functionalities. However, for features that only need to determine a connection between two users, such as connect communities, it will have the drawback of over-engineering the solutions.
- The optimizing solutions only work towards an unweighted graph. However, if there are any weights or different data structures, the optimization and the current implementation won't work.
- In a constrained memory environment, the space complexity of O(V * P), where P is the number of paths a node can have, will become a problem compared to the O(V) of the original solution.

Therefore, to avoid the aforementioned problems, we can optimize it by:

- We will store all the traversal paths of all the nodes during the initial utilization. When there is a user being added, or there are new connections between two users as a result of recommended friends features via a Pub/Sub system [6] (e.g. Kafka, RabbitMQ, etc.), we will add the node and all the traversal path of that node to the cache.
- Instead of adding every user and its traversal path to the cache, we can prioritize different characteristics of a node to determine if the current node is needed or node (e.g if the user is not active or has less than 5 friends or the user does not active in connecting friends[7]). The worst case will be the same. However, we would reduce a small amount of memory.
- Instead of storing everything in a cache, we can store a subset of similar traits user and their traversal path in a separate node. Whenever there is a new connection or a new user comes in, we will put the connections/users in the corresponding node based on certain factors (e.g., number of shared friends).

References

[1] Jayant Bisht (2022). *Degree Centrality (Centrality Measure*. GeeksForGeeks

[2] Jennifer Golbeck (2015). *Introduction to Social Media Investigation*. ScienceDirect.

[3] Vai Brav (2023) *Shortest Path Algorithm Tutorial with Problems*. GeeksForGeeks.

[4] Wikipedia *Shortest Path Problem*

[5] Matthew Woodward (2024). *Twitter User Statistics 2024: What happening after "X" rebranding?*. SearchLogistics.

[6] Estefania Cassingena Navone (2020). *Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction*. FreeCodeCamp.

[7] GeekForGeeks (2023). *Applications, Advantages and Disadvantages of Breadth First Search (BFS)*

[8] Alec Kirkley, Hugo Barbosa, Marc Barthelemy, Gourab Ghoshal (2018). *From the betweenness centrality in street networks to structural invariants in random planar graphs*. Nature.

[9] Nupur Jain (2024). *Methods of sampling*. Geeks For Geeks.

[10] Karthik Reddy (2022). *Cliques in Graph*. GeeksForGeeks.

[11] Dmitri Goldenberg Jun 2019 *Social Network Analysis: From Graph Theory to Applications with Python*

[12] William Campbell, C.K. Dagli, C.J. Weinstein Jan 2013 *Social Network Analysis with Content and Graphs*

[13] Adhe Rizky Anugerah, Prafajar Suksessanno Muttaqin, Wahyu Trinarningsih Apr 2022 *Social network analysis in business and management research: A bibliometric analysis of the research trend and performance from 2001 to 2020*

[14] Nguyen The Duy Khanh. *Final Assignment's Github Repository*