Social Network Analysis - Optimization, Scaling, and Final Evaluation

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 Optimization of Data Structures

The shortest path[1] 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. For social network analysis, we solved the shortest path problem, which was used to find the recommended friends and the shortest path between two cliques via Bread First Search (BFS).

2.1 Recommended friends

Before optimization, we must understand the implementation of the recommended friends feature. 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)

Preprint. Under review.

```
social_net.py > ...
class SocialNetwork:
    # 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 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

    # pathExistBFS will use breath first search
    # https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/#
```

Figure 2: Recommend Friends in Python

The time complexity of the algorithm 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.

```
Number of nodes 100
Number of edges 3824
Number of nodes ['u698', 'u7034', 'u7473', 'u3233', 'u3468', 'u6664', 'u3683', 'u5395', 'u607', 'u3248', 'u1262', 'u
1011', 'u2170', 'u2925', 'u3283', 'u376', 'u1687', 'u3219', 'u186', 'u1270', 'u7302', 'u1695', 'u1481', 'u255', 'u26
4', 'u115', 'u157', 'u664', 'u1617', 'u527', 'u724', 'u1327', 'u1469', 'u1330', 'u8365', 'u8093', 'u2218', 'u4819',
'u5602', 'u1588', 'u9745', 'u1975', 'u855', 'u817', 'u1414', 'u639', 'u6441', 'u1080', 'u741', 'u6701', 'u167', 'u37
```

Figure 3: 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 [2] (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.

```
   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      2/1    0.000    0.000  187.616  187.616 {built-in method builtins.exec}
        1    0.002    0.002  187.616  187.616 <string>:1(<module>)
        1    0.004    0.004  187.614  187.614 social_net.py:200(socialNetworkAnalysis)
  4400152   20.857    0.000  185.322    0.000 social_net.py:112(pathExistBFS)
        1    1.585    1.585  185.201  185.201 social_net.py:90(recommendedFriends)
410962039   41.469    0.000  159.294    0.000 coreviews.py:293(<genexpr>)
438916653   84.312    0.000  117.825    0.000 coreviews.py:341(new_node_ok)
443322805   19.983    0.000   19.983    0.000 filters.py:62(__call__)
406557887   13.764    0.000   13.764    0.000 filters.py:20(no_filter)
  4402152    1.006    0.000    5.651    0.000 graph.py:1318(neighbors)
  4402155    0.596    0.000    2.421    0.000 {built-in method builtins.iter}
        1    0.008    0.008    2.354    2.354 social_net.py:179(createSocialNet)
        1    0.000    0.000    2.248    2.248 social_net.py:11(__init__)
        1    0.051    0.051    2.248    2.248 social_net.py:78(traversePathWithAllNodes)
  4404152    1.739    0.000    2.224    0.000 coreviews.py:338(__getitem__)
  4404159    1.826    0.000    1.826    0.000 coreviews.py:286(__iter__)
  4404153    0.252    0.000    0.252    0.000 coreviews.py:279(__init__)
58559/57542    0.002    0.000    0.103    0.000 {built-in method builtins.len}
2005/1005    0.008    0.000    0.100    0.000 {built-in method builtins.sum}
     2004    0.001    0.000    0.098    0.000 coreviews.py:283(__len__)
```

Figure 4: 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.

```
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 4: 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:

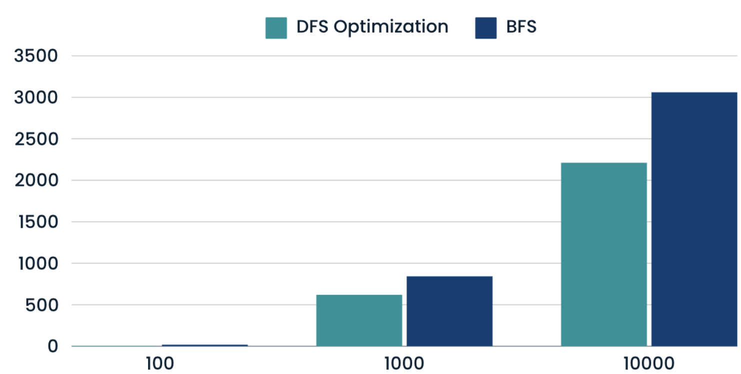

Figure 5: BFS Optimization for sorted degree Execution Time

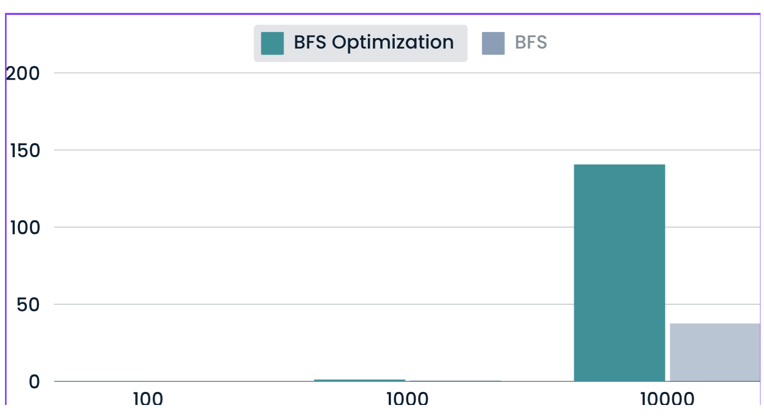

Figure 6: BFS Optimization for sorted degree Memory Usage (MiB)

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.

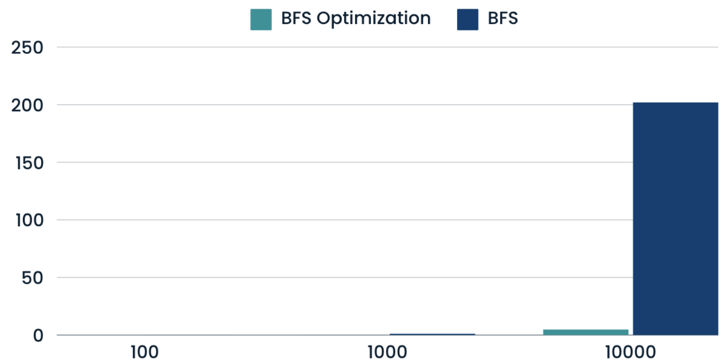

Figure 7: BFS Optimization for unsorted degree Execution Time (second)

2.2 Connect communities

Before optimization, we must understand how to connect community features. The feature has 3 steps:

- **Step 7:** Find the cliques[3] of the social network
- **Step 8:** Find the bottleneck user which serves as a "bridge-spanning role" in the network using betweenness centrality [8]
- **Step 9:** Calculate the minimum distance between the two bottleneck users to see if there are any connections.

```python
# 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 8: Connect communities in Python

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

	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 9: The shortest path algorithm's analysis

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.

qWe 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
```

Figure 10: 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:

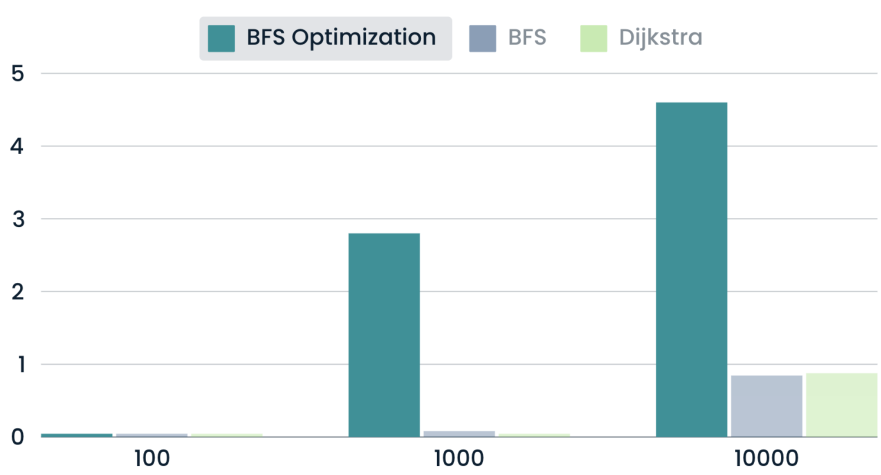

Figure 11: Execution time for Optimized BFS, BFS and Dijkstra

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.

3 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:**

```python
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 12: Test finding path between two nodes with BFS

- **Shortest path with BFS Optimization:**

```python
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 13: Test recording all traversal paths with BFS Optimization

- **Find the largest community:**

```python
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 14: Test for finding largest community

- **Find important people:**

```python
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 15: 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=largestCliqueBetweenessCentralit
    secondLargestCliqueBottleneckUser = max(secondLargestCliqueBetweenessCentrality, key=secondLargestClique
    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 16: 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 17: 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 18: 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 19: 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 20: 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 21: 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 22: Stress test for connecting communities

- **Find recommended friends:**

```python
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 23: Stress test for finding recommended friends

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

```
TERMINAL
10715', 'u4209', 'u1387', 'u8625', 'u3468', 'u10261',                                                74
', 'u719', 'u1481', 'u6552', 'u10813', 'u5132', 'u7459                                               89
00', 'u1204', 'u255', 'u2480', 'u7377', 'u8692', 'u3841', 'u741', 'u1687', 'u10742', 'u3233', 'u5269', 'u2170', 'u29
25', 'u2806', 'u4824', 'u7356', 'u3355', 'u1648', 'u172', 'u75', 'u167', 'u1039', 'u6441', 'u5882', 'u5395', 'u587',
 'u7034', 'u9610', 'u8523', 'u3428', 'u3683', 'u1414', 'u2594', 'u3669', 'u527', 'u7525', 'u1327', 'u1851', 'u7608',
 'u9437', 'u5470', 'u8726', 'u7368', 'u5108', 'u7302', 'u1330', 'u8057', 'u9369', 'u607', 'u855', 'u5817', 'u4296',
'u8365', 'u8494', 'u4951', 'u8125', 'u6245', 'u4604', 'u4077', 'u530', 'u8315', 'u10537', 'u9096', 'u1270', 'u10000'
, 'u9770', 'u9551', 'u10143', 'u8101', 'u10559', 'u2285', 'u6565', 'u6739', 'u5298', 'u9965', 'u6361', 'u1073', 'u32
42', 'u1588', 'u6949', 'u5331', 'u2994', 'u4588', 'u7471', 'u4819', 'u186', 'u3219', 'u157', 'u9745', 'u8175', 'u308
2', 'u3798', 'u10862', 'u2476', 'u6726', 'u6158', 'u9046', 'u5602', 'u4606', 'u4658', 'u5032', 'u3092', 'u10503', 'u
698', 'u5184', 'u9123', 'u724', 'u7025', 'u4312', 'u201', 'u664', 'u9489', 'u6196', 'u3283', 'u1011', 'u5457', 'u520
5', 'u2218', 'u4754', 'u3561', 'u639', 'u982', 'u4029', 'u10642', 'u397', 'u1695', 'u8871', 'u8924', 'u1080', 'u3777
', 'u1351', 'u3248', 'u3346', 'u817', 'u7473']
Distance between two bottleneck users: 2
Connect two largest clique together between u10623 and u2421
Facebook largest clique ['u10623', 'u1469', 'u8643', 'u8520', 'u7607', 'u6664', 'u7624', 'u17', 'u6438', 'u1571', 'u
10715', 'u4209', 'u1387', 'u8625', 'u3468', 'u10261', 'u3636', 'u6980', 'u6335', 'u10032', 'u2520', 'u1370', 'u10274
', 'u719', 'u1481', 'u6552', 'u10813', 'u5132', 'u7459', 'u3075', 'u8093', 'u1262', 'u7952', 'u6837', 'u10404', 'u89
00', 'u1204', 'u255', 'u2480', 'u7377', 'u8692', 'u3841', 'u741', 'u1687', 'u10742', 'u3233', 'u5269', 'u2170', 'u29
25', 'u2806', 'u4824', 'u7356', 'u3355', 'u1648', 'u172', 'u75', 'u167', 'u1039', 'u6441', 'u5882', 'u5395', 'u587',
 'u7034', 'u9610', 'u8523', 'u3428', 'u3683', 'u1414', 'u2594', 'u3669', 'u527', 'u7525', 'u1327', 'u1851', 'u7608',
 'u9437', 'u5470', 'u8726', 'u7368', 'u5108', 'u7302', 'u1330', 'u8057', 'u9369', 'u607', 'u855', 'u5817', 'u4296',
'u8365', 'u8494', 'u4951', 'u8125', 'u6245', 'u4604', 'u4077', 'u530', 'u8315', 'u10537', 'u9096', 'u1270', 'u10000'
, 'u9770', 'u9551', 'u10143', 'u8101', 'u10559', 'u2285', 'u6565', 'u6739', 'u5298', 'u9965', 'u6361', 'u1073', 'u32
42', 'u1588', 'u6949', 'u5331', 'u2994', 'u4588', 'u7471', 'u4819', 'u186', 'u3219', 'u157', 'u9745', 'u8175', 'u308
2', 'u3798', 'u10862', 'u2476', 'u6726', 'u6158', 'u9046', 'u5602', 'u4606', 'u4658', 'u5032', 'u3092', 'u10503', 'u
698', 'u5184', 'u9123', 'u724', 'u7025', 'u4312', 'u201', 'u664', 'u9489', 'u6196', 'u3283', 'u1011', 'u5457', 'u520
5', 'u2218', 'u4754', 'u3561', 'u639', 'u982', 'u4029', 'u10642', 'u397', 'u1695', 'u8871', 'u8924', 'u1080', 'u3777
', 'u1351', 'u3248', 'u3346', 'u817', 'u7473']
.
----------------------------------------------------------------------
Ran 1 test in 6988.362s

OK
```

Figure 24: Stress testing result

4 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)

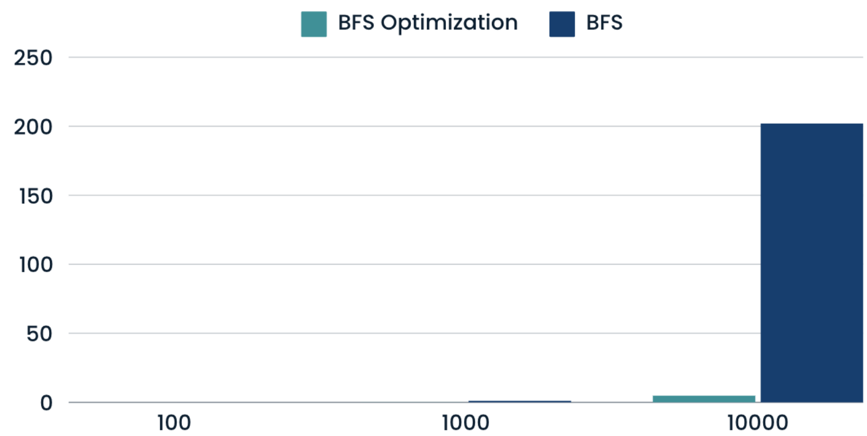

Figure 25: Reduction in execution time for Optimized BFS for sorted data (second)

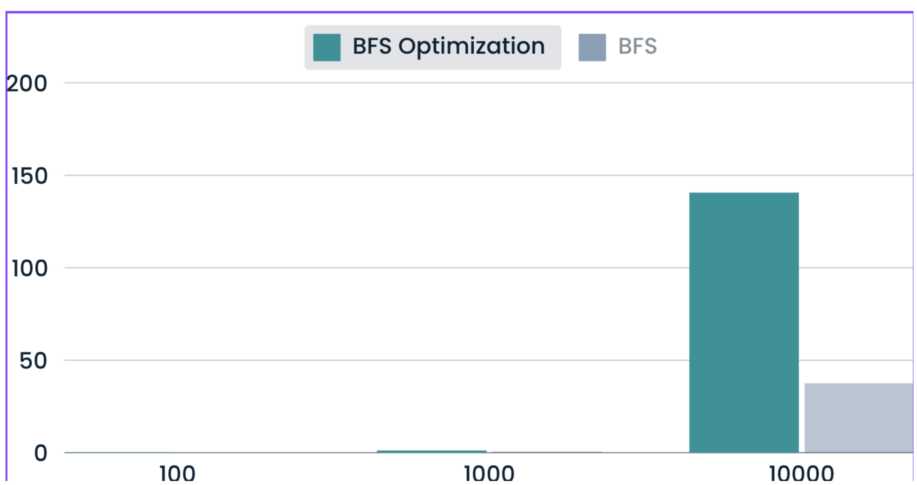

Figure 26: Increasing in memory usage for Optimized BFS for sorted data (MiB)

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] Vai Brav(2023). *Shortest Path Algorithm Tutorial with Problems*. GeekForGeeks.

[2] Saurav Paul(2023). *5 Python profiling tools for performance analysis*. Medium.

[3] Coen Bron, Joep Kerbosch(1973). *Finding all cliques of an undirected graph*. ACM Digital Library.

[4] Etsuji Tomitaa, Akira Tanakaa, Haruhisa Takahashia (2006). *The worst-case time complexity for generating all maximal cliques and computational experiments* . Science Direct.

[5] Shivani Naidu (2024). *Beyond The Breaking Point: How Stress Testing Ensures Software Stability*. QA Touch.

[6] Osama Ahmed (2023). *Pub/Sub System vs Queues*. Medium.

[7] Sven (2024). *Why is Facebook recommending me people I don't know or even common friends with*.

[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*