Friendster Network Analysis Final Report

Team The Da Vinci Code: Xi Chen, Yangyang Dai, Rose Gao, Liqiang Yu
June 2018

Contents

1. Introduction

1.1. Motivation

The world is big but it is also small. We socialize in different communities every day and our networks extend to every corner of the world. The increasing popularity of online community further reduces the geographic limitation between people and enables us to form more dynamic relationship compared to everyday life. Due to the large amount of data available for online networks, to explore the different communities, the relation between different users, the characteristics of those communities using big data are of great interest to our group. The purpose of this project is to serve as a grounding framework for social network analysis with big data.

1.2. Data Description

Our project uses the Friendster's dataset from the Stanford SNAP data collection. The original dataset is 30GB. Because our implementation is built on pairing up every single user, which potentially significantly increases our data size, we decided to use a smaller subset 'friends_000.txt' which is 140MB instead.

Since our data is well-formatted with user id and their friends' id, there was no data cleaning process needed for our project. However, there are many users with 'notfound', 'private' and blank friend lists that need to be removed depending on the analysis.

2. Community Analyses

2.1. Community Detection

2.1.1. Algorithm

In order to detect community, we first calculated the pairwise distance between every two users using Euclidean distance. For example, userA has friends user1, user2, user3, userB has friends user2, we potentially build two vectors to represent these two users, userA: [1, 1, 1], userB: [0, 1, 0]. And the distance would be square root of 2 (1+1). In our implementation, we used a trick instead of constructing the vectors, because we aimed to make our codes as simple and efficient as possible. The euclidean distance is equivalent to the sum of the number of friends of the two users minus twice the number of the common friends, and then take the square root.

After we obtained the distance output file, we used it as an input to implement community detection. Given entire distance information between every pair for every single user, we chose to rank the pairs based on distance in an ascending order, and then picked the first 50 users as a potential community. Originally we thought of give a fixed distance as a threshold, but after running several examples, we decided that an arbitrary threshold based on the number of

people composing one community would be more dynamic and accurate for our analysis due to the variety nature of distance values for different user.

2.1.2. Implementation

We mainly used MapReduce in our implementation. We read friends_000 text file into MapReduce, created a copy of the same file in the map init, and then produced key - user id, value - list of other users and the distance between them. After we obtained the pair distance output, we used another MapReduce to detect our community.

2.1.3. Results

An example output of the pair distance is shown below:


```
"16265" [["10000",4.582575695],["10001",8.0622577483],["10002",4.472135955],["10003",4.472135955],["10004",
14.6628782986],["10005",4.582575695],["10006",6.7823299831],["10007",4.472135955],["10008",5.1961524227],
["10009",5.3851648071],["10010",4.472135955],["10011",8.3066238629],["10012",4.472135955],["10013",
6.7823299831],["10014",4.472135955],["10015",4.472135955],["10016",8.5440037453],["10017",8.6602540378],
["10018",4.472135955],["10019",4.472135955],["10020",4.472135955],["10021",4.472135955],["10022",9.1104335791],
["10023",6.4807406984],["10024",4.472135955],["10025",6.8556546004],["10026",4.472135955],["10027",
8.1853527719],["10028",4.472135955],["10029",9.0],["10030",4.472135955],["10031",4.472135955],["10032",
4.472135955],["10033",5.9160797831],["10034",4.472135955],["10035",9.5393920142],["10036",10.1980390272],
```


Fig1. Sample Output for Pair of Distance

In this example, every user and their distance to user '16265' is yielded as a list of list.

A sample community output file is shown below:


```
"16265"
[[16265,18890,10002,10003,10007,10010,10012,10014,10015,10018,10019,10020,10021,10024,10026,10028,10030,1003
1,10032,10034,10037,10040,10042,10044,10046,10048,10051,10052,10055,10062,10064,10065,10067,10071,10073,1007
4,10075,10076,10077,10079,10080,10082,10084,10085,10087,10088,10091,10092,10095,10096]]
```


Fig2. Sample Output for Communities

In this example, all the users that are in user 16265's community are listed in a list of list. Overall, using 140MB friends_000 text data, running on 3 instances machine, would take roughly 12 hours.

2.2 Centrality Measures

2.2.1. Degree Centrality

In real-world social network, people with many friends and/or connections are always considered to be an important member in the community. Based on such idea, the measure of degree centrality ranks members with more friends higher in terms of centrality. The following formula shows the degree centrality Cd for node vi in an undirected graph with normalization by the maximum possible degree (Zafarani, Abbasi, & Liu 2014):

$$C_d^{\text{norm}}(v_i) = \frac{d_i}{n-1},$$

In terms of implementation, by using MapReduce, each "line" is a community. For each member in the community, we counted the number of friends, which was the degree centrality for each member. The member with the highest number of friends was considered as the most important person in this community. After normalization, we got the final value of degree centrality for each member in its community.

The following presents a sample output for the degree centrality:

```
"202" ["395", 0.0204]
"203" ["395", 0.0204]
"205" ["395", 0.0204]
"208" ["208", 0.0612]
"209" ["210", 0.1224]
"210" ["210", 0.1224]
```

Fig3. Sample Output for Degree Centrality

In this sample output, the first column shows the original center of the community, and the second column shows the most important person and its related value of degree centrality. For example, in the community whose original center is user "202", the person with the highest degree centrality is user "395", whose degree centrality is 0.0204;in the community whose original center is user "208", the person with the highest degree centrality is user "208", whose degree centrality is 0.0612 We can see that sometimes the value of degree centrality is zero. The possible reason is that in the last step of community detection, we grouped the users into communities based on the distance instead of their connections. In several communities we construct, there were few connections between the members, so the degree centrality may not be a very accurate measure in this case.

2.2.2. Closeness Centrality

The official definition of closeness centrality is "the sum of the length of the shortest paths between the node and all other nodes in the graph." Its intuition is that the more central nodes are, the more quickly they can reach other nodes. The following formula shows the closeness centrality for node x and d (y, x) denotes the distance between vertices x and y:

$$C(x) = \frac{N}{\sum_y d(y,x)}.$$

However, since another centrality measure, betweenness centrality, is also based on the concept of shortest path, we tried to come up with a new method to approximate the closeness centrality. The new measure was based on the distance instead of the shortest path. We computed the distance between each pair of users in the step of community detection. A member may be

considered as “the most central” if it has the shortest distance to all other members in a community. Therefore, based on this idea, we computed each member’s total distance to other members in the same community. After comparing the value of degree centrality after normalization, the member with the smallest closeness centrality/distance was considered to be “the most central” person. The incorporation of MapReduce made our computation much faster and more efficient.

The following presents a sample output for the closeness centrality:

```
"202" ["202", 0.2774]
"203" ["203", 0.5774]
"205" ["304", 0.7071]
"208" ["208", 0.3792]
"209" ["210", 0.1906]
"210" ["210", 0.1906]
```

Fig4. Sample Output for Closeness Centrality

The format of this sample output is the same as that for the degree centrality: the first column shows the original center of the community, and the second column shows the most important person and its related value of closeness centrality. For example, in the community whose original center is user “202”, the person with the highest closeness centrality is user “202”, whose closeness centrality is 0.2774; in the community whose original center is user “203”, the person with the highest closeness centrality is user “203”, whose closeness centrality is 0.5774. Comparing to the output of the degree centrality, the closeness centrality may be a better measure in this case.

2.2.3. Eigenvector Centrality

Eigenvector centrality seems to be an improvement from degree centrality. Degree centrality only cares about the number of friends each member has, but having more friends is not necessary to conclude that this person is important - having more important friends provides a stronger signal. The following formula shows the computation of eigenvector centrality: Ce is an eigenvector of adjacency matrix A and λ is the corresponding eigenvalue (Zafarani, Abbasi, & Liu 2014):

$$c_e(v_i) = \frac{1}{\lambda} \sum_{j=1}^{n} A_{j,i} c_e(v_j),$$

In terms of implementation, by using MapReduce, we firstly constructed adjacency matrix for each community; then selected the largest eigenvalue; lastly computed the corresponding eigenvector. The most central person is the one with the highest element in the eigenvector.

The following presents a sample output for the closeness centrality:

```
"202" ["202", 1.0]
"203" ["203", 1.0]
"205" ["205", 1.0]
"208" ["309", 0.5774]
"209" ["209", 0.5176]
"210" ["209", 0.3858]
```

Fig5. Sample Output for Eigenvector Centrality

The format of is the same as that for the degree centrality and the closeness centrality: the first column shows the original center of the community, and the second column shows the most important person and its related value of eigenvector centrality. For example, in the community whose original center is user “202”, the person with the highest eigenvector centrality is user “202”, whose eigenvector centrality is 1.0; in the community whose original center is user “209”, the person with the highest eigenvector centrality is user “209”, whose eigenvector centrality is 0.5176.

2.2.4. Betweenness Centrality

Betweenness is a measurement of centrality based on shortest paths in the graph theory. For every pair of vertices in a connected graph, there exists at least one shortest path between the vertices such that either the number of edges that the path passes through, for weighted graph, or the sum of the weights of the edges, for unweighted graph, is minimized. The betweenness centrality for each vertex is the number of these shortest paths that pass through the vertex.

In our project, in order to get the betweenness score for every user in a community, we calculated the total number of shortest path that passes through this user first. Then calculated the scaling factor, which is equal to the product of the length of community minus one and the length of community minus two and then divided by two. At last, using the quotient which obtained from the previous sum over the scaling factor, then divided by the total length of the community is the final score for each user.

Since the betweenness score depends entirely on the information of all the shortest paths between every pair of user, an it took a long time to get all the paths, this time we weren’t able to perform the betweenness centrality code on big data. Nevertheless this part serves as a framework for calculate the betweenness using Mapreduce if given an entire paths file.

3. Network Analysis

3.1. Shortest Path and Degrees

3.1.1. Algorithm

There are two questions that we are interested in. Firstly, how you could be introduced to someone you want to connect with? Secondly, how far are you from this person. Thus, we used Breadth-First Search algorithm in MapReduce, in order to find the shortest path between two users and its distance. Since we cannot store the graph information in memory, we created a file to store the graph information. Each line represents a node's information: ID | Children | Distance | Path | Color. The time complexity is O(|V| + |E|). Since BFS is an iterative algorithm, we used iterative MapReduce to iterate the algorithm 10 times to find a person within 10 degrees of connection.

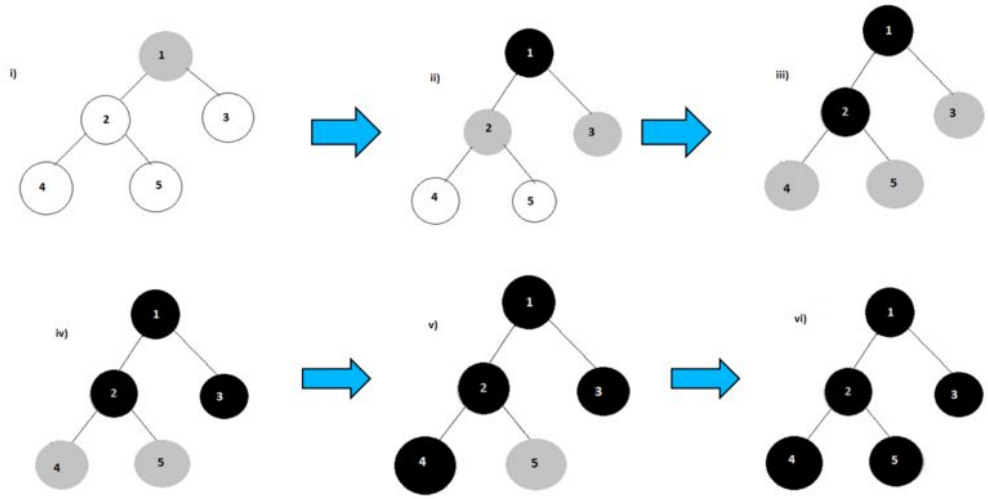

Fig6. Breadth-First Search Illustration

3.1.2. Implementation

We wrote a MapReduce version of BFS. The whole structure is shown below:

Breadth-First Search in MapReduce

1) Create a temporary file to store the graph:

   ID | Children | Distance | Path | Color (a Node)

2) Iterative Mapreduce Algorithm

   Mark the beginning node as 'gray'

   Expand each child of it

   Combiner - yield Child_ID, Child_ID | Null | Distance + 1 | Path + Parent_ID | 'gray' (a Node)

   Reducer - Reduce to the shortest path, save the darkest color, store adjacent nodes

3) Iterate 10 times:

```
for i in range(N)

        with mr_job.make_runner() as runner:
        runner.run()
```

3.1.3. Results

We randomly picked two nodes: 902 and 222 as our starting node and ending node then tried to find the shortest path and the distance between them. Here is the result:

```
running mrjob: 0
running mrjob: 1
running mrjob: 2
running mrjob: 3
running mrjob: 4
running mrjob: 5
running mrjob: 6
running mrjob: 7
running mrjob: 8
running mrjob: 9
The path from 902 to 222 is 902->899->779->222, the distance is 3
```

Fig7. Sample Output For BFS in MapReduce

BFS not only gives us the shortest path between two nodes, but it also gives us all the shortest path derived from the starting node. So we can examine the result by looking at the graph file at the end. A part of the graph file after 10 iterations is show below:

```
222|
170,217,220,233,261,298,299,302,334,342,548,779,874,980,987,22388,200728,227862,329963,444423,631216,777894,79310
7,1024174,1211003,1475118,2070592,3745539,5564172,6466512,7578558,16640016|3|902 899 779|black
2220190||9|902 899 779 217 210 249 323 125 937|black
2222869||9|902 899 779 217 210 249 323 125 937|black
2223052||8|902 899 779 217 210 249 323 325|black
2223512||6|902 899 779 217 261 881|black
22273275||6|902 899 779 810 921 924|black
222845||6|902 899 779 217 261 881|black
223||5|902 899 779 217 215|black
22317731||7|902 899 779 810 921 917 897|black
```

Fig8. Graph File After 10 Iterations

3.2. Graph Radius

3.2.1. Algorithm

After we created the Breadth First Search algorithm using MapReduce, a natural next step was to dig deeper into how nodes are connected. The motivation behind finding the graph radius was to find the most well connected person. To find the eccentricity for each node, we found the shortest path from one node to all other nodes and retrieved the maximal distance. The node with the minimum eccentricity was deemed to be the most well connected node.

3.2.2. Implementation

We wrote a program to write to a text file every viable path for nodes 100 to 400. We then implemented a MapReduce algorithm to find the minimum eccentricity from the document of paths. While the MapReduce algorithm was quick, getting the path from every node to every other node was very slow and took 9 hours.

3.2.3. Results

Unfortunately, our results were unexpected - users who were friends with 1 other user exclusively yielded a minimum max_distance of 1, such as user 102. Future work will be to implement our algorithm on the largest connected component of the network to yield more meaningful results.

3.3. Friends Recommender

3.3.1. Algorithm and Implementation

We wanted to recommend new friends to a user based on mutual friends shared with friends of friends. In our MapReduce algorithm, for each pair of people who aren't friends with each other, but share at least one mutual friend, yield the count of their mutual friends. Then, group by key (user id). Lastly, use heapq.nlargest to yield the top 5 people with whom a user shares the most mutual friends, and thus is likely to add as new friends.

3.3.2. Runtime

To recommend friends to 1,000 users, we ran our algorithm on one local machine and shaved down the computing time from ~10 seconds to 6.2 seconds. We then implemented it on a GCP dataproc and processed 1,000,000 users using 3 machines, which took 12 hours.

3.3.3 Result

We tested the recommender algorithm on a 1,000-line file and set the number of friends to be recommended is 5. A sample of the output looks like this:

```
"302"   ["217 5", "220 5", "874 5", "211 4", "222 4"]
"3023341"       ["119284 1", "131198 1", "1638431 1", "173275 1", "200507 1"]
"3027193"       ["119284 1", "131198 1", "1638431 1", "173275 1", "200507 1"]
"3028182"       ["106796 2", "1106549 2", "117425 2", "1256706 2", "13895 2"]
"3029121"       ["1032638 1", "107604278 1", "109573730 1", "111988512 1", "112609328 1"]
"3029289"       ["101 1", "1016487 1", "10177 1", "1081688 1", "10837 1"]
"303235"        ["1044265 1", "1189 1", "1194 1", "124269 1", "1275000 1"]
"30326270"      ["1007 1", "101 1", "1033050 1", "1040161 1", "1061 1"]
"3034834"       ["191 1", "3105270 1", "4777206 1", "497 1", "611 1"]
```

Fig9. Friend Recommender Result From A 1,000-line File

The first ID represents the user that we want to recommend friends to. The following list gives the top 5 friends. Each element in the list gives the friend's ID and the number of mutual friends in common, which are separated by a space.

4. NetworkX Comparison

4.1. Community Detection and Visualization

We decided to compare all of our algorithms against NetworkX. For example, this is the community detected around user 202:

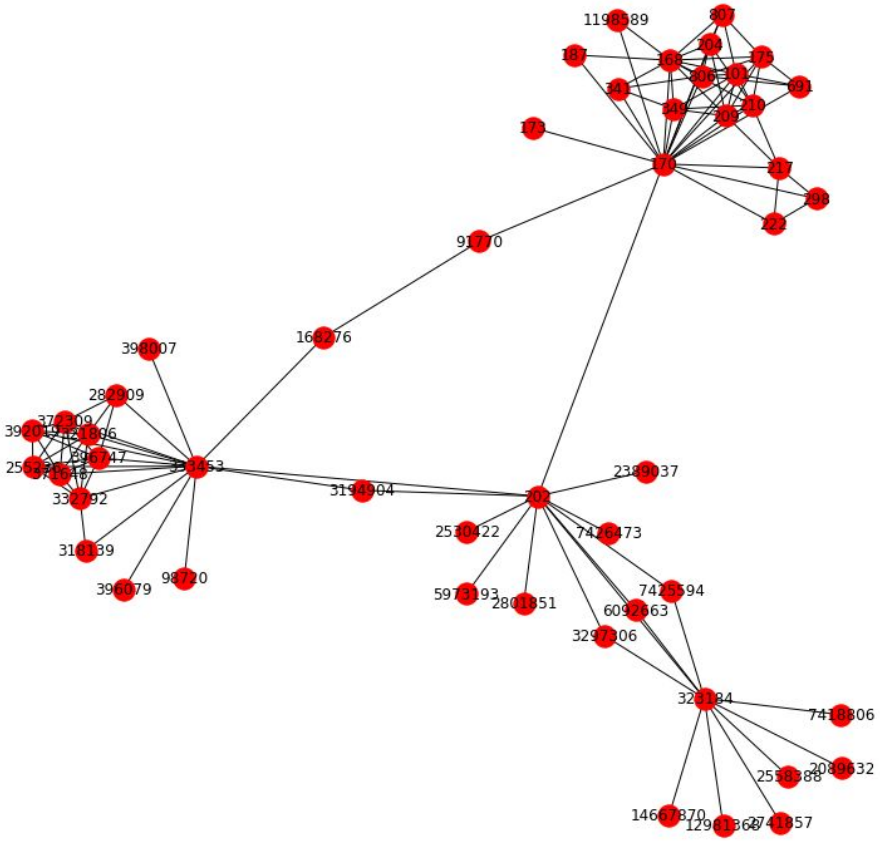

Fig 10. Sample Community Visualization

4.2. Community Centrality

When comparing user 202's community centrality measures against our MapReduce outputs, we obtained somewhat different results.

For each measure, here are the most central users and their centrality measures:

Closeness centrality: '202', 0.57

Betweenness centrality: '202', 1648.0

Degree centrality: '170', 0.388

Eigenvector centrality: '170', 0.446

We hypothesize that for degree centrality, we obtained a different result because we used distance in our MapReduce implementation of creating communities, restricting the number of nodes a user is connected to.

4.3. Friends Recommender

Using NetworkX's function to get neighbors of a user a.k.a. their friends list, we recreated our MapReduce algorithm to compute the top five recommended potential friends based on counts of mutual friends.

Runtime was much slower using NetworkX. Running our functions on 500 nodes averaged a runtime of 34 milliseconds per node. For 1,000,000 nodes, we project that to take 566 hours using NetworkX, compared to the 36 hours of combined dataproc time that our MapReduce algorithm took.

5. Challenges

5.1. Community Analysis Challenges

When using Mapreduce to calculate betweenness centrality, we confronted a challenge due to the big size of the data. Since the betweenness score depends entirely on the information of all the shortest paths between every pair of user, and it took a long time to get all the paths, this time we weren't able to perform the betweenness centrality code on big data. Nevertheless this part serves as a framework for calculate the betweenness using Mapreduce if given an entire paths file.

5.2. Network Analysis Challenges

It was a challenge to do BFS in MapReduce, but we managed to accomplish it by creating a graph file and running iterative MapReduce. The details are presented above.

5.3. GCP Related Challenges & Problems

5.3.1. When setting the number of nodes to be 8, we got errors:
Insufficient 'DISKS_TOTAL_GB' quota. Requested 4500.0, available 4076.0.
Insufficient 'IN_USE_ADDRESSES' quota. Requested 9.0, available 8.0.">

To solve this problem, we adjusted the quota limit in Google Dataproc.

5.3.2. When running on a larger file ~47 MB, we got errors:
BrokenPipeError: [Errno 32] Broken pipe : tried to output the file directly onto GCP

To solve this problem, we tried to stay connected to GCP when running, or running it in background.

5.3.3. How to run on a 9.3 GB file? Split first or upload the entire file onto GCP?

We managed to upload to a GCP bucket and use gs://yourfile when running on Dataproc.

5.3.4. Feasible to run in background?

We managed to use 'screen' program on a Google VM instance to do that.

6. References & Acknowledgement

Data downloaded from https://snap.stanford.edu/data/com-Friendster.html

Zafarani, R., Abbasi, M. A., & Liu, H. (2014). *Social media mining: an introduction*. Cambridge University Press.

We would like to express our sincere gratitude to Dr. Matthew Wachs for his enormous support and help on this project!