Social Computing P2: Social Analytics using Graph Databases

Debalin Das (ddas4)

September 25, 2016

1 Introduction.

I will go over the instructions for running my program and it will be followed by a report for all the social network metrics that I have calculated and the proof for the hypotheses.

2 Running instructions.

To run the program, do the following things:

1. Install Node.js.
2. Go to the "src" directory and run "npm install" to install all dependencies.
3. Run "node main.js" in the "src" directory to run my program.
4. Install Neo4j on your machine and create an account.
5. Make the default password as "password" or else change the DB configuration in top section of my code.
6. The default username is "neo4j".

3 Program walkthrough.

The following things are sequentially happening in my program:

1. It runs a loop for each ego center/network.
2. It deletes the current database first.
3. It creates a feature map with the feature ID as the key and the feature as the value.
4. Then, it reads all the individual features from the *.feat file and creates a map for individual features storing what feature each individual has.
5. It reads the features for the ego center and stores them.
6. It creates the ego center node with all its features with the node-neo4j library.
7. It reads the *.edges file and creates a node for each new user found in that file as well as the required relationship. If a relationship is found going from a -¿ b and b -¿ a, then it only creates one relationship for that, NOT two. Though Neo4j has a directed relationship architecture, we can mould the queries to make it such that the directionality does not matter. This is much more optimal.

8. Once this is done, you can head over to the web console (localhost:7474) to view the graph by doing a simple "match (n) return (n)".

9. After this, my program goes ahead to calculate the metrics.

10. First, number of nodes is calculated by the query

    "start n=node(*) match (n) return count(n)"

    This is displayed on the screen. This should contain all nodes in the ego network plus the ego center.

11. Second, the number of edges is calculated by

    "match (n:FBUser)-[r:FRIENDS]->(m:FBUser) return count(r)"

    This will have only half the edges in the *.edges file plus the ones added for representing the edges from the ego center to the other nodes.

12. Then the clustering coefficient is calculated first by getting all distinct neighbours of a particular node and all connections between those neighbours. All possible combinations is found by doing "n C 2". This is treated as the denominator and the actual connections as the numerator. The fraction value is calculated for all the nodes other than the ego center.

13. Then the betweenness centrality is derived by running a Cypher query on the remote machine. The query is:

    START n=node(*)
    WHERE EXISTS (n.name) and n.name <> '0'
    WITH collect(n.name) AS all_nodes
    START source=node(*), destination=node(*)
    MATCH p = allShortestPaths((source)-[r:FRIENDS*]-(destination))
    WHERE source <> destination AND LENGTH(p)¿ 1 AND source.name ¡¿ '0' AND destination.name <> '0'
    WITH EXTRACT(n IN NODES(p) — n.name) AS nodes, all_nodes
    WITH COLLECT(nodes) AS paths, all_nodes
    RETURN reduce(res=[], x IN all_nodes — res + [x, length(filter(p IN paths where x IN tail(p) AND x <> last(p)))])

    In this query, it first collects all the names except the ego center. It then finds the shortest paths for each of the nodes to every other node. It then aggregates the presence of each node in every path and then returns an array of the same with alternating node names and a centrality value.

    Note: this query takes a long time to execute!

4 Proofs

1. czhao13-01: This can be proved by the following Cypher query:

   match (m), (n)
   where exists (m.‘hometown;id‘) and exists (n.‘hometown;id‘) and exists (m.‘education;school;id‘) and exists (n.‘education;school;id‘)
   and m.‘hometown;id‘ = n.‘hometown;id‘ and m.‘education;school;id‘ = n.‘education;school;id‘
   and (m)-[:FRIENDS]-(n)
   return count(*)

   which for a sample dataset returns 12. Then running this query:

match (m), (n)
where exists (m.‘hometown;id‘) and exists (n.‘hometown;id‘) and exists (m.‘education;school;id‘) and exists (n.‘education;school;id‘)
and m.‘hometown;id‘ = n.‘hometown;id‘ and m.‘education;school;id‘ = n.‘education;school;id‘
return count(*)

returns 75, which shows only 16% of people are connected despite being in the same university and hometown. Which DISPROVES the statement. Depending upon the ego network, this would vary. You can run my program for each node and then run this query to see where it holds and where it does not.

5 References.

1. For betweenness centrality: http://www.markhneedham.com/blog/2013/07/27/graph-processing-betweeness-centrality-neo4js-cypher-vs-graphstream/

2. For clustering coefficient: http://mypetprojects.blogspot.com/2012/06/social-network-analysis-with-neo4j.html