Find two members that have the most common connections

Member’s connection data is one of the most important datasets at LinkedIn. 1st degree connection is the connection that member directly connect to. For example, if Mike sent a connection request to Sam, and Sam accepted the connection request, then Mike is Sam’s 1st degree connection, and vice versa.

More importantly, the beauty of the network is 2nd degree connections and beyond. Here, we bring the concept of common connection. To make it easier, we are using the below example to explain the concept.

Assuming that A and B are not directly connected, but both connected to C, then C is one common connection between A and B. Similarly, if both A and B connected to D, then D is another common connection between A and B. Note that if A and B are directly connected, then we do not calculate the common connection between the two.

Now, you are given a data set of 1st degree connection data from 200,000 members on a social network, with member\_id from 1 to 200,000. Each pair represents a 1st degree connection. For example, if you see “1, 100”, that means member 1 and member 100 are directly connected.

Can you find out the two members that have the most common connections? Once you find the result, can you optimize your method to improve the performance?

Dataset:

<https://www.dropbox.com/s/dv4v81k52eh78y2/common_connection_200k.csv>

Disclaimer: The dataset provided does not come from the member connection data on LinkedIn. It’s an artificially generated dataset that represents the member connections. At LinkedIn, we care about member privacy and won’t disclose member’s data to anyone.

**SOLUTION**

Andy Mishra

**Result:** Members 84644 and 99476 have the most common connections, with 163

**Languages:** Python, Neo4j (Cypher)

My initial naïve approach to this problem was to build a dictionary of members and their connections, where the keys are member\_ids and the values are sets of members that the member\_id is connected to. Then, I could iterate over this dictionary and calculate the intersection of connections for each pair of member\_ids. This obviously led to a combinatorial explosion and made for very slow performance on the full dataset, taking about 1.5 hours when parallelized across 40 cores on an AWS instance.

Since this type of social network data lends itself well to a graph data structure, I decided to set up a Neo4j instance in order to take advantage of its intuitive approach to dealing with graph data. Using the Cypher query language, I was easily able to load in the CSV of connection data and build the connection graph. Then, one simple query gave me exactly the result I was looking for.

Due to the overhead in loading the data into Neo4j, both the above approaches took about the same amount of time start-to-end. For a one-off query like this, the Python approach worked fine, but the graph database method is far more scalable as the data keeps growing. Once you have the initial connection graph set up, adding a connection or a new member takes a trivial amount of time and so you are just left with query latency.