Part 1: Actor Connections

Graph Design

Describe the implementation of your graph structure (what new classes you wrote, what data structures you used, etc)

Our graph structure implementation is designed around 3 basic classes used to keep track of connections: ActNode, Movie, and Edge. In the final submission for disjoint sets we added the member variable DisjointNode * dis as well, so that we can easily go back and forth between corresponding DisjointNodes and ActNodes to access all of the information we need.

To put these different classes into graph form we utilized C++'s unordered_map Data Structure within the ActorGraph class. One to store Actors and another to store Movies. For the final submission, we also created a DisjointNode class to use as our disjoint set structure for the actorconnections function. It has member variables of an ActNode* to keep track of the actor, size to keep track of how many elements are below it in the up-tree, and DisjointNode * parent to build the disjoint set structure. We also created a ConnectInfo struct that holds an actor name, movie title and year so that we can put it in a priority queue that will be used to find the year they become connected. The priority queue allows us to sort by year so that we can use a while loop to pop one year at a time to add to the graph, and the struct allows us to easily access the connections.

Describe *why* you chose to implement your graph structure this way (Is your design clean and easy to understand? what are you optimizing?, etc)

We chose this implemenation because we knew we would have to access both Actors/Actresses, and Movies very frequently when updating the graph's state while loading from the initial data file. Therefore, we realized we needed some sort hashing to be done. This makes connecting edges from actors with edges containing movie information quick and efficient.

All this accessing of movies and actors NEEDS a quick find method, making hash maps the optimal data structure to use. By using a string as a key for the movie, we can easily search for the element in the hashmap and when it is found we get the pointer to that element, so we are able to access the rest of its member variables quickly and easily. Since ActNode contains a vector of all of its edges, it is easy to iterate through this vector when needed to find all of that actor's connections. Since Movie contains a vector of ActNode *s, it is also easy to find all of the actors that were in a particular movie. Finally, the edges provide a single connection between 2 actors, as represented in a graph structure.

Actor connections running time

1. Which implementation is better and by how much?

   Our union implementation was significantly faster than BFS, roughly 4 seconds faster for test_pairs input file (14 times faster), Because we implemented our disojoint set using Path compression and union by size, the algorithm for insert, and find was optimized. After using the util methods given from PA2 for timing, we ran tests on the two different algorithms for finding actor connections, and union find showed a faster total runtime for finding connections between all pairs passed in the input file, as well as a faster average for finding each pair's connection. (summed up time between all pairs divided by the number of pairs).

   After running the program with an actor pair input file containing 100 queries of the same actor pair, the runtime for BFS showed an even greater difference in runtime compared to that of union find. Union find completed finding all actor pair connections roughly 30 seconds faster than BFS. In relative comparison 16 times faster than BFS.

   When running the program with pair.tsv (the input file containing pairs of actors all with no connection throughout all years), both algorithms ran extremely quickly (under 1 second), and ran in approximately the same time.

   Based on the results explained above, we can conclude that the disjoint set implementation is "better."

Part 2: Communities within Graphs

CSE 100 PA4 June 4, 2017
Authors: Allison Reiss and Sarah Gemperle
Extension Report:

The problem that we solved in our extension was finding different communities within a dataset. We wanted to find a way to split a social network graph into community clusters based on their connections.

To solve this problem, we used the idea that a community has more links within the community than external to it. So to find communities, we wanted to look for densely connected regions as kernels of communities, and then we could split them up by removing the links between the communities. To calculate which nodes were more central to the communities and which were border nodes, we evaluated their betweenness centrality. Betweenness centrality is the measure

of centrality in a graph based on shortest paths. The betweenness centrality for a single node is the sum of the number of shortest paths that pass through that node. Therefore, the node that is on the most shortest paths is likely to be a border between 2 communities.

We used Brandes' Algorithm to calculate the betweenness of each edge in the graph. The algorithm works by looping through every node and performing a breadth first traversal of the graph. In each traversal, we compute the number of shortest paths that go through each node. We then calculated the dependency for each node, which is defined as the ratio of shortest paths. We calculate this for each neighbor by summing the ratios of the neighbor's number of shortest paths/curr's number of shortest paths. Finally, we calculate the dependency of each node by summing these dependencies to get the total betweenness of the node throughout all iterations. This algorithm is O(m*n), where m is the number of edges and n is the number of vertices.

To test this, we downloaded multiple Facebook friend sets of varying sizes, the max of which was just over 50,000 nodes (filename: "107.edges"). In addition to testing multiple datasets, we also tested with different parameters for numbers of splits. We tested multiple splits from just 1 to 50 splits, where splits is the number of nodes that are deleted (once per iteration) in an attempt to find and cut the border nodes to separate communities. Note that for larger datasets, it takes many more splits to break up the communities because the nodes are often extremely interconnected and there are few nodes that are a definite border. To test on smaller datasets, we had to modify our original datasets so that they were more sparse. We have provided the a dataset with only 100 nodes in the file "fbFriends.txt".

Link: https://snap.stanford.edu/data/egonets-Facebook.html
Follow the link above and download the facebook.tar.gz file. We ran on a couple of these provided datasets, starting with the file labeled "0.edges". You can test our code on any of the files ending in ".edges".