DS210 Report - Charlotte Coco Greene

Git Repo - https://github.com/ccgreene/DS210_project

1. For Git, I created a git repository on git. I linked the Redhat server to that repo. For each commit, I would do git add file, git commit -m “message”, git push
2. I kept it updated throughout the project

Overview of the Data set

https://snap.stanford.edu/data/github-social.html

Dataset statistics	
Directed	No.
Node features	Yes.
Edge features	No.
Node labels	Yes. Binary-labeled.
Temporal	No.
Nodes	37,700
Edges	289,003
Density	0.001
Transitvity	0.013

Output Description:

1. First, it should find the number of walks. This takes a very long time because of how large the data set is and because the ideal number of walks is so large. I start the iteration at 200 for the sake of runtime and because I know it converges around 300. It should output the ideal walk number and the number of iterations
2. It should then output the 25 most connected vertices and their corresponding pagerank.
3. It then finds the similarity in friend circles between one random user from the top 25 and all other users
4. It then outputs 10 random social circles.
5. It then shows statistics for social circles and the general distribution of how big a circle is

Main.rs

1. Make adjacency list using Graph.rs
2. Find the most accurate number of walks
3. Calculate the pagerank with that number of walks
4. Sort the pagerank so the most connected are on top

5. Pick a random user from the top 25 pagerank and see their mutual friends with the other top 25
6. Find social circles and print 10 random ones!
7. Make a graph of the social circle proportions
    a. https://docs.rs/plotters/latest/plotters/series/struct.Histogram.html
    b. I used the plotters histogram code as the basis for my own code

Pagerank.rs

The pagerank.rs module has all functions relating to pagerank. It finds the most accurate pagerank.

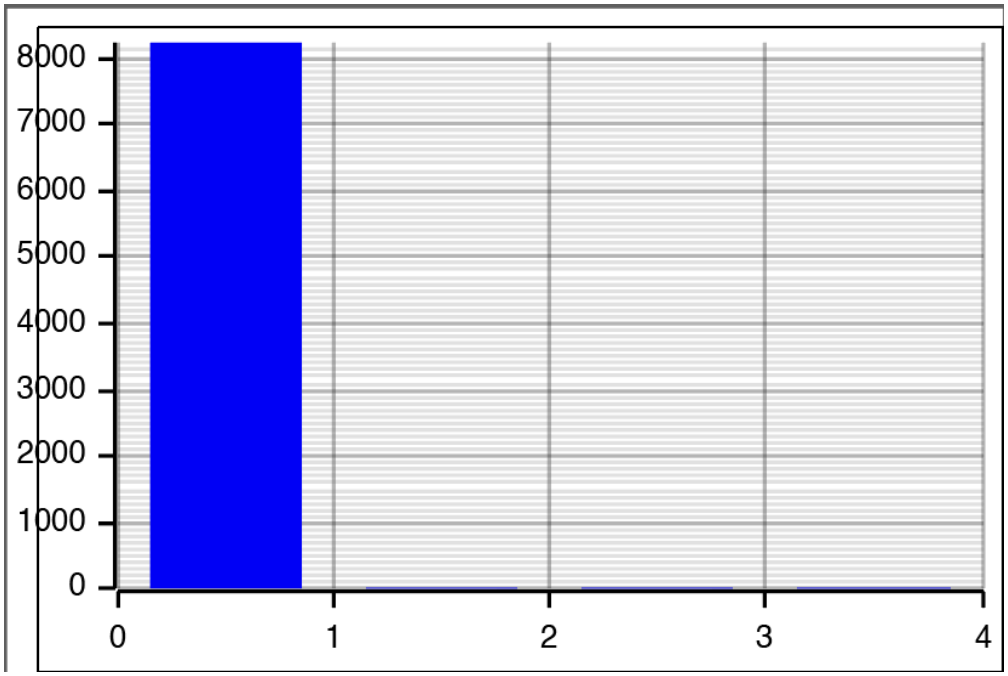
- https://medium.com/biased-algorithms/pagerank-algorithm-explained-5f5c6a8c6696
    - This explains the concepts of pagerank and convergence in detail, I used it for reference.
- fn random_walk is a helper function for pagerank. From a given node, it goes to a random edge with an 85% probability and a 15% of jumping to a random node. I chose the walk length as 120 based on computation costs. This is accurate ish just based on playing with the walk length on my own. I know that the accuracy truly converges at a number higher than 120, but any length more than 120 causes a massive amount of processing time and crashes the server 50% of the time (not good). My computer would spend 25 minutes just to process the number of walks function alone. The graph has 33770 nodes and low density so a higher walk length would be more accurate, but the processing power of the server limits me here.
- Fn pagerank finds the pagerank. Pagerank is a measure of connectivity and importance between nodes. It finds the probability of finding a node by dividing the total random walk counts by the number of total walks for each node. This takes a long time due to the number of nodes (37700), the number of walks, and the walk length (120). In my fn, I first assigned every node a pagerank of 0.0 to account for possible errors with shape if a node has no edges. I optimized the number of walks in the next fn.
- pub fn most_accurate_walk_count finds the number of walks for the most accurate pagerank, or the number of walks when it converges. First, I chose the tolerance to be 1e-3 based on playing around with the data. If it is anything smaller than that, it goes on infinitely. Initially, I had it at 1e-4 but the server timed out. I have experimented with it a lot and every num_walks output seems to plateau before 1e-4. Num_walks increases by 10 each time because I thought anything smaller would be too computationally expensive. The fn works with recursion and iterating over previous and current pageranks to see if they have converged, changing the number of walks each time. Initially, I had a separate fn that found the difference in pagerank but that was too computationally expensive because cargo would have to convert two pagerank adj_lists into array1 form then back then find the difference.

Graph.rs

- Fn add_edge is a helper function for making a graph and pushes an edge onto the end of an adjacency list. I use it throughout testing mainly.
- Fn make_graph creates an adjacency list with header id_1 and id_2. It first indexes the columns and then for every value in column 1 matching the start node, it pushes that edge into the adjacency list using fn add edge.

Bfs_friends.rs

- Fn bfs. The breadth-first search algorithm starts with one vertex and travels down to its children for as long as directed. It uses the queue system where it starts at the start vertex, then goes to its neighbor (adding to visited vector and pushing to the end of the queue), and then goes to that node's neighbor, and so on. I keep track of the visited order for use in the social circles function below.
- Fn find_common_friends finds the common friends between two users. This data is sparse with a density of .001 and transivity of .013. There are also 37700 nodes with the majority of those users having few connections. To obtain a meaningful result, I used pagerank to find the top 25 most connected nodes. I only printed the connections with the top 25 most connected users so I would most likely get a result, and because it was less computationally expensive. In previous iterations, I would choose one random user and compare it to another random user, and almost every time they had no friends in common which was uninteresting. I calculated the similarity by doing the number of shared friends divided by the total number of friends. I was going to make a separate similarity function, but they were so similar I just added more statistics to this. The similarity is really comparing their social circles.
- Fn find_social_circles. I found the circles using the bfs algorithm. I would start at a node and use bfs to find its friends and then its friends' friends with traversal order. The issue is that this graph was sparse. In main, I output 10 random circles, and almost every time, the majority of them are just one person. I have a graph below showing this. These numbers change slightly each time but ~8167 out of 8179 of the circles have less than 10 connections which is what the graph is supposed to show. The majority of circles have very few people. ~24 circles have 10-100 users, ~1 circle has 101-500 users, and ~5 circles have 500+ users. While the majority of circles have very few people, usually just themselves, the majority of users are in the large 500+ circles. The large circles probability represented well-connected software engineers or entire companies, while the unconnected circles were probably students or young coders. I imagine the 101-500 circle is a midsized company while the ~5 circles with massive amounts of users are probably big companies.
    - The number of circles is shown in the graph below, but because the numbers are so disparate, it is hard to tell.



Tests

1. Graph.rs
    a. I test the add edge fn to make sure it builds an adj_list correctly
2. Pagerank.rs
    a. test_random_walk() I test this to make sure that it returns a vertex and that it does not return something for every vertix
    b. test_most_accurate_walk_count() tests to make sure num_walks is less than the max walks and more than the initial walks
    c. test_pagerank() makes sure that all pageranks add up to around 1 because all probabilities should
3. Bfs_friends.rs
    a. test_find_common_friends(), I created a small adj_list where I could visually see the common friends and checked to make sure that the fn also got it right
    b. test_find_social_circles() I created a new adj_list and made three separate circles, 1 having no friends. I then made sure that each circle was grouped correctly and that the friendless node was it sown circle.

Conclusions

- Because this was a sparse graph, I cannot be completely confident in my conclusions. From the social circle function, I think that most developers in smaller social circles were either independent or students. Those with very large social circles were in large companies and were likely connected with their coworkers. Generally, I do not think

GitHub is very connected. The culture, from personal experience and this data, is that you only friend who you actively know and that it is normal to have few git friends.