Name: Keenan Willison
Date: 12/15/23
CDS DS 210
Final Project Writeup

Context:

For context, my dataset was an undirected social network in Facebook that had over 4000 nodes (friends) and showed all of their connections between the other nodes. Before looking at the data, my initial goal for this project was to try and find the most popular friend that interconnected the most friendship circles within my data. I was also curious to see if there were any disconnected friend groups from the rest of the Facebook users. I wondered if social media was as truly interconnected as I thought, or if it was more independent than I thought.

Code Description:

Let's break down what my code does. I broke my code into 4 different modules: file_manipulation, graph, algorithms, and tests. First, in the file_manipulation module, I implemented code that will read the file. The function opens the downloaded file, reads the file line by line, splits the data file to put the edges into a list of edges, and will return an error if there's a problem with reading the file.

The next module was the graph module, where I constructed the undirected graph for my dataset. Here, I included three functions. The add_undirected_edges function added undirected edges to the graph by pushing both ways to show the nodes are both connected to each other. The sort_graph_lists sorts the adjacent list to help it keep the order consistent. Lastly, the create_undirected function creates the undirected graph based on the given number of vertices and edges.

The algorithms module contains five separate functions. The first function is the friend_circles function, which will find the different friend circles (if there are different ones) from the data. It uses Breadth-First Search in order to find the friend circles. This function initializes a visited vector to track nodes, and it iterates over each node in the graph and keeps track of whether the node has been visited or not. If the node hasn't been visited, it gets set to true and then gets pushed to the queue for further exploration. Then it traverses the neighbors of the current node to see if the neighbor has been visited, and if it hasn't then it assigns that node to the next_neighbors set, showing that it belongs to the next BFS layer and pushes it into the queue. Once it's done the BFS, it checks to see if the circle vector is empty or not. If it's not, it will push the circle vector and then sort it, and then return the friend circle.

The next function is most_popular_friend, which essentially looks for the friend (node) that lies on the shortest path between other nodes the most frequently. This is considered the most popular friend. It uses a BFS to track different paths and keep track of how many times each node appears on these paths. It goes through the BFS (as explained above), but the difference is that this function returns two HashMaps, paths and number of paths. The "Paths" HashMap stores the paths from the node we are on to the other nodes getting traversed, while the "number of paths" HashMap counts the number of shortest paths through each node. Once I

call it in the main using the max_by_key feature, it finds the node with the largest count, which is our most popular “friend.”

The third function in the algorithms module is the shortest_path function. This function finds the shortest path between two “friends” from the graph. This function also uses BFS like we’ve seen in the first two functions. It initializes a queue and explores the neighboring nodes. If the current node in the BFS == node 2 (our ending node), then it creates a new path, but creates the path backwards, going from node 2 back to node 1. So towards the end of the function, it reverses the order to get the true path from node 1 to node 2. If no path is found, it will return None.

The fourth function is the get_random_nodes function, which I need to call in the tests and the main function when finding the average_path_distance (haven’t talked about this function yet). This function creates a vector containing the indices of nodes from the graph. It then shuffles the node indices, and then shortens the vector to the given sample size we want in order to get the random “friends” from the dataset.

The fifth function in the algorithms module is the average_path_distance function, which calculates the average distance between random pairs of nodes in a graph using parallel computation. It uses rayon (data-parallelism library) to speed up the iteration of random nodes in order to find the shortest path between them. The distance of this path is then added to the total_path_distance AtomicUsize variable, which will store all of the other shortest path distances. If there is a path between nodes, then the total_node_pairs counter increases by one. This function uses AtomicUsize to have thread-safe mutable counters (total_path_distance and total_node_pairs). Finally, it computes the average distance by dividing the total_path_distance by the total_node_pairs.

The last module I used was the tests module. I had three tests in this module, one to test the shortest path function, one to test the average path distance, and the last one to prove the number of nodes from the dataset was correct. The test_shortest_path function takes an example graph, and tests the shortest path between nodes 0 and 4. It then verifies that the path is what we expected it to be, along with the two nodes. The test_average_path_distance function also creates an example graph and generates random nodes from the graph. It then manually calculates the expected average distance and verifies that it’s equivalent to the average distance calculated by the average_path_distance function from the algorithms module. The last test was to test the node count. It reads the imported file, makes the graph, then checks to see if the number of nodes in the graph matches what the original website stated.

Output:

The first thing I wanted to output was to see if there were any distinct friend circles because that was one of my main curiosities before looking at this dataset. Once I ran the friend_circles function, it came back with only one friend group, which contained all the nodes in the graph. I knew that if the length of the friend circle was equal to the number of nodes in the graph, then we know every friend in the dataset is connected. This answered one of my original questions about social media’s interconnectedness, and it proved that everyone (in this dataset) was connected. There were no independent friend circles separated from the rest of the friends in this Facebook dataset.

The next outcome I got was when I called the most_popular_friend function. What my code outputted was the one friend (node) that lied on the shortest path between other nodes the most frequently. This was a very interesting result to me because I had always wondered if in a social network if there was one person who could lead to a lot of other connections. In this Facebook dataset, it was friend 1912. Applying this into other popular social media apps today like Instagram or Tiktok, there is most likely an individual who is most important for connecting two random individuals, just like my code proved. One interesting thing I noticed in my result was that the node 1912 stayed the same (meaning it was always the most popular friend), but the number of paths slightly varied each time. For example, one time I run it and it will output: `The most popular friend is node 1912 with 6915 paths`. And then the next time I run it, it will produce: `The most popular friend is node 1912 with 6910 paths`. This could be the case because if there are multiple shortest paths of the same length between nodes in the graph, the algorithm might treat them differently across runs. If there was a tie in the shortest path, running the function once might assign it to one node while running it again might assign it to another. However, the variation in paths was very minimal, and node 1912 was always the most popular.

The next output was the shortest path output between two random friends. Here is a screenshot of the last time I ran it (this will be different the next time because they are two random friends):

```
The shortest path between node 3906 and node 333 is: [3906, 3437, 1085, 107, 0, 333]
```

This shows the shortest path between node 3906 and node 333. Node 3906 would go to node 3437, then 1085, then 107, then 0 and then it would finally reach node 333. So overall, this particular example would take 5 steps for “friend 1” to reach “friend 2.”

The final output of my code showed the calculated average path distance between random nodes given a sample size. Here’s a screenshot of what my code outputted:

```
The average distance between 100 random pairs of nodes is 4.6664.
```

Given this specific sample size of 100 random pairs of nodes, the average distance between the nodes was about 4.67 steps. This means that on average, the shortest path between any two random friends is about 4.67 steps away from each other. The only limitation to this code was the computation speed in which this function ran. The reason why I implemented the parallel computation in the average_path_distance was to reduce the computation time, which it did. It decreased the computation time by about 10x. I then continued trying to parallelize the code to reduce the speed even more, but couldn’t find a great solution. I tried parallelizing parts of the shortest_path function because the average_path_distance function calls it, but once I ran it, I got the same computation speed, if not slower. This is why I used the sample size, because with a smaller sample size, it still results in an accurate average path distance, but it takes a lot less time to compute. If I’m looking at areas to improve my code, this would most likely be it.