DS210 Project Writeup

This project is an analysis of a YouTube Comment database I curated via Python Scripts that use the YouTube Api. To go over the data collection briefly, it is split into 3 separate programs in order to ensure data integrity and tackle each part of the collection problem one by one. The initial csv's from the collection have been removed, and only the comments.csv is in the project so it can be used by rust.

Now to get to the rust aspect of this project. To start, I first separated the structure of the project into 3 files: analyser.rs, grapher.rs, and main.rs.

Grapher.rs is a module file that contains all the functions for ingesting the data into rust and getting the data into its appropriate data structure. First, is the *comment_threads_neighbors* function that first iterates through the csv and creates a mapping of each comment thread to all the commenters who commented in that thread. Now that it has the mappings for each comment thread, I was able to create an adjacency list representation of the graph with the node being a commentator and the neighbors being people that person has commented *with*. An interesting thing to note in this part is the iterator section of this function. What this function does is for each of the comment threads, it iterates through each commenter. Then it takes all the commenters before that commenter, *(.take(i))*, and then takes all the commenters after that commenter *(.chain(commenters.iter().skip(i + 1))* and puts them into a vector of neighbors. This adjacency list of commenters and those they commented with is then returned. The *commenter_and_channels* function is a bit more simple, with it just iterating through the csv, and with the comment author being the key, and a hashset of channels visited being the value pair. Similarly, the *channel_and_commenters* function does the same thing except the channel is now the key, with a hashset of the commenters being the value pair.

Now with the graphs ingested via the *grapher.rs* the analysis is then computed via functions from *analysis.rs*. The first function *most_neighbors* simply just calculates who has the most neighbors for a graph, via a pretty simple for loop. The page rank implementation for this project uses three separate functions: *step, walk, and random_key.* Page rank starts by first taking in a map of string to vector of strings and the number of times you want to go. (In this context it was a commenter and their comment thread 'neighbors.') The reason for including this n is because since the dataset was so large, running pagerank on each of the 160,000 nodes was proving to be pretty time intensive, so I opted to run on 10,000 randomly selected nodes. Now with the random node selected it will then *step* to a new vertex via the step function. This step is done by first generating a random number between 1 and 10, if it is less than 9, step to a random neighbor, and if it is 10 or the person has no neighbors, jump to a random vertex (or key). This random jumping is done via the random_key function which is pretty self explanatory, (Puts all the keys into a vector of references, and then generates a random index to get the key). Now that it has stepped to a new vector, whatever the new vector is is a key to a ranking map, and it will increment the score by 1. Finally, it then divides each person's count value, by the total number of walks done, thus getting the page rank of each person. *PageRank* is then

tested by first a test that ensures it all sums up to 1, and then by a function to ensure that PageRank is properly ranking people based on their popularity. The other implementation in this program is a breadth first search algorithm. This runs via a VecDeque where the function will first add all the neighbors of the start vector to the queue. Then it will iterate through each of the neighbors and make the neighbor a key to a hashmap where its value pair is the distance from the start node. Then the neighbors of that node are added to the queue if they have not been processed with a distance of + 1, and the current vector is popped out because it has been processed. Finally, a hashmap of each person and their distance from the start node is returned. Now with this mapping, I then created the function channel_commentor_seperation which is a function that will get the average span of neighbors for each youtube channel. This function takes in a vector of all the bfs graphs generated, and the mapping from each commenter to the channels they have commented on. Then it first makes a hashmap of the channel distances. This hashmap uses a special struct *ChannelCounter* as the value which is just a sum and a count variable. Now for each bfs graph in the bfs vector, it then gets the channels associated with that commenter. Then, this function iterates through each channel. Then this channel is cloned and entered into the hashmap, where it then first uses a closure to define a *ChannelCounter* variable with zero values. It then uses the implemented *update_values* method to update the counter values for each channel. With the mapping of each channel to the sum of all the distances to that channel, the average distance is calculated for each channel by using the *ChannelCounter* structs implemented average method. Finally, these average distances are sorted and printed in Descending Order.

Finally, the main method mainly just implements the above functions and properly formats the output for how I want it to look. The first thing to note is calculating the average distance between all comment threads. This is done by first doing a bfs on each node and then getting all the distances from that node, and incrementing by the number of values in the bfs graph. Also, the bfs graph is pushed into the *bfs_vector* so it can be used later in the comment_thread_neighbors function. Another thing to note is the recasting of the *channel_comments* variable. This was done in order to save a bunch of memory because this graph would not be in use for the rest of the program. The last thing to note is that I converted the ranking map obtained from the page rank function into a vector of tuples so that I could rank the top 25 Commenfluencers obtained from the page rank algorithm. With that being said, those are all the main aspects of my project. : -)