DS210 Final Project Writeup

- **I used chatgpt to help develop my find_mutual_friends function, and will cover this in depth later in the writeup. A copy of the interaction that I could find in my history is also attached.**

For my final project, I chose to analyze this Facebook Friends Data set. I am a Facebook user, and chose this set because I am interested in exploring the levels of interconnectivity between users on social media networks. Specifically, I wanted to code a project that finds the first, second, and third degree mutual friends for a given user, and what percent of the entire dataset those mutuals represent. I am looking to find how much of the data set a given user is friends with, as well as who has the most friends in the dataset, and what the average number of friends a person has in the dataset.

The Read_file Function

After loading the dataset into my Rust project, I implemented a Read_file function to go over the data I provided. I used the same format for a file reader function that I used in Homework 10; importing BufReader to read the file and implementing a for loop that iterates over each line of data and changes it into a vector of tuples that represent the edges of the graph. Each tuple of edges is then pushed onto the graph vector, and I returned it at the end. Additionally, I put my file reader into a separate mod from my main code so that everything would stay organized. To do this I initialized a mod called File_reader and copied the function into it so that it wouldn't clutter the main area.

Graph Analysis

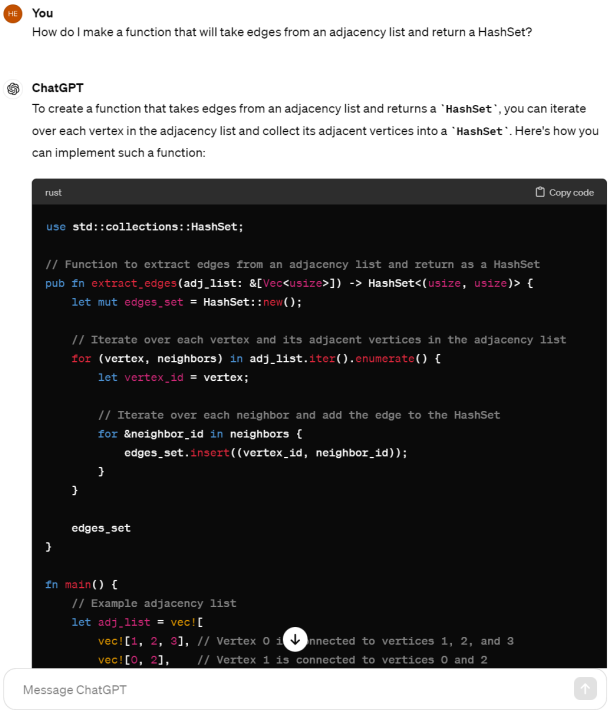
I put all my graph analysis functions into their own mod so that they would be organized and separated from the main. I named the mod graph_analysis, and inside are the functions adjacency_list, bfs, and find_mutual_friends. These three functions construct and analyze the graph of the data set and select a user and search their mutuals.

The adjacency_list Function: This function takes a reference to the edges in my graph in the form of a slice of tuples, and it returns a vec of vecs in usize. The first for loop in the function finds the maximum node in the edges by iterating over them (finding the max between i and j). then , I initialize the adjacency list with an empty vector and max_node +1 to represent the neighbors with the vec of vecs. I then implemented a second loop that iterates over all edges, and adds the neighbors of each node to the adjacency list based on the i and j values. Finally, I returned adj so that the function would execute.

The bfs function: The breadth first search function in my code takes in the adjacency list I made above and also a starting vertex that marks where the bfs function will start traversing from. It returns a vector of the vertices it visited. First, I initialized some items that I will use in the function; "visited" is a HashSet that ensures I won't revisit vertices I've already seen, "queue" is VecDeque, storing the vertices that need to be visited, and finally "bfs_order" stores the order that my function visited the vertices. I started the BFS by putting the start vertex in the queue and marking it as "visited". Then, I constructed a while loop that will keep iterating as long as there are vertices in the queue. From there, the loop pops a vertex from the front, marks it's order with bfs_order, and uses adjacency_list to find its neighbors. Then the for loop looks at each neighbor, and marks it as visited if it hasn't been visited yet, and adds it back to the queue.

Finally, I added an else statement that will take care of situations where the vertex is out of bounds. Then, I returned bfs_order.

The Find_mutual_friends function: this function takes in a user id, a reference to my adjacency list, and the degree of separation I want to consider between friends, and it returns a HashSet with the ID's of mutual friends in my selected range of separation from the selected user. I initialized three things to start with: "visited" keeps track of the user ID's I've already seen to avoid revisiting them, "mutual_friends" is my HashSet that stores the mutual friends, and "queue" is a VecDeque that stores a tuple of user ID and degree for the BFS. Next, I implemented a While loop that will continue running until the queue is empty. It begins by popping a (user id, degree) tuple from the front of the queue, and skipping over it if the degree is greater than the current degree. It then checks if the degree is greater than 0 (meaning that this user has at least one mutual) and inserts the correlated User ID into the "mutual_friends" HashSet. Then, I made a for loop that begins by getting the friends of the user from the adj_list, and marks unvisited friends visited, adding them back to the queue with their degree +1, to show that it's one degree away from the User ID we selected. Finally, I returned the "mutual_friends" HashSet. I used ChatGPT quite a bit in constructing this function to help me with understanding VecDeque and combining
my adj_list function and bfs to put the code all together. It also helped me to figure out what to pop from the queue and how that works.



```rust
fn main() {
    // Example adjacency list
    let adj_list = vec![
        vec![1, 2, 3], // Vertex 0 is connected to vertices 1, 2, and 3
        vec![0, 2],    // Vertex 1 is connected to vertices 0 and 2
        vec![0, 1],    // Vertex 2 is connected to vertices 0 and 1
        vec![0],       // Vertex 3 is connected to vertex 0
    ];

    // Extract edges from the adjacency list
    let edges_set = extract_edges(&adj_list);

    // Print the extracted edges
    println!("Extracted Edges: {:?}", edges_set);
}
```

This function, `extract_edges`, iterates over each vertex in the adjacency list. For each vertex, it iterates over its neighbors and adds the corresponding edge (represented as a tuple of vertex IDs) to the `HashSet`. Finally, it returns the `HashSet` containing all the extracted edges.

You
How do I implement VecDeque into my function?

ChatGPT
To implement `VecDeque` into your function, you can use it instead of `Vec` for your queue data structure in the breadth-first search (BFS) algorithm. Here's how you can modify your `bfs` function to use `VecDeque`:

rust Copy code

```rust
use std::collections::{VecDeque, HashSet};
```

Message ChatGPT

rust Copy code

```rust
use std::collections::{VecDeque, HashSet};

// Make breadth-first search
fn bfs(adj_list: &[Vec<usize>], start_vertex: usize) -> Vec<usize> {
    let mut visited = HashSet::new();
    let mut queue = VecDeque::new(); // Use VecDeque instead of Vec
    let mut bfs_order = Vec::new();

    queue.push_back(start_vertex);
    visited.insert(start_vertex);

    while let Some(vertex) = queue.pop_front() {
        bfs_order.push(vertex);
        if let Some(neighbors) = adj_list.get(vertex) {
            for &neighbor in neighbors {
                if !visited.contains(&neighbor) {
                    visited.insert(neighbor);
                    queue.push_back(neighbor);
                }
            }
        } else {
            eprintln!("Vertex index {} is out of bounds!", vertex);
        }
    }

    bfs_order
}
```

In this modified version, `VecDeque` is imported from `std::collections` at the beginning of the code. Then, instead of `let mut queue = Vec::new();`, you declare `queue` as a `VecDeque` using `let mut queue = VecDeque::new();`. The rest of the function remains the same, utilizing `push_back()` to enqueue elements and `pop_front()` to dequeue elements, making it suitable for a BFS algorithm.

You
Do I need a while loop in my function to find friends of friends?

ChatGPT
Yes, you do need a loop to implement the breadth-first search (BFS) algorithm to find friends of friends. The BFS algorithm iteratively explores neighbors of nodes to find all nodes reachable from a starting node within a certain degree.

Here's a basic outline of how you might structure your function to find mutual friends up to a certain degree:

rust Copy code

```rust
use std::collections::{VecDeque, HashSet};

// Function to find mutual friends up to a certain degree
fn find_mutual_friends(
    user_id: usize,
    adj_list: &[Vec<usize>],
    degree: usize,
) -> HashSet<usize> {
    let mut visited = HashSet::new();
    let mut mutual_friends = HashSet::new();
    let mut queue = VecDeque::new();

    // Start BFS with the user_id and degree 0
```

```
    let mut visited = HashSet::new();
    let mut mutual_friends = HashSet::new();
    let mut queue = VecDeque::new();

    // Start BFS with the user_id and degree 0
    queue.push_back((user_id, 0));
    visited.insert(user_id);

    while let Some((user, current_degree)) = queue.pop_front() {
        // Check if the current degree exceeds the specified degree
        if current_degree > degree {
            continue;
        }

        // If the current degree is greater than 0, add the user to mutual_friends
        if current_degree > 0 {
            mutual_friends.insert(user);
        }

        // Explore neighbors of the current user
        if let Some(friends) = adj_list.get(user) {
            for &friend in friends {
                // If the neighbor hasn't been visited yet, add it to the queue
                if !visited.contains(&friend) {
                    visited.insert(friend);
                    queue.push_back((friend, current_degree + 1));
                }
            }
        }
    }

    mutual_friends
}
```

The main function

The main function reads into the facebook_combined.txt file using the file_reader mod I established, and it stores the pairs of User ID's it finds in "edges". (pairs of user ID's represent friendships). If it encounters an error, it returns early and prints an error message. Then, it uses those edges to construct the adjacency list. Then, I began making the code answer the questions I asked in my project proposal by starting with some basic calculations: 1) I found the number of users by calculating the length of the adjacency list, 2) I found the number of friendships by calculating the length of the edges variable, and 3) I found the average number of friends per user by dividing the number of edges by the number of users. To find the user with the most friends, I constructed a for loop that iterates over all the users in the adjacency list and updates the max_friends_count and records the User ID if the user has the most friends. Then, the Breadth First Search begins. I selected 6 as my start vertex, and asked the function to find the first, second, and third degree friends for that user using the find_mutual_friends function. Finally, I asked the function to find the total number of mutuals for that user by finding the length of each HashSet representing the first, second, and third degree mutuals and naming it "total_mutual_friends". Then, it calculates what percentage of the dataset this user's mutuals represent by dividing total_mutual_friends by the number of users and multiplying by 100. Then all the results are printed.

My output

BFS order from vertex 6: [6, 89, 95, 147, 219, 319, 327]
User with the most friends: 107
Number of friends: 1043
Average number of friends per user: 22
First-degree mutual friends: {89, 147, 95, 219, 319}
Second-degree mutual friends: {95, 319, 327, 89, 147, 219}
Third-degree mutual friends: {319, 327, 89, 95, 147, 219}

Percentage of the dataset represented by mutual friends: 0.42%

- For my output, I selected the starting vertex of 6 for my BFS, and it found that the user with the most friends was User ID 107. User ID 107 has 1043 friends in total, and the average number of friends for a given user in the data set is 22. Then, I found the user ID's of the first, second and third degree mutual friends for user 107 and printed them out. Finally, I found that the percentage of the dataset represented by User 107's mutuals was 42%.

My tests

I Implemented three tests in my code. They test the accuracy of the find_mutual_friends function and ensure that it properly returns the lists of IDs for mutuals of each degree. All three of the tests read off of a small sample data set I created called little.graph.txt, which has 30 pairings of edges representing friendships. Each test initializes an adjacency list and feeds it edges just like in the main, and then I selected user iD 0 to find mutuals for. Test_mutuals_1 looks for the first degree mutuals, which I input in the expected_friends HashSet as [1,3]. This test passed. Test_mutuals_2 follows the same protocol, but finds the second degree mutuals for user 0 in the little.graph set, which I predicted to be [1,3,2,8,4,5,6]. This test passed. Finally, I found the third degree mutuals for User 0 with test_mutuals_3, and predicted the mutuals to be [1,2,4,9,7,8,3,6,5], which passed.