

DS 210 FINAL PROJECT

MIYA STAUSS

OVERALL:

The dataset I chose to use is a disease-disease network dataset with around 1100 vertices. I cleaned out the data to make sure there are only integer values with nodes and edges. This dataset explains the relationships between inherited, developmental, and acquired human diseases with the nodes representing diseases and edges representing the associations between them. This dataset is interesting to me because each node directly affects the other and will allow me to make connections about how different diseases are acquired in someone's lifetime. I have always found the relationships of diseases very interesting and wanted to learn more about how each virus interacts with one another. This showed me interesting findings and provided me more perspective on diseases.

The question I wanted to ask is how similarly related inherited (genetic), acquired (develops after birth) and developmental (gained throughout childhood) of diseases are. Specifically, are there similar characteristics of diseases that would lead to one being not mutually exclusive (can you get a disease at birth AND during development stage ect.) and what patterns are there? How connected are these diseases to one another and what is the statistical analysis?

I answered this question through measuring centrality with BFS, closeness, and degrees. I decided to use centrality as a measure because each node is directly related to one another (disease-disease). By using centrality, I could discover the most connected nodes/diseases and how they affect each other. Below, I will explain how I broke down the modules in my code, the significant findings, and how it relates to my overall dataset.

ALGORITHMS:

In my project, I have 5 modules: BFS, Closeness_Original, Degree, Tests, and Readfile. Each connects to one another and will be explained thoroughly below.

DEGREE FILE:

In the degree module, I want to measure the number of connections a node has (number of connections one type of disease has to another). I wrote a collection of helper functions to support my main `calc_degrees()` function.

Neighbors(takes in point as u32) -> returns Vector of u32

This function prints and returns a list of all the neighbors of a given node. It goes through all of the `unique_nodes()` in the dataset and iterates through the x and y values. Using `dedup()`, I push all the individual neighbors of the nodes into a vector called `neighbors_list`. I called this function in my test function - I picked a random node in the dataset and it returned a list of its direct neighbors. Below is an example output of the neighbors for node 1 using my `graph_test.txt` dataset.

```
Running
[2, 3, 4]
```

Unique Nodes () -> returns Tuple of Vectors

This function finds all of the unique nodes in the dataset with no duplicates. It creates a vector in which the nodes and edges get pushed into after iterating through the dataset. This function is also where I call on the specific dataset I want my functions to run on. Calling on my read_file() function from the readfile module, I pass in which dataset I want to use. At the end, the nodes get sorted and returned with the vector of all the unique nodes and a tuple of all the (nodes, edges). Here is an example output using the graph_test.txt dataset.

```
Running `target/debug/stauss_final_project`
[1, 2, 3, 4], [(1, 2), (1, 3), (1, 4), (2, 3)]
```

Adjacency_Lists() -> returns Tuple of Vectors

This function prints all the nodes and its neighbors from the dataset. It does this by iterating through all the unique nodes and keeping track using a variable called used nodes. In the loop, it checks whether the node has been processed and if it has not, it gets its neighbors by calling on the neighbors function. Then, it gets marked as a used node and continues through the list. The degree vector is then printed with the corresponding node and their neighbors. This screenshot below is an example using the graph_test.txt dataset.

```
pub fn adjacency_lists() -> Vec<u32, Vec<u32>> {
    let nodes: Vec<u32> = unique_nodes().0;
    let mut used_nodes: Vec<u32> = Vec::new();
    let mut degree: Vec<u32, Vec<u32>> = Vec::new();
    let mut number_of_neighbors: Vec<u32> = Vec::new();
    // Iterates through unique nodes in graph & finds degree of each unique node
    for i: usize in 0..nodes.len(){
        if used_nodes.contains(&nodes[i]){
            break;
        }
        else{
            number_of_neighbors = neighbors(point: nodes[i]);
            degree.push((nodes[i], number_of_neighbors));
            used_nodes.push(nodes[i]);
        }
    }
    degree.sort();
    degree.dedup();
    // Sorts, removes duplicates, and prints all unique (nodes, neighbors)
    for (node: &u32, neighbors: &Vec<u32>) in &degree{
        println!("node {}: {:?}", node, neighbors);
    }
    return degree;
} fn adjacency_lists
```

BLEMS 20 OUTPUT TERMINAL

TERMINAL

```
|
= note: `#[warn(dead_code)]` on by default
warning: `stauss_final_project` (bin "stauss_final_project") generated 5 warnings (run `cargo fix --bin "stauss_final_project"` to ap
4 suggestions)
Finished dev [unoptimized + debuginfo] target(s) in 0.52s
Running `target/debug/stauss_final_project`
node 1: [2, 3, 4]
node 2: [1, 3]
node 3: [1, 2]
node 4: [1]
```

Calculate Degrees (shows top 10 connected nodes)

In this function, I created a vector that would hold all of the degrees for the corresponding node. I iterated through all the unique_nodes and set the number of neighbors equal to the list of neighbors. Then I pushed those nodes and neighbors into the degree vector. After I sorted and removed duplicates, I printed out the top 10 nodes and degree centrality. The output below is from my disease dataset. This found all of the local first degree connections - you can see that diseases 131, 165, 172 ect.. Is connected to at least 4 other diseases within the dataset. This makes them the most connected and at risk for 4 other diseases if you have disease 131. The higher the degree, the higher the risk of disease.

```
Top 10 Nodes and Degree Centrality:
degree (131, 4)
degree (165, 4)
degree (172, 4)
degree (211, 4)
degree (217, 4)
degree (247, 4)
degree (257, 4)
degree (323, 4)
degree (341, 4)
degree (486, 4)
```

CLOSENESS FILE:

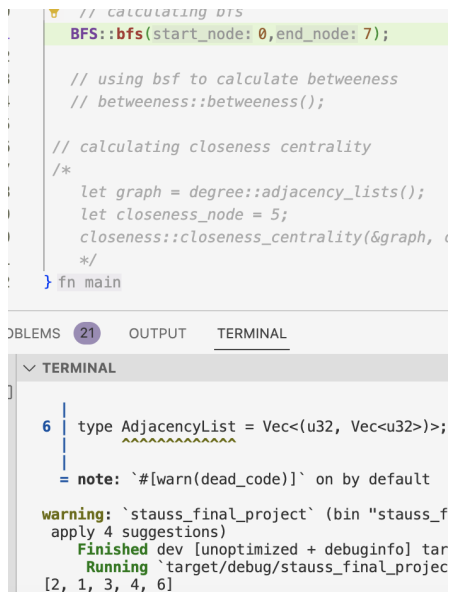
In the closeness module, there is a closeness Centrality() function that takes in the graph (in my case an adjacency list) and a node and returns a float (the centrality). I start by calculating all the distances from the given node to all other nodes in the whole graph and call on my BFS function to run it. Then, I calculate the sum distances and number of nodes (I subtract 1 in num_nodes) because I want to exclude the given node. Then to calculate the actual centrality, I divide the total number of nodes by the distances. Then, I push all of the nodes and their centrality measures into a node_centrality_vector that contains the (node, centrality). Lastly I sort and print out the top 5 nodes with the most centrality. Below is a screenshot of the output.

```
Top 5 Nodes and Closeness Centrality:
1. Node: 959 Closeness Centrality: 4.7075471698113205
2. Node: 549 Closeness Centrality: 1.3841886269070736
3. Node: 870 Closeness Centrality: 1.3841886269070736
4. Node: 118 Closeness Centrality: 0.852263023057216
5. Node: 821 Closeness Centrality: 0.7858267716535433
```

This shows me how close each node is to another in the network (globally) and how closely related each disease is to one another. In the output, you can see that disease 959 is most connected to all the other diseases in the dataset. The higher the centrality, the more connected the disease is in the whole graph.

BFS FILE:

In the BFS module, I made it return all the shortest paths from one node to another. It took in a start and end node to calculate what would lie in between. When I printed out the shortest path, I removed the start and end node so it only printed the path itself. I used a queue to keep track of the previous, visited, and current node I was on. I also called on the neighbors function to go iterate through all of the possible direct neighbors of that node. All of these shortest paths were pushed into a vector called `shortest_paths` and printed out. Here is an example of this function printing out the shortest path between the start node (0) and end node (7) with a small test dataset (`graph_test_two`).



```
// calculating bfs
BFS::bfs(start_node: 0, end_node: 7);

// using bsf to calculate betweenness
// betweenness::betweenness();

// calculating closeness centrality
/*
let graph = degree::adjacency_lists();
let closeness_node = 5;
closeness::closeness_centrality(&graph, closeness_node);
*/
} fn main
```

PROBLEMS 21 OUTPUT TERMINAL

TERMINAL

```
6 | type AdjacencyList = Vec<(u32, Vec<u32>)>;
   | ~~~~~
   | = note: `#[warn(dead_code)]` on by default
warning: `stauss_final_project` (bin "stauss_f
apply 4 suggestions)
Finished dev [unoptimized + debuginfo] tar
Running `target/debug/stauss_final_projec
[2, 1, 3, 4, 6]
```

MAIN FILE:

In the main module, I simply called the functions I needed to run in my files to make it all work. Because most of my print statements were in the functions individually, I just called the function. Lastly, I included my tests in a separate file.

TESTS:

In my tests module, I chose to write two tests of functions that return values so it was easy to test. I chose to test BFS and the neighbors function in my degree module. To test BFS, I chose two random points within the dataset (made sure they were far apart), and made sure the shortest path returned was the same when I called it in my main function. For the neighbors test, I chose a random point in the dataset and made sure it returned the exact same neighbors of that node in my test. Both run quickly and correctly. I would also like to mention that I used two “test” datasets with smaller vertices to run in my main function and test file to make sure it was correct before I used my large dataset.

```

▶ Run Tests | Debug
pub mod tests {
  use super::*;

  #[test] // test case 1: calculating bfs
  ▶ Run Test | Debug
  pub fn test_bfs() {
    assert_eq!(BFS::bfs(72,747), [360, 466, 623, 909, 859, 482, 351, 387, 710, 815, 869, 827, 279, 566, 131]);
  }

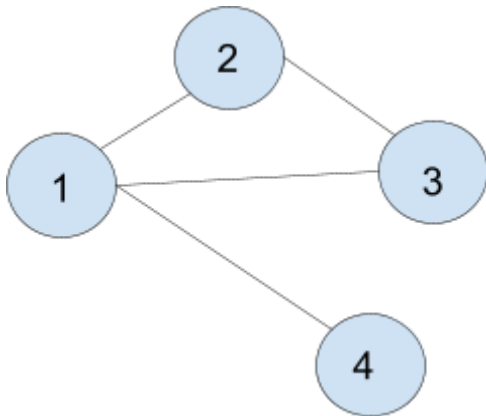
  #[test] // test case 2: calculating neighbors
  ▶ Run Test | Debug
  pub fn test_neighbors() {
    assert_eq!(degree::neighbors(131), [676, 566, 891, 747]);
  }
}

```

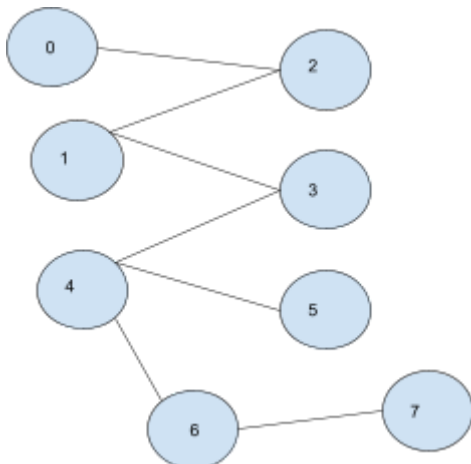
GRAPH TESTS:

Here are visuals of the two graph tests I used to test my code in the project.

Graph_Test.txt



Graph_Text_2.txt



TAKEAWAYS & CONCLUSION:

In conclusion, I learned how to measure centrality through BFS, calculating degrees, and calculating closeness. My dataset seemed significant as it resulted in the most connected diseases both locally and globally. If I had more time, I would like to try to calculate betweenness to continue more analysis on centrality. Overall, I learned a lot in this project and had effective results.