Data Used: twitter:https://snap.stanford.edu/data/ego-Twitter.html

Functions

Below is a description of each function, its purpose and its inputs and outputs.

The **read_file function** belonging to the mod read_in_graph, takes a filename as a string slice and returns a Graph. This function is used to read a file and convert it into a graph data structure. After all lines have been read and processed, a message is printed to show that the file has been successfully read into a graph. A Graph is returned, constructed with the number of nodes, named n and the vector of edges , named edges.

Adj_list, belonging to the open_adj_list mod, converts a list of edges in a graph into an adjacency list, each node in the graph has a list of nodes that is is connected to. The function takes two parameters: graph and num_nodes. graph is a reference to a vector of tuples, where each tuple represents an edge in the graph. Each tuple contains two usize values, representing the nodes that the edge connects. num_nodes is a usize that represents the total number of nodes in the graph.The function begins by creating graph_list, a mutable vector of vectors of usize. This will eventually be the adjacency list. The vec![vec![]; num_nodes] syntax creates a new vector with num_nodes empty vectors. Next, the function enters a loop over graph. For each edge in graph, represented by a tuple (v, w), it adds w to the adjacency list of v by pushing w onto graph_list[v].The commented out println! statement was likely used for debugging purposes. It would print a message when the function has finished creating the adjacency list. Finally, the function returns graph_list, the completed adjacency list.

The **clean_sample function** belonging to the open_adj_list mod, takes a vector of tuples, edges, as an argument. Each tuple represents an edge in a graph, with two usize values representing the nodes connected by the edge.

The function begins by creating a HashSet named unique_nodes that stores all unique nodes found in the edges vector. The function iterates over each edge in the edges vector, inserting both nodes of each edge into the unique_nodes set. (Since a HashSet only stores unique values, any duplicate nodes are automatically ignored.)It converts unique_nodes set into a vector called sorted_nodes and sorts it in ascending order. The function then creates a HashMap named node_map that associates each node with its index in the sorted_nodes vector. The function iterates over sorted_nodes and returns each item along with its index. It inserts each node and its index into node_map. Cleaned_edges to store the cleaned edges. It iterates over the edges vector again. For each edge, it retrieves the cleaned versions of its nodes from node_map (i.e., their indices in sorted_nodes) and pushes a tuple of these cleaned nodes into cleaned_edges. Cleaned_edges is returned, representing the same graph as the input edges, but with nodes renamed to their indices in sorted_nodes. This function was created to speed up the efficiency of running the program.

sample_edges, takes a reference to a vector of tuples, graph, and a sample_size as parameters. Each tuple in the vector represents an edge in a graph, defined by two usize values .It returns sampled_edges, which now contains a random sample of edges from the original graph.

The bfs module contains the functions compute_bfs, compute_avg_distance, and stats.

compute_bfs performs a Breadth-First Search (BFS) on a graph.It takes three parameters: start (the starting node for the BFS), graph (the graph on which to perform the BFS, represented as an adjacency list), and adj_list (the adjacency list of the graph). It computes bfs and returns the distance vector, which contains the shortest distance of each node from the start node (or None if a node is not reachable from the start node

The **stats** function in takes a slice of Option<u32> as input, which represents distances. It calculates and returns a tuple containing max_distance: The maximum distance found in the input slice, min_distance: The minimum distance found in the input slice, mean_distance: The average of all distances in the input slice, max_node: The index of the maximum distance in the input slice, min_node: The index of the minimum distance in the input slice, The function iterates over the input slice, updating the maximum and minimum distances and their respective indices as it encounters larger or smaller values. It also calculates the mean at the end.

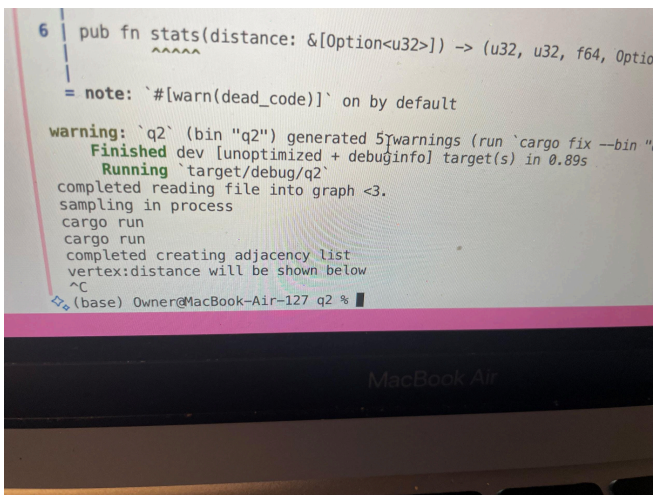
The **compute_avg_distance** function computes the average distance between nodes in a graph. It takes two arguments: adj_list containing the adjacency list of the graph, and graph. Using bfs it gets the distances. The function then iterates over each distance in the returned bfs vector and it adds the distance to total_distance and increments count by 1. It calculates the average distance by dividing total distance by count.

Main: The graph is read from a file named "twitter.txt" using the read_file function from the read_in_graph module. The edges of the graph are stored in the edges variable. A random sample of 50,000 edges is then taken from the graph using the sample_edges function from the random_sample module. This function randomly selects a specified number of edges from the graph. The reason for sampling is to reduce the size of the graph while still maintaining a representative subset of the original connections. The sampled edges are then cleaned using the clean_sample function. An adjacency list representation of the graph is then created using the adj_list function from the open_adj_list module. This function transforms the edge list into a vector of vectors, where each inner vector represents the nodes adjacent to a particular node.The bfs function is then applied to the graph using the compute_bfs function from the bfs module. This function computes the shortest distance from a specified start node (in this case, node 0) to all other nodes in the graph.

The stats function from the bfs module is then used to compute some statistics about the distances computed by the BFS algorithm. These statistics include the maximum and minimum distances, the nodes from which these distances are computed, and the mean distance.
The average distance in the graph is then computed using the compute_avg_distance function from the bfs module. This function computes the average of the shortest distances from each node to all other nodes in the graph. Finally, the computed statistics and the average distance are printed.

Process

Initially, the program would take a while to run, so a print statement was included in each function to show what stage it was at and to help me identify which functions were able to run.
As shown by these print statements in the earlier phase of my project. Many more print statements were included in my code but they were commented out during the process, and only the main print statements were included.

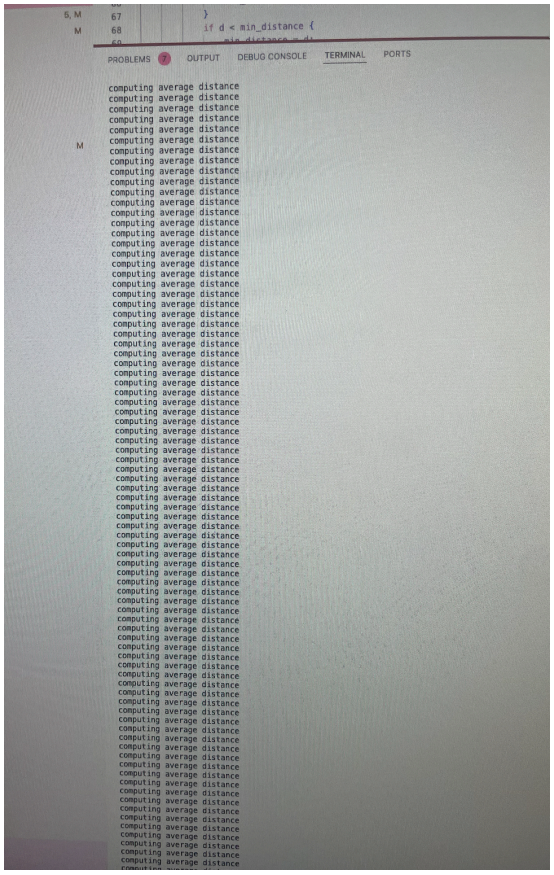


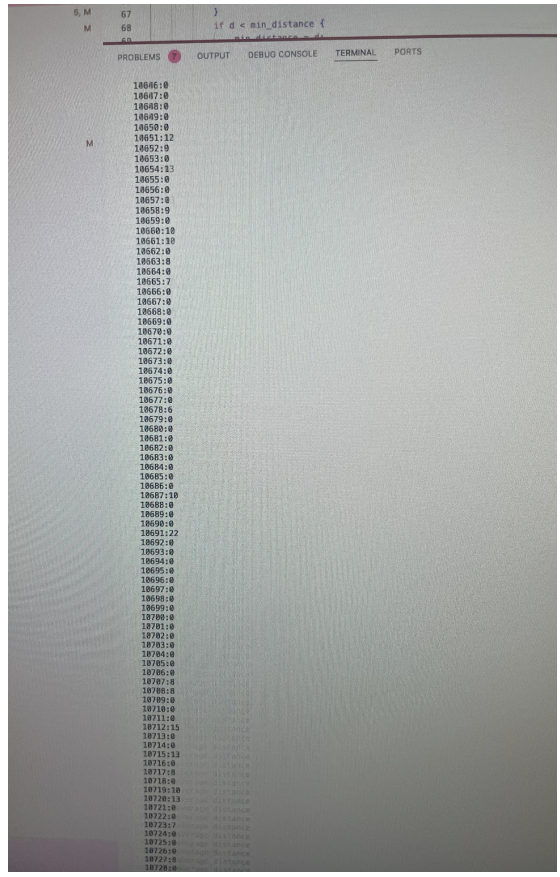
There are so many empty nodes displayed while the bfs program was printing, as depicted in the images below and it makes sense since we are taking a small sample from the data so there will be many broken connections. Below are also pictures of print statements to make sure the correct function works

```
5, M    67            }
   M    68            if d < min_distance {
        69                min_distance = d;
```

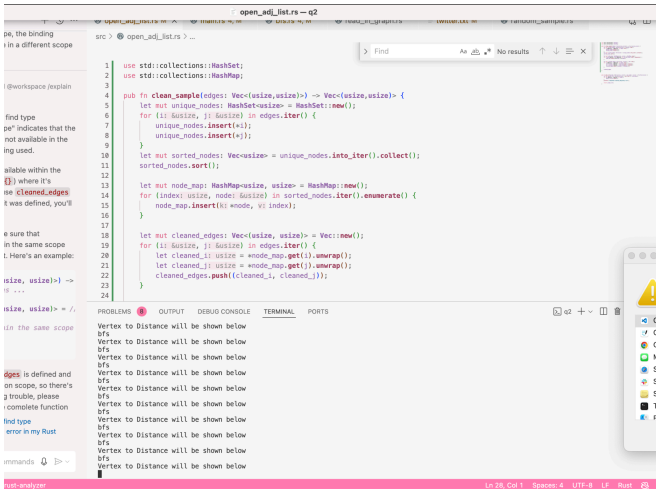
PROBLEMS 7 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
10046:0
10047:0
10048:0
10049:0
10050:0
10051:12
10052:0
10053:0
10054:13
10055:0
10056:0
10057:0
10058:0
10059:0
10060:10
10061:10
10062:0
10063:8
10064:0
10065:7
10066:0
10067:0
10068:0
10069:0
10070:0
10071:0
10072:0
10073:0
10074:0
10075:0
10076:0
10077:0
10078:6
10079:0
10080:0
10081:0
10082:0
10083:0
10084:0
10085:0
10086:0
10087:10
10088:0
10089:0
10090:0
10091:22
10092:0
10093:0
10094:0
10095:0
10096:0
10097:0
10098:0
10099:0
10100:0
10101:0
10102:0
10103:0
10104:0
10105:0
10106:0
10107:8
10108:8
10109:0
10110:0
10111:0
10112:15
10113:0
10114:0
10115:13
10116:0
10117:8
10118:0
10119:10
10120:13
10121:0
10122:0
10123:7
10124:0
10125:0
10126:0
10127:8
10128:0
```





Here is another print statement that was commented out



Here are more images from printing in different stages, all wich have been commented out

```rust
pub fn compute_avg_distance(adj_list: &Vec<Vec<usize>>, graph:&Graph) -> f64 {
    let mut total_distance: u32 = 0;
    let mut count: i32 = 0;
    for start: usize in 0..graph.n {
        let distance: Vec<Option<u32>> = compute_bfs(start, graph, adj_list);
        for dist: Option<u32> in distance {
            if let Some(d: u32) = dist {
                total_distance += d;
                count += 1;
                //println!("computing average distance");
            }
        }
    }
    if count > 0 {
        return total_distance as f64 / count as f64;
    } else {
        return 0.0;
    }
}


// creates analysis based off of bfs results, max, min, mean

pub fn stats(distance: &[Option<u32>]) -> (u32, u32, f64, Option<usize>, Option<usize>) {
```

PROBLEMS 8 OUTPUT DEBUG CONSOLE TERMINAL PORTS zsh

```
None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None,
one, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, N
ne, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, No
e, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, Nor
, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None
 None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None,
one, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, N
ne, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, No
e, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, Nor
, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None
 None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None, None, None]
Nodes statistics:
 As Analyzed From All Nodes
Maximum distance value is 23 and it is from this node: 13010
Minimum distance value is 0 and it is from this node: 0
Mean value: 11.088454586534116
(base) Owner@MacBook-Air-127 q2 % cargo run --release
   Compiling q2 v0.1.0 (/Users/Owner/Desktop/DS 210/FP2/q2)
```

Below is a print from when the sample size was 10 to see that sampling worked, because when I was in my debugging it would print the large amount being sample and it would cause my project to crash.

```rust
▶ Run | Debug
fn main() {
    let graph: Graph = read_in_graph::read_file(filename: "twitter.txt");
    let edges: &Vec<(usize, usize)> = &graph.edges;
    let list_edges: Vec<(usize, usize)> = random_sample::sample_edges(graph: edges, sample_size: 10);
    println!("{:?}", list_edges);
    let adjacent_edges: Vec<Vec<usize>> = open_adj_list::adj_list(graph: &list_edges, num_nodes: graph.n);

    let distances: Vec<Option<u32>> = bfs::compute_bfs(start: 0, &graph, adj_list: &adjacent_edges);

    let stats_results: (u32, u32, f64, Option<usize>, Option<usize>) = bfs::stats(distance: &distances);

    //let dist = &distances -> &Vec<Vec<usize>>;

    let avg_dist: f64 = bfs::compute_avg_distance(adj_list: &adjacent_edges, &graph);
    // print statements to visualize all the data

    //print statements for stats

    println!("Nodes statistics:");
    println!(" As Analyzed From All Nodes",);
```

```
PROBLEMS 9   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

warning: unused variable: `dist`
  --> src/bfs.rs:20:13
   |
20 |     for (v, dist) in distance.iter().enumerate() {
   |             ^^^^ help: if this is intentional, prefix it with an underscore: `_dist`

warning: unused variable: `avg_dist`
  --> src/main.rs:27:9
   |
27 |     let avg_dist = bfs::compute_avg_distance(&adjacent_edges, &graph);
   |         ^^^^^^^^ help: if this is intentional, prefix it with an underscore: `_avg_dist`

warning: `q2` (bin "q2") generated 7 warnings (run `cargo fix --bin "q2"` to apply 7 suggestions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.46s
     Running `target/debug/q2`
completed reading file into graph <3.
sampling in process
[(109635031, 107650489), (35659391, 30211023), (209424013, 156651075), (18923866, 248007541), (553139644, 325084625), (38264314, 321522130), (188
398607, 141021153), (79890469, 79332579), (175383671, 277401500), (46123952, 138003731)]
```

Ideally, I would make every part of the project code, but it wouldn't be efficient and would cause things to crash. The program now runs within two minutes and that's partially because there are fewer print statements being executed, only the most necessary information is printed.

Results and Analysis

Below is a screenshot of my results from running the program with cargo run release, I ran it a couple of times to see different outputs

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.05s
     Running `target/debug/q2 release`
completed reading file into graph <3.

sampling in process

list edges shown being made

adjacent edges in process

computing in process

stats finished calculating


Nodes Statistics:

As Analyzed From All Nodes
Maximum distance value is 5 and it is from this node: 1044
Minimum distance value is 0 and it is from this node: 0
Mean value: 2.9285714285714284
Average distance: 10.13033598608476
(base) Owner@MacBook-Air-127 q2 %
```

Ln 13, Col 3 Spaces: 4 UTF-8 LF Rust

```
   |              ^^^^^^ help: if this is intentional, prefix it with an underscore: _dist

warning: `q2` (bin "q2") generated 2 warnings (run `cargo fix --bin "q2"` to apply 2 suggestions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.05s
     Running `target/debug/q2 release`
completed reading file into graph <3.

sampling in process

list edges shown being made

adjacent edges in process

computing in process

stats finished calculating


Nodes Statistics:

As Analyzed From All Nodes
Maximum distance value is 25 and it is from this node: 8979
Minimum distance value is 0 and it is from this node: 0
Mean value: 12.92004048582996
Average distance: 10.076778072652669
```

Analysis

Each time the program runs a different sample from the data is made, so from the second picture, it is evident that the max distance value or “edge” is 25 from node 8979, and the minimum distance is 0 from node 0 . The Average distance from node to node is 10.07 , and the mean distance from node is 12.92.From the analysis, it's evident that the average distance between Twitter profiles or the average number of profiles between any two connected individuals is approximately 10.07.This finding suggests that within the Twitter network, individuals are typically separated by an average of around 10 profiles or connections. In other words, if there are about 10 profiles between two individuals, there's a high likelihood that they have a connection or some form of interaction on the platform. Considering the vast user base of Twitter, which spans millions of users, this average distance indicates a relatively close network where connections between users are common.

Tests

Below are the results of my three tests. There is a test for these functions: bfs, compute_avg_distance, and stats. As shown below all these tests passed successfully. The test_compute_bfs validates the Breadth-First Search (BFS) algorithm implemented in the compute_bfs function. It creates a graph structure represented by an adjacency list and then computes the BFS starting from node 0. The test checks whether the distances computed by BFS match the expected distances. The test_stats tests the stats function, which computes various statistical properties of distances in the graph. It calculates the maximum distance, minimum distance, mean distance, as well as the nodes with the maximum and minimum distances from a given node to all other nodes. The test validates that the computed statistics match the expected values. The test_compute_avg_distance tests the compute_avg_distance function, which calculates the average distance from each node to all other nodes in the graph. Similar to the other tests, it creates a graph structure and computes the average distance. Then it asserts that the computed average distance matches the expected value. As shown below all these tests passed successfully.

```
164 }
165
```

PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
   = note: `#[warn(unused_imports)]` on by default

warning: unused variable: `v`
  --> src/bfs.rs:20:10
   |
20 |     for (v, dist) in distance.iter().enumerate() {
   |          ^ help: if this is intentional, prefix it with an underscore: `_v`
   |
   = note: `#[warn(unused_variables)]` on by default

warning: unused variable: `dist`
  --> src/bfs.rs:20:13
   |
20 |     for (v, dist) in distance.iter().enumerate() {
   |             ^^^^ help: if this is intentional, prefix it with an underscore: `_dist`

warning: `q2` (bin "q2" test) generated 3 warnings (run `cargo fix --bin "q2" --tests` to apply 3 suggestions)
    Finished test [unoptimized + debuginfo] target(s) in 1.84s
     Running unittests src/main.rs (target/debug/deps/q2-ffddf46df82e16ff)

running 3 tests
test bfs::test_compute_avg_distance ... ok
test bfs::test_compute_bfs ... ok
test bfs::test_stats ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

(base) Owner@MacBook-Air-127 q2 %
```

CIES