Allison Smith

Prof. Alishah Chator

DS210

11 December 2024

DS210: Final Project Write Up

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

https://snap.stanford.edu/data/ego-Facebook.html

My goal for the project was to compare the Twitter and Facebook data sets and find which had a shorter average distance between each vertex pair. Initially, I had used a different data set which I had gotten from Kaggle but last minute after consulting with Sam (ta) he mentioned that the data set I had chosen would be difficult to use breadth-first search and shortest paths, meaning I had to restart my project. This also explains why my commits on github are inconsistent. I ended up editing and running my file on the csv file though the terminal. The new datasets presented the problems, What is the difference in the diameter of the network structures? What can this tell us about how sparse the networks are?

Beginning the code, I implemented "`use petgraph::graph::{DiGraph, NodeIndex};`", DiGraph is used to create directed graphs and NodeInex is used to identify the nodes in the graph. "`use petgraph::visit::Bfs;`" was implemented so I could use the Breadth-First Search algorithm which we covered in class. For my graphing algorithms I implemented "`use petgraph::algo::{connected_components, dijkstra};`" connected_components finds all connected components in an undirected graph and dijkstra implements the dijkstra algorithm which finds the shortest path from the starting node to all other nodes. I was more familiar with the HashMap, HashSet, and VecDeque. HashMap stores the key-value pairs which makes it easier when looking up items. HashSet stores unique elements where elements can not be repeated. VecDeque enables FIFO operations which I needed in order to use BFS. `(use std::collections::{HashMap, HashSet, VecDeque};)`. For handling the imported files I imported "`use std::fs::File;`" and "`use std::io::{self, BufRead};`". SliceRandom was imported for random selections, `use rand::seq::SliceRandom;` and "`use plotters::prelude::*;`" is for creating and plotting visualizations.

The function load_graph takes a string slice file_path as an input which directs it to the file being implemented and returns a directed graph where the node and edge are both represented by empty tuples. "`let mut graph = DiGraph::new();`" creates an empty directed graph called graph. "`let file = File::open(file_path).expect("Cannot open file");`" and “`let reader = io::BufReader::new(file);`” opens the file and send the message “Cannot open file” if the file cannot be opened and if the file can be opened it reads it. For reading the files, each line is split by white space and is parsed into an unsigned integer. “`.filter_map(|s| s.parse().ok())`” makes sure that only valid integers are kept and collected into a vector nodes. To make sure that each line has exactly two nodes I used to following lines to assign two numbers to node1 and node2, “`if nodes.len() == 2 { let node1 = nodes[0]; let node2 = nodes[1];`”. Then I created node1_idx and node2_idx which are indices of both the nodes created above and tries to find existing nodes by their index. If the node is not found then it adds the node to the graph. “`graph.add_edge(node1_idx, node2_idx, ());`” adds a directed edge from node1_idx to node2_idx in the graph. To conclude this function graph is returned, which returns a graph that contains the nodes and edges from the file.

The compute_distances function takes a reference graph, the directed graph created above which has no node or edge weights and returns a vector of unsigned integers which represents the distances between the nodes. First, I created an empty vector to store the distances. Then i made a for loop which iterates over each node in the graph, for each node a new BFS is started. Queue is defined as a VecDeque which is used to store the nodes in BFS order. Node_distances is defined as a HashMap which waps each node to its distance from the original starting node. The starting node is also added to the queue and its distance is set to zero, since its distance to itself is zero. To execute BFS “`while let Some(current) = queue.pop_front() {let current_distance = *node_distances.get(&current).unwrap();`” was written so the BFS process dequeues each node from the queue one at a time and the distance to get the current_distance from node_distances. The next for loop checks if each neighbor of the current node has already been visited. If the node had not been visited yet its distance is assigned as `current_distance + 1` since it is one away from the current node. The new node is then added to the queue. “`all_distances.extend(node_distances.values().cloned());`” makes it so after BFS is finished for the current starting node the function adds the distances that have already been computed to the all_distacnes vector, which we return after.

The function remove_high_degree_nodes takes graph, which is a mutable reference to the directed graph which means it can be edited in place and degree_threshold which is a usize value which represents the threshold for the size of each degree. The function does not return anything; it just modifies the input graph. Nodes_to_remove, first returns an iterator over all of the node indices in the graph, then returns the iterator over all of the edges connected to that specific node, because the graph is directed we get both the input and output edges. “`.filter(|&node| graph.edges(node).count() > degree_threshold)`” Filters the nodes to only keep the ones that are above the degree_threshold. “`.collect();`” then collects the nodes into a vector nodes_to_remove. The for loop “`for node in nodes_to_remove {graph.remove_node(node)}`” removes the node from the graph.

The graph_diameter function takes a reference to a graph and returns a value which represents the diameter of the graph, or the shortest path in the graph. I start by initializing max_distance to zero. The for loop iterates through the node_indices in graph. “let distances = dijkstra(graph, node, None, |_| 1);” uses Dijkstra’s algorithm to compute the shortest path from the current node to the attached nodes. “if let Some(&max) = distances.values().max() {max_distance = max_distance.max(max);}” finds the max distance form the current node to any other node while updating max_distance by adding the max distance found in each iteration, then returns this new max_distance.

Function extract_ego_network takes graph, center which uses the NodeIndex for the node that represents the center of the network, and depth which is a usize value that represents how far the network should expand from the center node. “let mut subgraph = DiGraph::new();” creates a new directed graph, “let mut queue = VecDeque::new();” performs BFS staring from the center node, and “let mut visited = HashSet::new();” makes a HashSet which keeps track of the nodes that have already been visited. “queue.push_back((center, 0)); visited.insert(center);” sets the center node depth to 0 to start BFS. The loop to perform BFS where the current node is popped from the front of the queue and its current_depth is checked to see if it is less than depth. If it is the function expires its neighbors and assigns the depth + 1 and marks it as visited. The function add a directed edge between the current node and the neighbor in the subgraph, then returns subgraph.

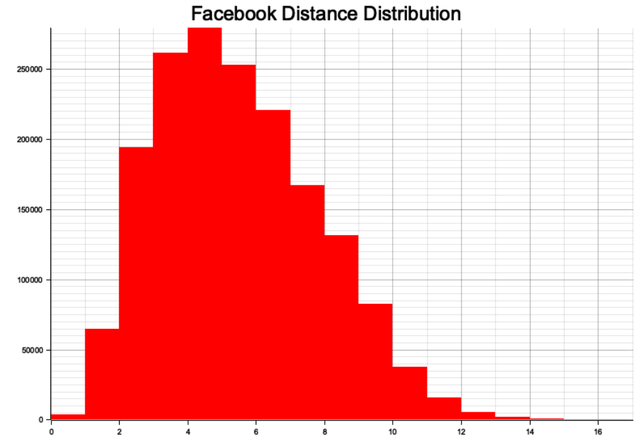
The random_subgraph function takes graph, and sample size which is a usize which specifies the number of nodes selected for the subgraph and returns a new directed subgraph that contains the randomly selected nodes and edges. The random number generator is initialized with “let mut rng = rand::thread_rng();” and the randomly selected nodes are chosen with the following code which iterates over the nodes, coverts the iterator of node indices into a vector, selects the random sample size and returns the referenced to the nodes. Fiter_map creates a subgraph which includes the select nodes and their edges where the first closure is applied to each node in the graph and the second closure is applied to each edge.

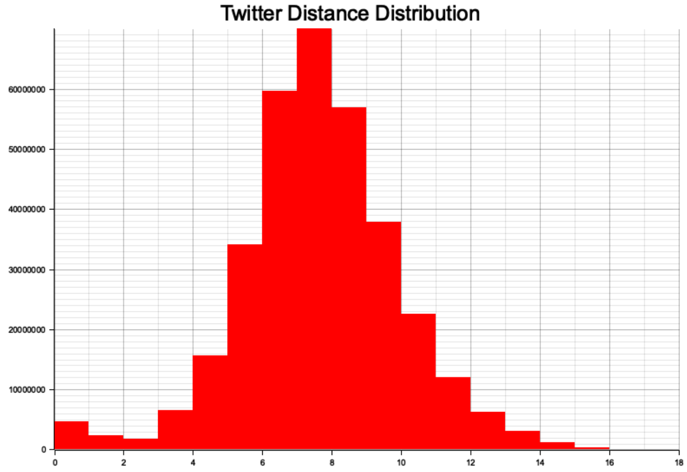
The plot_histogram function finds the min and max values of the dataset and assigns these values as to the x axis of the histogram. Histogram greates the histogram where for each value between the min and max values it counts how many times that value appears in the data array and computes the result which is a vector of tuples where each tuples contains a value that counts how many times it occurs. “let root = BitMapBackend::new(output_file, (800, 600)).into_drawing_area(); root.fill(&WHITE).unwrap();” creates a new BitMapBack where the background is white. “let max_count = histogram.iter().map(|&(_, count)| count).max().unwrap_or(1);” finds the maximum frequency of any bin in the histogram which determines the rang eof the y-axis. Then the plot is created and configured, the histogram bars are drawn all of which is where the font, size, labels, etc is determined. (Referenced chatgpt for solutions on how to make a good histogram in rust). “root.present().unwrap();

`println!("Histogram saved to {}", output_file);`” finalizes the histogram drawing and prints the message which says where the histogram image is saved.

Finally the main function processes the Twitter and Facebook graphs by first loading them, then removing high-degree nodes, computing the shortest path distances, calculating graph diameters, then plotting the histograms which show the node distances.

I tested the load graph function by creating a temporary file which has five nodes and four edges and checks if all of the nodes are present in the graph. The second test I performed was the test the compute distances function which first created a temporary file which loaded the graph and computed the distances between nodes using compute_distances, tests that the distances vector is not empty and makes sure that the max distance in the distances vector is less than or equal to four.





The results from analyzing the twitter and facebook graphs reveal the differences in the structures of the networks. The diameter of the Twitter graph is 18, while the diameter of the

Facebook graph is 17, indicating that Twitter’s longest shortest path is slightly longer than Facebook’s. This suggests that Twitter’s graph may be more spread out and less tightly interconnected compared to Facebook, where the smaller diameter hints at a more densely connected network.

The histograms further emphasize this difference. If the Twitter graph's histogram show a wider spread of distances, with a greater proportion of long paths, it reinforces the notion of a more fragmented network with isolated regions. In contrast, the Facebook graph's histogram reflects more short-range connections, supporting the idea that Facebook's network is more interconnected. These differences may arise from the nature of the platforms themselves, where Twitter, being a more global and open network, has a higher likelihood of isolated nodes or distant connections, whereas Facebook's social structure, with its focus on mutual connections and friend networks, which tends to create a more clustered and compact network.