Date: 12/15/24

graph_loader.rs

Imports:

- Beginning this project started with the building of the graph_loader.rs file which is used to take the dataset of frequently co-purchased product directed edges and transforming this into a directed graph or DiGraph in Rust using the petgraph crate. Which is the first line of code, which takes the import of DiGraph from the petgraph crate. Next is taking the 'File' import from Rust's standard library to handle files within the code. Next is Rust's standard library import of BufRead and BufReader, which are crucial for reading files with efficiency. After this is the import ParseIntError which helps with parsing errors specifically in cases when converting strings to integers. Finally, I implemented the 'fmt' or format for the 'fmt::Display' trait in cases for customer error formatting, which will come in handy during the MyError implementation next.

pub enum MyError:

- Starting off the custom error implementation, the first step, as with nearly all initializations of enums and structs is the '#derive(Debug]) trait. Next we initialize the public enum MyError which defines a custom error type called MyError which in our case handles multiple kinds of errors that might occur in running our program. We also make this enum public so that we can access it later outside of the module in which it is defined. The first variant and our first field is the IoError, which represents input/output errors, which in our case is helpful for failing to open the file, read from it, or write to a file. This variant wraps the standard 'io::Error' type from the 'std::io' module. The next variant and our second field is the ParseError, which is helpful in cases where errors form while parsing strings to integers. This variant wraps the standard 'ParseIntError' type from the 'std::num' module. Our third variant and third field is the 'FormatError' which is helpful for custom errors related to invalid formats, which is typical when encountering an unexpected line structure in a file. This variant holds a 'String' to describe the error in more detail.

impl for MyError:

- 'impl std::error::Error for MyError {},' which is our first iteration, for which this case handles implementing the std::error::Error trait for the MyError enum. By doing this, we can allow for MyError to be treated as a standard error type in Rust. This also allows for the Error trait to provide utilities like the '?' operator which is helpful for functions that have Result return types and stops the function if an error is returned. The next part 'impl From<io::Error> for MyError' implements the 'From' trait for MyError to handle conversions from 'io::Error.' This allows for future implementations to not have to explicitly convert errors to MyErrors. The second part 'fn from(error: io::Error) → Self { MyError::IoError(error)}' makes sure that when an io::Error needs to be converted into a MyError, this function wraps the IoError variant into a MyError. After this, the same is done for the ParseIntError.

pub fn load_graph:

- We make this function public so that we can call this later in main.rs. In the function we take a string slice 'file_path' as an argument which specifies the path to the file containing the graph data. If the function is successful, it will return a DiGraph of type u32 – representing the node

weights, and edges – that have no weight. But because the return is a Result, if the function is unsuccessful, it will return one of the three custom type errors mentioned above. Next, I initialized a new directed graph – 'graph.' After this I initialize a HashMap 'node_map,' which is created to map node weights to their inter indices in the directed graph. Following this, we create 'file' which tries to open the file specified by 'file_path,' which if successful will return a Result File, specified by 'Ok(File),' which will take the 'Ok(File)' and unwrap it, assigning it to file. If not it will return 'Error(io::Error)' – meaning that the file cannot be opened. In the case where a 'io::Error' is returned from Result, we use '.map_err(My_Error::IoError)' to convert the 'IoError' to a 'My_Error::IoError' and propagated to the calling function using our '?' operator. After this we define our reader – 'reader', which wraps the File in 'BufReader' to enable our efficient reading. Here 'BufReader' reads the file in chunks rather than a typical reader which reads line by line, which significantly reduces our I/O operations and improves performance in larger files, which is crucial in our case indicated by the size of the dataset. Moving forward we set up a for loop which calls our 'BufReader' defined earlier for more efficient reading, and the .lines() method tied to this call, which reads the file line by line, and returns an iterator over 'Result<String, io::Error>.' It should be noted that here, although this for loop is reading line by line, due to the BufReader, internally we are still reading the lines chunk by chunk that is processed into a buffered data, which is used to extract lines and return a 'Result<String, io::Error>.' This with the next line under the for loop 'let line = line.map(MyError::IoError)?' means that either a successful line is read, in which we return 'Ok(String)' or an unsuccessful line is read in which we return 'Err(io::Error).' If a line is read successfully, the 'Ok(String' is unwrapped and stored in 'line.' The next part is specific to the dataset I chose, in which lines that begin with the "#" symbol are irrelevant to the actual data. Therefore, we set up a if statement that if line – using .start_with() method here – starts with the '#' symbol we use the 'continue' statement so that we just skip to the next iteration of the loop, ignoring the rest of the logic. Next, we define a vector of referenced string slices with 'let nodes: Vec<&str>,' we do this instead with substrings so that we are only referencing portions of the original 'line' without allocating any new memory. We then use 'line.split(\t).collect();,' which splits the current line into substrings using the tab character, as for each line in the file is represented by two nodes which are now separated tab, and we call the .collect() method here to collect the result of our iterator produced by split into a collection which here is a vector of string slices. After this, we check the 'nodes.len() != 2' with an if statement, which is implying that if the node's length, aka two separated values after split, are not equal then we must return an error. The error is FormatError, in which we call MyError to produce a custom error, and use the format! macro to print out our message that the line format is invalid, followed by the line itself. Next, we define 'from_node' which attempts to convert all of the source node's substrings into u32 unsigned 32 bit integers. Then we define 'to_node' which does the same thing as 'from_node' but for all the destination nodes. If the parsing is unsuccessful, we use '.map_err(MyError::ParseError)?' to convert any ParseIntErrors into MyErrors. Next, we define 'from_index' which using 'node_map.entry(from_node)' checks if the our source node already exists within the node_map, if it does, this returns a reference to the corresponding graph index. However, if the source node is not already within our graph, it uses the 'or_insert_with' method with the 'graph.add_node(from_node)' to proceed past the first statement and add the source node to the graph, while returning its internal index. We do the same for our 'to_index' so that we handle

both the source and destination nodes. Lastly, we call our graph and '.add_edge(from_index, to_index, ())' to add our nodes to the edge and then add the edge to the graph. The edge weight is () or unit type, which is indicating that the edges have no weight. When the for loop is complete, and all the lines have been processed meaning the graph has been built, we use 'Ok(graph)' to return the graph.

graph_loader.rs

Imports:

- The first part of my graph analysis begins with adding in the proper imports. The first is fairly obvious which is the import of 'DiGraph' or a directional graph which is necessary to analyze the directed edges and nodes in our graph. The second is 'petgraph::visit::Bfs' which is useful for the first part of our analysis which uses a Breadth-first-search algorithm to perform a traversal on the graph. This explores the nodes layer by layer, starting from our given root node and visiting all of its neighbors, then visiting all of its neighbors and so on. The next is 'petgraph::Direction' which is an enum provided by petgraph to specify the direction to which the directional edges are pointing in a directed graph. Calls such as 'Direction::Outgoing' and 'Direction::Incoming' indicate that an analysis of the outgoing and incoming edges from a node respectively. Next is a HashMap, which we are using in this case to store mappings such as node identifiers to graph indices. Next is 'File' which was used in our last file, and has the same purpose here of opening, reading, and writing files. After this we then use 'std::io::{Write, BufWriter)' which is a trait to provide functionality to write data to a stream (in our case a file). Then 'BufWriter' is a wrapper that buffers writes to improve our I/O performance, which instead of writing small chunks directly to a file, it accumulates data in memory and writes it in much larger chunks.

pub fn degree_analysis:

- Moving forward, we define this function degree analysis to perform a degree analysis on the direction graph, specifically here to calculate the average in-degree and out-degree for each node. Of course we start by making the function public, so that we can later call this in main. We take the graph, which is a reference directional graph of unsigned 32 bit integers – for node weights, and () – for edge weights (given they don't have weights). We return a tuple of type f64,f64, which will be our average in-degree for nodes, and average out-degree for nodes in the graph. Then we define 'in_degrees' and 'out_degrees' as mutable vectors using our vec! macro, in which in_degrees will store the number of incoming edges for each node, and out_degrees storing the number of outgoing edges for each node. Then we define a for loop over each node in the graph, where '.node_indices()' returns an iterator over all nodes in the graph represented by their internal indices. Within this for loop we push our in_degrees to 'graph.neighbors_directed(node, Direction::Incoming).count())' in which 'neighbors_directed' is a method that returns an iterator over the neighbors of a node, and filters by the specified direction. Then we directly call Direction::Incoming to assist in this process, followed by a .count() to determine the total number of edges in which the direction is incoming. We do the same for out_degrees as well and finish our for loop. After this, we define 'avg_in_degree' and 'avg_out_degree' which calls our average function which is defined after this, and takes a reference to the vectors in which we stored the total number of in and out degrees in our for loop. Finally, we return our f64,f64 tuple back to the to function.

fn average:

- This function which was called in our previous function handles dealing with the averages of out in and out degrees calculation. We take the parameter ‘degrees :&[usize])’ which is a slice that we borrow from our previous function. We return a f64, which is important because this was the return value of our last function. Within our function, we firstly check if the degree ‘is_empty,’ if so, we return 0.0 to avoid any division by zero later causing errors. If the degree isn’t empty then we proceed, by using ‘degrees.iter().copied().sum::<usize>()’ – which creates an iterator over the references to the elements of the slice, and converts these references &usize into usize, which is important because the sum method only operates on owned values, not references. But we use .copied() here to avoid any ownership errors. Then we convert these usizes into a f64 to perform division, and call degrees.len() to get the length of the slice, which we also convert into an f64 for proper division. Then as both parts join together we get the sum of the slices as f64s and the lengths of the slices as an f64 and divide them. This will give us the average in and out degree averages, which we then return back as an f64 to be called in our previous function.

pub fn compute_shortest_paths_bfs:

- This function is used to compute the shortest paths from a given starting node to all other nodes in an unweighted graph, using Breadth-First-Search (BFS). The function takes in the referenced directed graph with node weights u32 and edges of no weight as a parameter, and a start or starting node as u32. The return type is a HashMap of <u32,u32>, where keys are node labels, and values are the shortest distances from the start node. We begin by finding the starting node in our graph. Which we do by letting ‘start_index’ be equal to ‘graph.node_indices()’ which returns an iterator over the internal indices of all the nodes in the graph. Paired with ‘.find(|&i| graph[i] == start’ which searches for the node whose weight at graph[i] matches the start node’s label. We also add a ‘.expect(“Start node not found in graph”)’ at the end of this search in cases where if the find operation fails, the program panics with our message “Start node not found in graph.” Next we initialize our BFS algorithm by creating a new BFS labeled ‘bfs’ which has the referenced graph and our starting node. Then we create ‘distances’ which is a new empty HashMap to which we will store distances from the start node. Then we insert our start nodes into our ‘distances’ map with the distance of 0, given the distance from itself is zero. Moving on, we create a while let loop which iterates over each node encountered in the BFS traversal with ‘while let Some(node)’ and returns the next node in the BFS traversal with ‘bfs.next(&graph).’ Then within the loop we initialize “current_distance’ which accesses the current distance of the node ‘graph[node]’ from the start node using our ‘distances’ map. Also within our while loop we define a for loop which for each neighbor in our graph, we check if the neighbor has already been visited which we do with ‘if distances.cointains_key(&graph[neighbor]),’ aka if the distance is already recorded in our ‘distances’ map. If the neighbor hasn’t been visited given the ‘!’ at the beginning of our if statement, we insert the neighbor’s label ‘graph[neighbor]’ into the ‘distances’ map, and assign it a distance equal to the current distance + 1, which guarantees this is the shortest path (BFS). At the end of this loop, we return our ‘distances’ map to our function.

pub fn display_shortest_paths:

- Moving forward, we define our display shortest paths, which is used to call upon our previous function ‘compute_shortest_paths_bfs’ and display a subset of the results during compilation; however, all the results are written to a file. We take a reference to the directed graph, and as before nodes have weights of type u32, whereas edges have no weights. We also take the starting

node ‘start’ which is type u32. Then we take ‘max_display’ – which is the maximum number of shortest paths to display in the console, and ‘output_file’ – the name of the file to write the full list of shortest paths. The return type of this function is ‘Result’ which will return ‘Ok(()) if everything succeeds and Err(std::io::Error) if not. The first step is defining ‘shortest_paths’ to the call of the previous function, where it computes the shortest path to all nodes from our starting node. Then we add a print header, which formats the values left-aligned with a width of 10 characters, and prints the title of the two columns “Node” and “Cost”, while adding a line of separation between the titles and the data. Next we define ‘file’ which creates a new file ‘output_file,’ if possible, if not an IoError will be propagated with the addition of the question mark. After this we create ‘writer’ which wraps our new ‘file’ into a BufWriter to buffer write operations. Then we begin by writing the file that will store all of our shortest paths from our bfs, which we start by using the macro ‘writeln!’ that writes the header “Node,Cost,” which is also handled by the error ‘?’ if an error occurred. Next we define a for loop that for each node and cost in our shortest_paths we write each node and its shortest path cost to the file in a CSV file format. After this we print the line “Full shortest paths written to {}” and our output file. When this is finished if no errors have occurred we return ‘Ok(()).

pub fn clustering_coefficient:

- This function calculates the clustering coefficient for a specific node in an undirected graph, specifically measuring how interconnected the neighbors of a node are, which further provides insight into the local connectivity of the graph. We begin by setting up our function which takes a reference to our directed graph, where node weights are u32, and edges are unweighted, but also the node we are concerned with of type u32. The return for this function is a f64. Starting off this function we create ‘node_index’ which is defined to find our specified node, and if it cannot find this node it has a ‘.expect’ where it prints that the node was not found in our graph. Next, we define ‘neighbors’ which is a vector. In this case when we use ‘graph.neighbors_undirected(node_index),’ although the graph is directed, we are returning an iterator over the neighbors of a node and treating the graph as undirected. This still works because this method ensures that neighbors whether in or out are collected in both directions. After this we define ‘neighbors_count’ which calculates the number of neighbors for that node using ‘.len().’ Then we use an if statement to see if the node has fewer than 2 neighbors, in which case there can be no connections between its neighbors. In such cases, the clustering coefficient is 0, and so that is what we return ending the process early. If not, we define ‘connected_neighbors,’ which is initially set to zero. Then we define a nested for loop in which the outer loop ‘i’ iterates over each neighbor, and ‘j’ iterates over the neighbors that come after i, in order to avoid duplicate pair comparisons. Then within this nested for loop we use ‘graph.find_edge_undirected(neighbors[i], neighbors[j]).is_some()’ to check if there is an edge in either direction between neighbors[i] and neighbors[j]. This will return ‘Some(edge)’ if an edge exists between neighbors, and None if no edge exists before the .is_some() method. Which will take this return and return true if an edge exists. After this we increment the count of connected_neighbors pairs for each edge found between neighbors. Then closing the nested for loop we define ‘total_possible_connections’ which for ‘n’ neighbors, the total number of connections between them is equal to ‘n * (n-1) /2,’ in which this formula counts the number of unique pairs of neighbors. Then we take ‘connected_neighbors’ as a f64 and divide this by our total_possible_connection as an f64 to

compute our clustering coefficient. The result is a value between 0.0-1.0 where 0 represents no connection between neighbors, and 1.0 represents that all neighbors are fully connected.

pub fn clustering_coefficient_summary:

- This function is for calculating the clustering coefficients for all nodes in a graph, and computing summary statistics for average and maximum coefficients, which is used in our main.rs file, and then writing the detailed results into a file. We begin by defining our function which takes our directed graph as a reference, with nodes weights as u32, and edges as unweighted. But also of output_file as a string slice. The return type is a Result where if the function returns 'Ok(())' it is successful, and if it returns 'Err(std::io::Error)' it is unsuccessful. Within our function we set coefficients to a mutable empty vector to later store the clustering coefficients for all nodes in our graph. Then we set up a for loop that iterates through each node in the graph by its internal index. Then inside this loop we define 'coeff' which calls our previous function to compute the clustering coefficient for our current node, and retrieves the label of the current node using its index. We then store the clustering coefficients to our coefficients vector. After this we define 'avg_coeff' which creates an iterator over the references to the clustering coefficients in the vector, sums the coefficients, returns the number of nodes in the graph, and divides the total number number of nodes in the graph to compute the average. Then we define 'max_coeff' which creates an iterator over the values of the coefficients (not references) but clones them, and uses '.fold(0.0 / 0.0, f64::max)' to find the maximum coefficient, with the '0.0 / 0.0' ensuring proper handling of edges cases where there could be an empty list. Next we write a couple of print lines, the first being our clustering coefficients summary, followed by the average coefficient with a call to 'avg_coeff' and maximum coefficient with a call to our 'max_coeff.' We limit our f64 to round to 4 decimal places to make the result look a lot cleaner. Following this, we create a new output file using 'let file = File::create(output_file)?' and use the '?' operator to handle ioErrors that may form. Then we define 'writer' which is used to wrap the file in a 'BufWriter' for efficient writing. Then we use the 'write!' macro to write a header row to the file in a CSV format which reads "Node,ClusteringCoefficient." After this define a for loop that uses the .zip method to combine the node indices with their corresponding clustering coefficients. Then within our for loop we use a 'writeln!' macro to write the nodes label 'graph[noe_index] and its clustering coefficient 'coeff' as a CSV row. Lastly, we print a line that says "Full clustering coefficients written to {}" and our output file. Then we return 'Ok(()) assuming we haven't run into any errors during the function.

Tests – tests.rs

Imports:

- The first part of my tests file is importing the other modules into tests, so that I can use them. The first is 'acm_210_finalproject::graph_loader{load_graph_MyError} which is a custom create module developed in the test folder. The next is "acm_210_finalproject::graph_analysis {degree_analysis, clustering_coefficient}" and the last is 'petgraph::graph::DiGraph.' The two custom crate modules are for analyzing the function written earlier, but because I'm working within a separate file, I needed to specify where I was pulling the function from, which is why including my project name beforehand was necessary.

fn test_load_valid_graph:

- This test is for validating the functionality of load_graph, where you would be loading in a valid graph. The first part is defining a dataset, which is done with a couple of strings. "0\t1" indicates

an edge from node 0 to node 1, “1\t2" indicates an edge from node 1 to node 2, “2\t0” indicates an edge from node 2 to node 0. After this we specify the file_path as “test_dataset.txt.” Then using the write file import, we write our new dataset into the file named “test_dataset.txt.” We also add a .expect() here in case that if it fails, the test will panic with the message that the error occurred in writing the test dataset, and not with the actual code. Then, we define ‘graph’ which calls the ‘load_graph’ function to load the graph dataset in from our test dataset, and we add another .expect() here in the case that the load_graph function failed to load in the graph. Then if the load_graph function works as expected, we use ‘assert_eq!’ here to compare the expected versus real values. We call graph.node_count() to get the node count for the graph, in which case expect it to have exactly three, if it doesn’t the test fails and writes “Expected 3 nodes in the graph.” This would indicate that load_graph is not working properly. We do the same for edge.count(), and if both pass the test passes, and then we use the file import to remove the file, as to not take up space and memory, and we add another .expect() in case the file failed to be deleted.

fn test_load_graph_empty_file:

- This test is testing that the load_graph function behaves correctly when given an empty file as input. First we create an empty dataset we call “empty.data.set.txt.” Then we use the file import to write an empty string to the file. After this we define ‘result’ which calls the load_graph function to load the graph from our empty dataset file. ‘Result’ here stores the result of the load_graph function, which is either ‘Ok(graph)’ or ‘Err(error).’ Next we use ‘assert!’ to check if the result is an ‘Ok’ variant – indicating that the graph was successfully loaded, or if the result is an ‘Err’ meaning that the graph was not loading in properly. Then we define ‘graph’ which unwraps the result – extracting the graph from the ‘Ok’ variant, if the result is an Err – the test will panic. Next, we use ‘assert_eq!’ to compare our expected and actual results from the load_graph result, if the graph contains 0 nodes and 0 edges then the load_graph function worked as expected. If not, the assert_eq! will print that we expected there to be 0 edges/nodes but somehow there are some. Lastly, we remove the file, which throws an error if not done properly.

fn test_load_graph_malformed_file:

- This function tests if the load_graph function behaves properly when dealing with malformed file inputs. First then we create our dataset with “0\t1\ninvalid_line\n2\t3” – where “invalid_line” is a malformed line that does not follow the expected format. Next we define the file path and give it the name of “malformed_dataset.txt.” Then we write the malformed data string to our newly created file. After this we define ‘result’ which calls our ‘load_graph’ function, and the result is stored in ‘result’ as either a ‘Ok(graph’ – indicating failure, or a ‘Result::Err’ – indicating that load_graph is working properly. Then we use ‘assert!’ to check if the value matches a specific pattern, in which case we are expecting a result error from trying to load in the malformed dataset. If the result is an error, the test passes, if not the test fails. Lastly, we remove the file.

fn test_load_large_graph:

- The test tests to validate that the load_graph function can handle a large graph dataset correctly. The first thing we do is define our dataset, by creating a for loop that iterates through the range 0-999. In which in the for loop we append a line to the dataset representing a directed edge from node i to node i + 1. After this we define our file_path which is called “large_dataset.txt.” Then we write the newly created dataset to this new file. After this we define ‘graph’ which calls the load_graph function, to load in this dataset. Then we use ‘assert_eq!’ to validate that our expected values of 1001 nodes and 1000 edges match our actual values that are in our graph. If not the test

will fail and return the message that the nodes/edges are expected to be this number. Lastly, we remove the file so as to not take up memory and space.

fn test_degree_analysis_small_graph:

- Firstly, we are testing to see if the in-degrees and out-degrees of all nodes in a small directed graph are computed correctly. The first thing we do is set up a new directed graph of nodes weight u32, and edges unweighted. The next step is to give nodes a,b,c labels 1,2,3 respectively. After this we add three edges, a→b, b→c, c→a. The graph forms a cycle. Then we define 'in_degrees' and 'out_degrees' as vectors using the 'vec!' macro. After this we iterate over all the nodes in the graph, and count the number of incoming edges to determine the in-degree and out-degree of the node. If the direction of the node is Incoming it gets pushed to the 'in_degree' vector, and if it is outgoing it gets pushed to the 'out_degree' vector. Then we use 'assert_eq!' to compare our expected value with our actual value. Here we are expecting that each node regardless of in or out degree should have 1, and if this isn't the case the test fails. Lastly we remove the file.

fn test_graph_analysis_degress:

- This test validates the correctness of the degree_analysis function by analyzing the average_in_degree and average_out_degreee of nodes in a directed graph. The first thing we do is create a directed graph with node weights of type u32 and edges that are unweighted. The next thing we do is create node a – labeled 1, node b – labeled 2, and node c – labeled 2. After this we create edges from a to b, b to c, and c to a. This forms a cycle. After this we call our 'degree_analysis' function which is used to calculate all of the out-degrees and in-degrees for all nodes in a graph, and calculate an average for both. We expect here that the average should be 1, given all of the nodes should have an in and out degree of 1. So we call 'assert_eq!' to check, if this is not the case then the test fails. Lastly, we remove the file so as to not take up memory or space.

fn test_clustering_coefficient_no_neighbors:

- This test is testing that the clustering coefficient for a node with no neighbors in a graph is correctly computed as 0.0. The first thing we do is define our directional graph, same as all other examples. The next step is to define 'node' which is the only node in our graph and add it. Then we define 'coeff' which calls clustering_coefficient on our graph and our singular node. When computing the clustering coefficient we should expect it to be 0.0, so we call 'assert_eq!' and if this is not the return value the test fails. If not, the test passes, and we remove the file.

fn test_clustering_coefficient_complete_neighbors:

- This test is testing to see that the clustering coefficient for a node whose neighbors are fully connected is computed as 1.0. First thing we do is define our directed graph, same as all other examples. Then we create nodes a, labeled 1, b, labeled 2, and c, labeled 3. Then we create a cycle by having our first edge from a to b, second from b to c, and third from c to a. Then we define 'coeff' which calls our clustering coefficient function, in which if all nodes are fully connected should return the value 1.0. We check this with 'assert_eq!,' if this is not the case the test fails. But if it is, the test passes and we remove the file from memory.

fn test_clustering_coefficient_partial_neighbors:

- This test is testing to see that for a node whose neighbors are not connected is computed as 0.0. The first thing we do is define our directed graph, same as for all previous tests. Then we define our nodes a, labeled 1, b, labeled 2, and c, labeled 3. After this we create our edges, which for the

first we have a to b, and the second a to c. Note that there are only edges between ‘a’ and the other nodes, meaning that b and c are not connected. After this we define ‘coeff’ which calls our clustering coefficient function on our node a, where we are checking how interconnected our neighbors are. For node ‘a’ its neighbors aren’t interconnected so we should expect a return value of 0.0. So we use ‘assert_eq!’ to check this, and if this is true, the test passes, but if any other value is returned the test fails. Lastly, we remove the file so as to not take up space or memory.

lib.rs

- The file is used for library crates, which contain reusable code that can be imported by other projects or the crate’s main.rs. In my case I used this file to define “pub mod graph_loader” and “pub mod graph_analysis.” This was because it makes the modules public which allowed me to external use them in my tests.rs file which was separate from my main src file.

main.rs

Imports:

- The first imports are our modules ‘graph_loader’ and ‘graph_analysis’ which we call in the main function to run alongside our other code. Then we call the individual functions we want to use from each module, such as ‘load_graph’ – to load the graph, ‘degree_analysis’ – for analyzing in and out degrees for all nodes, ‘display_shortest_paths’ – to display a preview of the shortest paths and write the entirety in a separate file, and ‘clustering coefficient_summary’ – to give the average coefficient, the maximum coefficient, and write the entirety into a separate file.

Main:

- For our main function we have no parameters, but the return type is of Result as to handle any type of error that might occur when trying to call the other functions. The first part of our main is defining our file_path which is “amazon0302.txt” which is the dataset I chose, in which it is based on the feature that amazon holds “Customers Who Bought This Item Also Bought.” Next we attempt to read the dataset from our dataset in our load_graph function, using ‘match’ we can handle return types of Result ‘Ok(())’ or ‘Err.’ If the graph is loaded successfully we print that the graph has been loaded successfully followed by the number of nodes and edges that are in the graph. After this we call our degree_analysis, which returns a tuple of f64,f64, and then we proceed to print these values with the up to the fourth decimal place as to keep these numbers clean. Then we print “Preview of the shortest paths:” which gives a preview to the computing of the shortest paths from our starting node which we define as 0. We only print out the first 15 nodes and their costs. After this is done the entirety of the shortest paths from starting node 0 is written to a file named “shortest_paths.csv.” Next we call our clustering coefficient summary which computes the clustering coefficient for all nodes in the graph, and writes the file to “clustering_coefficient.csv.” Lastly we print out an “Error loading graph” message if we encounter an error in the beginning of main, and return Ok(()) if the program runs successfully.

Output:

- Graph loaded successfully!
- Number of nodes 262111
- Number of edges 1234877
- Average In-Degree: 4.7113
- Average Out-Degree: 4.7113
- Preview of shortest paths:

- Node | Cost
- ----------
- 107315 | 22
- 244748 | 28
- 69226 | 15
- 108659 | 17
- 71394 | 20
- 220943 | 24
- 12265 | 11
- 175535 | 23
- 193772 | 21
- 29506 | 16
- 166957 | 22
- 134185 | 21
- 119799 | 21
- 131486 | 21
- 133276 | 21
- Full shortest paths written to shortest_paths.csv
- Clustering Coefficients Summary:
- Average Coefficient: 0.4105
- Maximum Coefficient: 1.0000
- Full Clustering Coefficients written to clustering_coefficients.csv

Looking at this output we can see that with 262,111 (nodes) or products that were analyzed in this dataset, and 1,234,8777 (edges) or co-purchase relationships. The graph is sparse given that the number of edges far exceeds the number of nodes. This sparsity indicates that most products are not directly co-purchased with all other products. Next, looking at our Average In/Out Degree, we can see that for the In-Degree this represents the number of products for which a given product is recommended as co-purchased. Whereas, the Out-Degree represents the number of products that are recommended as co-purchases for a given product. Since they are equal this means that the graph is balanced in terms of recommendations. This also implies that in the real world, a typical product listed on Amazon is associated with a small but significant number of co-purchases, about 5 products on average, which could mean that co-purchases recommendations are relatively targeted rather than overly generalized. Next, looking at the shortest path cost from our starting node ‘0,’ we see that to reach for example to reach 107315 it takes 22 hops, which indicates that the products are quite far from the starting product in terms of co-purchasing. Next, when looking at the clustering coefficient we see that the average was 0.4105 – which means that on average around 41% of a product’s co purchased products are also co-purchased with one another. Also we see that our Maximum Coefficient was 1.0, meaning that some products have fully interconnected co purchases, indicating that all of their neighbors are co-purchased with each other.