DS210 Final Project Write-Up

Alicia Cao

Github Link: https://github.com/aliciavcao/DS210_Final_Project

Project Overview:

The dataset I used for this project is a Konect network of the international E-road network, a road network located mostly in Europe. This network is undirected where the nodes represent cities and an edge between two nodes denotes that they are connected by an E-road. The main idea of what I wanted to discover in my graph analysis is how large and interconnected the international E-road network is. To get a better grasp on the size I wanted to calculate the average distance between all nodes pairs to see on average, how far would the average person in Europe have to travel to get to another city connected by this road. I also calculated this average in a random sample of 100 nodes to be able to visualize a more realistic number. Lastly, I wanted to calculate the average distance of shortest paths between all the node pairings. I did this by implementing a breadth first search algorithm as a function and using it to calculate my averages.

Breadth-First Search (BFS) is an algorithm that systematically explores a graph starting from a specified node. It begins by visiting the root node and then explores all its neighbors. Subsequently, it visits the neighbors of those nodes before moving on to deeper levels. BFS uses a queue data structure to keep track of the nodes to be visited, ensuring that nodes are visited in the order they were discovered, i.e., level by level. As it progresses, BFS marks visited nodes to avoid revisiting them. By traversing the graph layer by layer, BFS efficiently discovers shortest paths, identifies connectivity between nodes, and determines various properties of the graph, making it a fundamental algorithm in graph theory and pathfinding. I chose to use BFS because of how its particularly useful for calculating effective paths between graphs due to its nature of exploring all possible paths level by level. As it traverses level by level, the first time a node is encountered, it will be via the shortest path. In addition to that, BFS usually uses less memory compared to depth-first search (DFS) when exploring larger graphs. This is because BFS uses a queue to maintain the nodes to be visited, while DFS uses a stack (or recursion) which can consume more memory due to deep recursion. Since the purpose of this project is to analyze a large graph, it makes more sense to use BFS for memory purposes.

Code Explanation:

Going by the order of functions in my main.rs file:

The first thing I did was construct my graph. I created a variable with the file path of the csv I was using and then used the read_graph_from_csv function. Within this function it creates an empty undirected graph of string nodes and empty edges. Then it opens the file specified by file path. When successful, it creates a CSV reader from the file and iterates over each record (line) in the CSV file. For each record it retrieves or adds a node in the graph for the first and second columns of the record and adds an edge between the source and target nodes in the graph. It then adds another edge between the two but

in reverse since it is an undirected graph. If the creation of the graph was successful it will print the number of nodes and edges just to be able to visually verify it went through and check if the number of nodes and edges are correct.

The next step is printing out the average distances. For all the functions that I wrote to print these out I used a single breadth first search function, called bfs_shortest_path, that calculates the shortest paths between nodes. The function takes two main parameters: the graph itself and a starting node index from which the BFS algorithm begins traversing the graph. Upon invocation, the function initializes a HashMap named distances to store the computed shortest path lengths. It also creates a queue (VecDeque) to facilitate the BFS traversal, starting by enqueueing the provided starting node and setting its distance to 0 within the distances HashMap. The BFS algorithm proceeds by dequeuing nodes from the queue, examining their neighbors, and updating their distances from the starting node if these distances have not been calculated before. For each node removed from the queue, the algorithm evaluates its neighbors, increments the distance by 1, and enqueues these unvisited neighbors for further examination if their distances have not yet been determined. After completing the BFS traversal throughout the graph, the function returns a HashMap (distances) containing node indices as keys and their associated values representing the shortest path lengths from the provided starting node to each reachable node within the graph. This implementation computes and retrieves the shortest path lengths using BFS, providing a mapping of distances from the starting node to all other nodes in the graph.

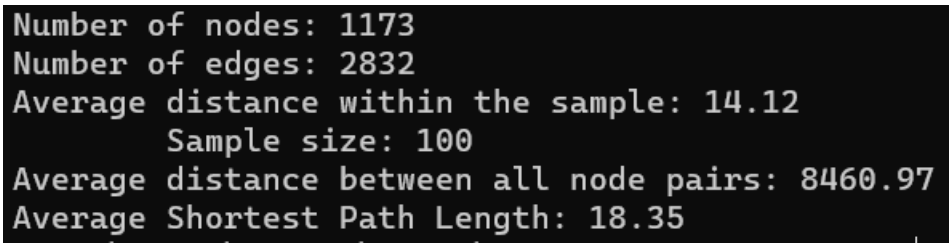
Continuing from the main function, to find the average distances between nodes of a sample of 100, I created the function calculate_average_distance_within_sample. This function takes in 2 parameters which are the graph and the sample size that you want to use. It first checks if the graph is empty or if the sample size provided is zero. If either condition holds true, the function returns a default value of 0.0. To carry out the computation, the function starts by initializing a random number generator by using the rand crate and collects all node indices from the graph into a vector named node_indices. Subsequently, it shuffles these node indices to randomize their order. From this randomized set, the function selects a sample of node indices according to the specified sample_size, storing these nodes in the sample_nodes variable. The core computation involves determining the total distance within the sample. It iterates through each sampled node and employs the bfs_shortest_path function to calculate the shortest path lengths from the current node to all other nodes in the graph. For each sampled node, it then iterates over all node indices in the graph and accumulates the distances obtained from the shortest path calculations, updating the total_distance_sample accordingly. Finally, the function computes the average distance within the sample by dividing the accumulated total_distance_sample by the product of the sample size and the total number of nodes in the graph. This calculation generates a f64 value representing the average distance between nodes within the specified sample, incorporating the graph's entire node set to derive this averaged distance metric.

The next part of the main function is the computation of the average distance between all node pairs in the csv file. The calculate_average_distance_between_all_pairs function only takes the graph as a parameter and starts by collecting all node indices from the graph and stores them in a vector named node_indices. The core computation involves iterating through each node in the graph using these collected indices. For each node, the function employs the bfs_shortest_path function to calculate the shortest path lengths from that particular node to every other node in the graph. It then traverses through the node indices once more, checking for existing distances between the current node and the target node. When a distance is found, it accumulates these distances into the total_distance_all_pairs. Following this

computation, the function calculates the average distance between all pairs of nodes. It accomplishes this by dividing the accumulated total_distance_all_pairs by twice the product of the number of node indices in the graph. This adjustment for doubling the count accounts for the fact that each pair of nodes is encountered twice during the iteration process. Ultimately, the function returns a f64 value representing the average distance between all pairs of nodes in the provided graph, utilizing BFS to compute shortest path lengths and considering every possible combination of nodes within the graph to derive this averaged distance metric.

The last part of the main function is computing the average shortest path length within the csv file. I created the calculate_average_shortest_path_length function to compute this. Initially, it performs a check to determine if the graph is empty. If the graph contains no nodes, the function promptly returns a default value of 0.0, signifying an absence of nodes or edges. Upon confirming a non-empty graph, the function proceeds by collecting all node indices from the graph, storing them in a vector named node_indices. It then initializes variables, namely total_shortest_path_length and total_pairs, to keep track of the cumulative sum of shortest path lengths and the count of processed node pairs, respectively. The core computation involves iterating through each node in the graph using the collected node_indices. For each node, the function employs the bfs_shortest_path function to compute the shortest path lengths from that node to every other reachable node. Subsequently, it traverses through all node indices again, examining each node pair. Upon finding the smallest distance between the nodes, the function adds this distance to the total_shortest_path_length and increments total_pairs. Finally, the function calculates the average shortest path length by dividing the accumulated total_shortest_path_length (treated as an integer) by the count of total_pairs. This division yields a floating-point value (f64), representing the average shortest path length within the graph. Overall, the function uses BFS to compute shortest path lengths between all node pairs and derives the averaged shortest path length metric by considering every possible pair of nodes in the graph.

Output:



Findings:

The output reveals information about the road network that shows that it is effective in getting people to the cities that they need to go to. With 1173 possible cities and 2832 connections between them it would make sense for the average distance between any random city to be extremely large. But after using the BFS algorithm to calculate the shortest paths between two nodes, we can compare that distance of 18.35 to the average of all distances: 8460.97. This difference is insanely large, indicating that the roads are built to allow users to effectively travel internationally.

Test Output:

```
running 5 tests
test tests::test_calculate_average_shortest_path_length_empty_graph ... ok
test tests::test_empty_graph ... ok
test tests::test_calculate_average_shortest_path_length_single_node_graph ... ok
test tests::test_calculate_average_distance ... ok
test tests::test_zero_sample_size ... ok

test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

The tests that I have implemented are simple. These tests ensure that the functions behave as expected under various scenarios such as empty graphs, graphs with known structures, and specific input conditions. To go into greater detail, I have implemented tests that check if my various average functions return a 0 if one chooses to pass in empty graphs or 0s for any of the parameters. They work by using assert_eq where it checks if the actual value obtained from the function matches the expected value. If they don't match, the test fails. I ensured that my functions would not fail by starting the functions off by checking if the parameters passed through the function are valid and returning a 0 if they weren't. The last ones I created were two test cases to see if my functions calculate averages properly. By creating a small graph with known distances between nodes and then comparing the calculated average distance against a manually computed expected value, I am able to check if the function runs as expected.