DS210 Final Project Report

This project aims to develop a graph data type in Rust along with functions that will find and calculate the distance between any two vertices. The graphs will be loaded from a text file that lists edges in two columns, separated by whitespace, indicating the "from" node and the "to" node. My approach will accommodate both directed and undirected graphs, but it will focus on unweighted graphs only. The main functionality will include calculating the distance between two randomly chosen vertices using a breadth-first search algorithm to identify the shortest path. By repeating this process with sufficiently many randomly chosen node pairs, the implementation will enable us to determine the average distance between any two nodes in the dataset, the longest path length, the proportion of nodes that are connected, and the total number of nodes over a predetermined number of iterations.

The graph.rs module manages graphs using the breadth-first search (BFS) algorithm. It starts by importing necessary components from Rust's standard library, including data structures like HashMap for storing graphs as adjacency lists, and utilities for file handling and input/output operations. An enumeration BFSError with a variant NodeNotFound is defined for error handling in graph operations. The main structure, Graph, uses a HashMap where each node key maps to a list of its direct neighbors. The graph functionality is implemented through methods to create a new graph, add edges, access the graph's internal map, and print its adjacency list for debugging. A significant function, read_graph_from_file, reads a graph from a specified file, expecting each line to denote an edge between two nodes separated by whitespace. This function helps build the graph by parsing the file and adding edges accordingly. The code also includes unit tests that validate various functionalities like graph creation, edge addition, and BFS-based shortest path calculations, ensuring that the graph and BFS operations work as intended.

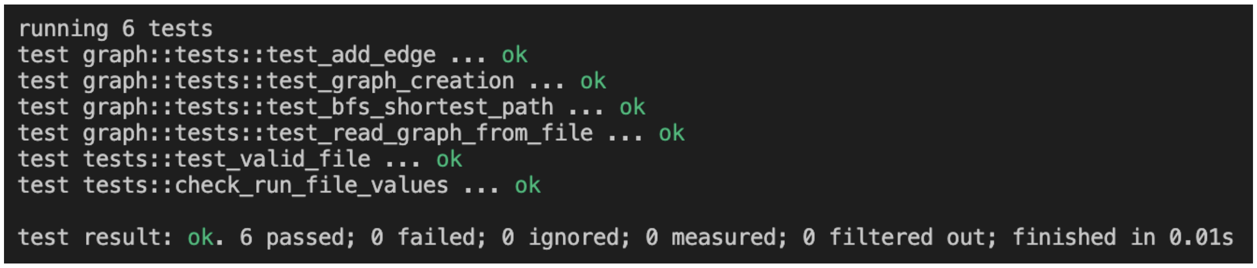
The bfs.rs module consists of two main functions: bfs_shortest and run_file. Bfs_shortest function implements the Breadth-First Search (BFS) algorithm to find the shortest path between a start_node and a target_node in a graph represented by a HashMap. It uses a HashSet to keep track of visited nodes, a VecDeque for the BFS queue, and another HashMap to record the distances from the start_node. The function returns the distance if the target_node is found; otherwise, it returns an error indicating the node was not found. Conversely, the run_file function simulates and analyzes graph behavior based on data from a specified file. It initializes a graph from the file, and over a specified number of test iterations, it randomly selects start and target nodes to find the shortest path using the bfs_shortest function. The function tracks the total distances, counts paths found, and determines the longest path. It optionally provides detailed outputs for each iteration. Finally, the function calculates and displays statistics such as the average path length, and the percentage of connected node pairs, and returns these metrics along with the longest path length found. Optionally, it can print the graph's adjacency list.

The main.rs module consists of a function and a test module, each utilizing a graph analysis function from a module: As for the main function, its purpose is to execute simulations on various graph data sets by processing graph files with the run_file function, imported from a module named bfs. It calls run_file multiple times with different parameters and processes different types of graphs (directed, undirected, connected) specified by filenames, such as "directed.txt" and "undirected.txt". The function parameters typically include the filename, number of nodes, number of iterations for testing, and boolean flags to control node tracking and adjacency list display. Moreover, it includes datasets from a Stanford website, handling larger graphs with fewer iterations to manage complexity. The test module's purpose is to contain tests to ensure that the run_file function behaves as expected when processing graph data. test_valid_file verifies that run_file can process a file without errors using predefined parameters, while check_run_file_values checks specific outcomes from run_file, such as the number of paths found, the percentage of node connections, and the longest path, ensuring the function's accuracy and correctness in graph analysis.

To evaluate the accuracy of these functions, I constructed three nine-node graphs named directed.txt, directed_connected.txt, and undirected.txt. The directed.txt graph includes vertices that are not traversable to all other vertices (for example, there is no route from node 7 to node 0). In contrast, directed_connected.txt adds three additional directed edges to ensure each vertex is reachable from any other. This was validated by performing 10,000 iterations where two random nodes were chosen to traverse. With directed.txt, connections between node pairs were found in about 70-71% of cases, whereas with directed_connected.txt, all node pairs were connected, reflected by a 100% connection rate. Expectedly, the undirected.txt graph also showed a 100% connection rate due to the bidirectional nature of its edges. To verify that data was read correctly, I enabled the “print_adj_list” argument in the “run_file” function. Additionally, I crafted six unit tests to examine the program’s critical functionalities. In the “main.rs” file, tests ensure the program accurately reads files from the “datasets” directory. The tests in “graph.rs” confirm proper instantiation of the HashMap, functionality of the “add_edge” method with manually entered data, and accurate distance calculations by the “bfs_shortest” function using both manually entered and file-read data.

I conducted practical testing using three datasets: the Email-Eu-Core Network (1,005 nodes), Epinions Social Network (75,879 nodes, directed), and Slashdot Social Network (82,168 nodes, undirected). I performed 1,000 iterations for the Email-Eu-Core Network and 100 for the other two. Despite variations in connectivity, the average node distance stayed under six edges, supporting the "six degrees of separation" theory. However, the longest paths sometimes exceeded six edges, and there were significant connection gaps, especially in the Epinions dataset with a 40-45% connection rate. The Slashdot dataset had a higher connection rate of 85-90%, likely due to its undirected structure facilitating more links. This research provided insights into node interactions and the extent of the "six degrees of separation" within large social networks.

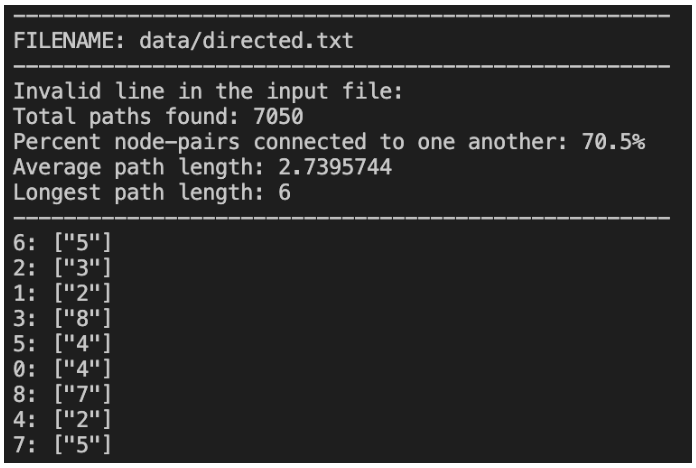
Tests:



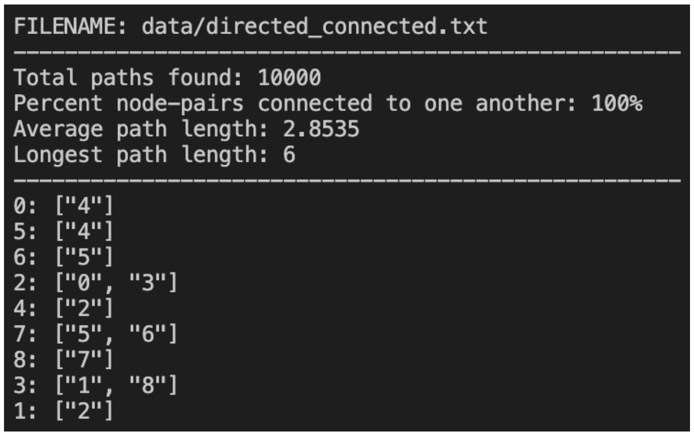
```
running 6 tests
test graph::tests::test_add_edge ... ok
test graph::tests::test_graph_creation ... ok
test graph::tests::test_bfs_shortest_path ... ok
test graph::tests::test_read_graph_from_file ... ok
test tests::test_valid_file ... ok
test tests::check_run_file_values ... ok

test result: ok. 6 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s
```

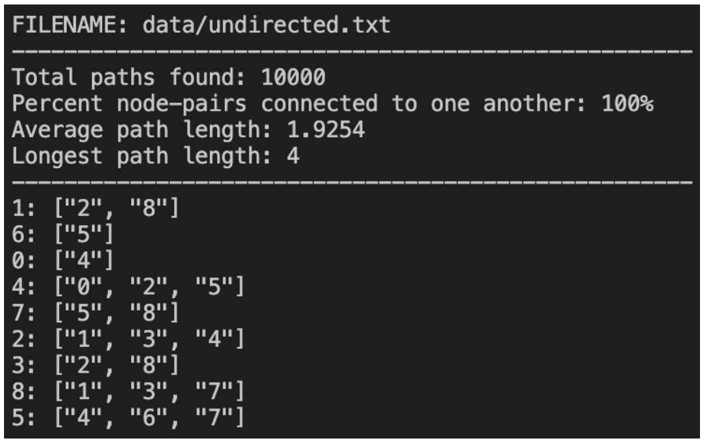
Output:



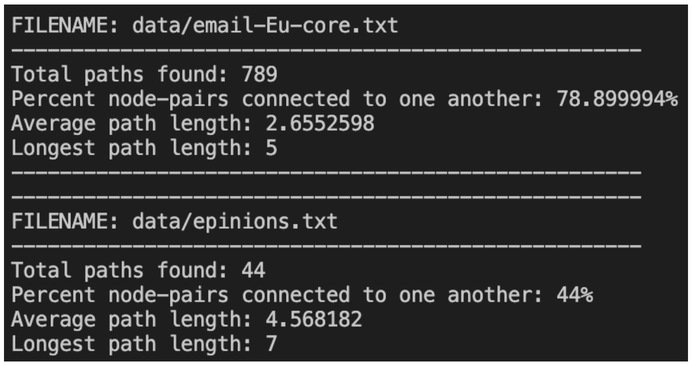
```
---------------------------------------------------
FILENAME: data/directed.txt
---------------------------------------------------
Invalid line in the input file:
Total paths found: 7050
Percent node-pairs connected to one another: 70.5%
Average path length: 2.7395744
Longest path length: 6
---------------------------------------------------
6: ["5"]
2: ["3"]
1: ["2"]
3: ["8"]
5: ["4"]
0: ["4"]
8: ["7"]
4: ["2"]
7: ["5"]
```



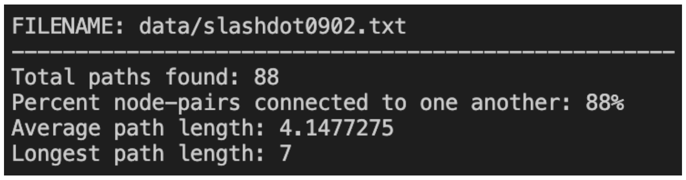
```
FILENAME: data/directed_connected.txt
----------------------------------------------------
Total paths found: 10000
Percent node-pairs connected to one another: 100%
Average path length: 2.8535
Longest path length: 6
----------------------------------------------------
0: ["4"]
5: ["4"]
6: ["5"]
2: ["0", "3"]
4: ["2"]
7: ["5", "6"]
8: ["7"]
3: ["1", "8"]
1: ["2"]
```



```
FILENAME: data/undirected.txt
----------------------------------------------------
Total paths found: 10000
Percent node-pairs connected to one another: 100%
Average path length: 1.9254
Longest path length: 4
----------------------------------------------------
1: ["2", "8"]
6: ["5"]
0: ["4"]
4: ["0", "2", "5"]
7: ["5", "8"]
2: ["1", "3", "4"]
3: ["2", "8"]
8: ["1", "3", "7"]
5: ["4", "6", "7"]
```



```
FILENAME: data/email-Eu-core.txt
-------------------------------------------------
Total paths found: 789
Percent node-pairs connected to one another: 78.899994%
Average path length: 2.6552598
Longest path length: 5
-------------------------------------------------
-------------------------------------------------
FILENAME: data/epinions.txt
-------------------------------------------------
Total paths found: 44
Percent node-pairs connected to one another: 44%
Average path length: 4.568182
Longest path length: 7
```



```
FILENAME: data/slashdot0902.txt
----------------------------------------------------
Total paths found: 88
Percent node-pairs connected to one another: 88%
Average path length: 4.1477275
Longest path length: 7
```