Luke Dunne

Professor Kontothanassis

DS210

5 May 2024

# Final Report - Pennsylvania Road BFS

My project is centered around analyzing the Pennsylvania road network using a graph and rust code. The primary motivation for this project is to better understand the basics behind how mapping applications like Google Maps or Waze compute road network data to suggest the best routes to users. Because I'm from Maryland, the Pennsylvania road dataset, being the closest relevant dataset from the Stanford Large Network Dataset Collection (SNAP), was interesting to me. The main goal of the project is to explore the dataset to calculate significant graph metrics, specifically the average shortest path length using a breadth-first search, and find other key insights about the graph that could be applicable to real-world scenarios such as traffic management.

I computed the average length of the shortest paths using a breadth-first search from a given node to all other nodes, for all nodes. This, however, was very strenuous, and was estimated to take over twelve hours to complete running. After running the original code with a timer and updates, I decided that I would have to take a random sample of 50,000 vertices of the 1,000,000 plus. The technique I used was to randomly scramble the nodes into a hashset, before iterating through. The issue I encountered was that many nodes were returning 0.0 for the average length of the shortest paths because of a disconnect in the dataset. To fix this, I changed

the code to run until 50,000 nodes with a distance greater than 0.0 were selected to eliminate the effect of these outliers.

In my Rust project, the main.rs file acts as the user entry point by declaring the graph_reader, distances, and bfs_sample modules, which are essential for reading graph data, performing breadth-first search (BFS), and performing the search from a sample, respectively. Within the main function, I use the read_graph function from the graph_reader module to load data from a specified file path where I gained inspiration from Lecture 27. This function returns the total number of nodes and the list of edges between them, which are then used to make an undirected graph, with inspiration from Lecture 28. After the graph is constructed, I call the bfs_sample function to run BFS on the given sample, calculating the average shortest path lengths among a subset of nodes on the main graph to maintain efficiency given the overly large size of the graph.

The distances.rs module is crucial for calculating shortest path statistics using a breadth-first search (BFS) approach. The avg_shortest function calculates several important metrics from a specified start node to all other reachable nodes in the graph, such as average distance, maximum and minimum path lengths, as well as the path for the longest shortest route found. This function applies a standard BFS algorithm, marking nodes as visited and computing distances incrementally as it navigates the graph.

The bfs_sample.rs module, in tandem with distances.rs, randomly selects a subset of vertices of user given size. This function provides five incremental updates on the code's progress while gathering the overall path statistics using the avg_shortest function before summarizing the results.

To run the project, make sure Rust and Cargo are installed on your machine. Download your dataset and make sure the formatting matches the format of PA_Roads.txt, and store it in an accessible location. In the main function, replace the file path on line 9 with your own, and select the sample size in line 11 (in place of 50000). To run the code, I used Cargo through the terminal. To run the tests, use “cargo test” to make sure the code works on the given dataset. To run the code fully, use “cargo run --release” to ensure it runs faster without safety catches. Tests below…

```
running 5 tests
test tests::avg_shortest_no_connect ... ok
test tests::test_avg_shortest ... ok
test tests::min_length ... ok
test tests::test_stats ... ok
test tests::graph_creation ... ok

test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.02s
```

These are explanations of the tests: The graph_creation test verifies an undirected graph from a small file, “project_test.txt”, created properly, ensuring that nodes are connected as expected and that edges are correctly bidirectional. The test_avg_shortest checks the calculation of the average shortest path length in a simple linear sequence of nodes, confirming that it produces the expected average distance. In the min_length test, it tests to ensure that the minimum length is 1, because the value should simply be one edge. The avg_shortest_no_connect test checks how the function handles an isolated node scenario, making sure 0.0 is returned when a node is disconnected. Finally, the test_stats test confirms the ability of the system to identify the longest path in a simple sequence of connected nodes with known endpoints.

Now, with all tests passed, we know the program works. We can now use “cargo run --release” in the terminal. Input code below…

```
fn main() {
    let (n, edges) = read_graph("C:/Users/ldunn/DS210/project/code/src/PA_Roads.txt");
    let graph = Graph::create_undirected(n, &edges);
    bfs_sample(&graph, 50000, n);
}
```

As the program runs, it will provide five updates on its progress and will output the computed average shortest path length throughout. After completion, it gives the overall average shortest path length for the sample, as well as the maximum length of the shortest path, including the end nodes. It also shows the computation time, which depending on the size of the graph, might add up. When running my sample of 50,000 vertices, some instances of running took less than five minutes, however, others took upwards of fifteen minutes. The average shortest path typically fell at 93.5 ± 0.5. Output below…

```
Average shortest path length currently is 93.27333351821937, with time taken for 10000 vertices = 355.3845304s
Average shortest path length currently is 93.28983446118194, with time taken for 20000 vertices = 598.8503272s
Average shortest path length currently is 93.54669175797795, with time taken for 30000 vertices = 717.5341596s
Average shortest path length currently is 93.48635802972218, with time taken for 40000 vertices = 834.5553988s
Average shortest path length currently is 93.49112003711613, with time taken for 50000 vertices = 955.4192399s
Average shortest path length for 50000 vertices: 93.49112003711613

Sample Statistics (not entire graph):
Maximum length of Shortest Path: 295
Path of longest shortest path (start node, end node): (253061, 257986)

Computation time: 955.4201006s
```

The project could provide a deeper understanding of road network connectivity, as well as highlight potential areas that could benefit from improvements. In the future, I think it would be interesting to perform with a dual-weighted graph to be able to map the quickest way to get from one node (intersection) to another where the edges are weighted on distance and speed limit. However, the graph provided did not include these metrics. I think the calculated value provided interesting insights into the Pennsylvania Road system. With the average shortest path typically lying at approximately 93, this would imply that on average, to get from one

intersection to a random one in the state, you would have to travel 93 intersections. This shows the vast span of roads in the state of Pennsylvania. A limitation to this is the fact that 0.0 values for average shortest path were eliminated, potentially skewing the data.