Network Graph Project Report

For my final project, I chose to compare the strongly connected components of two network data sets using the Kosaraju algorithm. My first dataset represents pages from Stanford.edu as nodes and hyperlinks between them as edges. My second dataset is similar but with webpages from Google as nodes and hyperlinks between the pages representing edges. Both datasets can be accessed under the "Sources" section of this report.

I. Explanation of Code

Within my rust coding environment, I created four modules: graph, Kosaraju, components, and main.

First, my graph module contains a function graph_builder that reads in a data file that has two columns of integers separated by white space and produces a directed graph. Both columns represent nodes, and the whitespace between them represents an edge. If a dataset does not have two columns, is not separated by whitespace, or is intended to be an undirected graph, then the graph_builder function will not run. For example, a file that has comma-separated values will not generate a graph using the graph_builder function.

My second module is where I implement the Kosaraju algorithm, which determines the total number of strongly connected components in a graph. A strongly connected component (SCC) is a set of vertices in a directed graph where every vertex is reachable from every other vertex. In short, the Kosaraju algorithm performs two depth-first searches on the directed graph to create a reversed graph and to find the SCCs. During the second depth-first search, the reversed graph is used to ensure that all the nodes within the SCC are traversed together.

My third module contains two functions that allow me to do an analysis of the strongly connected components produced by the Kosaraju algorithm. My first function, find_significant_components, takes the output of the Kosaraju algorithm, the vector of vectors of every strongly connected component, and the total number of nodes in the graph to print three things: the number of significant SCCs, the size of each significant SCC, and the number of nodes in each significant SCC as a percentage of the total number of nodes. This function filters through the SCCs to find the SCCs that contain enough nodes to represent at least 1% of the total number of nodes and labels them *significant*. My second function, find_large_components, takes the SCCs, the total number of nodes in the graph, and a threshold set prior to implementation and prints three things: the number of SCCs that have a size above the threshold set, the size of each large SCC, and the number of nodes in each large SCC as a percentage of the total number of nodes. This function filters through the SCCs to find the SCCs that contain a number of nodes above the threshold set and labels them *large*.

My last module, main, applies these three previous modules. Within the main function, I create a Stanford graph using the Stanford dataset and a Google graph from the Google dataset. The main function then prints the total number of nodes and the total number of SCCs in each graph. After setting the threshold value to 100 for graphs, the main function runs the find_large_components and find_significant_components functions on each graph to produce an output.

II. Explanation of Output

Number of total nodes in Stanford: 281904
Number of total strongly connected components in Stanford: 21411
Number of large components in Stanford: 93 (*Threshold = 100*)
Number of significant components in Stanford: 2
- 1. Size = 218049 nodes (77.35%)
- 2. Size = 3030 nodes (1.07%)

Number of total nodes in Google: 916428
Number of total strongly connected components in Google: 203180
Number of large components in Google: 25 (Threshold = 100)
Number of significant components in Google: 1
- 1. Size = 600512 nodes (65.53%)

Although both graphs have few *significant* SCCs, the Stanford graph has two compared to the Google graph's one. Surprisingly, despite the Google graph being more than three times larger than the Stanford graph (in terms of the total number of nodes), the Stanford graph has nearly four times the number of *large* SCCs. This is unexpected because since the threshold is the same for both graphs at 100, one would expect that since the Google graph has many more nodes, there would be more SCCs that have over 100 nodes compared to the Stanford graph since the threshold is not proportional to the number of total nodes. Therefore, we can say that according to this analysis, the Google graph has a higher percentage of isolated nodes that don't have many edges connecting them to a wide variety of other nodes. Putting this in context, the Google graph has more disconnected web pages than the Stanford graph. We can also say that the Stanford webpages are more interconnected than the Google webpages since 77.35% of all webpages on Stanford.edu can be accessed through each other and are connected. In reality, this makes sense because nearly everything on the Stanford.edu website has to do with Stanford and covers a smaller variety of information and different topics. Meanwhile, Google webpages can be very different, and still, 65.53% of webpages in this graph are interconnected.

III. Tests

Tests are important to algorithms and code to ensure they are effective before running them on large amounts of data. In addition to my four modules, I implemented a test module that contains six different tests. These tests ensure that the graph_builder function, the Kosaraju algorithm, the find_significant_components function, and the find_large_components function all run effectively. Of the six tests, five of them should pass without panicking, and the sixth will pass only because I told Rust it should panic.

The first test ensures the graph_builder function doesn't build a graph if the file is empty. This test panics since the file is empty but passes the test since the goal was the test to panic.

The rest of the tests ensure that the four functions all perform correctly. All five of these tests pass on their own.

To run these tests, one needs to type into the command prompt "cargo test –test test."

IV. Process

Implementing this project was difficult, and I encountered many obstacles. One of the biggest hurdles was getting my Kosaraju algorithm to run properly. I faced many Stack Overflow messages because of the expensive nature of my algorithm. However, after doing research, I discovered that my algorithm was failing to run due to its recursive nature. To fix this problem, I implemented a HashSet data function, and it enabled my algorithm to run. The most important thing is that I learned from the mistakes I made during this project and made the necessary adjustments to run a fast and efficient algorithm. I also was able to implement the many things I've learned this semester and use the Rust language to the best of my ability.

V. Appendix

Datasets: https://snap.stanford.edu/data/#amazon

Sources:
https://www.youtube.com/watch?v=18-7NoNPO30&list=PLai5B987bZ9CoVR-QEIN9foz4QCJ0H2Y8&index=15
https://www.youtube.com/watch?v=-L4nKAlmH3M&list=PLai5B987bZ9CoVR-QEIN9foz4QCJ0H2Y8&index=14
https://www.youtube.com/watch?v=5wFyZJ8yH9Q
https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm
https://www.geeksforgeeks.org/strongly-connected-components/
https://doc.rust-lang.org/book/ch11-01-writing-tests.html
https://doc.rust-lang.org/std/collections/struct.HashSet.html
https://stackoverflow.com/questions/53113010/strongly-connected-components-kosaraju-algorithm
https://doc.rust-lang.org/book/ch13-01-closures.html
https://doc.rust-lang.org/std/iter/trait.Iterator.html