Professor Chator

Programming for Data Science

May 1 2024

Six Degrees of Separation Analysis

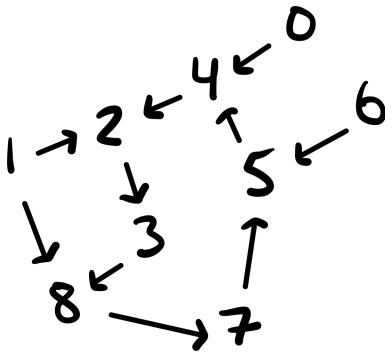
The concept of "six degrees of separation" describes an exciting attribute of many large social networks. It suggests that any node in a graph can be connected to any other node by, at most, six edges–something that may make individuals in a social network closer than we might think. Though this concept would typically be complex to examine in the real world, the more common accessibility of large datasets that describe the connections between humans on social media platforms allows us to prove its accuracy. In this project, I plan to implement a graph datatype in Rust and functions that may search for and compute the distance between any two vertices. The graphs will be read into the graph type from a text file containing two columns separated by whitespace: the "from" node and the "to" node. My implementation is designed to work with both directed and undirected graphs, yet only unweighted graphs. It will compute the distance between two randomly selected vertices in the graph and run a breadth-first search algorithm to determine the shortest path between them. After iterating through enough randomly selected nodes, we should be able to compute the average distance between any two nodes for that dataset, the most extended path length, the percentage of nodes found to have a link, and the total number of nodes found for a set amount of iterations.

The graph datatype is implemented in its file as a module for my main file. I will implement it as a tuple struct with only one feature: a Hashmap of String keys and Vector<String> values. Thus, the data from the sample files will be read through the adjacency-list graph format. This will be more useful as some vertices in extensive data become more spread out as the size of the dataset increases, allowing me to save on space. The graph

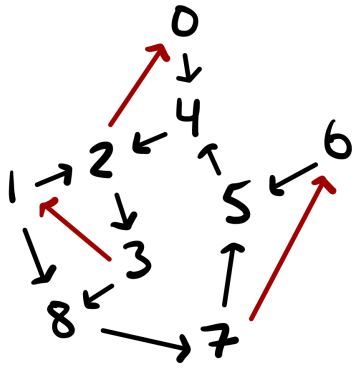
contains four methods in the implementation block: the “new” method, “add_edge” for when we read the data, “get_map” for running the HashMap through functions, and “print_adjacency_list” for debugging and visualization. Then, I implemented a simple version of the breadth-first search that returns when the shortest path from one node to another is found in the HashMap. This algorithm will always find the shortest route as it searches through a queue of nodes at the same proximity as the original node. In other words, the algorithm will look at all nodes one edge away before two edges away, and so on. The function will return the target node if found or a catchable “NodeNotFound” error to allow the program to continue running while letting the user know. The “read_graph_from_file” function performs a simple file read with the specified file name and checks the file format to ensure the data is not misread; the files used to represent our graphs must be separated by whitespace to work. Then, it returns a graph object after using “add_edge” to build the graph within the HashMap.

There are only two functions that I needed within the main file. The function “run_file” acts as the “formatter” for the output of the function, which takes in the name of the file, the number of nodes in that particular graph, the number of test iterations desired, a boolean argument for tracking individual node iterations, and a boolean argument to print the adjacency-list after the program is completed. Since some files were formatted differently, I thought it would be easier to input the number of nodes in the function header rather than read it directly from the file. The test iterations argument determines the number of randomly selected node pairs to compute the distance between them. Alhough it would be great to iterate through the entirety of the graph rather than picking randoms, the size of some large graphs simply takes too long to compute. Of course, the user can feel free to change the test iterations for each file and determine what gives the most accurate results without taking too much time to compute.

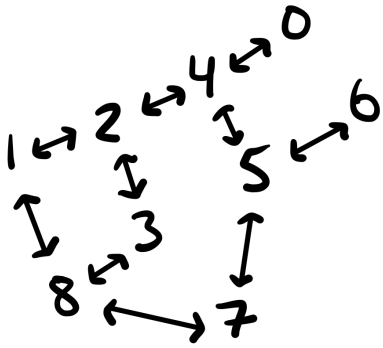
The “track_nodes” argument will print the distance between each node pair (or tell the user there was no connection between nodes). This feature was handy while debugging to ensure that each pair of nodes that came up was being computed correctly. While “print_adj_list” is also very useful for debugging, it should only be utilized with smaller graphs, as larger graphs would flood the terminal screen. The “run_file” function will also compute helpful information about the particular file’s graph structure and format prints for the user in the output.



directed.txt



directed_connected.txt



undirected.txt

To test the accuracy of these functions, I created three graphs of nine nodes called directed.txt, directed_connexted.txt, and undirected.txt. While directed.txt contains vertices that may not be traversable to all other vertices (e.g., there is no way to get from node 7 to node 0), directed_connected.txt provides three additional directed edges that ensure each verticy is accessible to all other vertices. This is also shown when the algorithm performs 10,000 iterations of choosing two random nodes to traverse. With directed.txt, the algorithm only finds node-pair connections between 70-71% of the selected nodes. However, with directed_connected.txt, the algorithm computes this percentage as 100%. As one might expect, the undirected.txt graph also computes this percentage to be 100% since nodes can then travel either way. I found it helpful to set the “print_adj_list” argument to true in the “run_file” function to ensure the data is being read correctly.

I have also provided six unit tests that test the program's most essential aspects. In "main.rs," I ensure that the program can read files correctly given some files in the "datasets" directory. The unit tests in the "graph.rs" file ensure that the HashMap is instantiated correctly, that the "add_edge" function works with manually entered data, and that when data is manually entered with the "add_edge" function, the breadth-first search function ("bfs_shortest") computes the distances correctly. I also ensure that "bfs_shortest" works correctly with data read from a file.

Once I knew that the program was running correctly, I added some test data to work with. The three datasets I used were the Email-Eu-Core Network (link), Epinions Social Network (link), and Slashdot Social Network (link). The number of nodes in these networks was, respectively, 1005, 75879, and 82168 nodes. While the Epinions graph was directed, the other two were undirected graphs. While I ran 1000 iterations of the Email-Eu-Core Network, I only ran 100 for the other two graphs due to time constraints. (The user may change the number of iterations as they please.) The average distance between nodes for all of these datasets seemed to have remained under six edges every time, giving some accuracy to the idea of "six degrees of separation." Still, however, the longest path was sometimes a bit larger than six edges, and there were a considerable number of cases where there was no connection between nodes at all. This occurred much more in the Epinions dataset, with a percentage of node pairs connected at around 40-45%.

Meanwhile, the rate was much better on a slightly larger dataset, Slashdot, ranging from around 85-90%. This is likely because the Epinions dataset is directed while the Slashdot dataset is undirected, allowing for more pathways between nodes. Moreover, some large datasets I experimented with needed more connectivity between random nodes despite having hundreds of

thousands of nodes. This could be due to the number of edges in the graph, where a graph with a small number of edges will not be very connected. Another potential reason, as already mentioned, could be whether or not these graphs are directed or undirected. Overall, this program gave me a pretty good idea of how the nodes in extensive social networks interact and how often random nodes are connected, and it proves, to some degree, the accuracy of the “six degrees of separation” concept.