Vijay Fisch
U30113778
DS210 Final Project
Due December 15th, 2023

Centrality in the Chicago Road Network

Data link: https://networks.skewed.de/net/chicago_road
Github Link: https://github.com/vijayfisch/DS210-Final-Vijay-Fisch
Runtime: Near instantaneous with cargo run --release, around 10 seconds with cargo run
Project Overview:

- Includes main, two modules (read_file, bfs), a tests file
- Includes two tests, 11 functions
- Main functions: BFS and centrality

Abstract
I am passionate about the intersection of urban policy and data analytics, so I found a data set containing Chicago's road network. The dataset is from the late 20th century, but I was mostly interested in the process of calculating what roads in a network may be more central than others, and creating an additional program to find the shortest path (not in terms of distance, so no weights) in terms of steps from one road to another.

The data set included over 10,000 nodes, each representing a road. Each edge was a tuple representing an origin and destination road. I wanted to use my rust skills to find the most central roads (in terms of connections to other roads), and also incorporate a shortest path function.

Using BFS and a centrality calculation, I found the top 5 central roads, their degrees of centrality based on how many nodes were connected to that node (in and out), and in order to incorporate the bfs function, the distance from each node to an arbitrarily selected node.

Data
My data set is a list of tuples of the following format: (origin, destination). Using the num_nodes and num_edges functions (removed them as tests), I found the number of nodesis 12,979 and number of edges is 39018. As seen on the right, each index includes an origin and destination node.

Methodology
First, I created a read file that took in a vector of tuples as (usize, usize), isolated an x and y from each tuple (origin and destination) and pushed them to read the csv file. Next, I created a bfs folder with the rest of my functions.

First, I created a type, Node, set as usize, and a struct for a Graph to create an adjacency list of each node and its connections.

Then, I created a graph implementation, including a bunch of functions relying on or connected to the graph struct. These functions were used for two main components: centrality and BFS for shortest path.

1) **Centrality** – end goal is to find which nodes have the most in and out connections
   a) fn new() -> Self

This function initialized an empty hashmap in which our data could be entered into. This hashmap would eventually allow us to determine the number of in and out nodes for each node.

   b) fn add_edge(&mut self, from: Node, to: Node)

This add edge function took each edge from the graph struct, found an entry in the hash map equal to the item we are looking for, and adds that connected node to the hashmap.

   c) in_degree(&self, node: Node) -> usize

c) in_degree(&self, node: Node) -> usize

The in degree function accessed the adjacency list, and found the values of nodes who's connected nodes vector contained the inputted vector. For example, if you input 8, it would find all of the nodes who connect to 8. Then, it found the number of nodes using .sum().

d) out_degree(&self, node: Node) -> usize

The out degree function accessed the adjacency list as well, just the adjacency list for the inputted node. It isolated just the neighbors of the node, and found the length of that vector.

e) centrality(&self, node: Node) -> f64

The general idea of the centrality function was that you could find the degrees of centrality by adding up the total number of in and out nodes for a given node, and dividing it by the total number of nodes. Nodes that were connected to more nodes would have a higher degree of centrality than other nodes. I did this by using the in_degree and out_degree function with the graph and the inputted node, returning an f64 rather than a usize value since the answers were going to be length decimals. This is because we were dividing in and out nodes of about 5-15 by the total number of nodes (over 10,000).

f) graph_centralities(&self) -> HashMap<Node, f64>

Lastly, I returned a hashmap containing a node and its centrality by iterating through the entire graph, calculating the centrality value for every element, and inserting the node and its centrality into a new hashmap.

Using the above functions, I was able to find the degrees of centrality for every road.