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.

```
22685   8455,3947
22686   8455,3948
22687   8455,8792
22688   8456,3984
22689   8456,12061
22690   8456,12062
22691   8457,8458
22692   8457,8459
22693   8457,11250
22694   8458,8428
22695   8458,8448
22696   8458,8453
22697   8458,8457
22698   8459,8448
22699   8459,8457
22700   8459,8460
22701   8459,8469
22702   8460,8446
22703   8460,8459
22704   8460,8461
22705   8460,8468
```

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

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.

2) **BFS for shortest path** – the end goal is to find the shortest path between any two nodes
   fn bfs_shortest_path(&self, start: Node, end: Node) -> Option<Vec<Node>>

This function took in a list of edges, a start node, and an end node, returning a vector of nodes to display the shortest traversal. I chose to use breadth-first-search rather than depth-first-search since BFS explores nodes level by level, rather than DFS which is more efficient but less effective for this task. DFS is more frequently used for directed graphs.

My shortest path function took in two nodes, an origin and destination, and a graph as an adjacency list. Using a hashset to track visited nodes, and a VecDeque to create a queue for traversal, the function stored shortest path nodes in a hashmap. By exploring neighbors of nodes in the queue until finding the destination node, it calculated shortest paths and returned them as Option<Vec<Node>>. In the case that there was no path between two nodes, the function returned None. This algorithm is best for an unweighted graph like the Chicago road network.

Tests

I had three tests;

1) Testing the data size
   a) fn num_edges(&self) -> usize
   b) fn num_nodes(&self) -> usize

```
assert!(graph.num_nodes() > 1000);
assert!(graph.num_edges() > 1000);
```

In my Graph implementation, I had two functions to calculate the the number of edges and nodes in the graph. These functions were in the Graph implementation rather than the test file since they relied on the Graph struct, but were labeled as #[cfg(test)] to clarify their purpose. The first test asserted that the number of nodes and number of edges in the dataset used met the requirements of the project (both over 1000).

2) Testing the shortest path

For the second test, I found a random node and another node two steps away from it. I went to node 8017, found a random neighbor (8739), and found a neighbor of that neighbor (8740). Then, I printed out the steps using the bfs_shortest_path function, and compared the two.

```
let path = graph.bfs_shortest_path(8017, 8740);
assert_eq!(path, Some(vec![8017, 8739, 8740]));
```

3) Testing a disconnected path

```
let path = graph.bfs_shortest_path(start_node, end_node);

// Assert that there is no path between these nodes
assert_eq!(path, None);
```

I ran my BFS algorithm on a disconnected node, 12983, and asserted that there was no path to test the else clause in BFS. All three tests passed, returning the following:

“test tests::tests::test_path ... ok
test tests::tests::test_data_size ... ok
test tests::tests::test_disconnected_nodes ... ok
test result: ok. 3 passed”

Results and Real World Applications

The top 4 central roads all had the same centrality, with 7 in and 7 out connections for each. The next highest road had 6 in and out connections. Here is an example output:

> Number 2 - Node 10339: Centrality: 0.0010786655366361044
> Shortest path to road 1: [10339, 6003, 6034, 10340, 6042, 6050, 6070, 10341, 6237, 6247, 6542, 6655, 6670, 4065, 4087, 7486, 4392, 4452, 4476, 4548, 4556, 10106, 10107, 7440, 4360, 7540, 7577, 7615, 7662, 7664, 7707, 7709, 7710, 9154, 7432, 2377, 7854, 10293, 1]

Earlier in the process, I printed the below results to do a quick check on the in-degree and out-degree functions, both of which were correct.

```
Top 1 - Node 10351: Centrality: 0.0010787486515641855
Node 10351: In-Degree: 7, Out-Degree: 7
Top 2 - Node 10340: Centrality: 0.0010787486515641855
Node 10340: In-Degree: 7, Out-Degree: 7
Top 3 - Node 10339: Centrality: 0.0010787486515641855
Node 10339: In-Degree: 7, Out-Degree: 7
Top 4 - Node 10410: Centrality: 0.0010787486515641855
Node 10410: In-Degree: 7, Out-Degree: 7
Top 5 - Node 6235: Centrality: 0.0009246417013407304
Node 6235: In-Degree: 6, Out-Degree: 6
```

While many roads had the same number of in degrees and out degrees, it was still important that I use both in my function. Given the total number of nodes was 12,979, the data contained one or a couple lines in which one node connected to another and not vice-versa. On top of this, doubling the degrees of centrality for every node did not skew the results.

The integration of data science techniques will revolutionize the urban planning process, especially in the context of road maintenance. In a more complicated project in which a weighted dataset was available, I could have found some interesting stats such as the shortest path including traffic as weights, or the most central roads including traffic as weights. Although the number of connections from a road yields some interesting results, without further data and analysis, we cannot really tell whether or not these roads necessitate evaluation or maintenance. An urban planner, using similar but more complex techniques could hypothetically predict road damage, accident risks, and intense traffic corridors utilizing a centrality technique.