Analyzing California Roads

By Joshua Lee

Collaborators: None

Chosen Dataset:

The dataset I have chosen is the California Road Network dataset from Stanford. The link to this dataset is here.

Overall Function of Project:

This project takes in a dataset which consists of 2 columns. The first column is the starting road/node and the second column is the road/node the first column leads to. It takes this dataset and constructs an undirected graph where it has the nodes as starting roads and the edges as all of the roads it can go to. Because there are multiple paths one road can take, it can have multiple edges. After it constructs the undirected graph, it then calculates the distance from one road to another road based on user input and it can also display the maximum length a road can take in relation to all its connections.

Function Implementations:

In this code I created 2 additional modules that contain structs. The first module I created was the **Graph** module. This module constructed a graph that would have a node and corresponding edges when read from a txt file. The second module was my **Path** module and in this module, this is where I computed my BFS algorithm for 2 nodes and my maximum distance for a single node.

graph.rs

In this module, I create a struct that represents the graph of nodes and corresponding edges. In this module I have multiple functions that each do a separate task that is crucial to my other calculations. The functions that correspond to the actual creation of my graph are *add_nodes()* which takes a new node that is not currently in the graph and adds it to my graph. The other function corresponding to the creation is *add_edge()* which adds an edge to the corresponding node. My other functions are mainly used in my other modules. I have 2 functions which correspond to getting the index value of any node I am given. This is used so that I can find the node I am looking for within the vector of nodes. One of them finds the index for unsorted nodes and the other finds the index for sorted nodes. *display_graph()* is used to print out all of my nodes and the edges that correspond to it in *main.rs*. *num_nodes()* function represents the number of nodes/vertices in my graph. My final function in this module *sort_edges()* is the most important. This is because this function takes a clone of my nodes and sorts them based on the smallest node to the largest node. After the function creates a sorted vector of my nodes, it creates a new vector for my edges so that the edges can also be sorted in the order of how my nodes were sorted. After it does this, my *sort_edges()* function returns a sorted 2D vector of edges corresponding to the sorted nodes.

path.rs

In this module, I create another struct called **Path** where I calculate a BFS implementation for 2 nodes. I initialize my path with a VecDequeue and a reference to my graph. My most important function *calculate_distance()* calculates the distance it takes to get from one node/road to another. It does this by first creating a variable, **distance,** which is a vector that will contain the *Option* enum as the type. The main purpose of my **distance** variable is to store the distance from a start node to all the other nodes it can connect to. If there is no connection at all, that indexed position will be filled with “None” and if there is a connection, it will be filled with “Some(any distance it takes to from one node to another)”. I then proceed to sort my vector of nodes to then get an index of my starting node in relation to the sorted nodes. I then make sure that the node I put in is valid by checking if it is contained in my given nodes, if it is not I push the “None” value in my distance vector and return it. If my node passes the validation, I set the distance from my starting node to the starting node as 0. After doing all of this initialization, I finally add my start node to the queue and then create an edge vector that contains all of the corresponding edges to the sorted nodes. I then loop over all of the nodes in the queue while after analyzing them I pop them. After this, I loop over all of the edges that are direct connections to this node and update my **distance** variable to change the value from “None” to Some(+1). After this, I change my queue by adding the new node to the queue and starting this loop over again. After traversing through all of my nodes and their edges, the **distance** variable will contain the distance from the starting node to all of the other nodes that it has a connection to in my graph.

My second implementation of graph analysis, *maximum_distance_for_node(),* was a little bit more simple as I calculated the max distance a certain node would have when traversing through all of its connected components. I did this by using my *calculate_distance()* to get the distance from a certain node to every other node and using this, I obtained the largest number in the Some() value by doing .unwrap(). This largest number is then returned as the max distance.

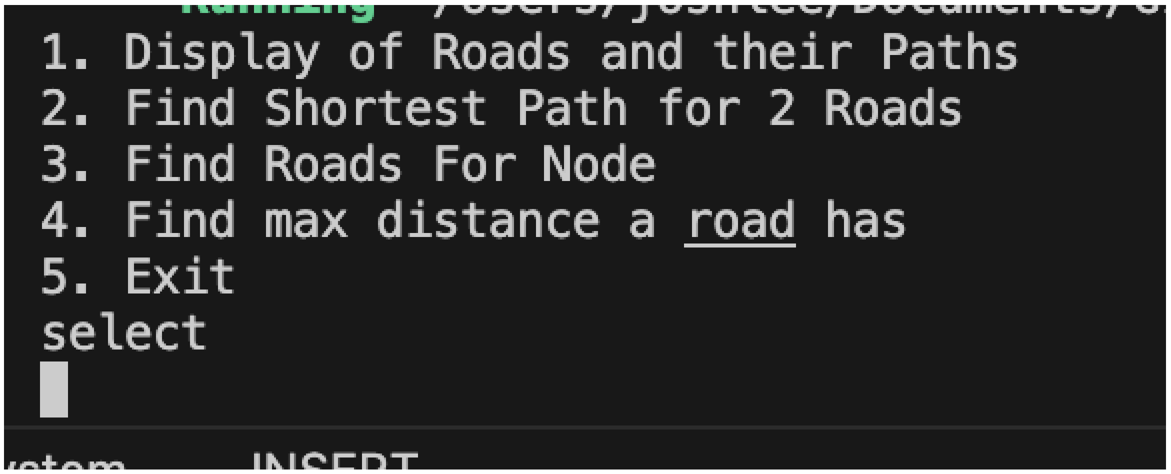
main.rs

My most important function *read_file()* takes in a txt file (in this case roadNet-CA (1).txt). After this, it proceeds to skip the first four lines as it is irrelevant to my project. After, it uses the shuffle function to shuffle the dataset as I only want to take the first 100,000-200,000 nodes and randomizing them is better than taking the first 100,000-200,000. ***My dataset has too many nodes and I am only taking a fraction of it because it would take too long to run on my original dataset.*** After this, this function takes the first and second column and makes a vector of nodes along with a vector of edges the nodes are connected/neighbors to. In my main function, I added some complexity by making the user input a number and depending on the number, it either prints out the graph itself, the edges corresponding to a certain node, the shortest distance from one node to another, the maximum distance a node has to any other node, or exiting and ending the program. My two functions *two_nodes()* and *display_edge()* both correspond to the number the user inputs in the beginning.

I will go over my main in more detail in the output analysis.

Output Analysis:

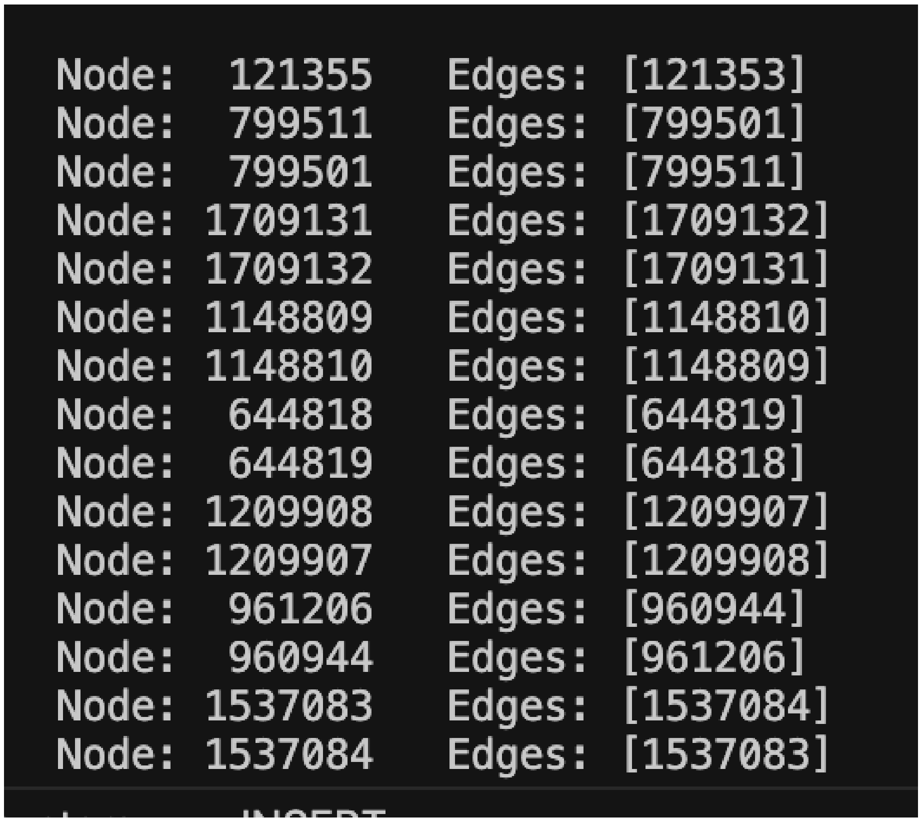
To run this project, use the command line *cd* to locate the file then finalize it by doing *cd src*. After that, you should be able to *cargo build* and then finally *cargo run —release* to get the output:



```
 1. Display of Roads and their Paths
 2. Find Shortest Path for 2 Roads
 3. Find Roads For Node
 4. Find max distance a road has
 5. Exit
 select
```

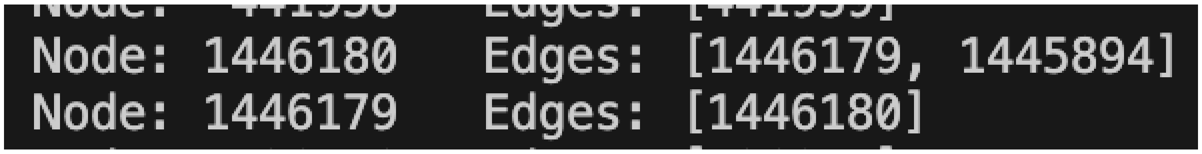
If the user inputs 1:

I used only 1000 nodes for this output because I wanted to save time and show just a small portion of my dataset in action.

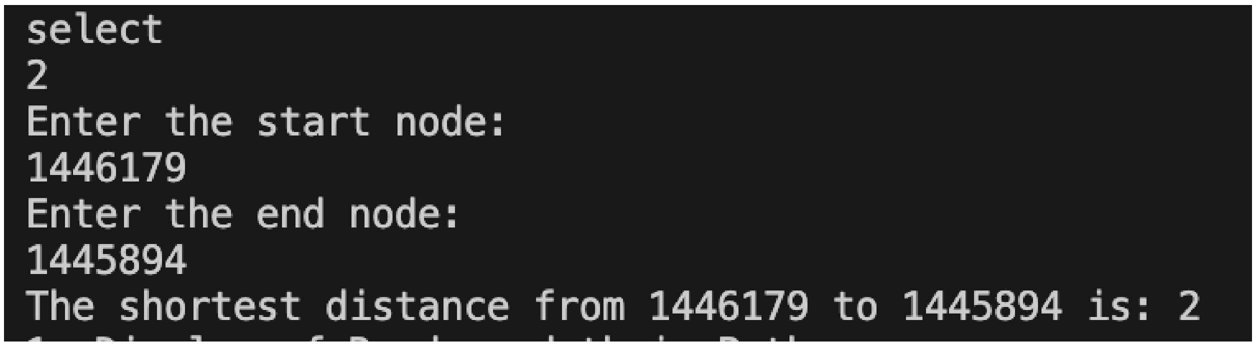


```
Node:  121355   Edges: [121353]
Node:  799511   Edges: [799501]
Node:  799501   Edges: [799511]
Node: 1709131   Edges: [1709132]
Node: 1709132   Edges: [1709131]
Node: 1148809   Edges: [1148810]
Node: 1148810   Edges: [1148809]
Node:  644818   Edges: [644819]
Node:  644819   Edges: [644818]
Node: 1209908   Edges: [1209907]
Node: 1209907   Edges: [1209908]
Node:  961206   Edges: [960944]
Node:  960944   Edges: [961206]
Node: 1537083   Edges: [1537084]
Node: 1537084   Edges: [1537083]
```

If the user inputs 2:

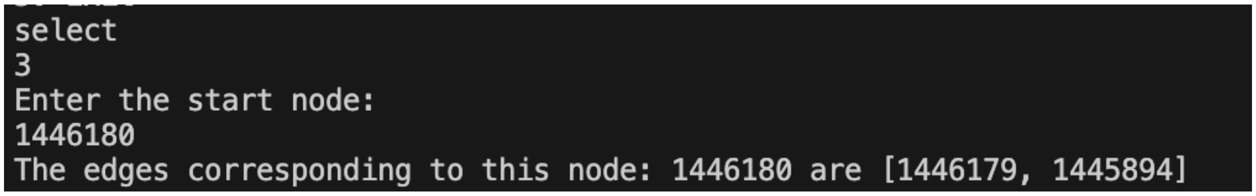


```
Node: 1446180   Edges: [1446179, 1445894]
Node: 1446179   Edges: [1446180]
```



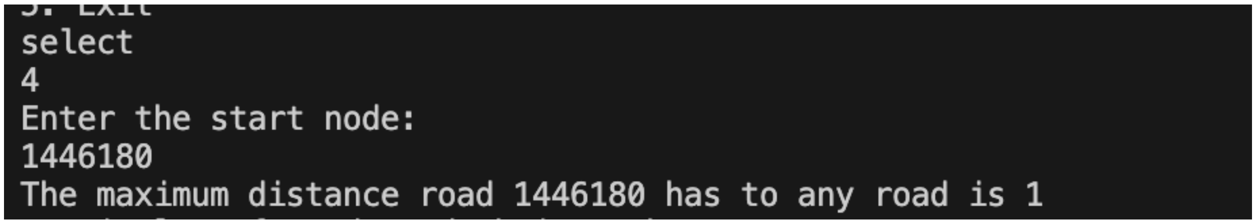
```
select
2
Enter the start node:
1446179
Enter the end node:
1445894
The shortest distance from 1446179 to 1445894 is: 2
```

If the user inputs 3:



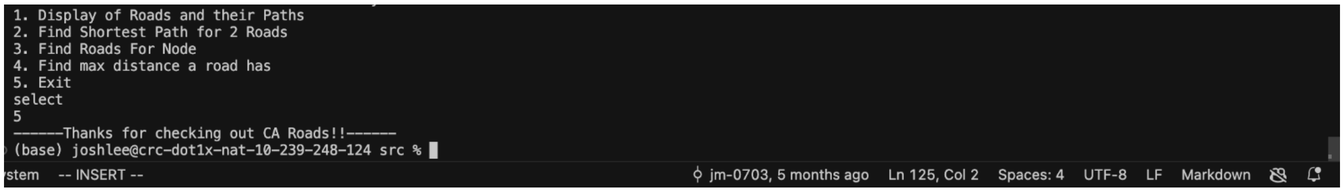
```
select
3
Enter the start node:
1446180
The edges corresponding to this node: 1446180 are [1446179, 1445894]
```

If the user inputs 4:



```
select
4
Enter the start node:
1446180
The maximum distance road 1446180 has to any road is 1
```

If the user inputs 5:



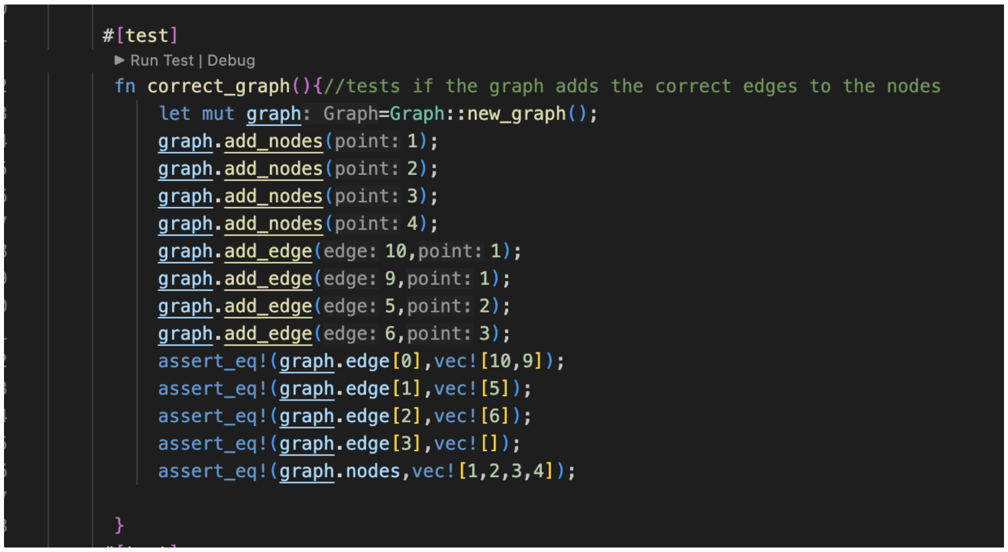
```
1. Display of Roads and their Paths
2. Find Shortest Path for 2 Roads
3. Find Roads For Node
4. Find max distance a road has
5. Exit
select
5
------Thanks for checking out CA Roads!!------
(base) joshlee@crc-dot1x-nat-10-239-248-124 src %
```

```
stem   -- INSERT --        jm-0703, 5 months ago   Ln 125, Col 2   Spaces: 4   UTF-8   LF   Markdown
```

As you can see, the user can see multiple implementations of my graph by selecting a number.

Tests:

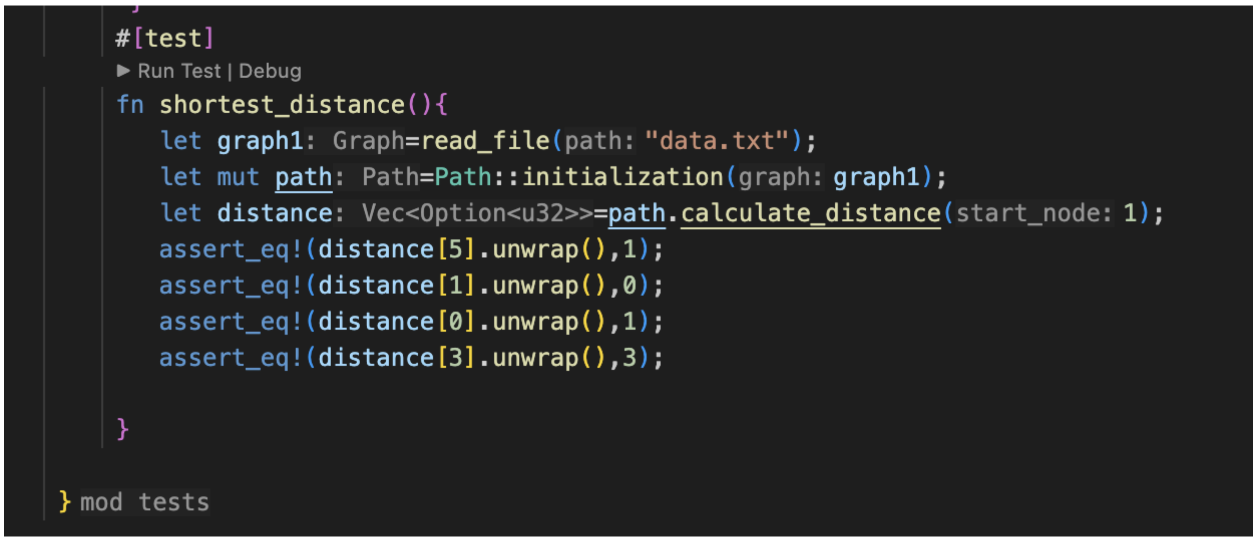
I have 2 tests:



```
#[test]
 ▶ Run Test | Debug
 fn correct_graph(){//tests if the graph adds the correct edges to the nodes
     let mut graph: Graph=Graph::new_graph();
     graph.add_nodes(point: 1);
     graph.add_nodes(point: 2);
     graph.add_nodes(point: 3);
     graph.add_nodes(point: 4);
     graph.add_edge(edge: 10,point: 1);
     graph.add_edge(edge: 9,point: 1);
     graph.add_edge(edge: 5,point: 2);
     graph.add_edge(edge: 6,point: 3);
     assert_eq!(graph.edge[0],vec![10,9]);
     assert_eq!(graph.edge[1],vec![5]);
     assert_eq!(graph.edge[2],vec![6]);
     assert_eq!(graph.edge[3],vec![]);
     assert_eq!(graph.nodes,vec![1,2,3,4]);

 }
```

This test makes sure that when I create a graph using *add_nodes()* and *add_edge()*, it has the correct corresponding edges to the nodes.



```
    #[test]
    ▶ Run Test | Debug
    fn shortest_distance(){
        let graph1: Graph=read_file(path: "data.txt");
        let mut path: Path=Path::initialization(graph: graph1);
        let distance: Vec<Option<u32>>=path.calculate_distance(start_node: 1);
        assert_eq!(distance[5].unwrap(),1);
        assert_eq!(distance[1].unwrap(),0);
        assert_eq!(distance[0].unwrap(),1);
        assert_eq!(distance[3].unwrap(),3);

    }

} mod tests
```

This test uses a smaller dataset which creates a graph and makes sure the shortest distances are correct.