A. Question I am answering: **How would it be possible to calculate relative carbon emissions for different states in terms of a centralized network format?**

Dataset: Link: https://www.kaggle.com/datasets/abdelrahman16/co2-emissions-usa/data

B. How I loaded it into Rust: Since the file was in standard CSV format, I uploaded it to my working directory on the cloud environment and simply used the csv crate in Rust to help me process the data.

Cleaning/Preprocessing: Since there were multiple different fuel types and sectors from which the fuel originated, I wanted to focus on a specific pair of fuel types and sectors. So I decided to take instances of data from the “Total carbon from all fuels” as the sector and the fuel type was “All fuels”. So, for each data instance, I would be paying attention to all the fuel produced by the country during that specific year. This was done in the graph.rs file when I was creating the graph from the already read data. Another thing I decided to do is that I compared these carbon emission values for location instances, and I filtered by time. I did this by having the user input a specific year from standard terminal input, and then my algorithm processed locations from that specific year. This was also done in graph.rs. If a user is interested in year-over-year changes, they can simply enter a different year and compare the outputs visually. In summary, I applied two filters, one that only cares about a specific sector and fuel, and the other is to restricts my algorithm to run on a specific year.

C.

Modules: So I had 2 modules. One is the Graph module, and the other is the data processing module. I created these 2 separate modules because I wanted to divide the workflow of the project. I thought it would be good to handle the data reading portion in one module, then I would handle all the graphical work/ graph filtering in the other module. As a

result, it makes it straightforward for me to call my functions in the main method. In my data processing module, the focus was to simply load the CSV data into a dataframe structure successfully. In my graph module, the goal was to filter the data in the CSV file based on the categories that I cared about and to form the adjacency list for the corresponding filtered data. After that, I performed my one-hot encoding on the nodes and implemented Dijkstra's algorithm along with closeness centrality calculation, all in the Graph module. This is the overall pipeline of the graph module.

Key functions and Types:

- Graph.rs(mod)
  - Node struct
    - Represents a node in my graphs, containing a location, year, and an emission variable each
  - State struct
    - This struct is another representation of the nodes, but for ease of use with Dijkstra's algorithm. It contains a cost variable and a position index.
  - Spatial Graph struct
    - This is my graph structure, which contains a variable to store the adjacency list, which is a hashmap mapping a node to a vector of its neighbor,s and each neighbor is represented by a node as well.
    - new() func
      - The new function creates an instance of the spatial graph, filling up the edge list along with the adjacency list. It takes in a dataframe and a specific year value. The function first iterates through the dataframe and parses the “year” column. Then I iterate row-wise along the year column. I extract the corresponding “location” and “emission” values in the row with the matching years(one that matches to the function parameter). I then apply my filters, which are just if statements, to see if these rows have “Total carbon emissions from all sectors” and “all fuels” as sector and fuel types. If they do, then I initiate a node. If a node meets these categories and is instantiated, then I run an annotated nested loop to create an edge between other nodes that meet these filters. As a result, this will create a complete graph. For every valid pair of nodes, I would represent the weight of the edge as the difference between the emission values, and I would populate the edge list. After I created the edge list, I created a corresponding adjacency list just by iterating through the edge list and mapping the nodes. I was able to do this since I used a hashmap for the adjacency list. After all this, i returned an instance of Spatial Graph with the edge list and adjacency list.
    - one_hot_encode_adjacency_list() func

        - The goal of this function is to turn the Nodes in the hashmap to simply numbers(u32) and put them in a vector format instead of a hashmap so I could perform Dijkstra's algorithm efficiently. This function returned the adjacency list in vector form, along with a hashmap of nodes: id numbers. In this function, I first created a hash set to represent all the unique nodes in the graph. Then I created the node: ID(u32) mapping and stored that in a hashmap. Then I iterated through the original adjacency list and used the hashmap i just created to get the index values to store those into a new empty vector. I also did this for all the neighbors.
    - Implement_dijkstars() func
        - This function implements Dijkstra's algorithm to find the shortest distances from one node to the rest, so I can use this information to calculate the closeness centrality of each node. This function takes in the one-hot encoded adjacency list and a starting node, and outputs a distance vector. This function first initializes a distance vector with high values, and the starting node has a distance of 0. Then I create a binary min heap and push the state struct to the binary heap. The state struct contains the cost, which is the weight, along with the position. Then I have a while let statement for the binary min heap and pop the minimum. If the cost of the current state is greater than its current distance in the dist array, we skip the iteration. Else, we update the distances to all of its neighbors if this new path is less than the current distance and push the neighbors to the heap. After that, i return the final distance array.
    - calculate_closeness() func
        - This function calculates the closeness centrality for all the nodes and returns a vector with the centrality measure for all nodes. It takes in an adjacency list. It then runs Dijkstra's algorithm starting from every node, then calculates the closeness centrality metric based on the distance array output.
- dataprocessing.rs(mod)
    - Column struct
        - Represents a column in the dataframe, storing a label for the column along with the data in the column.
    - Dataframe struct
        - This stores a Vector of columns, representing the entire dataframe. It is wrapped in an Option so that it can be initialized as none in the beginning
    - new() func
        - This simply initialized the dataframe as empty so that it can read the data later through read_csv

- read_csv() func
    - Based on the format from the starter file provided for the previous homework. It takes in the vector of encoded types and a file path. The read_csv function creates a csv reader builder instance that is formed from the file path. I had to use CARGO_MANIFEST_DIR in order to make reading from the file compatible with tests as well. After that, the function iterated through the rows of the CSV, and stored the first row into a column names vector. After that, for every element in every row, the function checked the corresponding type from the types array and wrapped the result in the ColumnVal() enum and pushed it to a row vector. After the end of this iteration, each row was pushed to a bigger rows vector. Then, for every column in the column names and for every row in the rows, I pushed the elements into a vector to store everything column-wise. After that, I put this vector and column name into the Column struct, and I updated the vector of columns in the dataframe correspondingly.
- ColumnVal enum
    - This enum has variants that represent 4 distinct types which can help be store the elements of the dataframe.
- Myerror type struct
    - This tuple struct was taken from the stater.rs file provided from previous homework, and it helps with error handling from the read_csv file to prevent the program from crashing.

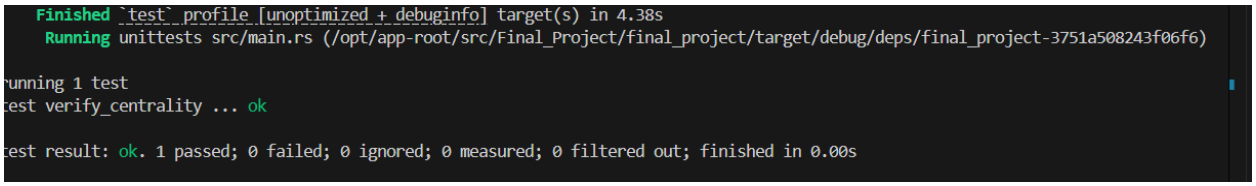
Main Workflow

- Fn main()
    - The user is first presented with a prompt to enter a valid year that they would like analyzed. After that, the program reads the year and parses it into an i64 and stores it. After that, I create a new dataframe and call the read_csv() function with the corresponding file name and types array. Then I created an instance of the spatial graph by passing in the dataframe along with the parsed year. After that, I one-hot encoded the adjacency list. Then, after that, I calculated the closeness centrality and stored that in a vector. Then I printed the corresponding closeness centrality for each node to the terminal output.

D. Tests

- Verify_centrality() func
    - This test function checked the closeness centrality on a small dataset with only 3 nodes, and I manually calculated the closeness centrality for each node. I did plus or minus epsilon to check if the computer output was in range, and if it was,

the test would pass. The function follows the same structure of the main function, more or less, but does the range check and reads from the smaller CSV.



E.

Main terminal output/interaction:

Please enter a specific year you're interested in anazlying Ex. 2000. Allowed years include(1970-2021
1985
Node { location: "Alabama", year: 1985, carbon_emission: 101.6322 }, Closeness Centrality: 0.00643884526522685
Node { location: "New Hampshire", year: 1985, carbon_emission: 13.261787 }, Closeness Centrality: 0.005948034981190529
Node { location: "Arkansas", year: 1985, carbon_emission: 49.14672 }, Closeness Centrality: 0.006668675470471052
Node { location: "Maryland", year: 1985, carbon_emission: 64.845451 }, Closeness Centrality: 0.0067565237497820485
Node { location: "New Jersey", year: 1985, carbon_emission: 112.20817 }, Closeness Centrality: 0.006243846450587642
Node { location: "District of Columbia", year: 1985, carbon_emission: 4.73737 }, Closeness Centrality: 0.005689665847404097
Node { location: "Connecticut", year: 1985, carbon_emission: 37.867124 }, Closeness Centrality: 0.006508812329509959
Node { location: "Maine", year: 1985, carbon_emission: 14.771797 }, Closeness Centrality: 0.005990232491806437
Node { location: "Mississippi", year: 1985, carbon_emission: 43.710564 }, Closeness Centrality: 0.006597978134986134
Node { location: "South Dakota", year: 1985, carbon_emission: 11.04116 }, Closeness Centrality: 0.005884031552899993
Node { location: "Michigan", year: 1985, carbon_emission: 166.725149 }, Closeness Centrality: 0.005210886499556337
Node { location: "Georgia", year: 1985, carbon_emission: 136.610775 }, Closeness Centrality: 0.005761855768527749
Node { location: "United States", year: 1985, carbon_emission: 4602.38824 }, Closeness Centrality: 0.00022162906202860555
Node { location: "Iowa", year: 1985, carbon_emission: 57.849266 }, Closeness Centrality: 0.006741155643449653
Node { location: "Utah", year: 1985, carbon_emission: 36.87705 }, Closeness Centrality: 0.006490768942698136
Node { location: "Virginia", year: 1985, carbon_emission: 82.420565 }, Closeness Centrality: 0.006663854164803565
Node { location: "Indiana", year: 1985, carbon_emission: 182.895788 }, Closeness Centrality: 0.004918343330229402
Node { location: "Missouri", year: 1985, carbon_emission: 100.66423 }, Closeness Centrality: 0.006451459987709682
Node { location: "Rhode Island", year: 1985, carbon_emission: 8.468955 }, Closeness Centrality: 0.005806259286679104
Node { location: "New York", year: 1985, carbon_emission: 187.919613 }, Closeness Centrality: 0.004829430851602547
Node { location: "Kentucky", year: 1985, carbon_emission: 107.711552 }, Closeness Centrality: 0.006334514593780389
Node { location: "Alaska", year: 1985, carbon_emission: 28.984221 }, Closeness Centrality: 0.006335507537737422

Node { location: "Nebraska", year: 1985, carbon_emission: 30.548067 }, Closeness Centrality: 0.006367670731392282
Node { location: "California", year: 1985, carbon_emission: 320.62755 }, Closeness Centrality: 0.0031076848869795296
Node { location: "North Dakota", year: 1985, carbon_emission: 38.221279 }, Closeness Centrality: 0.006514701436764265
Node { location: "West Virginia", year: 1985, carbon_emission: 102.207317 }, Closeness Centrality: 0.006430440854454619
Node { location: "Delaware", year: 1985, carbon_emission: 17.369309 }, Closeness Centrality: 0.006059984365981793
Node { location: "Oregon", year: 1985, carbon_emission: 26.881482 }, Closeness Centrality: 0.0062895061237436
Node { location: "Washington", year: 1985, carbon_emission: 59.54846 }, Closeness Centrality: 0.006750252228733559
Node { location: "Nevada", year: 1985, carbon_emission: 23.47013 }, Closeness Centrality: 0.006211115563720479
Node { location: "Arizona", year: 1985, carbon_emission: 61.078035 }, Closeness Centrality: 0.006755723048787086
Node { location: "Idaho", year: 1985, carbon_emission: 9.929281 }, Closeness Centrality: 0.005851006342734552
Node { location: "Florida", year: 1985, carbon_emission: 152.358689 }, Closeness Centrality: 0.005469900751821256
Node { location: "Tennessee", year: 1985, carbon_emission: 103.198138 }, Closeness Centrality: 0.006414413835669081 6
Node { location: "Minnesota", year: 1985, carbon_emission: 66.622979 }, Closeness Centrality: 0.006753343081405356
Node { location: "Oklahoma", year: 1985, carbon_emission: 84.14689 }, Closeness Centrality: 0.006645865015344662
Node { location: "North Carolina", year: 1985, carbon_emission: 107.215851 }, Closeness Centrality: 0.0063438886975425565
Node { location: "Colorado", year: 1985, carbon_emission: 61.525352 }, Closeness Centrality: 0.006756523749782047
Node { location: "Wisconsin", year: 1985, carbon_emission: 80.670785 }, Closeness Centrality: 0.0066791248427268465
Node { location: "Hawaii", year: 1985, carbon_emission: 17.014866 }, Closeness Centrality: 0.00605081025970699
Node { location: "Massachusetts", year: 1985, carbon_emission: 79.413137 }, Closeness Centrality: 0.006687937170880808
Node { location: "Ohio", year: 1985, carbon_emission: 241.22195 }, Closeness Centrality: 0.003991312396741374
Node { location: "Illinois", year: 1985, carbon_emission: 198.51798 }, Closeness Centrality: 0.0046430389461193065
Node { location: "South Carolina", year: 1985, carbon_emission: 54.071875 }, Closeness Centrality: 0.006714336235133237
Node { location: "Vermont", year: 1985, carbon_emission: 5.176967 }, Closeness Centrality: 0.005703651873024429
Node { location: "Texas", year: 1985, carbon_emission: 512.997091 }, Closeness Centrality: 0.0019887174797799264
Node { location: "Kansas", year: 1985, carbon_emission: 68.06627 }, Closeness Centrality: 0.006748184272210634
Node { location: "New Mexico", year: 1985, carbon_emission: 46.630001 }, Closeness Centrality: 0.0066380927940548375
Node { location: "Montana", year: 1985, carbon_emission: 22.145147 }, Closeness Centrality: 0.006179208081328651
Node { location: "Pennsylvania", year: 1985, carbon_emission: 250.961617 }, Closeness Centrality: 0.0038617940076221026
Node { location: "Wyoming", year: 1985, carbon_emission: 50.524977 }, Closeness Centrality: 0.006683128559417564
Node { location: "Louisiana", year: 1985, carbon_emission: 164.863567 }, Closeness Centrality: 0.005244804624882049

Interpretation: Now, for a specific year, we can see what countries are most involved and least involved in this emission network, and countries' relative performance, whether good or bad. California produces high carbon emissions, which is far from the center, having a low centrality measure. However, states like Colorado, Wisconsin, and Massachusetts have high closeness centrality, indicating they are the most centrality in this emission network. This states that states with high closeness centrality can be used as a pivotal marker for emission reduction strategies.

F.

How to build/run

- Simply do cargo run –release in the working directory for the code to run and follow the user prompt according to its rules
- The user prompt will ask to enter a year (1970-2021), so do as follows
- Does not take a long time to run/compile, especially with –release. It will take about a second

G.

No substantive AI use

Collaborators: None