DS210 Final Project Write-Up

Cooper Hassman

Introduction

In this project, I analyzed the North American Rail Network Lines - Passenger Rail View dataset from the US Department of Transportation. It can be viewed and downloaded here: https://geodata.bts.gov/datasets/usdot::north-american-rail-network-lines-passenger-rail-view-1/about. Each entry in the dataset represents a section of railroad used for passenger rail travel in the US, Mexico, and Canada. The dataset contains a total of 34 columns, which contain a variety of identifying information about the track section that it represents, such as length, county and state location, railroad and track ownership, and time zone. I did not use most of the columns in this dataset as they weren't useful in my analysis; the columns I used to construct my graph were "FRFRANODE," a numerical value representing the node where the track section begins, "TOFRANODE," a numerical value representing the node where the track section ends, and "STCNTYFIPS," which is a unique five-digit value that is the Federal Information Processing Standards code for each county in the United States.

Going into this project, I wanted to perform some analysis that could identify major transit hubs in the United States, or highlight where passenger rail infrastructure was needed. I wanted to perform a graph analysis where the nodes would be train stations or cities and edges would be the routes between them. I quickly started on creating the code for creating an adjacency list between nodes. However, I soon realized that the nodes of this dataset were not very intuitive to this type of analysis by themselves. Nodes not only represent train stations, but also where tracks change ownership or if there are any splits in the track. Therefore, the nodes couldn't be analyzed on their own as many of them were arbitrary and the connections between them didn't represent anything very meaningful to passenger transport. Furthermore, the vast majority of the nodes were only connected to two others (which node came before and after it on the train line).

I realized that I could generalize the analysis by grouping the nodes together geographically, so instead of pinpointing specific train stations that are more connected than others, I could find areas or regions in the United States that have higher connectivity. The only geographic marker that the dataset provides for each of its nodes is the county which it's in, represented by a 5-digit FIPS code. I then created an adjacency list of US counties that are connected to each other via passenger rail transit. With this adjacency list, I could perform analysis to find counties that are most connected and most vital to the US passenger rail network.

Code Explanation

I split up the functions of my code into several modules, the first of which is data_cleaning, which contains one function called parse_csv. Since my dataset was in the form of a CSV file, I was able to use a buffered reader to read the file line by line and parse it. The

outputted dataset is a vector of vectors with string values for all columns. During the reading of the file, the first line was skipped since it contains the column titles.

The next module was adjacency_lists, which contains three functions related to constructing and adjusting the adjacency lists of the dataset. The first function is build_node_adjacency, which takes a vector of strings as an argument (the format of the dataset), and returns a HashMap with keys corresponding to each node and with values corresponding to a vector of strings with all the nodes that the key is connected to within the transportation network. First, the function initializes an empty HashMap, then iterates through the rows of data and defines the from_node and to_node as the columns "FRFRANODE" and "TOFRANODE" respectively. It then uses the entry method to push the from_node as a key if it does not already exist, adding the to_node to the vector of connected nodes in the value. It then mirrors the relationship, ensuring that the to_node is also a key and that from_node is in its value of connected nodes since the dataset is undirected (i.e. the connections go in both directions).

Next, I defined the public function build_county_adjacency, which takes a data vector and node adjacency HashMap as its arguments. It outputs a HashMap with each county's FIPS code as its keys, and a vector of counties that each is connected to via rail as its values. I first initialized the county_adjacency HashMap, then initialized another HashMap called node_to_county which maps nodes to the county they're located in. Iterating through each row in the data, I inserted the "FRFRANODE" value as the key and the "STCNTYFIPS" column as the value. I then iterated through the items in the inputted node adjacency HashMap and used an if let statement to define a variable county representing the county location of the current node. Then, for each neighboring node in the node adjacency HashMap, I defined a new variable neighbor_county representing the county locations of all the neighboring nodes. If the node's county and the neighboring node's county is different, a new entry in the county_adjacency HashMap is added representing that the two counties are connected. The same logic is used as in the build_node_adjacency function with the entry method to ensure a two-way connection. The last few lines in the code serve to delete any repeating entries in counties' adjacent lists. I made use of the dedup method, which deletes identical entries in each of the vectors in the county_adjacency HashMap.

The final function in this module was remove_county, which is helpful later in the betweenness centrality analysis. The function takes a county FIPS ID string as well as an adjacency list HashMap, and returns a new adjacency list HashMap. Essentially, the function clones the inputted adjacency list and removes the inputted county from both its keys and its values. I made use of the retain method when iterating through the values, basically retaining every county that was not equal to the input.

The final module of functions is graph_analysis, which contains several functions related to analyzing the graphs constructed from the dataset. The first function is connectivity_analysis, which takes an adjacency list as its argument and returns a sorted list of nodes based on their number of neighbors. I used map to iterate over each (node, neighbors) pair in the adjacency list and replaced the list of neighbors with the length of the neighbors vector. Then, I sorted the

nodes by their number of neighbors in descending order, so the nodes with the most neighbors would be at the top.

The second function, bfs_component_size, uses Breadth-First Search to calculate the size of the connected component that a certain node belongs to. It takes three arguments: start, which is the starting node, an adjacency list, and a visited HashSet containing all the nodes that have already been visited. The function initializes a VecDeque and pushes the start node to the queue, then initializes a size variable to track the size of the connected component. It then begins the BFS algorithm by dequeuing a node. If the node hasn't been visited, the function inserts the node into the visited HashSet and adds one to the size variable. It finally adds all of that node's unvisited neighbors to the queue. This loop ends when the queue is empty, meaning that all the nodes have been visited that are within the connected component of the original node. The function returns the size of the connected component.

The next function is largest_connected_component, which takes an adjacency list as its argument and finds the largest connected component within the graph. It first initializes an empty HashSet and max_size variable, then iterates through each node in the keys of the adjacency list. If the node has not been visited yet, it uses the bfs_component_size function to find the size of its connected component. If its size is greater than the max_size variable, it is set to equal max_size. After iterating through all the nodes, the function returns the max_size variable.

The last function in this module is betweenness_centrality, which takes an adjacency list as its argument. The function computes a measure of betweenness_centrality and measures how removing a node from the adjacency list affects the size of the largest connected component. It first initializes a results vector and then iterates over each node in the graph, removes it from the graph using the remove_node function from the previous module, and computes the size of the largest connected component in the new graph using the largest_connected_component function. It then pushes the node and the largest component size once it's removed from the graph. Finally, the function sorts these tuples in ascending order by the size of the nodes' largest connected component, so we can see which nodes affect the connectivity of the graph the most.

In the main function, I first loaded my csv file and used the parse_csv function from the data_cleaning module to load it. Then, I used the build_node_adjacency and build_county_adjacency to construct the node and county adjacency lists from the dataset. I then used the connectivity_analysis function to find how many counties each county is connected to via passenger rail. I printed the FIPS code of the top 20 most connected counties, as well as the number of connected counties each has. Next, I used the largest_connected_component function to find and print the largest connected component of the graph without removing any counties. I then used the betweenness_centrality function to find the largest connected component after each county was removed from the system. I also printed the FIPS code of the 20 counties with the smallest largest connected component in the modified graph after their removal, also printing the size of that component.

In the tests module, I first imported the modules and libraries necessary to complete the tests, including all three modules I defined above. I created a test for each function I created to

ensure that they were functioning properly. The test_parse_csv function creates a sample CSV file and verifies that the correct number of rows are parsed, and that the specific values from the data are what is expected. The test_build_node_adjacency test creates mock data that represent node connections and verifies that the length of the neighbors is correct. The test_build_county_adjacency test creates sample data with sample nodes and counties, then verifies that the county connections are correct after building the adjacency list. The test_connectivity_analysis ensures that the function calculates the number of neighboring nodes correctly. Test_largest_connected_component creates a sample graph and makes sure that the largest component is calculated correctly. The test_remove_node function ensures that once a node is removed from a function, it is not present in the keys or values of the resulting adjacency list. The test_betweenness_centrality test creates a sample triangle graph and asserts that removing any node leaves the other two nodes connected.

Results and Discussion

Code Output:

Top 20 Most Connected Counties by Number of Neighbors:
County: 51153, Number of Neighbors: 5
County: 17031, Number of Neighbors: 5
County: 42045, Number of Neighbors: 4
County: 24005, Number of Neighbors: 4
County: 42101, Number of Neighbors: 4
County: 51183, Number of Neighbors: 4
County: 26159, Number of Neighbors: 4
County: 51015, Number of Neighbors: 4
County: 25017, Number of Neighbors: 4
County: 17067, Number of Neighbors: 4
County: 36081, Number of Neighbors: 4
County: 31025, Number of Neighbors: 4
County: 34023, Number of Neighbors: 4
County: 34027, Number of Neighbors: 4
County: 06071, Number of Neighbors: 4
County: 25021, Number of Neighbors: 4
County: 09009, Number of Neighbors: 4
County: 06077, Number of Neighbors: 4
County: 50027, Number of Neighbors: 4
County: 53033, Number of Neighbors: 3

Largest Connected Component without Removal: 322

Top 20 Counties by Smallest Largest Connected Component Size after Removal:
County: 08123, Largest Component Size After Removal: 161
County: 08001, Largest Component Size After Removal: 161
County: 08087, Largest Component Size After Removal: 162
County: 08059, Largest Component Size After Removal: 163
County: 08121, Largest Component Size After Removal: 163
County: 08125, Largest Component Size After Removal: 164
County: 08013, Largest Component Size After Removal: 164
County: 31057, Largest Component Size After Removal: 165
County: 08047, Largest Component Size After Removal: 165
County: 31087, Largest Component Size After Removal: 166
County: 08049, Largest Component Size After Removal: 166
County: 08037, Largest Component Size After Removal: 167
County: 31145, Largest Component Size After Removal: 167
County: 08045, Largest Component Size After Removal: 168
County: 31065, Largest Component Size After Removal: 168
County: 31083, Largest Component Size After Removal: 169
County: 08077, Largest Component Size After Removal: 169
County: 31137, Largest Component Size After Removal: 170
County: 49019, Largest Component Size After Removal: 170
County: 31099, Largest Component Size After Removal: 171

The results of this project have important implications for the passenger rail network in the United States. The initial connectivity analysis I performed found highly connected counties within the network, indicating that these counties have high levels of rail infrastructure. I manually looked at the locations of these counties based on their FIPS code to try and extract patterns from the analysis. 10 of the 20 most connected counties are located in the Northeast Corridor region, while all the others are located in California, Seattle, or Chicago. This aligns with my previous knowledge about passenger rail infrastructure being concentrated in the Northeast with pockets of infrastructure around major cities. However, this analysis cannot be completely taken at face value, since the number of connected counties by rail depends heavily on how many counties each county borders. A more meaningful measure that takes this variance into account would be to find the counties with the highest ratio of connected counties by rail to their total number of bordering counties.

The second part of my analysis identifies counties that are crucial in maintaining connectivity among the US passenger rail network. The largest connected component in the full graph without removing any counties is 322 connected counties. There were two counties that cut this largest component directly in half when they were removed from the system, both in the state of Colorado. In fact, thirteen of the top 20 counties with the highest impact are in Colorado, while six are in Nebraska, and one in Utah. Upon further research on the dataset's website, most

of these counties lie on a critical rail line that connects the East and West coasts of the US, as well as the northern and southern regions of the country. Without many of these counties, the rail system is significantly less connected. These results indicate that many rail lines in the states of Colorado and Nebraska serve as chokepoints in the passenger rail system. The dramatic cut in the US rail network's connectivity with the removal of a single county highlights the importance of these counties within the network as well as the system's vulnerability. If these rail lines were to be disrupted, there would be major disruptions to cross-country rail travel. However, it's also important to consider that most passenger rail travel in the United States is regional; most of it is centered around traveling between neighboring cities rather than traveling across the country. Nonetheless, these findings suggest that increased attention should be paid to these critical counties as they are crucial to the overall connectivity of the US passenger rail network.