Cecily Wang
Professor Kontothanassis
DS 210
December 12, 2023

California Road Network Project

Overview

This project aims to analyze routes within a road network represented by a graph data structure where intersections and endpoints are nodes, and the roads connecting these are undirected edges. In this project, I am using the California Road Network dataset from the Stanford Large Network Dataset Collection (SNAP) to find the shortest paths from randomly selected locations and compute measures for the network such as degree centrality, average distance, degree distribution, and graph densities.

Features

- **Graph Creation**: Generates an undirected graph representing the road network where intersections are nodes and roads are edges. The programs related to reading files/creating a graph read through each line of the file, ignoring any comment lines. It takes each line and splits it by whitespace, converting each piece into an integer. If a line has exactly two integers, it's assumed to be an edge in the graph, representing a connection between two nodes (source->src and destination->dst).

-**Centrality**:

In calculating each centrality for each node, I made a HashMap data structure in order to count the connections (the degree) for each node in the graph. It goes through each edge and increments the count of both the source and destination nodes by one. The degree centrality that I am using is a measure of how connected a node is in the network, so the more edges it has, the higher its centrality. The centrality program then sorts the degree counts in ascending order by the node identifier and prints out the degree centrality in descending order, showing the node with the highest number of connections first.

-Distribution of Centrality: The program is designed to analyze a road network and provide insights into how well-connected the nodes are, which can represent intersections or endpoints in a road system. This can be useful for tasks like urban planning or traffic optimization.

- `calculate_degree_centrality`: This function calculates the degree centrality of every node in the graph, which is a measure of how many connections (or "edges") each node has. It then returns a HashMap where each node is associated with its degree centrality.

- `shortest_path`: This function calculates the shortest path between a start node and a goal node using Dijkstra's algorithm. It returns an Option<Vec<usize>>, which is an optional vector of node indices. If a path is found, the vector contains the sequence of nodes from start to goal; otherwise, it's None.
- `degree_frequency`: This function computes the frequency of each degree of centrality (i.e., how many nodes have a particular number of connections) and returns this frequency distribution as a HashMap.
- main: It uses the load_road_network_from_file function to load the graph from a file, calculates the degree centrality for each node in the graph, computes the frequency of each degree of centrality, and then prints out this information. The program will print the number of nodes for each number of connections they have.

- **<u>Shortest Path Computation</u>**: Utilizes Dijkstra's algorithm to find the shortest path between two intersections.The code is structured to first preprocess data to make distance queries faster and then calculate an approximation rather than the exact shortest path to improve performance in scenarios where an approximation is sufficient.

- `select_city` function: Given a graph and a number k, this function selects k random nodes (cities) from the graph. It shuffles the list of all nodes and then truncates the list to keep only k nodes.

- `precompute_city_distances`: This computes the shortest paths from each of the selected cities to all other cities using Dijkstra's algorithm, which is a well-known algorithm for finding the shortest paths between nodes in a graph. The results are stored in a nested HashMap, which maps a city to another HashMap containing distances to all other cities.

- `approximate_shortest_path` function: To find an approximate shortest path between start_node and end_node, this function iteratively evaluates precomputed distances. It uses an alpha parameter to potentially scale the distances for approximation. The smallest distance obtained through this iteration is considered the approximate shortest distance.

- `load_road_network_from_file` function: This function reads a text file representing the road network where each line indicates a road between two cities (nodes). It skips lines starting with '#' (comments) and constructs an undirected graph with unit weights (weight of 1) for each edge (road).

- main function: It tries to load a road network from a specified file, selects a number of cities, precomputes distances between them, and then calculates an approximate distance between a start node and an end node. The resulting distance is printed out.

- **<u>Average Path Distance</u>**: This program is an example of how simulations can be used to estimate properties of complex systems like road networks by relying on random sampling and pathfinding algorithms. It calculates the average distance of all paths within the graph to gain insights into network efficiency. Contains the function `calculate_average_distance` which takes a reference to an UnGraphMap, the number of Monte Carlo iterations, and the sample size as parameters. The function calculates the average distance by generating random start nodes. Then for each start node, pick a sample of distinct end nodes and use Dijkstra's algorithm to find the shortest paths from each start node to the end nodes. Finally, we sum the distances and count the number of paths found.

- Module `average_distance`:This module contains a single public function:
  - `calculate_average_distance`:The function takes a reference to an undirected graph (UnGraphMap<usize, u32>), the number of Monte Carlo iterations to run (monte_carlo_iterations), and the size of the sample of end nodes for each start node (sample_size).
  - The goal of Monte Carlo simulations is to estimate the average distance by randomly sampling paths in the graph. It verifies that there are at least two nodes in the graph to calculate distances.For each iteration, it randomly selects a start node and a distinct set of end nodes and computes the shortest paths between them using Dijkstra's algorithm. Then it sums up the distances and counts the number of paths found. Finally, it returns the average distance as an Option<f64> (and if no paths are found, it returns None to indicate the absence of an average distance).
  - Function load_road_network_from_file: same as the other load road text file function.
  - The main function is the program's entry point. It loads the graph from file "california_roads.txt". It sets the number of Monte Carlo iterations and sample size. It then invokes the `calculate_average_distance` function from the average_distance module with the loaded graph, number of iterations, and sample size. Finally, it prints out the average distance if it could be computed.

- **Graph Density**: This code is using data structures to represent a graph (nodes, edges, and adjacencies), processes the data from a file to populate those structures, and applies algorithms to analyze the graph in terms of density, which is an indicator of how closely knit the graph is. The main output includes the density of the graph as a whole and the densest subgraph within it. This is useful because it can provide important insights into the infrastructure's efficiency and connectivity. The function `compute_densest_subgraph` is particularly useful in identifying sub-networks that are highly connected, which might be either at risk of high traffic load or inherently important to network connectivity.

- `calculate_density` Function: This function takes a set of nodes and a list of edges as input and calculates the density of this part of the graph. In graph theory, density is the ratio of the number of edges to the number of maximum possible edges in a graph.
- `compute_densest_subgraph` Function: This function determines which subgraph (a smaller graph within the larger graph) has the highest density. It does this by iterating through each node in the adjacency map's keys, generating potential subgraphs, and computing their densities, ultimately keeping track of the densest one.
- `edges_in_subgraph` Function: This helper function generates a list of all the edges that are fully contained within a given subgraph, by checking the provided adjacency map for connections between nodes in the subgraph.

- main Function: Reads from the file then initializes empty sets and a map to keep track of nodes, edges, and an adjacency map (showing which nodes connect to which). Then it iterates through each line of the file, ignores comments (lines starting with '#'), and parses the nodes from each line, updating the sets and map accordingly. Finally it calculates the overall density of the graph and prints it as well as finds and prints the densest subgraph and its density.

How to Run the Project

1. Make sure to have Rust and Cargo installed.
   Under `Dependencies` in the .toml file we add in:

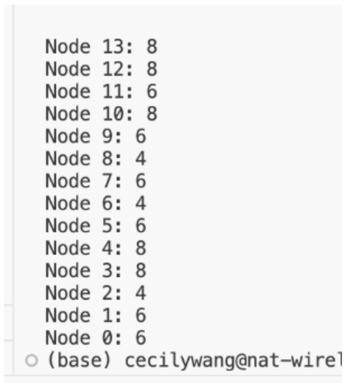
```
petgraph = "0.6"
rayon = "1.5.0"
rand = "0.8"
itertools = "0.10.1"
```

2. Clone/download the project.

3. Navigate to the project directory and run cargo build to compile the project.
4. You can then run the project using cargo run.
5. Be sure to have the **“california_roads.txt”** file in the same directory for all features/files that calculate different measures of the network.

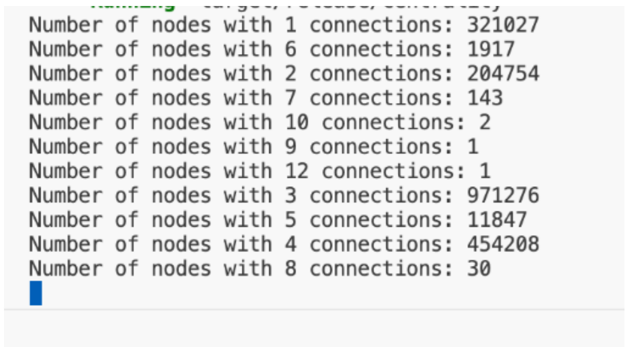
Outputs for Each Measure:

-In the **degree centrality:** The degree centrality of a node is a measure of the number of direct connections (roads) that a node (intersection or endpoint) has to other nodes. The more roads a particular intersection is connected to, the higher its degree centrality. So in my code output here, we see that Node 2 for example has a centrality of 4. This indicates that Node 2 could be a four way intersection.



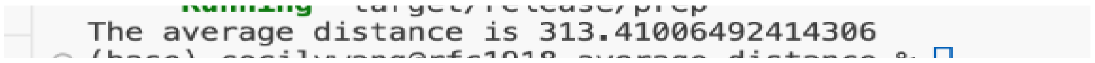
```
Node 13: 8
Node 12: 8
Node 11: 6
Node 10: 8
Node 9: 6
Node 8: 4
Node 7: 6
Node 6: 4
Node 5: 6
Node 4: 8
Node 3: 8
Node 2: 4
Node 1: 6
Node 0: 6
(base) cecilywang@nat-wirel
```

-In the **frequency of degree centrality**: By looking at the output, we can see that most of the nodes have four connections. This can be attributed to the fact that the California road network may have many four way intersections, which makes sense because they are the most common intersection to have in the US. One thing that I found interesting from the output is the singular node with 12 connections. This makes me curious as to what this would look like in real life, or if it may be a highway merge of some sort.



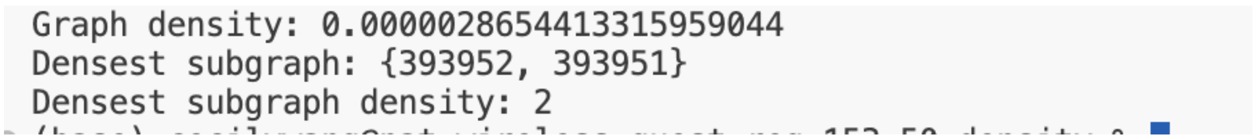
```
Number of nodes with 1 connections: 321027
Number of nodes with 6 connections: 1917
Number of nodes with 2 connections: 204754
Number of nodes with 7 connections: 143
Number of nodes with 10 connections: 2
Number of nodes with 9 connections: 1
Number of nodes with 12 connections: 1
Number of nodes with 3 connections: 971276
Number of nodes with 5 connections: 11847
Number of nodes with 4 connections: 454208
Number of nodes with 8 connections: 30
```

-In the **average distance**: We are estimating the average distance between nodes in a graph by simulating random road trips between places in California's road network using Monte Carlo sampling. Here we see that in this code run, we got an average distance of about 313.



```
The average distance is 313.41006492414306
```

-**Graph density**: The value 0.0000028654413315959044 is a very low graph density, suggesting that the California roads network is very sparse with many more possible connections between intersections than actual roads.

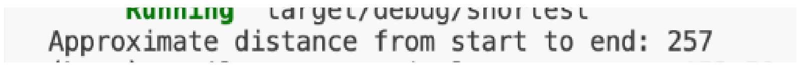


```
Graph density: 0.0000028654413315959044
Densest subgraph: {393952, 393951}
Densest subgraph density: 2
```

- **Densest Subgraph:**
    - The "densest subgraph" refers to a smaller network within the larger network that has the highest density. In road terms, it would be a part of the network where there are many roads packed closely together. In my output, {393952, 393951} refers to the IDs of nodes (intersections) that form the densest subgraph found by the program. There are only two nodes in this subgraph. This could be a large part of the highway system in California and one interesting measure that could be computed in the future is looking at the traffic efficiency of this densest subgraph point in order to figure out where one could break it down and enable for easier and more efficient access in the future.
- **Densest Subgraph Density:**
    - The "densest subgraph density" is the density calculation applied to the densest subgraph. A density of 2 in this context seems strange because, usually, subgraph density should be between 0 and 1. However, considering that density is calculated with $D = 2E / (N*(N-1))$, if there are multiple edges (roads) directly between the two nodes (intersections) that comprise the subgraph, the density can be greater than 1. This could indicate the presence of multiple road segments directly connecting to two intersections (perhaps parallel roads merging).

-**Shortest distance between cities:** Utilizes Dijkstra's algorithm to find the shortest path between two intersections. Here we are finding out how busy or important an intersection is within the network based on how many roads lead directly to or from it. An intersection with a high degree of centrality could be a major junction or hub in the road network. This concept is useful in network analysis, helping to identify key nodes that are central to the network's connectivity.

Reminder: because we are grabbing from randomly selected nodes, the distances will vary each time the code is run.



```
     Running `target/debug/shortest`
Approximate distance from start to end: 257
```

Testing

To run the tests, use the command `cargo test` from the terminal within the project directory. The test will automatically build and run.
*** Something that I found fun later on when I was in the latter half of building my project was that I could put the tests separately in a different file and run it from the main. I did this for my `shortest_distance` which helped me visually de-clutter which is something that I struggled with when keeping track of my code.

Conclusion/Wrapping Thoughts

This project provides a foundation for road network analysis and route optimization, which can be built upon and extended. It illustrates important concepts in graph theory and their applications in real-world scenarios like urban planning and traffic management. Further development may include more advanced optimizations, dynamic routing updates, and integration with live-streaming traffic data.

One issue that I found in my project that I wish I could have simplified was running a main function in each file. I started doing this because when calculating the centrality for EACH node, I would get a long output that would overshadow any other main function calculation.

The reason that I generated random samples to calculate the shortest path and average path distance was due to the large amount of data. That is something that I would want to work on more in the future, is figuring out how I can create an efficient and optimal program for this dataset to easily be able to configure the shortest distances from *any* point, not just the randomly sampled nodes that made my code compile in the first five minutes.

One thing that I regret doing in my project is the redundant file reading, I wish I could have come up with a singular module that could be shared amongst *all* of the types of measures that I did for the network, but one thing that threw me away from doing this was that some of the files needed to read the file differently, such as the ‘average distance’ file. Another thing that I regret was how long it took for the `average distance` function to run which is why I shifted my idea in calculating this to using Monte Carlo iterations instead of trying to calculate every distance for all nodes.