Overview:
Goal: Find the nodes with the highest closeness centrality, find the average distance between nodes in the dataset.
Dataset: Amazon product co-purchasing from March 2nd, 2003. Accessed from Stanford Network Analysis Project: https://snap.stanford.edu/data/amazon0302.html, 262111 nodes.

Processing:
I didn't change anything about the dataset except remove header lines I didn't need. I wrote a function similar to one from a previous homework assignment to turn it into an adjacency list.

Code Structure:
Modules: I split the code into modules called 'bfs', 'histogram', 'filereader', and 'closeness'. The filereader module just contains the function that reads into an adjacency list. It isn't really related to any other function so I made it its own module. The bfs module contains the bfs function and the avg_distance function, which basically descends from the bfs one. The histogram module contains any function that works with the histogram, and the closeness module has all 4 closeness functions.

Functions:

- read_graph
    - Purpose: loads the dataset in to be worked with
    - Input: path, output: Vec<Vec<usize>> (adjacency list)
    - Logic: uses BufReader to go through each line, splitting at whitespace and parsing into floats, then collecting so there is a vector of tuple edges (from, to). Then, these are each processed into the adjacency list, where the index corresponds to the "from" node and the inner vector contains each "to" node.
- bfs
    - Purpose: Performs breadth-first search on a given start node, finding the shortest distance to each other reachable node
    - Input: reference to djacency list (vector format) and start node (usize) Output: Vec<Option<usize>> where the index is the end node and the vector either contains Some(distance) or None if it is unreachable.
    - Logic: First initializes the distance vector to return, also initializes a vector that is the length of the graph that just stores a boolean value, this tracks whether each node has been visited. Also initialize a queue, which is used to store nodes that have been checked. This is iterated through to find neighbors of the already visited nodes. Set the conditions for the start node to be correct (0 distance, visited true). Start a while loop that runs as long as something is in the queue. For each node in the queue, for each neighbor of that node, check if it is visited, if so, ignore that node. If not, mark the node visited and mark the distance as one more

than the neighbor it traced from. Put this node in the queue. Return the distance vector.

- avg_distance
    - Purpose: Get a quick one-number summary of the distances in the graph.
    - Input: reference to adjacency list (vector format) and sample size (usize given by user) Output: f64 average distance.
    - Logic: Start a random sampler, filter out nodes without edges, and sample however many nodes the user specified (each of these steps is repeated in other functions as well). Initialize a total distance and count each at 0. Call bfs on each node in the dataset, then iterate through the vector of distances it outputs. Increment total distance by each distance and count by one for each. Return their quotient as an f64.
- avg_distance_histogram
    - Purpose: Give a more detailed view of the distances between nodes present in the dataset.
    - Input: reference to adjacency list (vector format) and sample size (usize given by user) Output: HashMap using distance keys and count of each distance as values.
    - Logic: Works almost the same as avg_distance, but instead of initializing and incrementing those counts, initialize a HashMap, check the distance, and increment the corresponding count.
- print_histogram
    - Purpose: Helper function for avg_distance_histogram. I wrote them separately because I wasn't sure whether I was going to use rust for visualization or export and use python.
    - Input: The HashMap given by avg_distance_histogram Output: Nothing, but prints to terminal.
    - Logic: Use .iter().collect() to change the HashMap to vector and sort it. (not sure if it would be better to do this in the other function). Figure out what the maximum count for any distance is. Set a maximum bar length (I chose 80 repeated characters so it would display well in a quarter of my screen). Then, iterate through the sorted vector. For each distance, normalize the bar length from the count, then use std::iter::repeat().take(bar_len) to create a repeated sequence of *'s and take the right number of them. Print the distance and corresponding bar. I chose not to print empty bars so the histogram is more readable, but that could easily be changed if I wanted to know the full range of distances that are present.
- reverse_al
    - Purpose: Reverse an adjacency list, useful for calculating in_closeness.
    - Input: reference to adjacency list Output: adjacency list (vector)

    - Logic: Make a new vector of the same length. For each neighbor in the original list, push the index of that neighbor in the original vector into the new vector index corresponding to the neighbor. Return the vector.
- out_closeness
    - Purpose: Calculate the out_closeness of a node in a graph.
    - Input: reference to adjacency list, start node: usize Output: f64 closeness score
    - Logic: Call bfs on the start node. Initialize a count and total distance, iterate through all of the distances given by bfs, incrementing the corresponding counters. Return the closeness score (count/total_distance).
- in_closeness
    - Works the exact same as out_closeness but reverses the adjacency list before running
- get_all_out_closeness and get_all_in_closeness
    - Purpose: Calculate the closeness for all nodes in a given graph.
    - Input: reference to adjacency list, sample size Output: vector of tuples (node, closeness)
    - Logic: Does the same sampling process present in bfs, then makes an empty vector, calls a closeness function on each node, and pushes the node and result into the vector as a tuple. Return that vector.

Types: I didn't really use any custom types, it didn't seem necessary for my code. The graph could be represented as a vector without any trouble, so I didn't write an "adjacency list" struct or anything.

Workflow: First, the function read_graph loads the dataset into an adjacency list. The bfs function that accepts the adjacency list is the heart of my code. That returns all shortest distances given a start node, which is used in the avg_distance function and closeness functions as well as to build a histogram of distances between nodes. That histogram is produced with a function called avg_distance_histogram, which feeds into print_histogram, a function that does what it sounds like.

Tests:
Output:
running 3 tests
test closeness::tests::test_out_closeness ... ok
test closeness::tests::test_reverse_al ... ok
test histogram::tests::test_avg_distance_histogram ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

What they do: Each test basically initializes a small graph then calls the function on that test graph and asserts that they output what I hand-calculated. One test checks that the reverse_al function works properly, it should just reverse all edges. This is important for in_closeness. The other test checks the out_closeness function because it utilizes the bfs function, so it tests that implicitly as well. It is also so similar to in_closeness and is called in get_all_out_closeness (and similarly in get_all_in_closeness), so it is important to test. Basically every function is tested in part through these two tests (all closeness functions, bfs, avg_distance). The final test checks that histograms are being built properly. This is important because I wanted to make sure that I wasn't double counting edges or anything so my histogram is as accurate as possible.

Results:

Output:

What sample size would you like to use? Note: higher sample size increases runtime
5000
sample_size: 5000
Average shortest path for 5000 samples: 19.95
Distance:
7: *
8: **
9: ****
10: ********
11: **********
12: **********
13: **********
14: **********
15: **********
16: **********
17: **********
18: **********
19: **********
20: **********
21: **********
22: **********
23: **********
24: **********
25: **********
26: **********
27: **********
28: **********
29: **********
30: *********
31: *******
32: *****

33: ***
34: **
35: **
36: *
37: *
Distances with very low counts are omitted
Top 10 In Closenesses:
(0, 1.0)
(3591, 0.11135250803833344)
(5765, 0.10393221475644514)
(16587, 0.09322678652475679)
(5442, 0.09321352313990511)
(3660, 0.09188613544163221)
(6067, 0.09128147235001727)
(7, 0.09075889563988356)
(450, 0.09059198197782976)
(13138, 0.08955519114064296)
Top 10 Out Closenesses:
(109348, 1.0)
(210137, 1.0)
(244446, 1.0)
(196271, 1.0)
(201237, 1.0)
(27861, 1.0)
(48920, 1.0)
(253119, 1.0)
(111140, 1.0)
(40989, 1.0)
Elapsed: 538.677296458s

Interpretation:
Each node corresponds to a product. If I was marketing, I would understand that the vast majority of nodes are not closely related. 19-20 steps is a lot. I would also understand that the products with the highest out closeness are good at getting people to buy additional products, while those with the highest in closeness are frequently bought as a complement. This might change the strategy with which I market my products.

Usage Instructions:
Make sure the dataset is in the project file, it should be that way in GitHub. Run with cargo run or cargo run --release (recommended). After you run, input a sample size. Using a sample size of 100 will give a good idea of how it runs and take about 10 seconds using --release. My output was calculated with a sample size of 5000, which took 9 minutes.

Citations:

First one: I asked “How do I reverse my adjacency list to compute in closeness?”

Second one: I asked: “How do you calculate closeness centrality?”

Also got this snippet: std::iter::repeat("*").take(bar_len).collect::<String>() to display my histogram in the terminal, but I lost the interaction.

1. **Create an empty adjacency list for the reversed graph:**

rust — Copy code

```rust
let mut reversed = vec![Vec::new(); adj_list.len()];
```

2. **Loop over all nodes and their neighbors**: For each edge `u → v`, add `u` to `reversed[v]`.

rust — Copy code

```rust
for (u, neighbors) in adj_list.iter().enumerate() {
    for &v in neighbors {
        reversed[v].push(u);
    }
}
```

3. **Use the reversed list in your BFS** Now you can pass this `reversed` list into your `bfs()` function to compute distances *to* a node, rather than *from* it.

For a node `u`, closeness centrality is often defined as:

python — Copy code

```python
C(u) = (reachable_nodes) / sum of shortest distances from u to all
```

In a directed graph:

- You may want to compute both **out-closeness** (reachable nodes from `u`) and **in-closeness** (nodes that can reach `u`).