Project explanation:

My project aims to look at the connectivity of the Amazon product network by calculating the average shortest path length between nodes. The analysis provides insight into the closeness of products in terms of co-purchasing behaviors. The dataset consists of roughly 403,394 nodes and 3,387,388 directed edges with detailed metrics like clustering coefficients and network diameters. My method involves extracting data from a compressed file, constructing a directed graph, implementing a Breadth-First Search (BFS) algorithm, and calculating average distances using the BFS results. The question I tried answering by doing this is: what is the average shortest path between nodes in the Amazon product co-purchasing network? I hope to determine how closely products are related in terms of co-purchasing behavior.

Coding explanations:

```
//main.rs
mod graph_ops;  // Import the graph operations module that includes
functions to manipulate and analyze graphs.
mod metrics;    // Import the metrics module that calculates various graph
metrics like centrality.
mod stat_util;  // Import the statistics utility module for statistical
calculations like median and standard deviation.
use std::env;   // Import the environment module to interact with the
environment in which the program runs.
```

Importing three modules (graph_ops, metrics, stat_util) and one standard library module. The first three modules contain specific instructions and data structures for graph analysis and statistical computations used in this project. The last module is used to read the environment variables such as in this case the current directory.

```
fn main() {
   println!("Current directory: {:?}", env::current_dir().unwrap());  //
Print the current working directory to help in troubleshooting paths.
```

In the start of the main function, it prints the current directory where the program is running. Using env::current_dir().unwrap(), this retrieves the current directory and unwraps the result, assuming there is no error.

```
   let graph = graph_ops::construct_graph("../amazon0601.txt.gz");  //
Construct a graph from a gzipped text file specifying edges.
```

Creates a graph data structure by reading from a gzipped file that contains edge data. The function construct_graph from the graph_ops module is used here, and it handles file reading, decompression, and graph construction.

```
   println!("Number of nodes: {}", graph.node_count());  // Output the
number of nodes in the graph to the console.
   println!("Number of edges: {}", graph.edge_count());  // Output the
number of edges in the graph to the console.
```

Printing the number of nodes and edges in the graph, helping to understand the graph's size and complexity.

```
   graph_ops::analyze_degree_distribution(&graph);  // Analyze and print
the degree distribution of the graph.
```

Calls the function analyze_degree_distribution that calculates and prints the degree distribution of the graph. This is important in network analysis showing how many connections each node has.

```
   let closeness_centrality =
metrics::calculate_closeness_centrality(&graph, 100);  // Compute the
closeness centrality for a sample of 1000 nodes.
   println!("Closeness centrality for 100 sampled nodes: {:?}",
closeness_centrality);  // Print the computed closeness centrality values.
```

Calculates the closeness centrality for a sample of 100 nodes using the calculate_closeness_centrality function in the metric module and prints the results.

```
   let (average_distance, max_distance, median_distance, std_deviation) =
metrics::calculate_path_metrics(&graph, 100);  // Calculate and print path
metrics like average, max, and median path lengths.
   println!("Average shortest path length from sample of 100 nodes: {}",
average_distance);  // Display the average shortest path length.
   println!("Maximum shortest path length from sample of 100 nodes: {}",
max_distance);  // Display the maximum shortest path length found.
   println!("Median shortest path length from sample of 100 nodes: {}",
median_distance);  // Display the median value of shortest path lengths.
```

```
    println!("Standard deviation of shortest path length from sample of 100 nodes: {}", std_deviation);  // Display the standard deviation of shortest path lengths.
```

Calculates average, maximum, median, and standard deviation of the shortest path lengths for a sample of 100 nodes. These are important for understanding the nature of a graph and its connectivity.

```
#[cfg(test)]
mod tests {
    use super::*;
    use graph_ops::*;
    use metrics::*;
    use petgraph::graph::DiGraph;

```

The #[cfg(test)] attribute indicates that this module and its contents are only compiled and run when the tests are executed. use super::*; imports all the functions, types, and traits from the parent module (main module in this case), allowing access to its public items. use graph_ops::*; and use metrics::*; import everything from the graph_ops and metrics modules respectively, which include the functions to be tested. use petgraph::graph::DiGraph; directly imports the DiGraph type from the petgraph crate, which is used to construct graphs for the tests.

```
    /// Tests graph construction from a mock data source to ensure nodes and edges are added correctly.
    #[test]
    fn test_graph_construction() {
        let mut graph = DiGraph::<u32, ()>::new();
        graph.add_node(1);
        graph.add_node(2);
        graph.add_edge(NodeIndex::new(0), NodeIndex::new(1), ());
        assert_eq!(graph.node_count(), 2);
        assert_eq!(graph.edge_count(), 1);
    }
```

This test ensures that the graph construction functionality is working correctly. A new directed graph is initialized, and two nodes are added followed by an edge between them. assert_eq! is used to verify that the number of nodes and edges in the graph are as expected, confirming the graph's integrity after these operations.

```
    /// Tests the calculation of degree distribution.
    #[test]
    fn test_degree_distribution() {
        let mut graph = DiGraph::<u32, ()>::new();
        let n1 = graph.add_node(1).index();
        let n2 = graph.add_node(2).index();
        graph.add_edge(NodeIndex::new(n1), NodeIndex::new(n2), ());
        graph.add_edge(NodeIndex::new(n2), NodeIndex::new(n1), ());

        let distribution = analyze_degree_distribution(&graph);
        assert_eq!(distribution.get(&1), Some(&2)); // Both nodes have 1
outgoing edge
    }
```

Tests the analyze_degree_distribution function from the graph_ops module. A graph is constructed with two nodes and two edges, creating a bi-directional link between them. The test checks if the degree distribution correctly reports that each node has one outgoing edge. This verifies that the function accurately counts and records the degrees.

```
    /// Tests basic functionality of BFS for shortest paths.
    #[test]
    fn test_bfs_shortest_paths() {
        let mut graph = DiGraph::<u32, ()>::new();
        let n1 = graph.add_node(1).index();
        let n2 = graph.add_node(2).index();
        graph.add_edge(NodeIndex::new(n1), NodeIndex::new(n2), ());

        let paths = bfs_shortest_paths(&graph, n1);
        assert_eq!(paths[n2], Some(1));
    }
```

Tests the bfs_shortest_paths function. A graph with two connected nodes is used to check if the BFS correctly calculates the shortest path from one node to the other. The test asserts that the shortest path distance from node n1 to n2 is 1, ensuring the BFS implementation works as expected for simple cases.

```
    /// Tests the closeness centrality computation on a small, controlled
graph.
    #[test]
```

```
    fn test_closeness_centrality() {
        let mut graph = DiGraph::<u32, ()>::new();
        let n1 = graph.add_node(1).index();
        let n2 = graph.add_node(2).index();
        graph.add_edge(NodeIndex::new(n1), NodeIndex::new(n2), ());
        graph.add_edge(NodeIndex::new(n2), NodeIndex::new(n1), ());

        let centralities = calculate_closeness_centrality(&graph, 2);
        assert!(centralities.contains_key(&n1));
        assert!(centralities.contains_key(&n2));
    }
```

Tests the calculate_closeness_centrality function for accuracy. A simple graph with two nodes and bi-directional connectivity is used to compute closeness centrality for both nodes. The test ensures that the centrality values are computed and stored correctly for each node, validating the centrality calculation logic.

```
/// Tests calculation of path metrics to ensure the function computes
correctly.
#[test]
fn test_path_metrics() {
    let mut graph = DiGraph::<u32, ()>::new();
    let n1 = graph.add_node(1).index();
    let n2 = graph.add_node(2).index();
    graph.add_edge(NodeIndex::new(n1), NodeIndex::new(n2), ());
    graph.add_edge(NodeIndex::new(n2), NodeIndex::new(n1), ());

    let (average, max, median, std_dev) = calculate_path_metrics(&graph,
2);
    println!("Average: {}", average);
    println!("Max: {}", max);
    println!("Median: {}", median);
    println!("Standard Deviation: {}", std_dev);

    assert_eq!(average, 0.5, "Average should be 0.5"); // Adjusted
expectation
    assert_eq!(max, 1.0, "Max should be 1.0");
    assert_eq!(median, 0.5, "Median should be 0.5");
    assert!(std_dev >= 0.0, "Standard deviation should be non-negative");
}
```

Tests the calculate_path_metrics function, which computes several path metrics from a small, controlled graph. The graph setup includes two nodes with bi-directional edges, and the test checks the calculated average, maximum, median, and standard deviation of path lengths. The assertions verify that the computed metrics match expected values, ensuring that the function handles basic calculations correctly.

```
//graph_ops.rs
pub use petgraph::graph::{DiGraph, NodeIndex};  // Re-export DiGraph and
NodeIndex from petgraph for direct use in other modules.
use std::collections::{HashMap, VecDeque};     // Import HashMap for node
indexing and VecDeque for queue management in BFS.
use std::fs::File;                             // Import File for file
operations.
use std::io::{BufRead, BufReader};             // Import BufRead and
BufReader for efficient buffered reading.
use flate2::read::GzDecoder;                   // Import GzDecoder to
handle gzip compressed files.
```

First line is used to export DiGraph and NodeIndex from the petgraph crate, making them directly accessible from other modules in the project. The second line imports HashMap for associating node identifiers to indices in the graph, and VecDeque for implementing a queue in the breadth-first search (BFS) algorithm. The third line imports the File class for handling file I/O operations. The fourth line imports classes for buffered reading, which enhances file reading performance. The fifth line imports the GzDecoder class for decompressing gzipped files.

```
/// Constructs a directed graph from a gzipped file containing edge
definitions.
pub fn construct_graph(file_path: &str) -> DiGraph<u32, ()> {
   let file = File::open(file_path).expect("Failed to open file"); // Open
the specified gzip file, panicking if it fails.
   let decoder = GzDecoder::new(file); // Create a GzDecoder to decompress
the gzip file.
   let reader = BufReader::new(decoder); // Wrap the decoder in a
BufReader for efficient reading.
   let mut graph = DiGraph::<u32, ()>::new(); // Initialize an empty
directed graph.
   let mut index_map = HashMap::new(); // Create a hashmap to track the
mapping of node values to graph node indices.
```

This function, construct_graph, initializes the process of constructing a directed graph from a gzipped text file containing edge definitions. It opens and reads the file, handling decompression and buffering. An empty directed graph is created using DiGraph::<u32, ()>::new(), which will store nodes and edges where nodes are represented by u32 and edges have no associated data (represented by ()).

```
   // Read lines from the file, skipping comments and empty lines.
   for line in reader.lines() {
       let line = line.expect("Failed to read line"); // Read each line,
panicking if an error occurs.
       if line.starts_with('#') {
           continue; // Skip comment lines that start with '#'.
       }
       let parts: Vec<&str> = line.split_whitespace().collect(); // Split
the line into parts using whitespace.
       let from_node: u32 = parts[0].parse().expect("Failed to parse from
node"); // Parse the 'from' node ID.
       let to_node: u32 = parts[1].parse().expect("Failed to parse to
node"); // Parse the 'to' node ID.
```

This loop reads each line from the file. It splits each line into components, parsing them into ‘from’ and ‘to’ node identifiers, which are u32 values.

```
       // Use or create node indices for 'from' and 'to' nodes, adding
them to the graph if they are new.
       let from_index = *index_map.entry(from_node).or_insert_with(||
graph.add_node(from_node).index());
       let to_index = *index_map.entry(to_node).or_insert_with(||
graph.add_node(to_node).index());

       graph.add_edge(NodeIndex::new(from_index),
NodeIndex::new(to_index), ()); // Add an edge between the 'from' and 'to'
nodes.
   }

   graph // Return the constructed graph.
}
```

This code manages the indexing of the nodes. If a node does not exist in index_map, it is added to the graph, and its index is stored in index_map. Edges are then added between the from and to nodes using these indices.

```
/// Performs a breadth-first search (BFS) to find shortest paths from a
start node in the graph.
pub fn bfs_shortest_paths(graph: &DiGraph<u32, ()>, start_index: usize) ->
Vec<Option<u32>> {
   let mut distances = vec![None; graph.node_count()]; // Initialize
distances vector with None for each node.
   let mut visit_queue = VecDeque::new(); // Create a queue for nodes to
visit.
```

This function performs breadth-first search to determine the shortest paths from a specified start node to all other nodes in the graph. The distance vector is initialized with None for each node, indicating that the distance to each node is unknown initially. The VecDeque is used as a queue to manage the nodes as they are visited, maintaining the BFS order.

```
   let start_node = NodeIndex::new(start_index); // Get the start node
index.
   distances[start_index] = Some(0); // Set the distance to the start node
as 0.
   visit_queue.push_back(start_node); // Add the start node to the queue.
```

The start node's index is converted into a NodeIndex, and its distance is set to 0 because the distance to itself is zero. This node is then added to the queue to begin the BFS process.

```
   // Process the queue until empty.
   while let Some(node) = visit_queue.pop_front() {
       let current_distance = distances[node.index()].unwrap(); // Get the
current node's distance.

       // Visit each neighbor of the current node.
       for neighbor in graph.neighbors(node) {
           if distances[neighbor.index()].is_none() { // Check if the
neighbor has not been visited.
               distances[neighbor.index()] = Some(current_distance + 1);
// Set the neighbor's distance.
```

```
                visit_queue.push_back(neighbor); // Add the neighbor to the
queue for further exploration.
            }
        }
    }
```

The loop continues until the queue is empty. For each node dequeued, its neighbors are visited. If a neighbor has not been visited (distance is None), its distance is set to the current node's distance plus one (since it's an adjacent node), and it is then enqueued for further exploration.

```
    distances // Return the vector of distances.
}
```

The function returns a vector of distances, where each index represents a node and the value at that index represents the shortest distance from the start node to that node. The distances are wrapped in an Option, with None meaning no path exists from the start node to that node.

```
/// Analyzes and returns the degree distribution of the graph.
pub fn analyze_degree_distribution(graph: &DiGraph<u32, ()>) ->
HashMap<usize, usize> {
    let mut degree_count = HashMap::new(); // Create a hashmap to count
degrees.

```

This function calculates the degree distribution of the graph, which is a measure of the number of edges connected to each node. The degree distribution is represented as a hashmap where the key is the degree (number of connections) and the value is the count of nodes having that degree.

```
    // Iterate over all nodes to compute their out-degrees.
    for node_id in graph.node_indices() {
        let degree = graph.neighbors_directed(node_id,
petgraph::Direction::Outgoing).count(); // Count outgoing edges for each
node.
        *degree_count.entry(degree).or_insert(0) += 1; // Increment the
count for this degree in the map.
    }

```

The loop iterates over all nodes in the graph. For each node, the number of outgoing edges is counted using graph.neighbors_directed with the Outgoing direction specified. The degree is then used to increment the count in degree_count.

```
    degree_count // Return the degree distribution map.
}
```

The function returns the degree distribution map. This is important to understand the connectivity patterns within the graph, such as identifying hubs or nodes with many connections.

```
//metrics.rs
use crate::graph_ops::{bfs_shortest_paths, DiGraph}; // Import necessary
functions and types from graph_ops.
use crate::stat_util::{median, standard_deviation};   // Import
statistical functions from stat_util module.
use std::collections::HashMap;                        // Import HashMap
for storing centrality values.
use rand::seq::SliceRandom;
```

The first line imports the bfs_shortest_paths function and the DiGraph type from the graph_ops module. The second line imports functions for statistical calculations. The third line imports the HashMap data structure. The fourth line is used for for random sampling.

```
/// Computes closeness centrality for each node in a sample of the graph.
pub fn calculate_closeness_centrality(graph: &DiGraph<u32, ()>,
sample_size: usize) -> HashMap<usize, f64> {
    let mut closeness_centralities = HashMap::new();   // HashMap to store
closeness centrality values.
    let mut rng = rand::thread_rng();                  // Random number
generator for sampling nodes.
    let node_indices: Vec<usize> = graph.node_indices().map(|n|
n.index()).collect(); // Collect all node indices.
    let sampled_nodes = node_indices.choose_multiple(&mut rng,
sample_size).cloned().collect::<Vec<_>>(); // Randomly sample nodes.
```

This function calculates the closeness centrality for a random sample of nodes. Closeness centrality measures how close a node is to all other nodes in the graph, useful for identifying nodes that can spread information efficiently. The function initializes a HashMap to store the centrality values, and uses a random number generator to sample nodes.

```
    // Iterate over each sampled node to calculate closeness centrality.
    for &node in &sampled_nodes {
        let distances = bfs_shortest_paths(graph, node); // Calculate
shortest paths from the node to all others.
        let total_distance: u32 = distances.iter().filter_map(|d|
*d).sum(); // Sum all distances for the node.
        let closeness = if total_distance > 0 {          // Check if total
distance is greater than 0.
            (graph.node_count() - 1) as f64 / total_distance as f64 //
Compute closeness centrality.
        } else {
            0.0                                          // Return 0.0 for
isolated nodes.
        };
        closeness_centralities.insert(node, closeness); // Store the
centrality value in the HashMap.
    }
```

For each sampled node, the function calculates the shortest paths to all other nodes using the previously imported bfs_shortest_paths. It then computes the sum of these distances and uses this sum to calculate the closeness centrality. Nodes with zero total distance (isolated nodes) are assigned a centrality of 0.

```
    closeness_centralities                              // Return the map
with centrality values.
}
```

The function returns the HashMap containing the closeness centrality for each sampled node, which can be used to analyze the efficiency and connectivity of nodes within the graph.

```
/// Calculates path metrics for a sample of nodes within the graph.
pub fn calculate_path_metrics(graph: &DiGraph<u32, ()>, sample_size:
usize) -> (f64, f64, f64, f64) {
    let mut rng = rand::thread_rng();                   // Random number
generator for sampling nodes.
    let node_indices: Vec<usize> = graph.node_indices().map(|n|
n.index()).collect(); // Collect all node indices.
    let sampled_nodes = node_indices.choose_multiple(&mut rng,
sample_size).cloned().collect::<Vec<_>>(); // Randomly sample nodes.
```

This function calculates several path metrics (average, maximum, median, and standard deviation of path lengths) for a sample of nodes. It again uses random sampling to select nodes for analysis.

```
   let mut all_distances = Vec::new();                // Vector to store
distances from sampled nodes.
   // Calculate distances from each sampled node to all others.
   for &node in &sampled_nodes {
       let distances = bfs_shortest_paths(graph, node); // Get shortest
paths from each node.
       all_distances.extend(distances.iter().filter_map(|&d| d).map(|d| d
as f64)); // Store valid distances as f64.
   }
```

The function collects all valid distances from the sampled nodes to all other nodes, converting them into f64 for subsequent statistical analysis.

```
   let max_distance =
all_distances.iter().cloned().fold(f64::NEG_INFINITY, f64::max); //
Calculate max distance.
   let mean_distance = all_distances.iter().sum::<f64>() /
all_distances.len() as f64; // Calculate mean distance.
   let median_distance = median(&mut all_distances.clone());
// Calculate median distance.
   let std_deviation = standard_deviation(&all_distances, mean_distance);
// Calculate standard deviation.
```

Using the collected distances, the function computes the maximum, mean, median, and standard deviation of the path lengths. These metrics are essential for understanding the distribution of path lengths in the graph, which can indicate the graph's overall connectivity and compactness.

```
   (mean_distance, max_distance, median_distance, std_deviation)
// Return the calculated metrics as a tuple.
}
```

The function returns these metrics as a tuple, providing a comprehensive view of the path characteristics in the graph.

```
//stat_util.rs

/// Helper function to calculate the median of a list of numbers.
pub fn median(numbers: &mut [f64]) -> f64 {
   numbers.sort_by(|a, b| a.partial_cmp(b).unwrap());  // Sort the numbers
in ascending order.
   let mid = numbers.len() / 2;  // Compute the middle index of the sorted
list.
```

This function calculates the median of a list of floating-point numbers. The list is first sorted in ascending order. Sorting is crucial because the median is the middle value in a sorted list, or the average of the two middle values if the list has an even number of elements.

```
   if numbers.len() % 2 == 0 {
       (numbers[mid - 1] + numbers[mid]) / 2.0  // If even, return the
average of the two middle numbers.
   } else {
       numbers[mid]  // If odd, return the middle number.
   }
}
```

The calculation of the median depends on whether the count of numbers is odd or even: Odd: The median is the middle element, Even: The median is the average of the two middle elements. Returning the median value provides essential information about the data set because it is important in understanding the distribution's center without the skewing effects of outliers.

```
/// Helper function to calculate the standard deviation of a set of
numbers.
pub fn standard_deviation(numbers: &[f64], mean: f64) -> f64 {
   let variance: f64 = numbers.iter()
       .map(|value| {
           let diff = mean - (*value as f64);  // Calculate the difference
from the mean for each number.
           diff * diff  // Square the difference.
       })
       .sum::<f64>() / numbers.len() as f64;  // Calculate the average of
the squared differences (variance).
   variance.sqrt()  // Return the square root of the variance (standard
deviation).
}
```

This function calculates the standard deviation of a list of floating-point numbers, which measures the amount of variation or dispersion in a set of values.

1. Mean Difference: Compute the difference between each number and the mean of the dataset.
2. Square Differences: Square each of the differences to make them positive and to weigh larger differences more heavily.
3. Variance: Calculate the average of these squared differences.
4. Standard Deviation: Take the square root of the variance to return to the original units of the data.

Program Output:

```
cargo run --release
   Compiling part1 v0.1.0 (/home/mbakhat/finalproject/part1)
    Finished release [optimized] target(s) in 5.94s
     Running `/home/mbakhat/finalproject/part1/target/release/part1`
Current directory: "/home/mbakhat/finalproject/part1/src"
Number of nodes: 403394
Number of edges: 3387388

Closeness centrality for 100 sampled nodes: {99911: 0.09101675648917414, 1
27653: 0.09015245057340769, 321513: 0.07817419582762698, 403351: 40339.3, 352660:
0.056978078313871604, 111926: 0.07200559775589964, 324129: 0.09036964154620346,
393966: 0.07266593122808636, 347419: 0.08781485677265875, 128763:
0.08793794795395755, 268176: 0.09123755922401947, 73470: 0.09096001004771537,
353230: 0.06668967946902588, 7571: 0.08293779228677484, 56026: 0.08725906810283943,
348928: 6020.79104477612, 132639: 0.08109694670929457, 266474: 0.0947003950535205,
330983: 0.08561978636707288, 22732: 0.08807597809372242, 280618: 3993.990099009901,
386214: 0.0837293456938313, 297705: 0.08711760130026464, 144010:
0.08066466114655388, 105921: 0.07684277759649374, 125431: 0.08443683276423651,
246140: 0.07817080248546383, 355667: 0.07962401181313969, 148789:
0.09636234207075135, 103797: 0.08148719746039955, 56422: 0.0772597331907368, 378807:
0.07819145492515858, 148300: 0.09462542202755726, 136334: 0.08868861905642378,
46043: 0.054119099772049185, 100220: 0.08710603216501114, 157161:
0.08793574344813936, 5820: 0.07857639846600002, 146183: 0.08765139085441599, 231249:
0.06696384254957054, 121842: 14406.892857142857, 330212: 0.06823445114715876,
164250: 0.08442034611686874, 76225: 0.07066834423683871, 114271:
0.08151899562974618, 157780: 0.09386430020902271, 114784: 0.07656888747499938,
380376: 0.06957363248099843, 343058: 0.08554019122826735, 389451:
```

0.06178133347382679, 188554: 0.08053691945130127, 208677: 0.09136501796076264, 14613: 0.09604313968900434, 34760: 0.08913680969977344, 115507: 0.09653365419245469, 153181: 0.08519981329216186, 263169: 0.08029316071132138, 59106: 0.08751911123819996, 322723: 0.08541212276098949, 289323: 0.07597578978335788, 393754: 7757.557692307692, 200961: 0.08710953079884971, 75068: 0.08443930720051208, 324981: 0.07826033984666993, 288783: 0.08375778754346022, 370017: 0.06633497507297549, 15852: 0.09301655811087794, 353438: 0.0896124206658625, 320103: 0.09106319136649982, 37917: 0.08292606213601014, 196177: 0.07304115399058958, 223246: 0.06994047154635621, 301097: 0.08643847997630937, 319864: 0.08125340282224512, 274317: 0.08957276336770124, 302609: 0.0812473804113074, 395962: 0.061693734994963785, 17521: 0.09658376296774444, 307815: 0.08446043418525254, 20700: 0.0691094085680707, 91096: 0.08711365051832874, 344123: 0.08644616725459224, 4838: 0.08518038325502825, 346643: 0.08010676586256665, 309860: 0.07628498173402516, 169207: 0.07739282567184143, 5607: 0.09643917162238891, 374782: 0.08341187364019124, 247707: 0.09468410192542547, 1961: 0.08950888414389391, 165774: 0.08530539395746499, 222461: 0.09548161059490289, 159606: 0.08417432609290804, 347101: 0.08781049843444703, 205693: 0.08369800555022434, 257076: 0.08561680614482098, 357088: 0.0757770096002158, 182938: 0.07463964099819023, 353555: 0.07795901534161097, 361695: 0.08168519026256757}

Average shortest path length from sample of 100 nodes: 12.411367676786535
Maximum shortest path length from sample of 100 nodes: 43
Median shortest path length from sample of 100 nodes: 12
Standard deviation of shortest path length from sample of 100 nodes: 2.903
1160223466252

Output explanations:

The output confirms the successful construction of the graph with 403,394 nodes and 3,387,388 edges. This indicates that your graph accurately represents the vast number of products and the complex web of co-purchasing relationships among them.

The closeness centrality values provided in the output are supposed to measure how close a node (product) is to all other nodes in the graph, which helps identify influential products in spreading potential influence (e.g., sales promotions, product recommendations). However, there are very high values which could suggest some errors, possibly from nodes with no outgoing paths (disconnected nodes).

Average Shortest Path Length (12.41): This suggests that on average, any product can be linked to any other product through approximately 12 steps of co-purchases. An average shortest path

length of 12.41 indicates a relatively efficient network in terms of the co-purchasing connections. It means that starting from any given product, a customer can be expected to traverse through about 12 other products (on average) before reaching any other arbitrary product in the dataset. A shorter path length between products implies higher discoverability. The average path length directly impacts the effectiveness of recommendation systems. Understanding that any product is, on average, only about 12 steps away from another, marketers and sales strategists can design campaigns that capitalize on the interconnected nature of product purchases.

Maximum Shortest Path Length (43): This value indicates the longest direct or indirect co-purchasing link between any two products in your sample. Such a high value could suggest the existence of very distinct or niche products which are less integrated into the core of the Amazon co-purchasing network. This length suggests that while most products in the Amazon network are relatively well-connected, there are outliers or extreme cases where the product association is tenuous. A maximum path length of 43 also poses challenges for product discovery algorithms. Products at the ends of such long paths might suffer from visibility issues, as standard recommendation systems based on co-purchasing patterns may less frequently suggest them.

Median (12) and Standard Deviation (2.903): The median being very close to the average suggests that the data is symmetrically distributed around the central value, with most products having a similar level of connectivity. The standard deviation tells us about the variability of the path lengths, with a relatively low value indicating that most products are not far from the average path length.

Network Connectivity: The calculated metrics illustrate a network where most products are relatively closely linked via co-purchasing paths, supporting the hypothesis that Amazon's product network is tightly connected. Such connectivity is crucial for efficient recommendation systems, influencing customer purchasing behavior by recommending products that are closely connected in this network.

Network Resilience and Efficiency: The average shortest path length being low suggests that the network is resilient and efficient in terms of information flow. It implies that changes in one part of the network (like changes in a product's pricing or availability) can quickly influence connected parts.