Obioma Eke DS 210 Final Project Write Up

About that data set & my approach

Link: https://snap.stanford.edu/data/amazon0302.html

The dataset that I chose is an Amazon product co-purchasing network from March 2, 2003. It is based on the *Customers Who Bought This Item Also Bought* feature on Amazon. It contains 262111 nodes and 1234877 edges. This is a directed, unweighted graph and if a product *i* is frequently co-purchased with a product *j*, the graph contains a directed edge from *i* to *j*. Since the graph is unweighted, I performed a Breadth-first search on the dataset to find the shortest path between nodes. This can show us how closely related certain items on Amazon are. As you can see in my commit history, I was actually going to use Dijkstra's algorithm at first, but I realized that that algorithm is specifically for weighted graphs and would not be appropriate for this data.

Instructions for running the code

For this project, the only required installation is Rust. Everything else that is needed to run this code is included in this repository, so all that needs to be done is to run cargo run and cargo test in the terminal for the code itself and tests respectively. After running the code, you will be prompted to enter a start node and an end node. To get a valid output, you can enter any number between 0 and 262110. The dataset, called amazon0302.txt, is already located in the src file. I used its relative file path in the main function, so it is not specific to my computer and should work on different machines. The visualization, called graph.png, should already be in the project-code folder, but a new image will regenerate whenever the code is run and appear in the same place. You can delete it before you run it to see it generate again if you would like, but the image will regenerate regardless.

Code

I decided to separate my code into separate modules for clarity and organization. The algorithm.rs file is responsible for loading the dataset and performing BFS. The load_graph function loads the dataset from a file and returns an adjacency list that is stored in a HashMap. The for loop reads each line of the dataset, skipping the header lines and specifying edges between nodes. The generate_edge_list function generates a list of edges in the sampled subgraph and ensures that only edges between sampled nodes are included. It iterates over the adjacency list and checks if both endpoints of an edge are in the sampled set. This function will be used later to create the visualization. The bfs function performs a BFS traversal from a starting node and keeps track of the order in which the nodes are visited. It uses a queue to manage traversal and a set to track visited nodes. It then prints the traversal order. I added a limit to just print the first 20 nodes of the graph to make it easier to read in the terminal output. The shortest_path function takes a graph represented as an adjacency list and a start and end node as inputs to find the shortest path between nodes. The visited HashSet keeps track of the nodes that have already been visited and the queue is used for the BFS process. Nodes are added to the back of the queue when they're discovered and removed from the front to be processed. The predecessors HashMap stores the predecessor of each visited node so that a path could be reconstructed from the end to the start nodes. The while let section then uses BFS to explore each layer of the graph. I also included tests here to ensure that the functions are working correctly. The test_load_graph function verifies that the graph is correctly loaded from the input data, the test_sample_connected_nodes function ensures that the sampled nodes belong to the same connected component, the test_generate_edge_list function checks that the edge list contains only edges between sampled nodes, and the test_bfs function ensures that the BFS function produces the expected traversal order. The visualizations.rs file's purpose is to generate a representation of the sampled graph. The visualize_sampled_graph function takes a list of edges and a list of node IDs and returns an image. I used the plotters crate for this and started by creating a drawing area using the BitMapBackend. The sampled nodes are randomly assigned coordinates within the canvas bounds. Each node is represented by a red circle with its ID next to it. The blue lines represent the edges between nodes. The visualization is saved to a file called graph.png. Something to note here is that since the nodes are randomly placed on the canvas, the image will look different every time you run it. The nodes and edges are still the same, it's just the location of the nodes that change. The main.rs file processes the functions defined in the previous files. I specified the data set (amazon0302.txt) and selected a sample size to be used for the visualization. I chose a small sample size of 15 because it is much easier to see the connections between the nodes. I then generated a list of edges with the generate_edge_list function from algorithm.rs and performed a traversal of the graph with BFS. The next part of the main function utilizes the shortest_path function from algorithm.rs and takes an input that includes start and end nodes. I used if else statements to ensure that user inputs are valid. If the user types in a number that is invalid, like a negative number or non-numerical value, they will get a message stating that the input is invalid. If they type in a number that is beyond the number of nodes, the user will be told that the node does not exist in the graph. The final part of the main function uses the visualize_sampled_graph function from visualizations.rs to create the graph image. The final output will print the starting node for BFS, the traversal order, and prompt the user to enter start and end nodes of their choosing to find the shortest path. This path will then be printed and the graph.png file will also be generated at this time.