DS210 Final Project: What are the best products to bundle on Amazon?

Goal: If you are given a product on amazon, can you find the best products to bundle with it, which are often bought together?

Why: I chose this project because I thought this is something that can be used in real life. Sellers and marketplaces can utilize this tool and find the best things to bundle together, maybe at a little discount as well which increases the incentive for people to buy the bundle. This algorithm can also be used in other fields to find the densest subgraphs for a given node.

How: This can be achieved by finding the densest subgraph that has the specified product present in it.

Dataset: https://snap.stanford.edu/data/com-Amazon.html

Procedure: I found this idea very interesting because it can be utilized by sellers to increase their sales. To begin with, I found a dataset which had data of products that are often bought together. It is given as a node to node dataset, and although it said that the data set is undirected, it has many nodes that are directed which means that the data is incomplete and needs to be cleaned first. The initial data also had many nodes missing and was discontinuos, so I converted that file into a continuous node file using python. I then made a function to read my dataset and store it. The function also looks if the given node is present, and adds the two nodes and their reverse only if they are not already present to avoid double counting issues. This way, I complete the dataset and convert all the incomplete nodes into undirected nodes.

It is next to impossible to find a densest subgraph which has the given product by the general method and it also does not guarantee the presence of our product in that subgraph. Therefore, I decided to first conduct a 6 degrees of separation algorithm and find all the nodes that are present within 6 steps of the initial product. After finding all the nodes, I find all the other nodes connected to these nodes and convert them into a nodes graph dataset and store them. I chose 6 degrees because it is highly unlikely that a product forms a dense subgraph with a product that is more than 6 steps away from it. After finding this subgraph, I implemented the densest subgraph algorithm.

The algorithm is:

Algorithm 2.1: DENSEST-SUBGRAPH(G = (V, E))

$n \leftarrow |V|$, $H_n \leftarrow G$

for i = n to 2,

Let v be a vertex in H_i of minimum degree

$H_{i-1} \leftarrow H_i - \{v\}$

return (H_j , which has the maximum density among H_i's, i = 1, 2, .., n)

Of course an algorithm that takes into consideration all the possible subgraphs would give the accurate result but even for a small dataset of 100 nodes, it will require examining 2^99 subgraphs, which is practically not possible to process. Therefore, the algorithm we use, given in https://www.cs.umd.edu/~samir/grant/ICALP09.pdf document, does not guarantee the most accurate outcome but does guarantee a result that is less than two times worse than the best result. To perform this algorithm, we find the density of the subgraph, then take the node with the least number of edges away from the graph and calculate the density of the subgraph again. We continue to do so till only 2 points are left. The density is calculated as the total number of edges divided by the total number of nodes.

So, with the subgraph created using the breadth first search algorithm, we perform the densest subgraph algorithm but with a little change, we make the presence of the starting node mandatory. We then find the densest subgraph for the given product. We also find the nodes with the highest density in the subgraph created by the six degrees of separation algorithm, the graph density associated with it and the total number of nodes present in the densest subgraph. The start_node allows us to change the product number that we want to find the densest subgraph for and the max_depth can be changed to find the densest subgraphs for different densities. I also print out three random results to show the working of our algorithm.

I divided my entire program into 6 different modules:

1. file_operations.rs

   Contains a function read_graph_from_file that reads the given data file.

2. graph_operations.rs

   Contains function bfs and function find_edges. The bfs function conducts a breadth first search and the function find_edges converts the found nodes into the node node connection hashset.

3. graphs.rs

   Contains structs Edge, Graph and Subgraph; also contains implementation for Graph and enum GraphOperationResult. Edge contains to and from, graph contains nodes and Subgraph contains nodes and density. GraphOperationResult contains NoteNotFound or Success.

4. graph_analysis.rs

   Contains enum GraphData and functions calculate_top_nodes, calculate_graph_density and find_highest_density_subgraph_with_start_node. The enum contains edges and topnode. The first function finds the top nodes in the subgraph, the second one finds the graph density and the third one finds the densest subgraph.

5. main.rs
6. test.rs

Explanations:

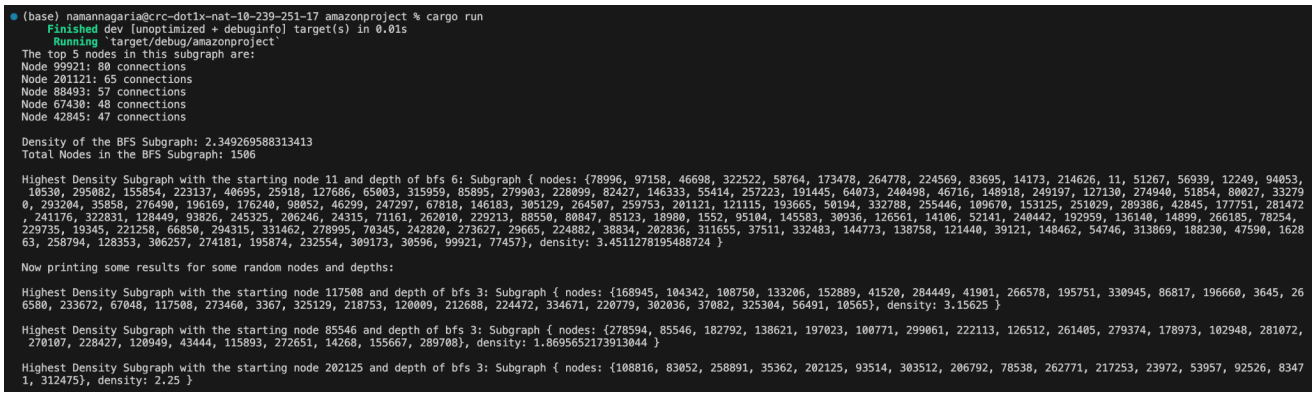
1. The struct Edge represents a directed edge in the graph.
2. The struct graph represents the graph structure.
3. The struct subgraph represents the subgraph with the graph.
4. The struct nodeconnections represents a node and its connection count in the context of graph analysis.
5. The enum graphdata is used to encapsulate different types of graph-related data like edges and top nodes.
6. The graphoperationresult enum provides a way to handle the result of operations on the graph, indicating either success or the specific issue encountered (like a node not being found).

Functions:

1. new: Creates a new instance of a graph.

2. add_edge: Adds an edge between two nodes in the graph.
3. find_node: Determines if a specific node is present in the graph.
4. bfs: Conducts a Breadth-First Search starting from a specified node up to a given depth.
5. find_edges: Retrieves all edges within a specified set of nodes.
6. read_graph_from_file: Reads graph data from a file.
7. calculate_top_nodes: Identifies and ranks nodes based on the number of connections they have.
8. calculate_graph_density: Computes the density of a graph or subgraph based on the number of nodes and edges.
9. find_highest_density_subgraph_with_start_node: Locates the subgraph with the highest density starting from a given node and within a specified depth.

Cargo run results:



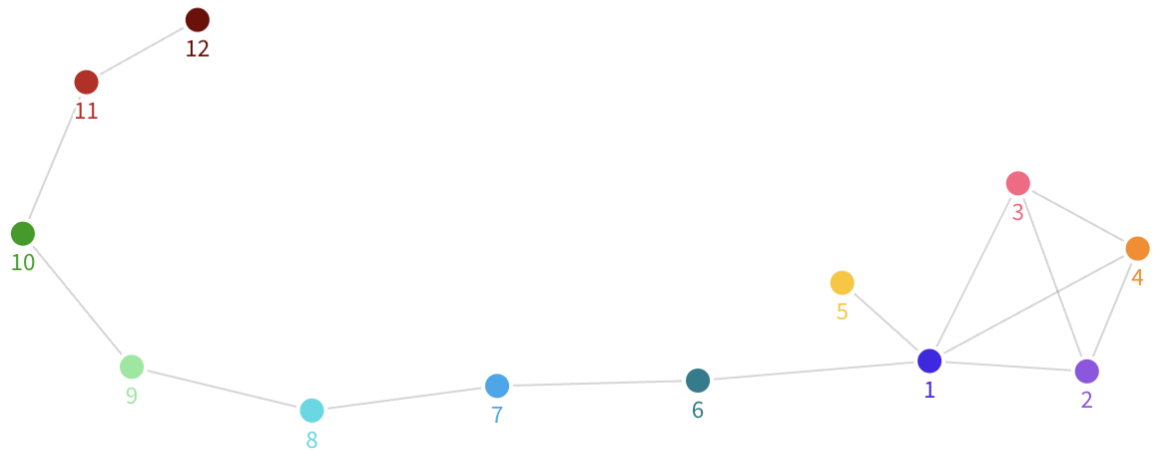
These are the cargo run results of the code. The first part tells the top 5 nodes with the most connections present inside the 6 degrees of separation subgraph. After that , I print the density of the subgraph and the total nodes present in the subgraph. Then I find the highest density subgraph for the given node and depth and return all the nodes and the density. Lastly, I print 3 random results for random nodes and random depths, to show the functioning of the algorithms.

```
(base) namannagaria@crc-dot1x-nat-10-239-251-17 amazonproject % cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.01s
     Running unittests src/main.rs (target/debug/deps/amazonproject-547264fcf097569d)

running 3 tests
test tests::tests::test_top_node_and_connections ... ok
test tests::tests::test_densest_subgraph_density ... ok
test tests::tests::test_nodes_in_densest_subgraph ... ok

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

These are the results for the 3 different tests. My network graph of my test dataset was:



The tests were as follows:

Test 1:

```rust
#[test]
fn test_densest_subgraph_density() {
    let filename = "test_data_3.txt";
    let graph_data = read_graph_from_file(filename).expect("Failed to read graph data");

    let mut graph = Graph::new();

    if let GraphData::Edges(edges) = graph_data {
        for edge in edges {
            graph.add_edge(edge);
        }
    } else {
        panic!("Invalid graph data format");
    }

    let start_node = 1;
    let max_depth = 2;
    let subgraph = find_highest_density_subgraph_with_start_node(&graph, start_node, max_depth);

    assert_eq!(subgraph.density, 1.5, "Density of the densest subgraph did not match expected value.");
}
```

This function finds the densest subgraph for start node 1 and depth 2. It can be clearly seen that the densest subgraph here is 1,2,3,4 and they have a density of 1.5.

Test 2:

```rust
#[test]
fn test_top_node_and_connections() {
    let filename = "test_data_3.txt";
    let graph_data = read_graph_from_file(filename).expect("Failed to read graph data");

    let mut graph = Graph::new();

    if let GraphData::Edges(edges) = graph_data {
        for edge in edges {
            graph.add_edge(edge);
        }
    } else {
        panic!("Invalid graph data format");
    }

    let start_node = 1;
    let max_depth = 2;
    let reachable_nodes = bfs(&graph, start_node, max_depth);
    let edges_set = find_edges(&graph, &reachable_nodes);

    match calculate_top_nodes(&edges_set.into_iter().collect::<Vec<_>>(), 1) {
        GraphData::TopNodes(top_nodes) => {
            assert_eq!(top_nodes.len(), 1, "Expected only one top node.");
            let top_node = &top_nodes[0];
            assert_eq!(top_node.node, 1, "Expected top node to be node 1.");
            assert_eq!(top_node.count, 5, "Expected node 1 to have 5 connections.");
        },
        _ => panic!("Invalid top nodes data"),
    }
}
```

This function checks the top node of the subgraph for start node 1 and depth 2 and then checks the number of nodes it is connected to which are 1 and 5 respectively.

Test 3:

```rust
#[test]
fn test_nodes_in_densest_subgraph() {
    let filename = "test_data_3.txt";
    let graph_data = read_graph_from_file(filename).expect("Failed to read graph data");

    let mut graph = Graph::new();

    if let GraphData::Edges(edges) = graph_data {
        for edge in edges {
            graph.add_edge(edge);
        }
    } else {
        panic!("Invalid graph data format");
    }

    let start_node = 9;
    let max_depth = 4;
    let subgraph = find_highest_density_subgraph_with_start_node(&graph, start_node, max_depth);

    let expected_nodes: HashSet<u32> = [12, 8, 6, 11, 10, 9, 1, 7].iter().cloned().collect();
    let subgraph_nodes: HashSet<u32> = subgraph.nodes;

    assert_eq!(subgraph_nodes, expected_nodes, "The nodes in the densest subgraph do not match the expected set.");
}
```

This test checks the nodes in the densest subgraph for start node 9 and depth 4. From the graph it can be seen that this is [1, 6, 7, 8, 9, 10, 11, 12].