Report for Lab activity: analysis of music networks.

Marino Oliveros Blanco NIU:1668563
Pere Mayol Carbonell NIU:1669503

1. Introduction

In this report, we will explain how we have faced the Lab activities. In these exercises, we have explored the capabilities of the spotipy libraries and Spotify's API, connecting them to our code and completing our objectives. This is separated in the acquisition and preprocessing of the data, and afterward analyzing and visualizing it. Here we will explain the problems and setbacks we found along the way, as well as how our code works.

2. Part 1: Data acquisition

For acquiring the data, we will use the client object specific for our connection to our spotify app. Through this, we can access the artists and their information, including their neighbors, connections and personal creations.

There are three functions in this part.
The first one {`def search_artist`} simply gets the client object to connect to and a name of an artist[str] to get their ID using the function search in the spotipy library.

The second one {`def crawler`} is the one we will use to search through the found items in neighboring artists. This initializes a queue for BFS or a stack for DFS, where we will put the artists found.
Afterward, everything happens inside a while that stops when we reach the maximum number of nodes we want to crawl. What we do inside is, basically, first pop the oldest item in the stack or queue. Moreover, we add the artist we currently are in to the visited list, to have them all in one place and also control the amount of elements we visit. We then get the information of the artist, and add all the information to a node along with their name. Then, we get the neighbors of this node through the function '`artist_related_artists` ' in the spotipy library, which returns a dictionary with information on every item related to the aforementioned artist. In the end, we add the related_artist to the queue or stack. In case there is any problem processing the current artist, an error pops up, and then it continues to the next iteration in the while.

The last function in this part {`def get_track_data`} gets the graphs that result of our findings in the crawler function and creates a dataset. We simply iterate inside the graphs and their nodes, extracting the information from the dictionaries we added before, so we can get the features for the artist as a pandas DataFrame.

All of this is implemented together as a hole in the main function, where we can visualize the created graphs, the dataset and see which are the properties of the nodes inside of them, along with another function {`def degree_statistics`} which is basically used to get the degree's information for each node.

2.1. Questions to answer [Part 1: Data acquisition]

2.1.1. Regarding the graphs Gb[BFS] and Gd[DFS]:.

They are created using[the same inputs except for the search algorithm that they use to crawl through the graph. The main difference between them is then going to be that the storage of artists in Gb is going to be in a queue and for Gd a stack. So, for the graph using the BFS search, we see the structure tends to be wider and respect more levels. The second one, shows a deeper and more linear structure, reflecting the depth-first nature.

1. Order and size.

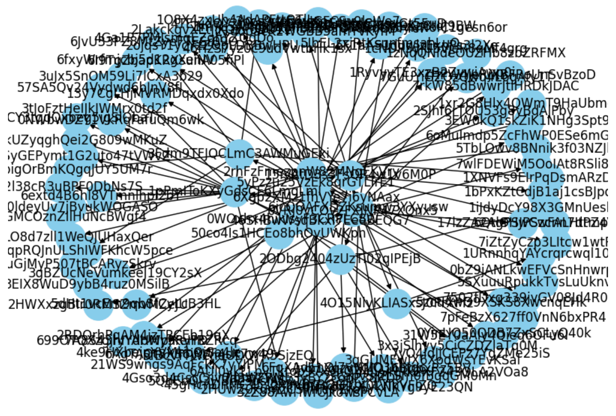
- Graph Gb(BFS): 100 nodes, 99 edges
- Graph Gd(DFS): 100 nodes, 99 edges

The order of the graphs Gb and Gd is the same in this case (both have 100 nodes). However, if they had different orders, it would be because BFS explores nodes level by level, which can lead to discovering more immediate neighbors, whereas DFS explores as deep as possible before backtracking, potentially missing some immediate neighbors in favor of depth. Nevertheless, since our function limits the discovered nodes to the same value, it makes sense that no matter the search algorithm, the order is the same:

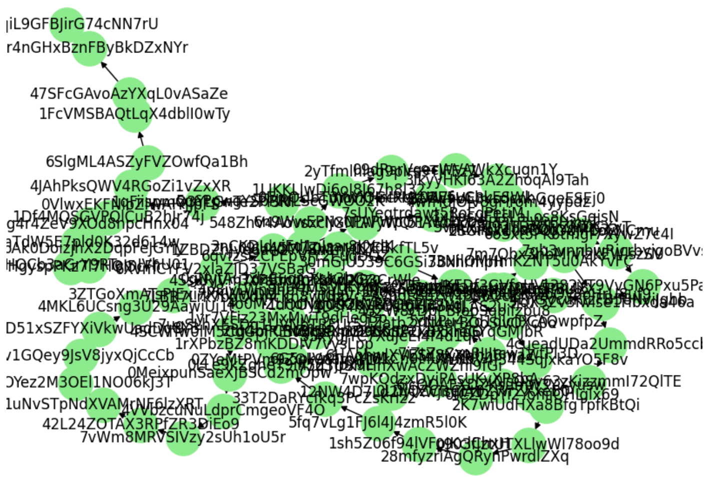
```
    Gb = crawler(sp, id1, 100, strategy='BFS', out_filename="gB.graphml")
    Gd = crawler(sp, id1, 100, strategy='DFS', out_filename="gD.graphml")
```

The size of the graphs is 99, which is the number of edges. This can vary based on the number of connections a certain artist and its neighbors have. However, the size can be similar if both methods explore the same overall region of the graph.

BFS:



DFS:



2.1.2. Indicate the minimum, maximum, and median of the in-degree and out-degree of the two graphs (gB and gD). Justify the obtained values.

Using the function mentioned before for the degree statistics, we get the following statistics for the graphs Gb and Gd:

a. Degree statistics for Gb: {'in_degree': {'min': 0, 'max': 1, 'median': 1}, 'out_degree': {'min': 0, 'max': 20, 'median': 0}}
b. Degree statistics for Gd: {'in_degree': {'min': 0, 'max': 1, 'median': 1}, 'out_degree': {'min': 0, 'max': 1, 'median': 1}}

Given the characteristics mentioned before and the visualizations of the graphs, we can see these goes hand in hand what we could expect. The in degrees are the same meaning for each artist one points at it. For the out degree, since the DFS is more linear and prospects less the level by level feature, then for every node created there is simply one out degree.

On the other hand, for the BFS, since it is created layer by layer, we can clearly see the connection between one neighbor in the layer x and then all of its connected artist in the layer x+1.

2.1.3. Indicate the number of songs in the dataset D and the number of different artists and albums that appear in it

Number of songs: 1819
Number of different artists: 197
Number of different albums: 1247

Part 2: Data preprocessing
Part 3: Analysis
Part 4: Visualization