**Setlist Song Recommender Using Graph DB**

1. **Problem statement:**

The construction of smoothly-transitioning sets of your favorite electronic music can be a difficult and time-consuming process that can require skills honed over years. Popular DJs are often so popular because of their ability to do this and are paid good money to showcase their carefully crafted sets. Providing fluid sets/track lists is most certainly a highly valued service because it typically requires lots of resources like time and skill to accomplish. If we were able to produce the same behavior of creating smoothly-transitioning sets made up of popular and related songs while eliminating any need for any human effort, the implications could be great. Understanding these facts, we hope to build a module capable of accepting raw setlist data as input that returns the most frequent songs played after every song in the data as well as similar artists as output. The module will be able to accept a specific song as input and return the top songs commonly played by DJs after they play the inputted song.

1. **Datasets and tools:**

<https://www.1001tracklists.com/>

The above link directs you to the website we scraped for our data. The data consists of a series of csv files each representing an artist set. Each “set” is stored as a list of song strings, with each song string made up of component artist, song, and remix strings. To parse the data form the website, we used Beautifulsoup, an HTML parser that allowed us to grab all the divs with a certain id (and that id was the one associated with each track in the tracklist). Psuedo code for parsing:

1. Get all links for sets from a particular festival (built in to 1001)
2. For each link:

4. Download webpage

5. Parse HTML for specific divs

6. For each div:

7. Get song info and save to list

8. Serialize and save list to file

The data model we will be using to represent our dataset is a neo4j graph database which is labeled and directed. Each Node possesses 2 attributes: type (song or artist) and name. Each Relationship possesses 2 attributes: type (played\_by, originally\_by, or played\_after) and count (only for relationships of type played\_after). Once all the data was loaded into our database, it contained 25,431 nodes and 168,872 edges or relationships. The average density of our graph is 0.000261125 and loading process itself took approximately 20 minutes.

1. **Data structure and auxiliary structure:**

As stated earlier, every node possesses a type and name attribute. Song and artist nodes are associated respectively with the specific song or artists stored in the name attribute. Relationships can be of type played\_by, originally\_by, or played\_after. Relationships of type played\_by represent that an artist associated with a specific artist node played the song associated with a specific song node. Furthermore, relationships of type originally\_by represent an artist associated with a specific artist node created a song associated with a specific song node. Finally, relationships of type played\_after represent a song associated with a specific node was played after a song associated with a different specific song node. This relationship will utilize the count attribute, and increments whenever this behavior is encountered at load time. The following pseudocode outlines the population of the database:

1. For festival in directory:

2. For set in festival:

3. Get cur\_artist

4. Open set\_file

5. Prev\_song = null

6. For line in set\_file:

7. Get cur\_song\_obj

8. If cur\_song\_obj != null:

9. Cur\_song = add\_song\_node (cur\_song\_obj, cur\_artist, prev\_song)

10. Prev\_song = cur\_song

This function simply loads all of the artists and songs into the database by looping over every set file (line 2), and, for every line in the file making the function call Cur\_song = add\_song\_node (cur\_song\_obj, cur\_artist, prev\_song) which actually adds the node and its relationships to the database (line 9).

1. **Algorithm Description:**

find\_next\_songs(SONG):

MATCH (SONG NODE)-[PLAYED AFTER]->(CURRENT SONG NODE) ORDER BY PLAYED AFTER.COUNT

Being that our graph database was designed to contain the PLAYED AFTER relationship, we can run this simple query to return songs that share a PLAYED AFTER relationship with the inputted song ordered by count. The returned songs will be the most frequently played songs after the inputted song within the data.

Find\_similar\_artists(artist):

1. get set of songs played by input artist
2. for each other artist:
3. get set of songs played by other artists
4. calculate jaccard similarity between sets of songs played
5. return artist w/ max jaccard similarity

The purpose of this function is to find the artist who behaves the closest to the input artist based upon the songs they’ve played. To do this we simply take the song set of the input artist (line 1), loop over every other artist (line 2) and calculate the Jaccard similarity of the input artist song set and the other artist song set (line 4) and return the artist with the max Jaccard similarity (line 5).

1. **Optimization:**

As mentioned above, we designed our database with the priorities in mind of keeping runtimes down and implementation simple. Consisting of only one simple query to our database, our find\_next\_songs function is very compact and highly efficient compared to a relational model equivalent. Looking to possibly use our good design to our advantage further, we found that we could optimize our find\_similar\_artists function by only looping through only artists that have at least one song in common with the input artist. This was found with the following query:

Match (a:Artist)<-[:PLAYED\_BY]-(s1:Song)-[:PLAYED\_BY]->(b:Artist {name: {artist\_name}}) RETURN DISTINCT a

This technique cut the average number of artists from 10466 to <500. Another way we found to optimize this function was to parallelize the for each loop using joblib in Python.

1. **Plan:**

Being that we are designing a novel algorithm, we don’t have other models to compare it against. Conserving time, however, was one of our leading priorities in this experiment, and the metric we will use to judge our algorithms is runtime. By continuing to find areas of our code in which we can optimize by using the good design of our database like when we used a query to drastically reduce loop iterations in find\_similar\_artsists. we expect to see further improvements in runtime. We will also be monitoring the accuracy of our find\_similar\_artists function by verifying that our outputs align with the information we find on the internet. For example, if Dion Timmer is outputted as a similar artist to Excision, we will confirm this based on research and make any necessary adjustments.

1. **Results:**

To reiterate, the problem we were seeking to solve was to drastically reduce the resources required to create smoothly-transitioning track-lists made up of popular and related music. These costly resources of manually creating a tracklist can include hours of time and/or experience possibly developed over years. By implementing the module outlined above and sticking to our guidelines detailed in the experimental plan, we were able to achieve our goal of eliminating the need for such costly human resources. We were able to directly verify this by comparing sample outputs of our find\_similar\_artists function to information we found through internet research. Some sample input/output pairs we used in this verification process were Excision/Snails, Nightmare/Slander, Armin Van Buuren/Andrew Rayel, 3LAU/rl grime, Alison Wonderland/Boombox Cartel. All of which were found to be similar artists on the internet. Furthermore, we were guaranteed at least some success from the beginning given the nature of our data. We know the tracklists were authored by famous DJs who have built up credibility in the music industry in many cases for a lot of years. By using their carefully crafted tracklists that were originally performed at high production venues, we can be assured at least some level of quality.

1. **Moving Forward:**

Judging by our current successes, we think the future of the project looks very bright. The music industry is massive, and music streaming services are on the rise. With such a huge content pool of music and artists to draw from, there’s a growing need for methods to organize it thoughtfully so that its enjoyment can be maximized. To do this, many people spend personal time to create tracklists for themselves or end up just relying on others to do it for them. Even when time is conserved by relying on others to organize your music, they still lose a factor of input. If we were to extend this project to an application and create an easy to use GUI that represents songs as trees to easily choose a next choice, a user could have higher quality tracklists of music that’s personally tuned to them via filter criteria that we could give them control over. Possible criteria may include artists they like, artists they don’t like, songs they like, etc. To alleviate the need for webscraping, we could use setlist.fm api to much more efficiently add data to the database. Lastly, we could integrate with rave.dj to send them a tracklist and have it created into an actual playlist.