Zachary Irons

Dr. Terrence Harvey

Honors CISC 220-080

December 9, 2016

Semester Project

Spotify Playlist Optimization Using Graph Theory

Concept

The idea for this project was to take a playlist of songs of various genres and determine an optimal order to play them in. This end result would be a playlist where each song was evenly similar to those before and after it, as to eliminate any jarring transitions of pace, genre, or artist. The Spotify API was used to gather data on songs using various GET methods including: get playlist, get tracks, get artist, and get related artists. All songs' data was collected and organized into a dictionary. Tying this idea into graph theory, each songs similarity to any other song was boiled down to a weight. This weight was decided to be a combination of the song's artist, genres, other artists related to the artist, and the year released. Each song was then graphed as a vertex with all other songs and those with weights were connected by edges of that weight. A minimum spanning tree was calculated using Prim's algorithm. Prim's algorithm is preferred to Dijkstra's algorithm because it encourages the completed tree to have more roots by adding the next closest thing available compared to a tree created by Dijkstra's algorithm which is only looking for the shortest path to all the vertices. This tree was traversed using an altered version of depth first search. Instead of travelling down each root completely, the algorithm would print every other node on the way down the root. Upon reaching the end of a root, the nodes skipped

would then be recollected and printed. This was done to create a seamless transition from each song to the next while still leaving enough songs to play before going down a different root of potentially completely different genre. This ensured each song was at most only two songs away from the next one played. Everything described was completed and can be examined in the attached, fully commented file: project.py.

Execution

The first step in this project was to collect the data from the Spotify API. Due to authentication issues, the initial steps were taken outside of any code. Once a playlist was decided on, the get playlist command was used to retrieve the id to the playlist's list of tracks. In order to access a playlist tracks, a Spotify user must input username and password. This was done manually and an OAuth Key was received to use in the get command. This command looked like this:

curl -X GET "https://api.spotify.com/v1/users/spotify/playlists/{playlist_id}/tracks" -H "Authorization: Bearer {access token}"

This command was entered into the Windows Command Prompt and the output was written to a .txt file. For the purpose of testing a playlist of size 93 was chose and will be the example for the remainder of the report. The output can be found in the attached file: code.txt.

The text file contained all the data about each song in the playlist in JSON format. This file was read into a Python program and initialized as a dictionary. Through several for loops, the dictionary was iterated over and the ID's of each song were extracted and appended to a list. This list can be seen in the attached file: id.txt. This list was iterated through and the relevant information about each song was fetched from Spotify and added to a list of dictionaries. Each

dictionary contained the song title, artist name, list of genres of the artist, list of related artists to the artist, year released, and a currently empty list of weights. The information retrieved from Spotify for each song can be seen in the attached file: track_list.txt.

This dictionary of song information was looped through to fill in the weights of each song to the others. The maximum a weight could be was 40 with 10 points coming from genres, 20 from related artists, and 10 coming from year released. If the artist of a song was the same as another, 30 points were subtracted immediately because the genres and related artists would be identical. The year released was also only factored into the weight if points were deducted for genres or related artists. This was done because, without this condition, all songs would have a weight with every other song. Since the graph was already very dense, any reduction to the number of edges would help speed up the algorithm. After all these calculations, if a songs weight with another song was still 40 it was set to None. This was also done for the weight between a song and itself. The list of weights between every song in the playlist can be seen in the attached file: weights.txt.

Having calculated the weights between all the songs, they were transferred from the dictionary to a class Graph. The Graph class contains a set of all the vertices. The class also has a dictionary called “edges” that has each vertex as keys and those it points to as values. Since the graph is initially undirected these edges go both ways. It also contains a dictionary called “weights” that contains as keys tuples of either end of an edge and the value being the weight as an integer. These weights go both directions as well.

Prim’s algorithm was used to create a minimum spanning tree of the graph from the arbitrarily picked first song. The function returns a dictionary of with all the keys being children

and all the values being the parents that point to them. This dictionary can be seen in the attached file: path.txt. In order to use this for further traversals this dictionary needed to be reversed so the parents were keys and their children, if any, were values. This dictionary can be seen in the attached file: graph_dict.txt. An example drawing of an early version of this minimum spanning tree can be found in the attached image: tree.jpg.

With a completed minimum spanning tree the only thing left to do was to run a depth first search on the tree. A custom depth first search was written that skipped every other node on the way down and collected them on the way back. An example of this program being run on a much smaller tree can be found in the attached file: dfs.py. A sample drawing and traversal visualization of this DFS can be seen in the attached image: dfs.jpg. The results of the DFS are the final, optimized order of the playlist. The final order can be seen in the attached file: results.txt.