PlaylistGenomeProject: Using Spotify Public Playlists to know more about your Songs

Bianca Trombetta, Calvin Tonini, Christopher Ramirez

Our awesome project video

Overview

We implemented a graph data structure, Floyd-Warshall's algorithm, and various Centrality algorithms to find the shortest distance between all nodes. We also kept track of the fastest path between all nodes. For every pair of nodes, the algorithm checks if it is faster to travel between any and all third nodes. If it is, the distance is updated to reflect as such, and the path matrix is updated to reflect that the fastest path goes through the third node. This data is the core of our project, which we use to implement various features to generate interesting information about the graph.

Installing and Running our Code

Detailed instructions are located in our README file, which can be found, here.

Planning

We planned to implement a crawler, a graph maker, a list of paths maker, a distance maker, and more! Here are our original and final draft specifications that show which extra features we wanted to implement but couldn't, and modifications to our original ideas. The feature list was aggressive, and we encountered difficulties, but ultimately we were able to achieve the majority of our goals and objectives, though perhaps not as cleanly or as integrated as we wanted. For example, we were unable to crawl and traverse 10,000 playlists as we had originally planned (because of the memory limitations of our environment and to limit the size of our final submission).

In terms of the schedule and milestones, we unfortunately encountered difficulties and fell behind our schedule during our first week of work. However, we were able to regain lost time during the second week, and eventually were able to achieve our milestones on time towards the end of the project.

Design and Implementation

We all were completely unfamiliar with Python (and Python modules), graph data structures, IDEs, and Git Hub when we first started out. Some of us had experience in imperative programming from CS50, but generally we had minimal familiarity with our development.

Implementing crawler was difficult because of problems interfacing with the Spotify API, which required us to first learn more about HTTP and how APIs work. Fortunately we were able to use the spotipy library to abstract away much of the HTTP calls, but understanding the Spotify authentication and credential model proved challenging, and implementing a method for our project to obtain authorization tokens took time.

Things became extremely messy when we attempted to scale our crawler to larger numbers of playlists. We realized that there were playlists that included songs that were not on Spotify, such as local files of inconsistent names and contents (Spotify has a consistent index of songs). In addition, we came across issues such as playlists being destroyed after being being discovered and added to our frontier, user accounts being deleted after being discovered and added to our frontier, and other emergent behavior that our crawler was ill-equipped to handle, raising exceptions and errors. Finally, we discovered that crawling at full speed takes a long time (it took us roughly an hour to crawl 100 playlists).

Our first implementation of exporting our data worked well for test situations, which involved small numbers of playlists that we ourselves were intimately familiar with. However, we encountered difficulties reimporting data into another python file in a useful form. Things became extremely messy attempting to export larger datasets as json or txt or even as CSV files, as we encountered the differences between encoding UNICODE and ASCII and UTF-8. We needed to stop to understand the nature of encoding and the errors we were getting. Ultimately, we used pickle to store our datasets and variables. Since we did not use the data sets outside of python scripts, we realized it wasn’t necessary to export.

Many of our functions were implemented using lists, and sometimes lists of lists. Lists are somewhat inefficient, and we later went back and redid our work whenever possible as sets, or frozen sets inside of sets. Sets have better O time than lists because they use hash tables instead of lists, and frozen sets are immutable, allowing them to be hashed inside of sets.

We generally collaborated in person, asking each other for help and ideas. When one group member knew how to implement a part of the code and was excited to implement it, a flurry of git pulling and pushing would happen so that we could code what we felt confident coding. It would be too difficult to say what we coded individually, because most of the documents contain interwoven contributions from all of us; we would have to say which lines of which scripts we coded, which would be too detailed.

Results

As a sample of what our project can do, we ran our crawler over one of our teammates’ personal spotify account. Here’s a list of our results:

- The playlist data containing their playlists (generated from crawler.py).

- The graph data structure containing their playlists as a graph (generated from graphmaker.py).

- The shortest distance and fastest path matrices from running the graph through fasterpathfinder.py.

- The songlist, a list of songs ordered in the same way as the above matrices (also generated from fasterpathfinder.py). This is what we used to convert index numbers to song names and refer to cells in the matrices.

- From running centrality.py we get the following results:

  0: Bavelas's Centrality - Rag And Bone
  1: Harmonic Centrality - Rag And Bone
  2: Dangalchev's Centrality - Rag And Bone
  3: Betweenness Centrality - Rag And Bone

  This data is not very interesting, but you can see how the different definitions affect the output if you run centrality.py on the hour data set, for example:

  0: Bavelas's Centrality - Cello Sonata, Op. 8: III Allegro molto vivace
  2: Dangalchev's Centrality - Pictures

- From running generatepath.py on 'Rag and Bone' and 'Flourescent Adolescent', the shortest path between these two songs is: [Rag and Bone, In One Ear, Float On].

- When running neighborfinder.py with the following inputs (the number refers to the number of songs we request), we can get a list of related songs:

  Rag And Bone, 2 - Gold On The Ceiling, Tighten Up
  Henrietta, 2 - Fluorescent Adolescent, Creepin Up The Backstairs

- When running playlistneighborfinder.py on the songs returned from neighbor finder, it returns 'In One Ear'. We interpret this as a suggestion for another song to add to a playlist containing the songs returned from neighbor finder.

- When running triangles.py on 'Rag And Bone' from the ramchris data set, we find that the trio of songs contains 'Henrietta' and 'Perhaps Vampires is a Bit Strong But'. We interpret this as a suggestion for a new playlist starting point.

- The atlas visualization of the graph data structure, showing how all of the songs are connected to each other.

Reflection

Had we more time, we would have gone for either faster data importing and exporting, or implemented more of our extra features. It would have been cool to give a playlist suggestion given two friends' music tastes.

If we could start this project over, we would not worry about visualizing our data. It is cool to look at, but was frustrating to implement and did not really teach us anything interesting (besides seeing how our graph worked).

The Floyd-Warshall algorithm works incredibly slowly, with a running time of $O(n^3)$. Optimizing the algorithm was difficult to figure out. We were able to take advantage of the fact that distance matrices are symmetric for undirected graphs, meaning that it was only necessary to input data from half of the song pairs. However it was difficult to reason how to get the correct output and how to update the path matrix when only going over unique sets of song pairs; we went through several different implementations of pathfinder and neighborfinder before choosing ones that we thought were more efficient.

The different measures of centrality are incredibly interesting. Just by changing when values are inverted, it's possible to generate radically different nodes as the most central to a graph. The reusability of code to implement betweenness centrality was also very interesting. We found that the structure of Floyd-Warshall's was the same for the calculations of betweenness centrality, and the code from generatepath.py was what we needed to give us the paths to search.

In the future, we would love to be to make this a website with a user-friendly interface instead of python shell scripts. Imagine users being able to explore graph data structures in their browser!

Advice for Future Students

Some of our group members were worried about using public libraries to implement features that were tangential to our goal. It was much better to use the public libraries and focus on learning the new data structures and algorithms that we were interested in. Also, getting to work earlier than later is incredibly useful; don't procrastinate!