# Graphsify: An Interactive Music Exploration Tool

Alec Bernardi and Alexandra Zytek

Spring 2018

## 1 Introduction

Graphsify is an interactive web application that allows users the visualize their music library and discover trends in their tastes. It harnesses Spotify's Web API to visualize the similarity of various audio features among a set of user-defined songs. In addition to showing links between selected songs, the visualization also suggests new music by leveraging the recommendation endpoint of the Spotify Web API.

## 2 Motivation

Graphsify was developed with the goal of answering several questions: What kinds of similarities exist within the songs that an individual listens to, and how can we use these similarities to introduce the individual to new songs? Do links exist between seemingly different genres of music within an individual's tastes? How can we intuitively display the songs to which an individual listens, and introduce new songs in a way that makes it clear where the suggestions are coming from?

Our hypothesis was that the songs that one enjoys form clustered groups with similar audio features. These clustered groups might be interconnected by intermediate songs that bridge the gaps between various genres that one listens to. This similarity network, which very closely resembles a social network, can be used to explore one's music tastes and possibly recommend new music that they will enjoy.

## 3 Background

Muelder et. al. produced an algorithm to make a graph visualization specific to music libraries. This method used Fourier transforms to convert the song data into a form that could be compared. The result was frequency-over-time data, which could be analyzed and converted into a series of similarity metrics. These

metrics were then plotted using several different graph algorithms. Though this project uses audio features that are extracted by Spotify, this seems to be a reasonable method of computing such features. Perhaps an extension of the visualization would be to create novel similarity metrics.

Kobourov (in one of the paper's read in class) described a series of force-directed graph drawing algorithms for creating well-spaced graphs with no crossing lines where possible. These algorithms can be used to render the song network in this project. Furthermore, for small enough data sets, the drawing algorithms should be able to run dynamically, and show updates in real qtime. When applied to the visualization, this could be used to make it seem responsive and further engage the user in the interactivity of the visualization.

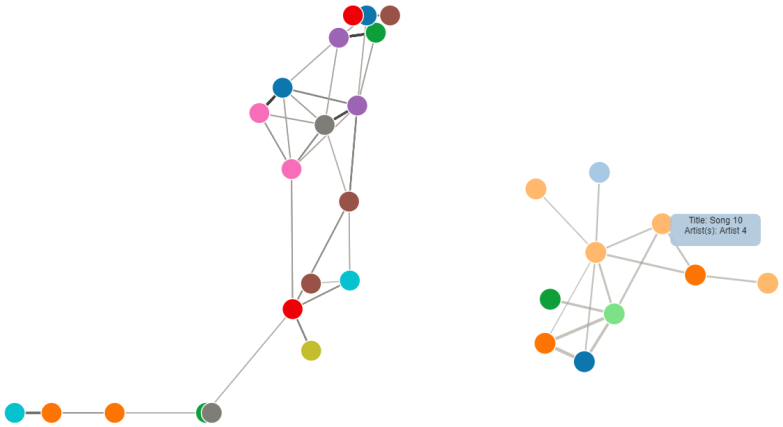
Existing work in finding music suggestions includes a heterogenous graph method proposed by Guo and Liu. This method uses six types of nodes (songs, artists, albums, genres, users, and playlists) connected by sixteen types of edges (including interactions such as listening to, creating, and commenting on) to classify multiple types of "walks," or connections between songs. For example, a song may be recommended if it is featured in the same playlists as a song the user listened to. Pauws and Verhaegh proposed another method of suggesting songs, using eight attributes that include more nominal attributes (such as name, artist, album, and genre) and numerical attributes (duration, year, and tempo). With these attributes, they automatically generated a playlist given an specified input using a local search with a series of heuristics to improve efficiency. Using this idea of a walk through the song graph, playlists could be generated via this tool in such a way that any two adjacent songs are similar, but the overall playlist might transition between genres multiple times. However, generating the longest path in a graph without repeating vertices is known to be a rather expensive computation, and so an approximation would need to be used.

# 4 Design Evolution

We had planned on using a node-graph "social network for songs" style visualization from the beginning of the project, but the details of the visualization changed throughout the project. Specifically, we needed to make decisions about how the nodes and graphs should be colored and displayed, and which edges should be rendered.

## 4.1 Node Colors

The color scheme of the nodes is an important design decision that can greatly impact the final appearance of the visualization. Originally, we looked at discrete values associated with the tracks for use with a qualitative color scheme. One option was key, which is a value from 0 to 11, which would result in an acceptable number of colors. However, key is not very meaningful to the common




(a) The node graph colored by artist. (b) The node graph colored by key.

Figure 1: Some coloring methods that were tried. The color schemes are qualitative Color Brewer schemes.

user, and we did not want it to appear more important than the other audio features. Additionally, it came across as a rather arbitrary choice.

Another option was to color the nodes by artist. This was the scheme used when showing the visualization to test subjects in class. However, the number of distinguishable colors ran out fairly quickly, and users stated that having multiple artists share colors (which became necessary as the number of artists rose above about ten) was confusing. Figure 1 shows examples of the rejected coloring schemes. Several users suggested coloring by genre. However, Spotify does not provide this information by track, and many tracks do not nicely fit into a particular genre. Therefore, this idea was deemed impractical.

Our final decision was to color based on a combination of top audio features. The three highest weighted features (ties are broken alphabetically) would be used together to form the R, G, and B component of the node color. The benefit of this method is it creates a very pretty gradient throughout the graph, and is updated to represent the information that the user cares most about. The downside is that it is less immediately meaningful, and may confuse the user. Hopefully, a legend explaining the color scheme will help with the confusion. Although it is not ideal, we deemed this option to be the best of those considered.

Another problem that came about related to color was how to distinguish between suggested songs and songs added by the user. The original plan was to have suggested songs be greyed out (lower saturation) or faded. However, with the large number of colors being used, nodes would already exist that appeared grey or faded, so it would be difficult to distinguish. Therefore, the decided-upon design was to have the suggested nodes be white with colored outlines, while the user-selected nodes were solid colored.

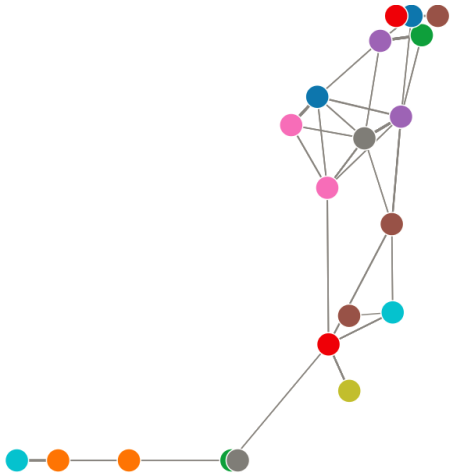


Figure 2: The original edge design, where thickness increases with similarity.

### 4.2 Edge Design

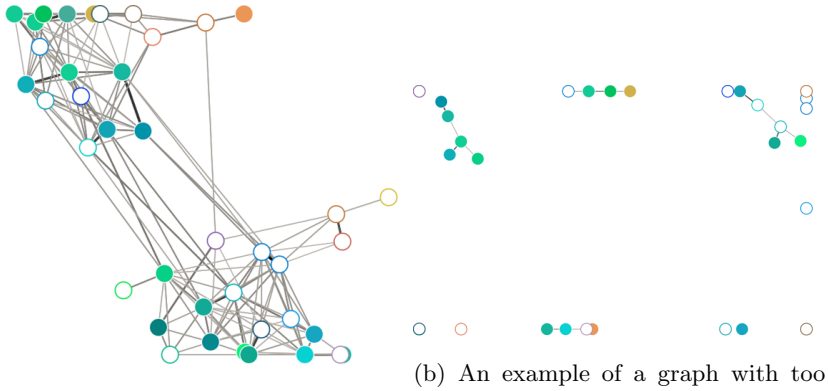
In the original version of the visualization, the similarity of nodes was represented by the thickness of the edges: the thicker the edge, the more similar the audio features of the nodes. However, when shown to test users, most did not realize that the lines were varying in thickness until this was pointed out. When they were told about this design choice, the general consensus was that they would not have guessed this meaning without being told. To make the line strength differences more apparent, we added line darkness to the design, so that more similar nodes would be connected by thicker and darker edges. Figure 2 shows an example of the original rejected design.

### 4.3 Edge Selection

A very important design decision was when to draw edges. Unlike other applications of the node graph, connections between nodes in this context are not a binary "exists or does not exist." Rather, every pair of nodes technically has a connection of some varying strength. Therefore, some threshold needs to be set to determine whether or not an edge is drawn.

The objective is to have few enough edges that all edges can be seen and followed fairly easily, while having enough edges that trends can be seen. Additionally, no node should have an edge that is drawn that is weaker than an edge that is not drawn. Figure 3 shows some examples of flawed numbers of edges.

Setting a specific threshold value for the edge connections (for example, any edge of strength greater than 0.5) is not ideal, because that relies on the algorithm used to generate connections as well as the weights used, which may not be a constant. Additionally, if the user inputs songs that are especially different or similar, the resulting graph may not be useful. Therefore, it makes



(a) An example of a graph with too many edges, resulting in clutter.

(b) An example of a graph with too few edges, resulting in little information being provided.

Figure 3: Some examples of failed attempts at getting the correct number of edges.

sense to use a value that dynamically changes with the graph.

One method tried was to draw the top twenty percent of connections. However, the number of edges grows exponentially with the number of nodes, so it rapidly becomes too chaotic.

Instead, we decided to draw the top edges based on the total number of nodes. Specifically, drawing three times as many edges as nodes tended to create a good spread, and scaled well even as the nodes increased. Additionally, we ensured that every node had at least one connection, so that the user could see the relationships with even their most obscure music.

## 4.4 User Interface

The implementation of the user interface as a whole was fairly linear in nature. First, a sidebar was created to show the user the songs they have added, suggestions, and a method of adding new songs. This was housed in a Stepper component to suggest an order to using the app. More specifically, it was intended that the app be used to first add songs, view suggestions, add those, and then export the results. The first iteration can be seen in figure 4.

The use of a Table for the song list was decidedly poor, since it takes up too much space, and creates clutter with a scroll bar. Since it doesn't provide all the track information at a glance, a revision was made, using a List instead, allowing the album art to also be displayed. Furthermore, a floating action button was created for opening a Dialog to add songs to the list. This revision can be seen in figure 5.

The Dialog for adding songs was created next. It leverages the same method of displaying the active songs to display the results of a search. It uses the Spotify Web API to generate search results. Though songs were intended to

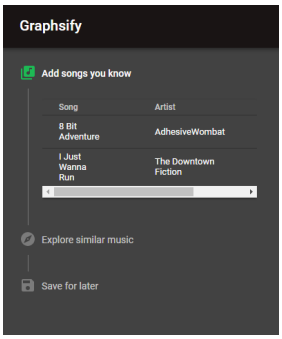



Figure 4: First iteration of the UI.

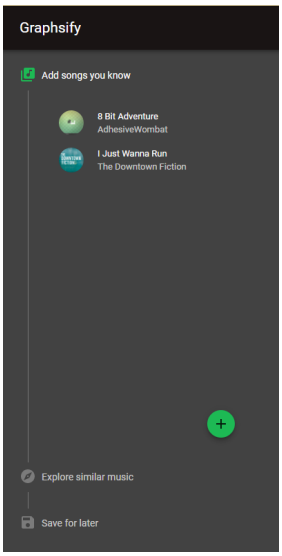



Figure 5: Second iteration of the UI.

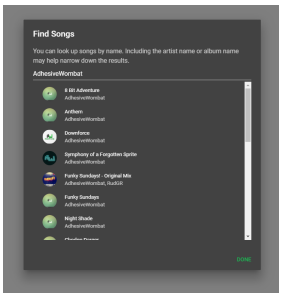



Figure 6: The song selection functionality.

be searched via name, typing in the name of an artist seems to generate fairly relevant results, as seen in figure 6.

The final piece of the UI to be implemented was the graph. This posed significant issues, due to fundamental incompatibilities in the frameworks used for the two aspects of the visualization. As a result, only static images were able to be generated via the original framework. With the deadline approaching, an executive decision was made to try to reimplement the graph with a compatible framework. However, it was not completed in time, and so the new graph does not quite match the specs of the original.

# 5 Feedback

At the time of showing our work to the class, our project was in two pieces: the UI component and the visualization component. Therefore, we received feedback on these components separately.

## 5.1 Visualization Feedback

Generally, the visualization was effective. The dragging and animation was considered helpful by most reviewers. One comment that came up multiple times was that using line thickness to represent similarity between nodes was not effective because the variation was difficult to discern; several people did not even realize they were different. As a result, we are now experimenting with using a combination of line lightness and thickness. Additionally, users were unsure if coloring nodes by artist was a good choice because it may result in having to repeat colors or use too-similar colors when the number of artists gets large. Some people suggested coloring by genre instead; unfortunately, Spotify does not readily provide this information on a per-track basis.

## 5.2 UI Feedback

Most of the feedback given on the user interface was positive. The peer reviewers thought that the proposed workflow was intuitive and streamlined. Even so,

there were a few suggestions for additional features to improve the experience. Several of these related to the addition of songs to the active set. One suggestion was to allow users to import songs from their own library or playlist. Another peer suggested adding songs by manually specifying feature values, instead of searching by track information. The same peer also recommended adding a way to add all the songs in an album or by an artist easily. Adding a feature to import playlists from Spotify would address most of these concerns. Spotify has built its own clients that are quite intuitive for creating playlists. By indirectly harnessing these technologies to improve the app, most of the concerns about adding tracks could be addressed without reinventing the wheel. At its current state however, the visualization does not allow importing playlists.

# 6 Data Collection

All song data is retrieved from Spotify, using the Spotify Web API. Specifically, Spotify provides a list of track features for every song in its database, calculated automatically by Spotify's music analysis algorithms. The features are as follows:

1. Acousticness: a value from 0.0 to 1.0 representing how likely the track is to be acoustic. A higher value represents a higher confidence that the song is acoustic.

2. Danceability: a value from 0.0 to 1.0 representing how suitable the track is for dancing. A higher value represents a more danceable track.

3. Energy: a value from 0.0 to 1.0 representing the energy of the track.

4. Instrumentalness: a value from 0.0 to 1.0 representing how likely the song is to be instrumental. A higher value represents a higher confidence that the song is purely instrumental.

5. Key: the key of the track, given using standard Pitch Class notation.

6. Liveness: a value from 0.0 to 1.0 representing how likely the song is to be live. A higher value represents a higher confidence that the song is live.

7. Loudness: the track's overall average volume, given in decibels, typically from -60 to 0.

8. Mode: 0 for a minor track, or 1 for a major track.

9. Speechiness: a value from 0.0 to 1.0 representing how much spoken language occurs in the track. A higher value represents a track that is more likely to be purely spoken (such as podcast or talk show).

10. Tempo: the overall tempo of the track in beats per minute, generally between 0 and 200.

11. Valence: a value from 0.0 to 1.0 representing the positivity of the track. A higher value represents a track that is more "happy."

12. Popularity: a value from 0 to 100 representing how popular the track is. A higher value represents a more popular track.

It is important to note that all features are on a continuous 0.0 to 1.0 scale except key, mode, loudness, tempo, and popularity.

# 7 Algorithms

## 7.1 Computing Track Similarity

There are various ways to compare the features of two songs. The method used by Graphsify is essentially a weighted Euclidean distance between feature vectors with normalized components. Component-wise normalization is desirable to ensure that each feature has an equal effect on unweighted similarity. To this end, each feature defines a function that computes the square distance between two values after normalizing them both to a scale from 0 to 1. Most features are already on this scale, however there are some exceptions. To address this, each feature specifies minimum and maximum values so that feature values can be normalized.

$$d_i(\vec{u}, \vec{v}) = \left( \frac{v_i - \min_i}{\max_i - \min_i} - \frac{u_i - \min_i}{\max_i - \min_i} \right)^2 = \left( \frac{v_i - u_i}{\max_i - \min_i} \right)^2 \tag{1}$$

The resulting function is suitable for every feature except for key and mode, since similarity in key is a bit more complex. As such, the key feature uses a distance function that simply returns 0 if the keys match, and 1 otherwise. With these component-wise square distance functions, the weighted Euclidean distance between two feature vectors can be expressed in terms of the two feature vectors and another vector of corresponding weights. The result is a function with a value that approaches 0 as two feature vectors become more similar. More specifically, smaller values of this function indicate that the corresponding components of the feature vectors are closer in value.

$$d(\vec{u}, \vec{v}, \vec{w}) = \sum_i w_i^2 \cdot d_i(\vec{u}, \vec{v}) \tag{2}$$

## 7.2 Making Track Recommendations

The Spotify Web API specifies an endpoint for retrieving song recommendations. To do so, it requires a seed, which can be a group of songs, artists, or albums. Furthermore, minimum, maximum, and target values can be specified for each feature to further refine the results. So, to generate recommendations for a

single track, the seed is merely the song itself, and the feature thresholds should be computed based on the track's features and the user-selected weights.

For each feature, the target value is the track's value for that feature, since similar songs are sought. The range between the minimum and maximum values should be centered around the target, and should also decrease with an increasing weight. Hence, the square distance between values for a feature i should observe the following inequality, for a track with feature vector u and a suggestions with feature vector v:

$$d_i(\vec{u}, \vec{v}) \leq \left(\frac{1 - w_i}{2}\right)^2 \tag{3}$$

Lastly, the recommendations that Spotify provides are sorted in ascending order, and only the top four are selected as suggestions. This culling is an attempt to prevent the graph from becoming overcrowded with suggestions.

# 8 Final Version

The visualization has a sidebar that allows users to add songs by searching Spotify. A user can type in a phrase (usually a track title or artist), which will bring up a list of query results from which the user can choose. Clicking one of these results will add the song to the visualization, as well as up to four automatically suggested songs. As more songs get added, the visualization will begin to render edges and the social network structure will be formed.

As mentioned above, the final version of the node graph is a bit different from the final version with the UI. Figure 7 shows the final version of the node graph, as deemed most user friendly. In this version, solid nodes represent songs added by the user, and hollow nodes are suggestions.

Figure 8 shows the final version put together with the UI. In this version, the radial nodes are the suggestions while the central nodes are the user-added tracks.

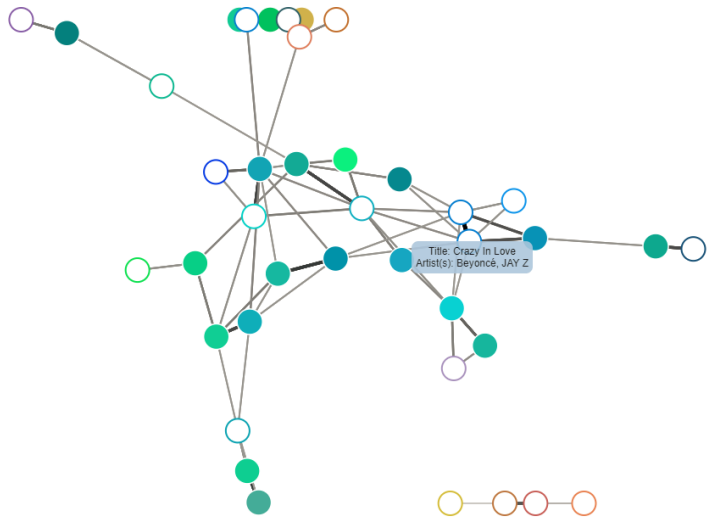



Figure 7: The final version of the node graph.


Figure 8: The final version of the visualization with the UI components.

# 9 Work Breakdown

For this project, Alec worked on the project UI (such as adding songs), retrieving suggestions from Spotify, and combining all elements of the project. Alexandra worked on the node-graph visualization. Both team members worked on retrieving track information from Spotify.

# 10 References


1. Get Audio Features for a Track. (n.d.). Retrieved from https://beta.developer.spotify.com/documentation/web-api/reference/tracks/get-audio-features/

2. Guo, C., & Liu, X. (2015). Dynamic feature generation and selection on heterogeneous graph for music recommendation. SIGIR '15 Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval,807-810. doi:10.1109/bigdata.2016.7840658

3. Kobourov, S. (2013). Force-Directed Drawing Algorithms. In R. Tamassia (Ed.), Handbook of Graph Drawing and Visualization (383-408). CRC Press.

4. Muelder, C., Provan, T.,& Ma, K. (2010). Content Based Graph Visualization of Audio Data for Music Library Navigation. 2010 IEEE International Symposium on Multimedia. doi:10.1109/ism.2010.27

5. Pauws, S., Verhaegh, W., & Vossen, M. (2008). Music playlist generation by adapted simulated annealing. Information Sciences,178(3), 647-662. doi:10.1016/j.ins.2007.08.019

6. Web API. (n.d.). Retrieved March 26, 2018, from https://beta.developer.spotify.com/documentation/web-api/

**Coding references:**

7. Bostock, M. (2017, December 4). Bounded Force Layout. Retrieved from https://bl.ocks.org/mbostock/1129492

8. Bostock, M. (2018, April 26). Force-Directed Graph. Retrieved from https://bl.ocks.org/mbostock/4062045

9. What is the javascript equivalent of numpy argsort? (2017, November 26). Retrieved from http://takeip.com/what-is-the-javascript-equivalent-of-numpy-argsort.html