6-2016

Blending Two Automatic Playlist Generation Algorithms

James Curbow
Union College - Schenectady, NY

Blending Two Automatic Playlist Generation Algorithms

By

James E. Curbow

* * * * * * * * *

Submitted in partial fulfillment

of the requirements for

Honors in the Department of Computer Science

UNION COLLEGE

June, 2016

Abstract

CURBOW, JAMES E. Blending Two Automatic Playlist Generation Algorithms. Department of Computer Science, June 2016.

ADVISOR: Anderson, Matthew

We blend two existing automatic playlist generation algorithms. One algorithm is built to smoothly transition between a start song and an end song (Start-End). The other infers song similarity based on adjacent occurrences in expertly authored streams (EAS). First, we seek to establish the effectiveness of the Start-End algorithm using the EAS algorithm to determine song similarity, then we propose two playlist generation algorithms of our own: the Unbiased Random Walk (URW) and the Biased Random Walk (BRW). Like the Start-End algorithm, both the URW algorithm and BRW algorithm transition between a start song and an end song; however, issues inherent to the Start-End algorithm lead us to believe that our algorithms may create playlists with smoother transitions between songs.

Contents

1 Introduction **1**

2 Expertly Authored Stream Algorithm **2**

3 Converting Edge Weights to Similarities **4**

3.0.1 One-Way Similarity 4

3.0.2 Two-Way Similarity 5

4 Playlist Generation Algorithms **6**

4.1 Start-End Algorithm 7

4.2 Unbiased Random Walk Algorithm 9

4.3 Biased Random Walk Algorithm 10

4.4 Deterministic URW & BRW 12

5 Data & Implementation **13**

5.1 Data 13

5.2 Implementation 13

6 Results **14**

6.1 Start-End Algorithm Results 15

6.2 Unbiased Random Walk Algorithm Results 15

6.3 Biased Random Walk Algorithm Results 21

7 Analysis **21**

7.1 Start-End Song Analysis 21

7.2 Unbiased Random Walk Algorithm Analysis 22

7.2.1 Greediness ”Sweet-Spot” For Adjacent Similarity 22

7.2.2 Greediness Does Not Indicate Playlist Cohesion 23

7.2.3 Spikes Indicate Forced Bad Decisions 24

7.2.4 Flaws of the URW Algorithm 24
7.3 Biased Random Walk Algorithm 25

8 Conclusion **25**

List of Figures

1 Building a weighted graph using Algorithm 1. 3

2 Calculating the one-way similarities for song A (right) based on the edge weights. 5

3 Calculating the two-way similarities (right) based on the edge weights. 6

4 Similarity Requirements 7

5 Ideal Similarity Ratios and Real Similarity Ratios 8

6 Selecting Songs Based on Ideal Similarity Ratios and Real Ratios 9

7 As the URW & BRW algorithm's greediness parameter approach infinity the algorithms converge. 12

8 Implementation Structure 14

9 A Similarity Matrix 16

10 Average similarity to next song using one-way similarities 17

11 Average adjacent similarities using one-way similarities 17

12 Average similarity to next song using two-way similarities 18

13 Average adjacent similarities using two-way similarities. 18

14 Similarity Matrices for URW Algorithm for the One-Way Similarities 19

15 Similarity matrices for the average URW Algorithm for the two-way similarities 20

16 The Start-End algorithm allows similarity to change without a change in similarity ratio. . . 22

17 Being too greedy can lead to a dead end. 23

1 Introduction

A playlist is a sequence of recorded songs designed to be listened to in order. The importance of the playlist has become increasingly evident to music streaming services such as Spotify [1] and Pandora [2] [8, 3]. These services typically encourage user to generate playlists, as well as promote the playlists that the service itself has generated, because playlists often escalate service usage [1]. Service-generated playlists are either expertly curated (i.e., the songs are selected and ordered by an expert), or created automatically by a computer. Some services now have billions of playlists available [2].

Automatic playlist generation's heightened influence makes the field a particularly relevant and exciting one. Researchers are developing new, exciting concepts for automatic playlist generation algorithms; for example, Platt developed an algorithm that generates a playlist based on multiple seed songs [6]. However, while there exist several interesting mechanisms designed to automatically create a playlist, many listeners still prefer user-made playlists [4]. This prompts the questions: In what way are these automatic playlist generation algorithms flawed? And, more importantly, how can we improve them?

One specific technique that may be improved upon is proposed by Flexer et al. which we call the Start-End algorithm [5]. This method selects and orders a set of songs to create a playlist that transitions between a given start song and end song. This is a relatively novel feature that has not been implemented by most music streaming services. However, their algorithm appears to have a few major flaws–namely, it does ensure a smooth transition between songs that are adjacent in the playlist.

Implementing a playlist generation algorithm requires a method to determine song similarity. In their implementation of the Start-End algorithm, Flexer et al. use a mechanism known as audio fingerprinting which relies on acoustic content to determine song similarity. Fletcher et al. make the claim that "different approaches towards computation of similarity" can be applied in their algorithm [5]. In our implementation we will use a different technique to determine song similarity.

We define the term "expertly-determined smoothness". Unlike Flexer et al.'s interpretation of smooth transitions, which focus on temporal and acoustic elements of songs, expertly-determined smoothness focuses on relationships between songs that a user would first notice. The method we use to determine song

[1] spotify.com
[2] pandora.com

similarity is proposed by Ragno et al. Their algorithm infers similarity between songs based on the number of adjacent occurrences in expertly authored streams (the EAS algorithm) [7]. By taking into account expertly authored streams, such as a radio DJ's set-list, or a commercially-produced sequence of songs like a movie soundtrack, we believe that the EAS algorithm accounts for expertly-determined smoothness. This is due to the fact that the algorithm is based on a expert's playlist decisions–this is an element of song similarity that an audio fingerprinting techniques does not account for.

Implementing the Start-End algorithm using the EAS algorithm to determine song similarity prompted the questions: Does the Start-End algorithm produce a smooth playlist? How can the Start-End algorithm be improved so that it does ensure smooth transitions between adjacent songs?

The necessary improvements to the Start-End algorithm led us to two proposed playlist generation algorithms: the Unbiased Random Walk (URW) and the Biased Random Walk. These algorithms were also implemented using the EAS algorithm to determine song similarity; this ensured that any difference in result is a function of the altered playlist generation algorithm (the Start-End algorithm and the algorithms introduced that improve upon it), and not a difference in the song-similarity algorithm (the EAS algorithm). This allowed us to compare the Start-End song algorithm to the URW algorithm and the BRW algorithm.

We proceed by providing an overview of the EAS algorithm in Section 2. In Section 3 we detail the methods we used to prepare edge weights in the graph from the EAS algorithm to be input for the playlist generation algorithms that are outlined in Section 4. In Section 5 we provide an overview of the data we collected as well as our implementation. In Section 6 we present the results of the playlist generation algorithms, and analyze the results in Section 7. We conclude and discuss plans for future work in Section 8.

2 Expertly Authored Stream Algorithm

The EAS algorithm allows us to infer song similarity by exploiting the order of expertly authored streams (playlists). The idea is intuitive: songs that are adjacent in an expertly curated playlist are similar–an expert (such as a DJ) probably wouldn not follow up Beethoven's 5th Symphony with Justin Bieber's latest single. The EAS algorithm quantifies these expert decisions and allows us to compute song similarity [7]. The

algorithm takes a playlist of songs as input and constructs a weighted graph. The nodes in the graph represent each unique song in the playlist. The edge weight between a pair of songs is equal to the number of times those songs appear next to each other in the playlist. The Algorithm 1 outlines the algorithm.

Algorithm 1 Converting an EAS to a Weighted Graph

Input: A stream of songs S
Output: A weighted graph $G = (V, w)$
V = songs in S.
$w_{u,v} = 0, \ \forall \ u, v \in V$.
for $i = 1...(|S| - 1)$ **do**
    $w_{S[i],S[i+1]}$++.
end for

Figure 1 illustrates the EAS algorithm. The playlist on the left is read into the EAS algorithm which builds the graph depicted on the right. The edge weights between two nodes correspond to the number of playlist adjacencies between the songs that those nodes represent. When songs are never adjacent they have weight 0, and that edge is not drawn. We will calculate similarities based on these edge weights. We can aggregate the information of multiple expert playlists by repeatedly running Algorithm 1 and accumulating the weights on the same graph $G = (V, w)$.

Playlist

1	Song A	“Party in the U.S.A”
2	Song B	“Love Don't Live Here”
3	Song C	“Cowboy Casanova”
4	Song B	“Love Don't Live Here”
5	Song A	“Party in the U.S.A”
6	Song C	“Cowboy Casanova”
7	Song D	“Raise 'Em Up”

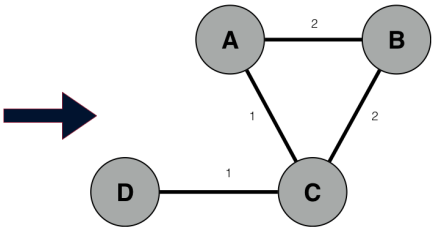


Figure 1: Building a weighted graph using Algorithm 1.

3 Converting Edge Weights to Similarities

In this section we explain how we have interpreted similarities from the output of the EAS algorithm. We calculate two different types of similarities: the one-way similarity, and the two-way similarity. We expect that the different similarities will impact the smoothness of the playlist in different ways, which should be reflected in our results.

3.0.1 One-Way Similarity

The first similarity type is what we call the one-way similarity. To calculate the one-way similarity from node u to node v, we divide the weight between the two nodes by the sum of the weights of all the edges that land on node 1. This similarity is asymmetric, and it is simply the transitional probability between two nodes (or two songs). Equation 1 details the calculation.

$$sim(u, v) = \frac{w_{u,v}}{\sum_{x \in V} w_{u,x}}. \tag{1}$$

Figure 2 provides a simple example of how we convert the edge weights from the graph produced by the EAS algorithm (left) to one-way similarities (right). To calculate the similarity between songs *A* and *B* (transitioning from *A* to *B*), we divide the edge weight between the two songs (2) by the sum of all edges that land on *A* (3), which leaves us with $2/3$.

It is important to note how this calculation will actually affect the playlist. These similarities are the probabilities of transitioning *from* one node *to* another, and they are based solely on the node that we are transitioning away *from*. This has an interesting consequence. In Figure 2, node *C* is connected to every other node in the graph, and node *D* is connected only to *C*. It is safe to assume that a node's degree indicates the corresponding song's popularity in the graph, so we can say *C* is a popular song, and song *D* is not as popular.

But there is something significant to be said about the transition between *C* and *D*. *D* rarely appears, but when it does, it appears **only** next to *C*–perhaps this apparently weak relationship is indicative of a surprisingly strong similarity. This idea motivates the two-way Similarity described in the next section.

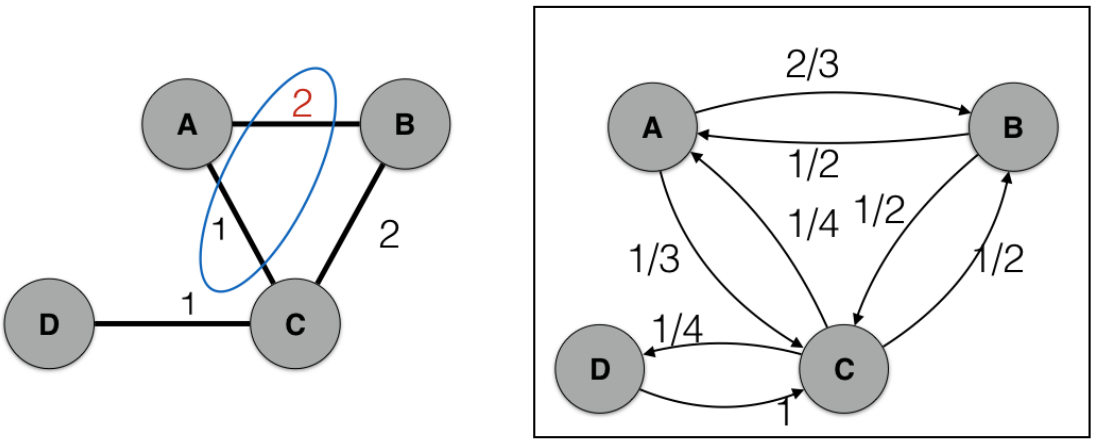


Figure 2: Calculating the one-way similarities for song A (right) based on the edge weights.

3.0.2 Two-Way Similarity

The second similarity calculation is the two-way similarity. This similarity is symmetric, i.e., the similarity from node u to node node v is the same as the similarity from node v to node u. To calculate the similarity between the two nodes, we divide the edge between the two nodes by the sum of all the edges landing on both nodes. Equation 2 outlines the calculation.

$$sim(u, v) = \frac{w_{u,v}}{\sum_{x \in V} w_{u,x} + \sum_{y \in V} w_{v,y}}. \tag{2}$$

Figure 3 provides a simple example of how to calculate the similarities (right) from the edge weights (left). To calculate the similarity between A and B, we divide the edge weight between A and B (2) by the sum of every edge landing on both A and B: A's edge weights sum to 3 $(1 + 2)$, and B's edge weights sum to 4 $(2 + 2)$, so we have $2/7$.

We anticipate that this similarity calculation will have a different effect on the resulting playlists. With this symmetric two-way similarity, we are taking into account both the node we are transitioning from *and* the node we are transitioning *to*. This levels the playing field for popular songs and less popular songs. Our example in Figure 3 illustrates this effect. The similarity between D and C is greater than that between A

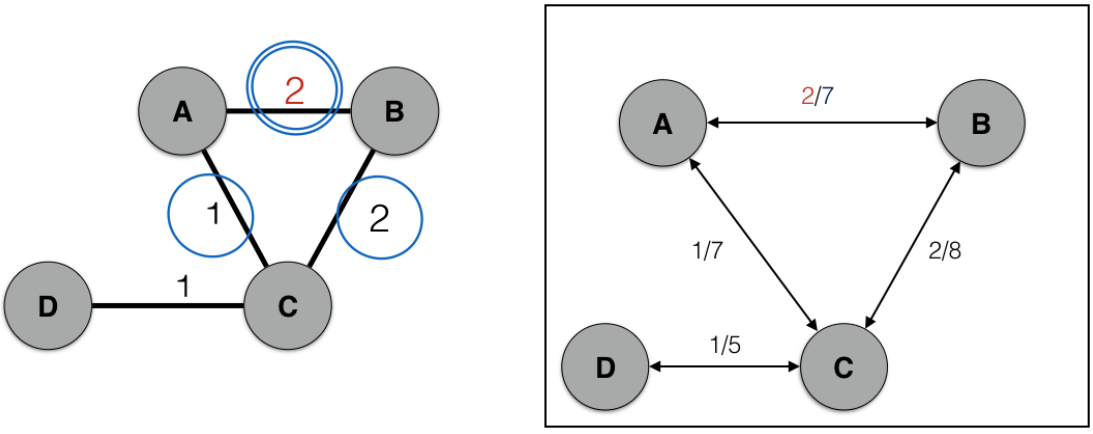


Figure 3: Calculating the two-way similarities (right) based on the edge weights.

and *C*. This similarity does not let *A*'s popularity overshadow the significance of the relationship between *C* and *D* as described above in 3.1.1.

4 Playlist Generation Algorithms

This section details the Start-End algorithm, and our improved playlist generation algorithms: the Unbiased Random Walk algorithm and the Biased Random Walk algorithm.

Note that there is one trivial difference between Flexer et al.'s implementation of the Start-End algorithm and ours: Flexer et al. pick and order songs by divergence–a measure that quantifies how different two songs are. A high divergence indicates that songs are very different, and a low divergence indicates that they are alike. In our implementation, however, we pick and order songs by similarity, which is a measure between two songs that describes how alike they are. Divergence is the inverse of similarity. This means implementing their algorithm to use similarities instead required minimal changes, and resulted in no changes in the algorithm's behavior. We denote the similarity between two songs, S_i and S_j by $sim(S_i, S_j)$; the divergence between the same two songs is equal to $\frac{1}{sim(S_i,S_j)}$.

Flexer et al. require that similarities satisfy three properties: (i) the larger $sim(S_i, S_j)$ is, the more alike they are; (ii) a song's similarity to itself must be finite; and (iii) similarities must be symmetric, i.e.,

$sim(S_i, S_j) = sim(S_j, S_i)$. The two-way similarity calculated using the techniques from Section 3 meet all these requirements. The one-way similarity, however, meets all but the last. We set self-similarity to some number between 0 and 1. While the one-way similarity is not symmetric, we believe asymmetry will not prevent the algorithm from running as intended. (Figure 4).

Requirement	Description
"Low to High"	A low similarity indicates that two songs are dissimilar.
"Finite Self-Similarity"	A song's similarity to itself cannot be infinity.
"Symmetry"	Similarity from A to B equals similarity from B to A.

Figure 4: Similarity Requirements

In the following discussion of playlist generation algorithms we will refer to a playlist as a sequence of p songs, excluding the start and end songs, S_s and S_e. A playlist is derived from a library of n songs, $(S_1, S_2, \ldots, S_n)$.

4.1 Start-End Algorithm

Conventional automatic playlist generation tools, such as Spotify Radio and Pandora, are primarily a function of a start song (and in some cases intermittent user-feedback). These services simply prompt the user for a start song, and then deliver a playlist based on that song. The Start-End algorithm is different; it is primarily a function of a start song *and* and an end song. It is this unconventional feature that motivates this research. The Start-End algorithm consists of two (supposedly) independent parts:

1. Read in a library of songs, then determine the similarity between each song (again, in their implementation Flexer et al. compute song similarity based on"spectral similarity" which is a form of audio fingerprinting; we use the EAS algorithm).

2. Select and order the songs based on their similarities to smoothly transition from a start song to an end song. This is the part of the algorithm that we classify as a playlist generation algorithm because

it actually creates the playlist.

In our research, we replace Step 1 with the EAS algorithm. The Start-End algorithm is detailed below:

Algorithm 2 Creating a playlist using the Start-End algorithm.

1: **Input:** A database of n songs (S), a start song and an end song (S_s and S_e, respectively), a threshold (t), and a similarity graph (from the EAS algorithm) ($simGraph$), we will calculate a playlist of length, p.
2: **Output**: A smooth list of p unique songs with that begins at S_s and ends at S_e.
3: Create an ordered list, $startSimilarities$, of S such that the song with the highest similarity to S_s is at the head of the list, and the song with the smallest similarity to S_s is at the tail.
4: Create an ordered list, $endSimilarities$, of S such that the song with the highest similarity from S_e is at the head of the list, and the song with the smallest similarity from S_e is at the tail.
5: Use t to calculate Threshold Index: $thresholdIndex = \lfloor t * n \rfloor$.
6: Eliminate $t\%$ songs from S by removing $t\%$ songs that are least similar to both the start song and the end song. This leaves us with a m songs for further processing.
7: Calculate Playlist Step-Width: $step = \frac{\hat{R}(S_s) - R(S_e)}{p+1}$, where R(S) = $\frac{sim(S,S_e)}{sim(S_s,S)}$.
8: Find ideal song position, $\hat{R}(j) = \hat{R}(S_s) + j * step$.
9: Select p songs, S_j, that best match these ideal positions: $S_j = \underset{i=1,\ldots,m}{\operatorname{argmin}} \|\hat{R}(j) - R(i)\|$.

Figure 5 is a number line depicting step 7 of the Start-End algorithm. Each mark on the number line represents an ideal position, and each position is separated by the step width calculated in in step 6 (in this example, the step width is .4). The numbers below the number line are the *ideal* similarity ratios at each position. The songs are depicted above the number line according to their *real* similarity ratio (e.g., song D has a real similarity ratio between ideal similarity ratios .4 and .8).

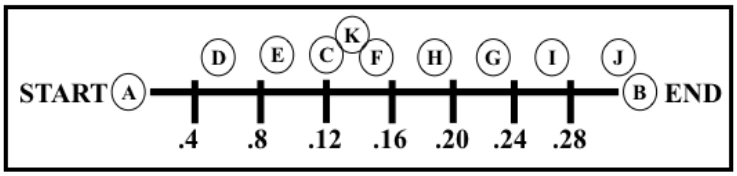


Figure 5: Ideal Similarity Ratios and Real Similarity Ratios

Figure 6 illustrates the matching process outlined in step 8. This step is responsible for ordering the playlist. The resulting playlist is **ADECFHGIB**.

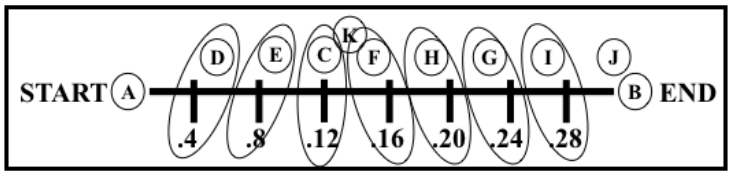


Figure 6: Selecting Songs Based on Ideal Similarity Ratios and Real Ratios

4.2 Unbiased Random Walk Algorithm

In this section we detail the Unbiased Random Walk algorithm. Like the Start-End algorithm, this algorithm transitions from a start song to an end song. The goal of the URW algorithm was to ensure that each song in the playlist has at worst a non-zero similarity to the previous song, and at best a high similarity to the previous song. Recall that the structure of the EAS algorithm's output is a graph. We can use this to our advantage to create a playlist generation algorithm. We want to perform a random walk from the start node to the end node in the given number of steps. The URW algorithm is outlined below:

Algorithm 3 URW Algorithm

Input: A number of attempts, $attempts$, a greediness parameter, z, a playlist length and range, $length$ and $range$, a similarity graph (from the EAS algorithm), $simGraph$, and a start song and end song, S_s and S_e, respectively.
Output: A smooth sequence of unique songs with a length between $length - range$ and $length + range$ that transitions from S_s to S_e, or, an empty playlist.
$runner = startSong$
$playlist$ = [$startSong$]
while $len(playlist) <= length + range$ **do**
  select z random neighbors of $runner$; let *next* be the neighboring node most similar to $runner$
  $append$ next to playlist
  $runner = next$
  if ($runner == endSong$) && ($length - range <= len(playlist <= length + range$) **then**
    return $playlist$
  end if
end while
return empty $playlist$

We can make the URW algorithm greedier by increasing the z parameter. As z increases, the algorithm gets less random and more greedy because it is picking more of the runner's neighbors, so the odds are one

of them has a higher similarity to the runner node. In the limit of large z, the algorithm always picks the most similar neighbor.

Algorithm 3 is a Monte Carlo Algorithm, that is, it does not *guarantee* that we find a playlist, so we run this algorithm until it either finds a playlist or reaches the $attempts$ parameter. It returns an empty list if $attempts$ is reached.

4.3 Biased Random Walk Algorithm

The Biased Random Walk algorithm is a slight variation on the Unbiased Random Walk algorithm (the required inputs are the same). The distinguishing feature of this algorithm is that we do not choose the next song completely at random and as we do in the non-greedy URW algorithm–we pick based on probability determined similarity. We can do this easily using the one-way similarities because they function exactly like transitional probabilities. To use this method with the two-way similarities, we have to normalize the similarities so that all edges leaving a node sum to 1. This allows us to treat them like probabilities. The BRW algorithm is outlined in Algorithm 4.

An example may clarify the algorithm. In Figure 2, if our start song is **A**, and our randomly chosen r value is $.9$, we would choose **C** as our next song.

Algorithm 4 BRW Algorithm

Input: A number of attempts, $attempts$, a greediness parameter, z, a playlist length and range, $length$ and $range$, a similarity graph (from the EAS algorithm), $simGraph$, and a start song and end song, S_s and S_e, respectively.
Output: A smooth sequence of unique songs with a length between $length - range$ and $length + range$ that transitions from S_s to S_e, or, an empty playlist.
$runner = startSong$
$playlist$ = [$startSong$]
while $len(playlist) <= length + range$ **do**
  $acc = 0$
  create list of z random numbers, R, such that $0 <= r <= 1 \forall r \in R$.
  create an ordered list, $neighbors$, such that $neighbors[0]$ has the highest similarity to $runner$
  for $node$ in neighbors **do**
    if $(min(R) <= similarity(node, runner)$ **then**
      $playlist.append(node)$
      $runner = node$
    else
      acc += $similarity(node, runner)$
    end if
  end for
  if ($runner$ is $endSong$) && ($length - lengthRange <= playlistLength <= length + lengthRange$) **then**
    return *playlist*
  end if
end while
return empty *playlist*

Like in the URW algorithm, we can make the URW algorithm greedier by increasing the z parameter. As z increases, the algorithm gets less random and more greedy because it is picking z random numbers, and setting the minimum to *r*. As the *r* value decreases, the similarity between the *runner* and the next node increases.

Agian, Algorithm 4 also does not *guarantee* that we find a playlist, so we run this algorithm until it either finds a playlist or reaches the $attempts$ parameter. It returns an empty list if $attempts$ is reached.

4.4 Deterministic URW & BRW

The URW and BRW algorithms turn deterministic when the greediness parameter reaches infinity (see Figure 7). In this case both algorithms pick the most similar choice for the next song, and so, in theory, they will produce the same playlist.

But the most greedy choice is not always the best choice. The algorithms are actually more likely to fail in this case (i.e., produce an empty playlist). When the greediness parameter reaches infinity, the algorithms are tasked to find a path between the start song and the end song such that each song is followed by its most similar neighbor; this path is unlikely to exist. As the greediness parameter increases beyond a certain point, the algorithms are actually more likely to fail.

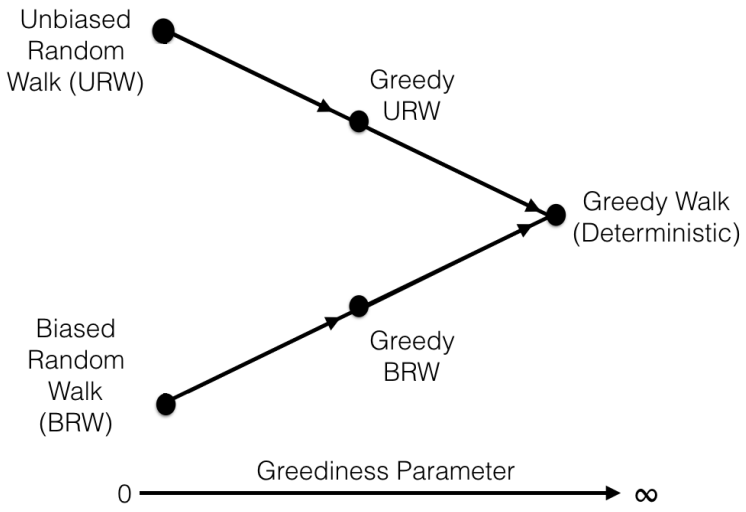


Figure 7: As the URW & BRW algorithm's greediness parameter approach infinity the algorithms converge.

5 Data & Implementation

In this section we discuss the data used and the implementation of each playlist generation algorithm. The data consisted of thousands of expertly authored playlists, and the playlist generation algorithms were implemented in Python.

5.1 Data

One expertly curated playlist will produce a small graph with few connections. But we can expand upon a graph produced by the EAS algorithm as many times as we would like in order to make it a large (and hopefully denser) graph by simply iteratively reading in and incorporating more expertly curated playlists into our graph. We scraped playlists from Spotify using the Spotify Web API.[3] We ended up with over $250,000$ songs from more than $2,600$ playlists. We used a Python library to interface with the web API called SpotiPy.[4]

Our initial tests were run using a significantly small, manually scraped data set from Spotify. The data set was comprised of about 200 playlists and $16,000$ songs. The algorithms appeared to be running correctly, but produced empty playlists. We realized that we were not taking the density of the graph into account. The Start-End algorithm and our algorithms both depend on the existence of many relationships between songs. A sparse graph lacks these relationships, and so the algorithms struggle to piece together a playlist.

In an attempt to make our graph more dense, we scraped playlists that we were able to identify as "country playlists". We made the identification using simple keyword searches. By only looking at country playlists, we were more likely to have repeated songs, and therefore more connections in our graph. We used playlists that were created by Spotify or created by a user that Spotify has acknowledged as an expert.

5.2 Implementation

We compile our *SpotiPy* scrape into a text file, and read it into the system in `RawDataEntry.py`, where we do some reformatting to make it easier to work with. We turn each song into a list of strings (e.g.,

[3] https://developer.spotify.com/web-api/
[4] http://spotipy.readthedocs.org/en/latest

["ArtistName","SongName", "AlbumName", "ID"]). Once the data is handled in `RawDataEntry.py`, we build a song library in `SongLibrary.py`. This functions just like a dictionary–mapping each song to its unique ID. This simplifies the expertly authored algorithm so we only have to work with IDs instead of working with all of the song's information when building the graphs. We build graphs according to the

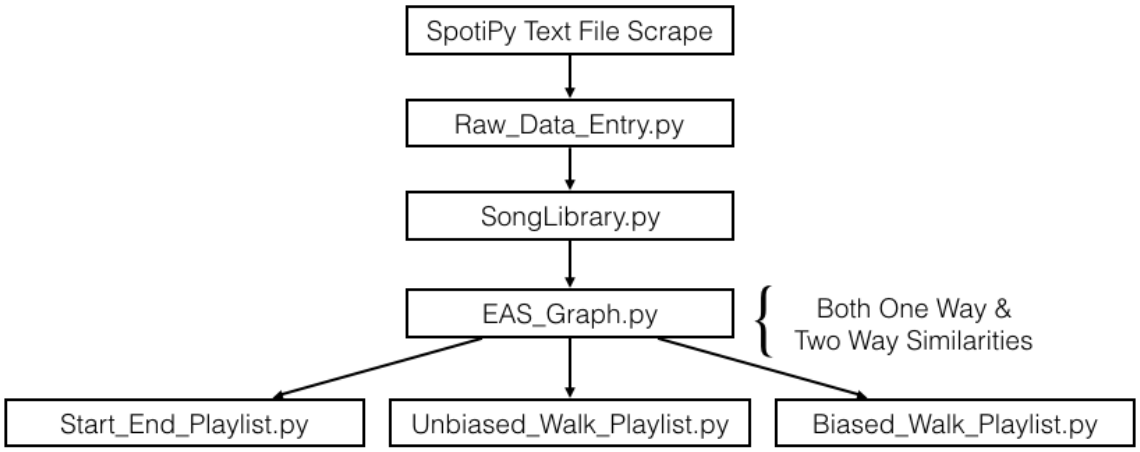


Figure 8: Implementation Structure

EAS algorithm (see Section 2) in *OneWay_EASGraph.py* (this graph will contain one-way similarities) and *TwoWay_EASGraph.py* (this graph will contain two-way similarities). Once we have a graph that tells us song similarities, we can create a playlist with a playlist generation algorithm.

Once every algorithm was implemented and running, we created a program ($RunBatch.py$) that allows us to generate hundreds of playlists at once. We also built a file ($Matrices.py$) that we used to visualize the data. Figure 8 depicts a file hierarchy of the implementation.

6 Results

In this section we present the results for the Start-End algorithm, the URW algorithm, and the BRW algorithm. For the URW algorithm and the BRW algorithm, we ran three trials for each algorithm: one trial with greediness parameter 1, one trial with greediness parameter 4, and one with greediness parameter 7.

We randomly generated pairs of songs (a start song and an end song) to be used as input for each playlist generation algorithm in order to produce multiple playlists at once. The adjacent pairs used were different for each playlist generation algorithm, and different for each trial of the URW and BRW algorithm. The number pairs used by the algorithms ranged between 200 and 260.

6.1 Start-End Algorithm Results

No results were generated using the Start-End algorithm. This reveals a flaw in the Start-End algorithm. Every song in the playlist must be connected to both the start song and the end song because the algorithm picks songs based on similarity ratio (otherwise, the similarity ratio $R(i)$ is either 0 or undefined). This flaw is discussed in detail in Section 7.

6.2 Unbiased Random Walk Algorithm Results

The URW algorithm gave us more interesting results. The results depicted in Figures 10-15 are the average results of the 239 playlists that we generated. The playlists created ranged between 15 and 25 songs in length (the *length* parameter was 20 and the $range$ parameter was 5), the results in the figures depict the desired length 20.

The greyscale matrices included in this section are what we will refer to as "Similarity Matrices". These matrices indicate the similarities between all songs in the playlist. To clarify, Figure 9 is an example of what a similarity matrix might look like. Each box is shaded based on similarity; the darker the box, the more similar the corresponding songs are. Take the box that corresponds to the row S_s and the column S_1, for example. We know that these two songs are more similar than S_2 and S_e because of the darker box at (S_s, S_1). The light diagonal stripe is the result of the low self-similarity.

We ran the algorithm using greediness parameters 1 (not greedy), 4, and 7. The playlist *length* parameter was 20 for all of the data collected, and the *range* parameter was 5. This was done for both one-way similarity and two-way similarity.

Figure 10 depicts the adjacent similarities of the URW algorithm with one-way similarities for greediness parameters 1, 4, and 7. Adjacent similarities are the similarities between a song and its successor. This

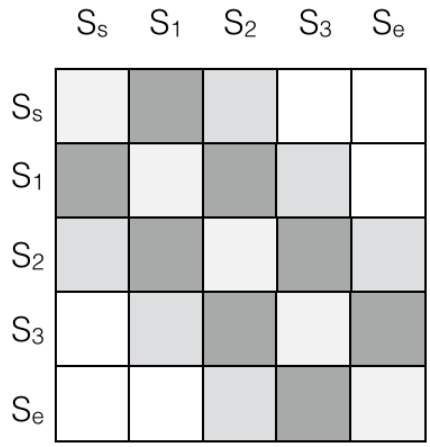


Figure 9: A Similarity Matrix

graph is intended to show us how smooth the transitions between are–the higher the point, the smoother the transition.

Figure 11 depicts the average adjacent similarities using one-way similarity. This graph allows us to visualize the impact that the greediness parameter had on the smoothness of the playlist. Figure 12 and 13 depict the same graph as Figures 10 and 11, respectively, but use the two-way Similarity. Figure 14 and Figure 14 are the similarity matrices for the same data.

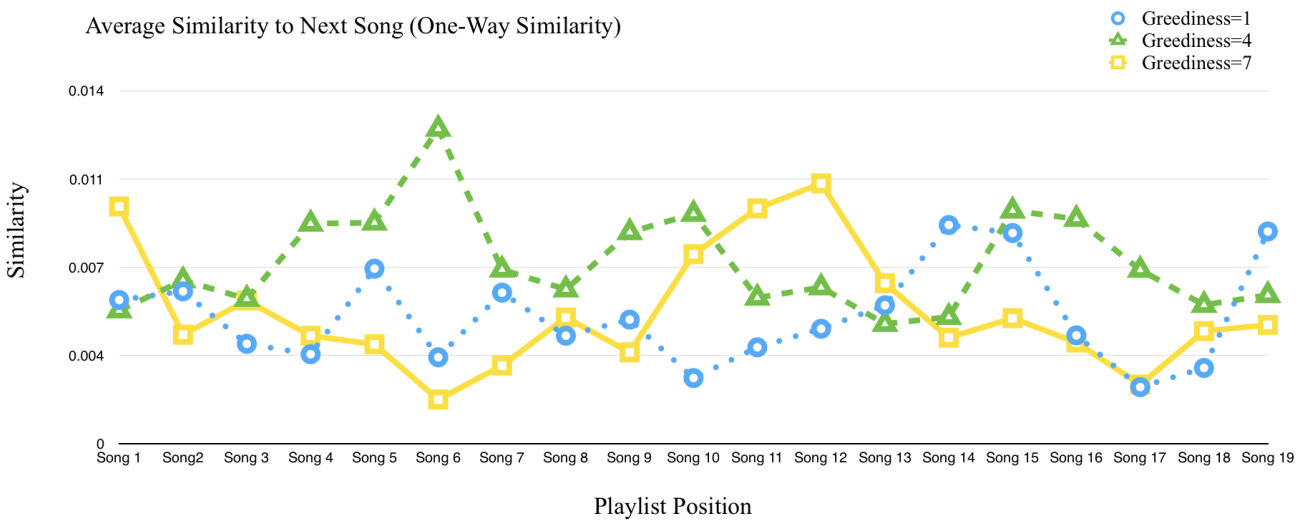


Figure 10: Average similarity to next song using one-way similarities

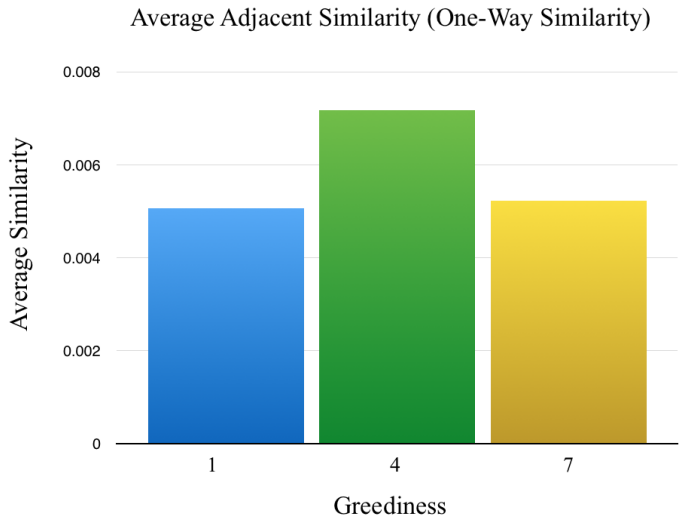


Figure 11: Average adjacent similarities using one-way similarities

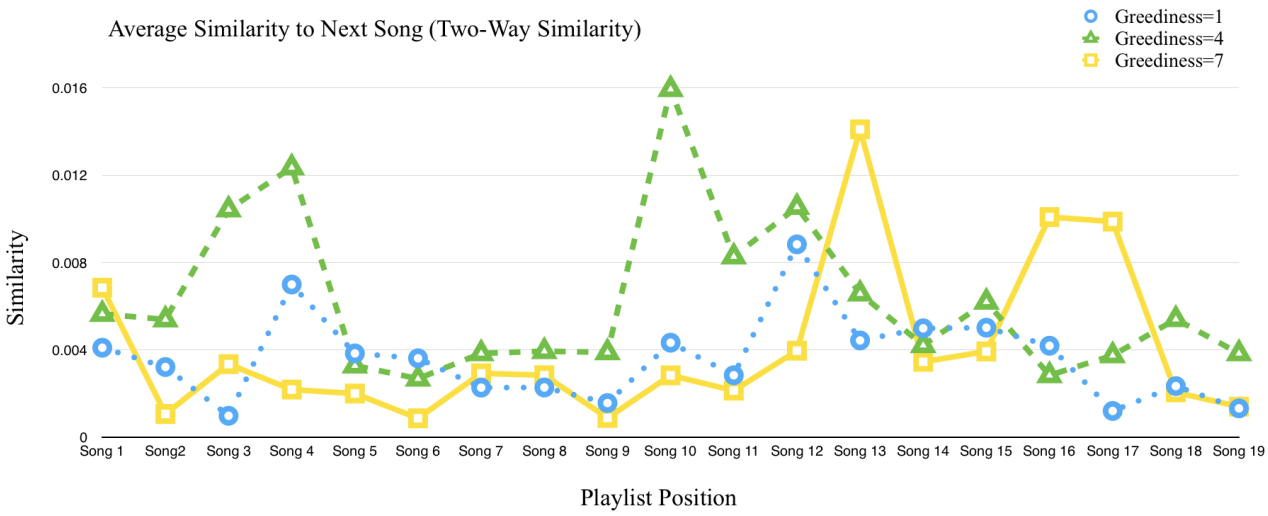


Figure 12: Average similarity to next song using two-way similarities

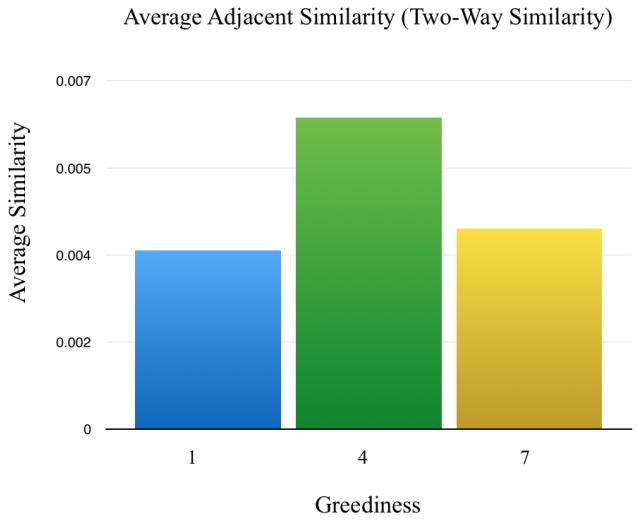
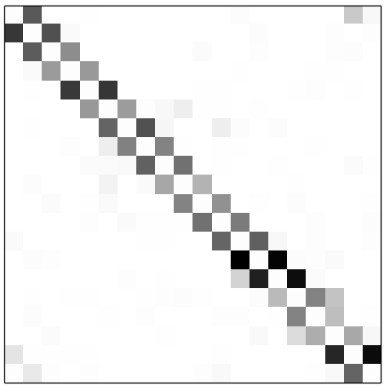
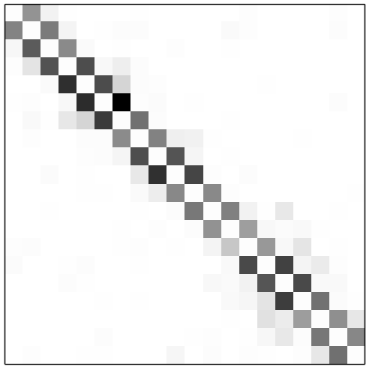


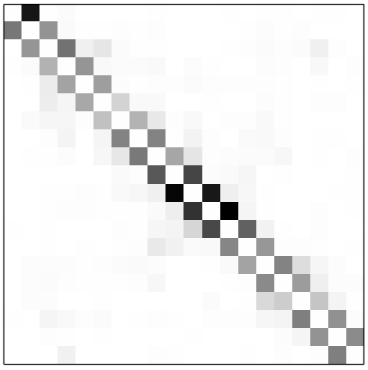
Figure 13: Average adjacent similarities using two-way similarities.



(a) One-way similarities; greediness parameter: 1

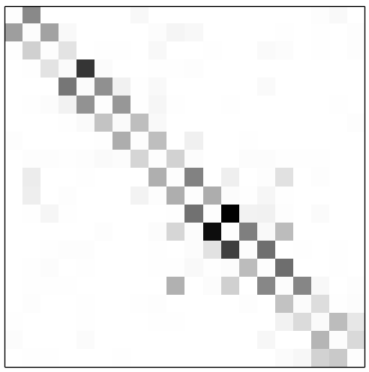


(b) One-way similarities; greediness parameter: 4

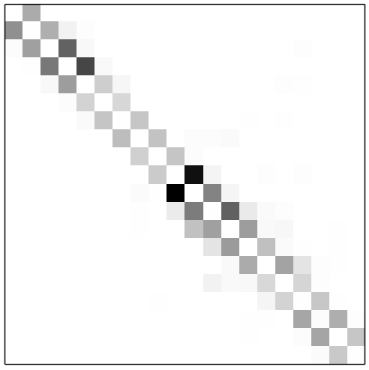


(c) One-way similarities; greediness parameter: 7

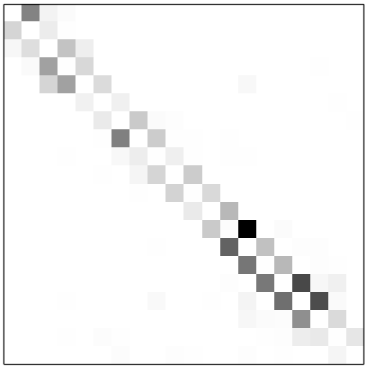
Figure 14: Similarity Matrices for URW Algorithm for the One-Way Similarities



(a) Two-way similarities; greediness parameter: 1 (not greedy)



(b) Two-way similarities; greediness parameter: 4



(c) Two-way similarities; greediness parameter: 7

Figure 15: Similarity matrices for the average URW Algorithm for the two-way similarities

6.3 Biased Random Walk Algorithm Results

We did not collect results for the BRW algorithm due to lack of time. We anticipate the BRW algorithm to behave in a similar manner to the URW algorithm because of how closely related the algorithms are. For this reason, we believe there is a bug in our implementation that has prevented us from collected results.

7 Analysis

In the following discussion we analyze the results of the three playlist generation algorithms. Our aim is examine the data in the results in a way that will expose the strengths and weaknesses of each algorithm in order to determine whether or not our algorithms out performed the Start-End algorithm.

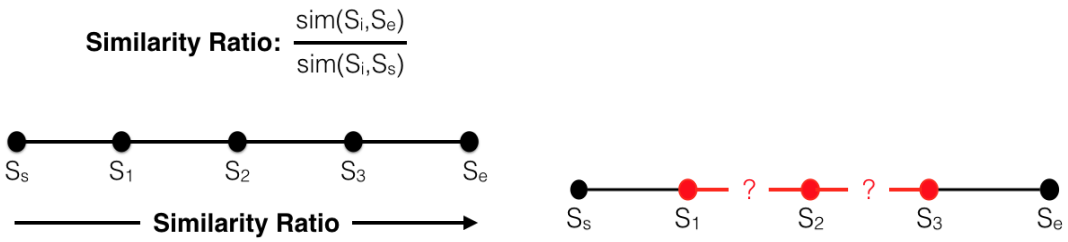
7.1 Start-End Song Analysis

The major flaw in the Start-End algorithm that prevented us from collecting results is that every song in the resulting playlist must have a non-zero similarity to **both** the start song and then end song (this includes the start song and the end song–they must have a non-zero similarity with each other). This is true because the similarity ratio (See Section 4.1, step 5) of the start song and the end song is needed for the step-width calculation. While this limiting factor is not explicitly made clear by Flexer et al. in their paper, it is evident when running the algorithm. If every song in the playlist is *not* connected to both the start song and the end song, we cannot calculate the step with (see Section 4.1, step 6).

This is an incredibly restrictive requirement that must be considered when analyzing the algorithm. Only songs that have non-zero edge weights to the start song and the end song are to be included in any playlist produced by the Start-End Algorithm; thus, we are forced to disregard the songs that do not fall into this category, and we are left with a fraction of the songs in our graph. Building a playlist of length n is only possible if the start song and the end song have n common neighbors.

Another issue with the Start-End algorithm was that it never ensures adjacent similarity–i.e., high similarity between a song and its successor; or, more colloquially, a "smooth transition". Figure 16 illustrate this phenomenon.

The Start-End algorithm picks and orders songs based on increasing similarity ratio. Because we have a constant step-width (see Section 4.1, Step 6), we might incorrectly assume that similarities between adjacent songs will be constant too. Figure 16a illustrates this assumption: the distance between song in the playlist corresponds to the similarity. In Figure 16b, however, we alter the similarity between S_1 and S_2, and S_2 and S_3 without changing the similarity ratio, but still end up with the same playlist. These similarities are never accounted for, and so the smoothness of the playlist can suffer.



(a) (b)

Figure 16: The Start-End algorithm allows similarity to change without a change in similarity ratio.

7.2 Unbiased Random Walk Algorithm Analysis

The data collected from the URW algorithm provided insights into the algorithm. Specifically, the data pointed to three interesting trends.

7.2.1 Greediness "Sweet-Spot" For Adjacent Similarity

We predicted that there would be a positive direct relationship between the greediness parameter and playlist smoothness–when the URW algorithm is non-greedy, it picks the next song completely randomly, and so the transitions may not be particularly smooth; as it gets greedier, however, it picks based on similarity, making the transitions smoother. This predicted relationship holds true to a degree. See Figure 11 and Figure 13. The average adjacent similarity between songs is significantly higher with a greediness parameter of 4 than it is with a greediness parameter of 1 or 7.

It appears as though the the greediness parameter has a "sweet-spot" that maximizes adjacent similarity.

As the algorithm gets greedier, it approaches a deterministic version of itself in which it would always add the most similar song (see Figure 7), which can lead to a dead end, and thus, no playlist. Figure 17 illustrates this concept: Transitioning from song A to song F is very unlikely even in the non-greedy algorithm. If we increase the greediness parameter, we will never arrive at song F. It is clear that the overly greedy URW algorithm has a dangerous side effect.

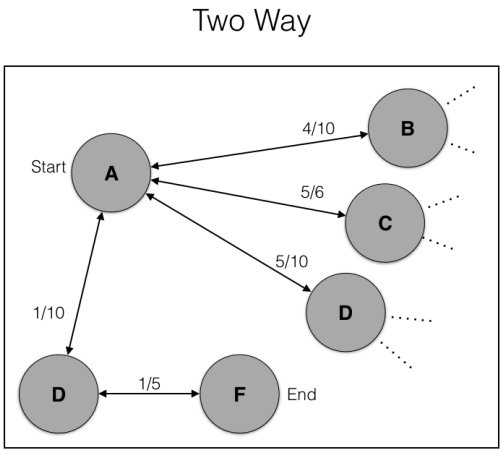


Figure 17: Being too greedy can lead to a dead end.

7.2.2 Greediness Does Not Indicate Playlist Cohesion

See Figure 14 and Figure 15 The dark bands running diagonally across the matrix indicate adjacent smoothness. Darkly shaded boxes that are not on this diagonal band indicate that songs that are not adjacent to one another are similar–we can get a general understanding of how cohesive our playlist is. A cohesive playlist is one in which there exist high similarities between songs that are adjacent, and also between songs that are not adjacent. In Figure 14a, we see few of these high non-adjacent similarities throughout our playlist. While the playlist does have adjacent similarity (indicated by the dark diagonal band), the playlist is not very cohesive. However, as the greediness parameter increases (see Figure 14c), we see more dark boxes surrounding the diagonal band. This indicates that songs that are a few songs away from each other are also fairly similar. The playlist is slightly more cohesive.

In Figure 14b the increase in playlist cohesion is less evident, and in Figure 15 we do not see cohesion increase as greediness increases. The figures clearly indicate that playlist cohesion is not predicated on greediness.

However, this behavior can be expected. Our algorithms do nothing to ensure playlist cohesion. Had that been a goal of the algorithm, we would have to construct the algorithm differently. One method we might use to ensure playlist cohesion is to select songs based on similarity to not only the previous song, but the songs that come before it.

7.2.3 Spikes Indicate Forced Bad Decisions

In Figure 10, the lines graphs each songs similarity to its successor song. The dashed line corresponds to the data collected using a greediness parameter of 4. The steep increase in similarity between Song 5 and Song 6 reflect a good decision–while the similarity between Song 5 and Song 6 is below $.011$, the similarity between Song 6 and Song 7 is almost $.014$. However, instead of increasing again, or plateauing, the similarities dip back down, forming a spike. In both Figure 10 and Figure 12, we see these spikes appear throughout the data.

From these spikes we can infer that a pair of adjacent songs with a high similarity is often followed by a pair of adjacent songs with a relatively low similarity.

Recall that the URW algorithm does not guarantee a playlist on the first attempt (See Section 4.2), and that we run the algorithm hundreds of times until we find a playlist. While it may sound obvious, this data represents cases where the algorithm actually succeeded: the spikes in the graphs tell us *why* these attempts succeeded. The spikes show us that in order to reach the end song, you may be forced into some bad decisions.

7.2.4 Flaws of the URW Algorithm

The URW algorithm has some flaws of its own. While tweaking our implementation of the URW algorithm may enhance performance slightly, there is no avoiding the trial-and-error-nature of the algorithm, which slows performance significantly.

Another issue with our algorithm is that it works best (and when the start and end song have at least one neighbor in common). This was a restrictive requirement of the Start-End algorithm that hoped to do away with in the URW algorithm. While the URW algorithm does not *require* that start song and the end song have a neighbor in common, it appears to function best when they do.

7.3 Biased Random Walk Algorithm

The BRW algorithm should behave similarly to the URW algorithm in that it should guarantee adjacent song similarity to some degree. Like the URW algorithm, each song that is added to the playlist must have a non-zero edge weight to the song before it.

However, we found that the BRW algorithm struggles to produce playlists. This may be because it is fundamentally greedier way than the URW algorithm. This is due to the normalization of the similarities. We expect that normalizing the similarities makes the many of the similarities very small. When we make the ordered list of *runner* node's neighbors in in Algorithm 4, the end of the list is likely to be cluttered by these neighboring nodes that have a low similarity to the *runner* node. In turn, the algorithm will be less likely to choose these songs and add them to the playlist.

8 Conclusion

We sought to determine the effectiveness of the Start-End algorithm and attempted to improve it. Our results and analysis indicate that the Start-End algorithm functions as the authors intended (the algorithm's flaw is not a result of faulty implementation) but behaves in a way that may not be ideal for the user, i.e., it does not produce a smooth playlist because in an EAS, many similarities are 0. The lack of adjacent similarity translate to a lack of smooth transitions. We conclude that the Start-End algorithm was that it requires an unrealistic data set in order to work when using the EAS algorithm to determine song similarity. In order for the Start-End algorithm to work alongside the EAS algorithm would require collecting even more data, or extending the existing data. There are several ways data can be extended–all relate to calculating similarity.

Ideally, we would be capable of determining the similarity between every pair of songs in the data set. To do that, we would need non-zero edge weights between every node in the EAS graph (i.e., a complete graph). As noted, however, the EAS produces a graph with many similarities of 0, and so the graph is far from complete. However, if we create a similarity matrix the containing every song our existing data set, we can raise the matrix to a power large enough to make all the zero similarities non-zero. Unfortunately, this method can get computationally expensive, and it would be better if we did not have to manipulate our data. So, with the URW algorithm BRW algorithm we attempted to address the issues apparent in the Start-End algorithm without having to extend the data.

The results reveal that the URW algorithm is more effective than the Start-End algorithm. By not calling for similarity ratios, the URW algorithm does not require that each song in the playlist have a non-zero similarity to both the start song and the end song. This allowed our algorithm to work with our data set. Furthermore, the URW algorithm created a smoother playlist. This claim is supported by our data–every song in the playlist has a non-zero similarity to the following song which results in a smooth transition from start song to end song.

References

[1] Playlists. https://www.youtube.com/yt/playbook/en-GB/playlists.html/. Accessed: June 2nd, 2015.

[2] The spotify playlist system. https://developer.spotify.com/web-api/ working-with-playlists/. Accessed: June 2nd, 2015.

[3] Streaming is now the mainstream. http://advertising.pandora.com/2015/04/ 30/streaming-is-now-the-mainstream-how-young-listeners-are-driving-online-radio-usage/. Accessed: June 2nd, 2015.

[4] Why mighty google still needs songza's human-made mixtapes. http://www.bloomberg.com/bw/articles/2014-07-02/ google-buys-songza-and-its-human-made-mixtapes/.

[5] Arthur Flexer, Dominik Schnitzer, Martin Gasser, and Gerhard Widmer. Playlist generation using start and end songs. In *ISMIR*, pages 173–178, 2008.

[6] J.C. Platt. Auto playlist generation with multiple seed songs, January 17 2006. US Patent 6,987,221.

[7] Robert Ragno, Christopher JC Burges, and Cormac Herley. Inferring similarity between music objects with application to playlist generation. In *Proceedings of the 7th ACM SIGMM international workshop on Multimedia information retrieval*, pages 73–80. ACM, 2005.

[8] Chareen Snelson. Youtube and beyond: Integrating web-based video into online education. In *Society for Information Technology & Teacher Education International Conference*, volume 2008, pages 732–737, 2008.