Musical Relationships

Exploring Connections Between Songwriters

Hannah An, Jack Armstrong, Li-An Chu, Zack Draper

March 10, 2017

I. Introduction

In this paper we will explore the connections that exist between songwriters in the popular music industry. Using the techniques of graph theory that we have developed in this course, we will model the community of songwriters as a network of individuals, (represented by vertices) and collaborations (represented by the edges that connect them). From the outset of our research, we have been interested in finding the most central individuals in this musical collaboration graph as well as the most central genres. In addition to this, we want to identify clusters of well-connected people in order to better understand the types of songwriters that collaborate with one another, and find out who collaborates the most. In order to put our calculations into context, we will explore some of the similarities and differences between our musical collaboration graph and other social networks. By investigating a variety of different properties of our graph, we hope to shed some light on the hidden connections that exist within this multi billion-dollar industry.

II. Background

History

The origins of graph theory date back to the 18th century when Leonhard Euler, the famous Swiss mathematician, investigated a puzzle known as the Seven Bridges of Königsberg problem.[1] The city of Königsberg, Prussia (now Kaliningrad, Russia) was divided by the Pregel River. Inside the river were two large islands, which were connected to each other and to the mainland by a system of seven bridges. The puzzle was to find a path through the city that would cross each of the seven bridges once and only once. Euler's insight was to simplify the problem by representing the landmasses as vertices and the bridges as edges that connect them. Euler observed that, except at the endpoints of the walk, a path that enters a vertex by one bridge must leave the vertex by another bridge. Therefore, unless the vertex in question acts as the starting point or ending point of the path, in order for a solution to exist, each vertex must be connected by an even number of edges. Following this line of reasoning, Euler showed that if a graph (the collection of nodes and edges) contains more than two vertices with an odd number of edges, then no such path between the nodes could exist.[6] Because all four of the landmasses in

Königsberg were connected by an odd number of bridges, Euler was able to prove that no path through the city crosses each of the seven bridges once and only once. In solving this problem Euler provided the foundations for graph theory.

Since graph theory was first developed nearly three hundred years ago, it has been used in countless areas of science. It has been used to analyze electrical circuits, diagram the structure of molecules, model communication and transportation networks, and even explore connections in the human brain.[1] Graphs are especially useful in exploring the relationships between people. These so-called "social graphs" are some of the most well known examples of graph theory. In the 1950's and 60's mathematicians began to study the connectedness of people by creating graphs in which vertices represented people and edges represented connections between them. During this time, mathematician Manfred Kochen and political scientist Ithiel de Sola Pool wrote an important manuscript, "Contacts and Influences", which laid the groundwork for the mathematical structure of social networks. Sociologists and mathematicians were particularly interested in finding how far apart two randomly chosen people would be from one another. That is to say, they wanted to know the minimum number of intermediaries separating two average people in a graph. The theory was that the number of vertices separating two randomly selected people living in the U.S. would be surprisingly small—perhaps only two, three, or four.

This estimate, which came to be known as the "small-world hypothesis" was put to the test in a famous experiment conducted by Stanley Milgram. He addressed packets to an individual living in the Boston area and asked volunteers living in Nebraska and Kansas to send the packets to personal acquaintances who might be able to get them closer to the intended recipient. Inside the packets were instructions to follow the same procedure. Although most of the letters never made it to the destination, among those that did, the average path length was between five and six steps.[1] There are many methodological critiques of the experiment suggesting that the average path length might be shorter or longer than this, but regardless of these criticisms, Milgram's research struck a chord with the public. Despite the fact that Milgram never used the phrase, this experiment is credited with popularizing the concept of "six degrees of separation."

There are many different kinds of social graphs that reside inside the world-wide acquaintanceship graph. Two of the most well-known and well-studied social graphs belong to the worlds of mathematics and film. The mathematical collaboration graph is defined such that an edge exists between two people if they have coauthored a mathematical paper together. The central figure of this graph—in other words, the person with the smallest average distance to all other mathematicians—is Paul Erdős. A prolific mathematician, Erdős published over 1,500 mathematical papers, a figure that remains unsurpassed today.[2] Because of his central place in the collaboration graph, the concept of Erdős numbers, which measure the shortest path between a mathematician and Paul Erdős, was born. A mathematician is considered to have an Erdős number of one if they have published a paper with Paul Erdős. They have an Erdős number of two if they have not published a paper with Erdős, but they have published a paper with someone who has published a paper with Erdős. This pattern goes on indefinitely. A similar concept exists in the film industry, in which people measure their separation from actor Kevin Bacon. In this graph, the vertices represent actors and the edges connect those who have appeared in a movie together.

Mathematical Tools

It is one thing to simply tabulate the connections that exist in a graph, and another thing entirely to understand these connections. Due to the large number of vertices, these collaboration graphs are difficult, if not impossible, to visualize. For instance, the mathematical collaboration graph contains more than 400,000 different authors,[2] while the film equivalent contains nearly three million actors.[5] Mathematical tools have been developed in order to understand the connections that exist within these large graphs. One method for investigating these graphs is to calculate the degree distribution of the vertices. (The degree of a vertex is the number of edges that connect to it.) For the case of the mathematical collaboration graph, Jerrold W. Grossman investigated this in an excellent paper entitled "The Evolution of the Mathematical Research Collaboration Graph."[3]

It is also possible to introduce weighting to these collaboration graphs, by assigning a value to each edge that depends on the number of times two people have collaborated. In the paper "Rational Erdős Numbers,"[4] Michael Barr explores one method of weighting graphs in which each edge is assigned a weight of $1/n$ where n is the number of times two people have collaborated. His argument is that people who have collaborated together multiple times should be closer together than those who have collaborated only once. To complete his analysis, he introduces a new method for calculating the distance between two people, which is very similar to the way resistances are added in electrical circuits.

Our Inspiration

The concept of representing social connections using the mathematical tools we have developed in this class inspired us to model a real world community using graph theory. We noticed that, unlike connections within the movie industry, which have been well-modeled using graphs (and for some reason have become synonymous with actor Kevin Bacon), connections within the music industry have not been as extensively explored. This surprised us, because collaboration graphs have been used to model everything from chess, to baseball, and have been used to investigate the overlap between the seemingly unrelated fields of mathematics and movies (leading to the unusual concept of Erdős-Bacon numbers). What really piqued our interest was the idea that we might be able to find the most central figure in music, just like Paul Erdős is the most central figure in math. We also thought that, since music is such an important part of popular culture, a wide audience might find our results interesting and informative.

III. The Model Itself

We define our songwriters graph as $G = (E, V)$, where V is the set of all people who have written songs and $(v_1, v_2) \in E$ if v_1 and v_2 are songwriters that have collaborated to write a song, for all v_1 and v_2 in V. Because our goal is to study how songwriters are connected, we will focus on popular and well-known songs, whose writers will likely have written other popular songs. We use the Hot 100 chart that Billboard publishes weekly, and we consider the writers of all songs that made it to a peak position of at least 50 on the chart between January 1, 1959, and February 25, 2017.

Unfortunately, there are a few issues with this model that mean it will not completely represent connections among popular songwriters. Primarily, what it means to be the writer of a song is not always completely clear, because a songwriter may have written the lyrics to a song or they may have written the score. Especially in modern music there are a great many layers to songwriting including producing, mixing, etc. An artist may even be listed as a songwriter if one of their works is sampled in a song and if otherwise the artist had no part in writing the rest of it. Because credits for songwriting are rather decentralized across the Internet, and there is no true equivalent of the IMDb for songs, artists may be listed as writers for a song on some websites but not on others.

Since our graph considers many songs over a long time period, we first simplify our model by considering just one website, allmusic.com, which has a very extensive database of songs and lists writers for the vast majority of them. We conducted a simple web scrape of allmusic.com as well as the online charts that Billboard publishes in order to collect writer data for each song. We consider two artists to be writers of the same song if they are given the generic credit of “writer” for that song. Simply listing a song’s writers without describing their specific contributions is common for many album credits and for many online databases including allmusic.com. We only consider artists who are listed as writers, and we don’t include those who were only involved in other ways, such as producing.

The exclusive use of the Hot 100 chart, specifically focusing on songs that made it into the top 50, will also not completely accurately model popular songs and songwriters during the time period that we consider. The Hot 100 only included songs specifically released as singles until 1998, when the current policy of considering all forms of song releases was implemented. Some popular songs never appeared on the chart despite having large sales and airplay on the radio due to the policies of selecting songs for the chart.

The problems we have discussed could cause our graph to include extra connections or to omit connections, but because we are focusing our analysis on the overall structure of the graph, the effects of adding or losing a small number of connections between writers should be negligible.

We use several types of software to conduct our analysis of the graph. Gephi (https://gephi.org) is a graph visualization program that also calculates various statistics about graphs. The NetworkX package in Python (https://networkx.github.io) contains useful functions for analyzing large and complex networks.

IV. Results and Extensions

Basic Statistics

In our data, there are 19,887 nodes and 59,352 edges. The graph is disconnected into 1,278 disjoint subgraphs, the largest containing 16,399 nodes and 56,268 edges, which we will call the giant subgraph. The next largest subgraph contains just 19 nodes, and despite the large number of subgraphs, this shows already that writers of popular songs are generally well-connected.

We also analyzed our main graph (the giant subgraph) only, and the main graph was selected instead of the whole big graph since it is the biggest among all these graphs and its study might exhibit interesting properties from structural viewpoint. A brief discussion about the type of the main graph is that it has 16,399 vertices, 56,268 edges, and the out-degree of each node has values that are generally close to each other. In conclusion, we can approximate it with a sparse graph and write $|E| = O(|V|)$[12]. In a way, this confirms the intuitive observation of the sparse structure for the main graph, considering the number of writers implied in the design of the songs for a long period in time. It was unlikely for the collaboration to be kept roughly at constant level between all of them.

Connectivity at network level can be studied based on a series of parameters called indexes[13, 14]; among these, we have selected α, β, and γ indexes. Main advantage of this approach is the fact that only basis data about the graph is necessary. However, since two of these indexes treat differently the planar and non-planar graphs, it would be useful to first study the planarity of the main graph.

At this point, we perform a simple test derived from Euler's criterion for planarity[15]. Namely, if a graph is planar, then the following inequality always holds:

$$m \leq 3n - 6 \text{ where } m = |E| \text{ and } n = |V|.$$

We are to prove that our main graph is non-planar. Assume that main graph is planar, and assume for reductio that the inequality holds. However, verifying whether $56{,}2683 \leq 3 \times 16{,}399$ -

6 leads to contradiction, since 56,268 > 49,191. This certifies the main graph is non-planar. It must also be said there are graphs that satisfy this inequality without being planar.

The simplest measure of connectivity for any graph is given by β index, which is calculated as the ratio between number of edges and number of vertices. For the current graph, its value is 56,268: 16,399 ~ 3.43. This confirms the claim that complex networks have a value of β greater than 1. Practically, higher value shows the number of network paths between writers is bigger (e.g. fixing a subset of writers, they have collaborated on more songs). This index can be seen as the average number of edges per vertex (e.g. how many songs, as average, were written by each pair of writers). At the same time, since $\beta > 1$, it means the network has more than one cycle (a single cycle, which corresponds to a closed circuit among the same writers, would require $\beta = 1$); not last, because the number of vertices is fixed (writers), more links (edges) would have showed stronger collaboration between them as overall. If $\beta < 1$, then it means the network is either simple enough or a tree (none of these situations occur here). Another analysis would be to study what value β has for the sequence of writers having highest out-degree sequence.

Because we have proved above that main graph is non-planar, computation of α and γ indexes takes this finding into account. This parameter is relevant in regards to overall connectivity because the number of existing cycles vs. the maximum possible number of cycles from the graph. The formula applied for α is given below:

$\alpha = (e - v + 1) / [\frac{v(v-1)}{2} - (v-1)]$. Applied for the main graph, this leads to a value of 39,870 / (16,398 × 16,399/2 – 16,398) = 39,870 / 134,439,003 = 0.0002965657220769. This value shows the network is far from being strongly connected (since the value is closer to 0 than to 1, which corresponds to a complete connected network. If instead we are interested in computing α's value for the whole graph (including the disjoint graphs), then the formula slightly modifies as $(e - v + p) / [\frac{v(v-1)}{2} - (v-1)]$, where p is the number of non-connected graphs. Here we have considered p=1 because the measurement of connectivity was made only for the main graph.

Finally, γ index offers a measure of the ratio between observed network links and maximum number of links (considering the network structure). This parameter is useful when it is computed periodically because it can show the evolution of the network in time (e.g. when writers begin new collaborations with other writers for the 1st time). For the non-planar graphs, it is computed using the formula from below:

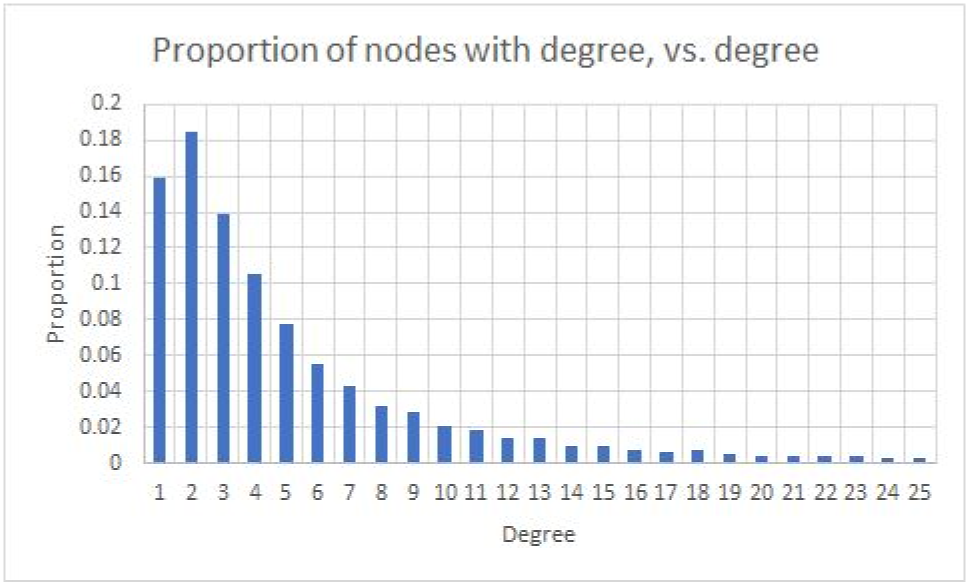
$\gamma = e/[\frac{v(v-1)}{2}]$. Applied for our graph, $\gamma = 56{,}268 /(16{,}398 \times 16{,}399 /2) = 56{,}268 / 134{,}455{,}401 = 0.0004184882093357$. This shows a proportion of the possible number of edges from the graph (e.g. on how many songs two writers have collaborated, in report with the number of songs they could have collaborated).

Less usual is the degree of connectivity, which is practically the reciprocal of the γ index. In the current scenario it is equal with 1: 0.0004184882093357 ~2,389.55.

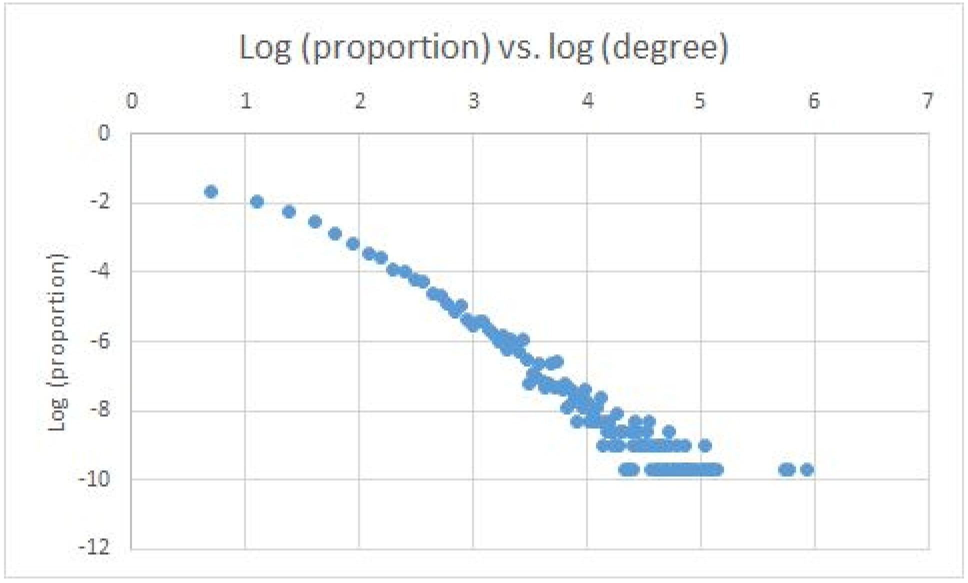
These indexes, even if are limited with respect to the amount of information they can reveal about the relationships from the graph, might represent a solid research base for more complex measurements of connectivity at node level. In addition, these values can be combined with classical results (e.g. computing MST where weight of each edge is given by the number of collaborations between two writers) for obtaining predictions of the future activity for some chosen writers.

Degree Distribution

Having created our musical collaboration graph, we want to investigate as many of its properties as possible in order to learn about the connections that exist within the songwriting community. The degree of a vertex in this graph indicates the number of other writers a person has collaborated with. The average degree of the graph is therefore the average number of different people each songwriter has collaborated with. This is one measure of how connected songwriters are to each other. We calculated the mean degree of the giant subgraph to be 6.862. Below, we have included a histogram of the degree distribution for this subgraph.



This distribution shows that the majority of nodes in the giant subgraph have a degree of two, and the proportion of nodes with larger degrees decreases steadily after this point. This distribution has a long tail, which can be seen most clearly by plotting the logarithm of the proportion of nodes for each degree against the logarithm of the degree number, omitting degrees which did not occur in the graph, and omitting the proportion with degree one in order to focus on the decay of the degree proportions.



This figure reveals an overall linear trend with some variance at large degrees. A linear trend in the log-log plot indicates that the degree distribution follows a power law:

$$p \propto k^{-\beta}$$

where p is the proportion of nodes with degree k. Networks with degree distributions that follow power laws are said to be scale-free,[11] indicating that the graph displays small-world characteristics. That is, the power law for the degree distribution gives a long tail where there are relatively large numbers of nodes with high degrees compared to graphs that are not scale-free. These high degree nodes provide a large number of connections within the graph bringing all the nodes closer together.

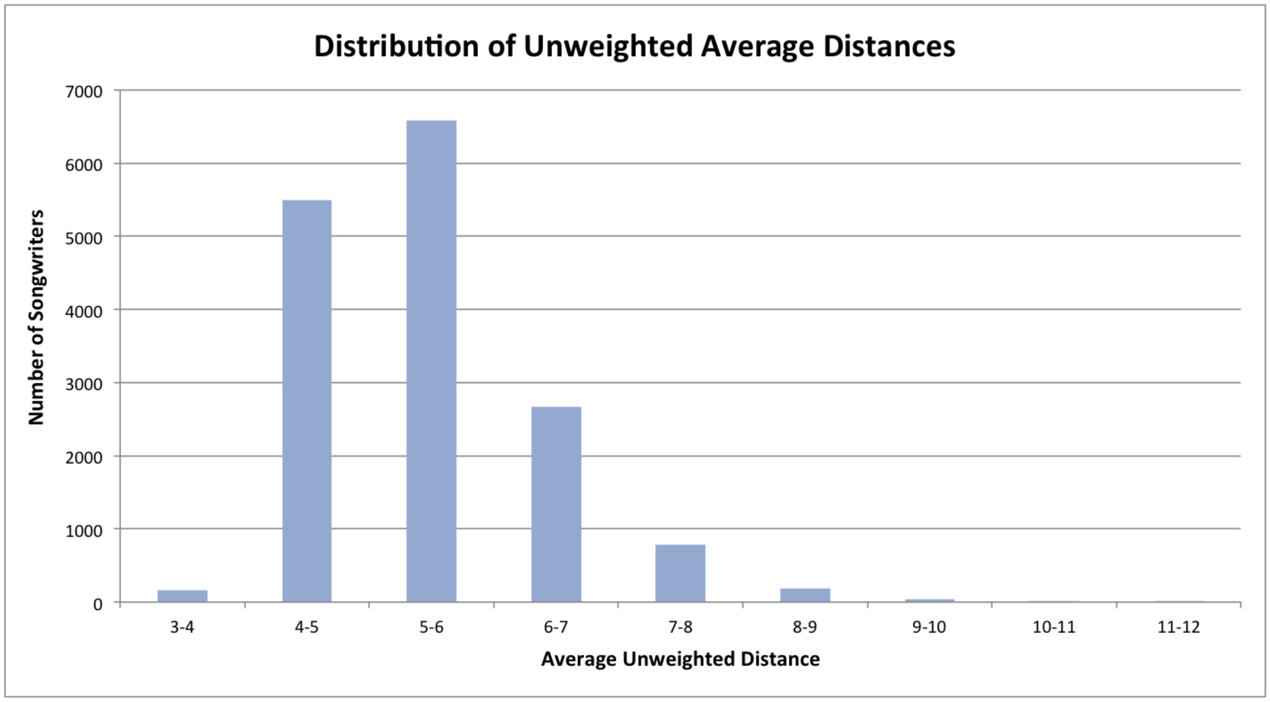
Fitting a linear regression to this plot gives $\beta = 2.079$ with $R^2 = 0.934$. This value is similar to values of β for other collaboration graphs, and is slightly smaller than that of the mathematical collaboration graph: $\beta = 2.81$.[3] A smaller β value tells us that that the proportion of vertices with each degree decays more slowly as degree increases. This indicates that levels of collaboration are higher among songwriters than mathematicians.

Measuring Centrality

When we first began discussing our project, we were interested in finding out who the most central figure in the music industry is. There are a number of metrics we could use to determine a person's centrality, however. Ultimately, we decided to define our centrality metric as the average distance from one person to everyone else in the main graph. The smaller this average distance, the more central the person. This raises another question, however. How do we determine the distance between two people? We will return to investigate this question in more detail later, but for now we will use a simple definition that has been used to study many other collaboration graphs, including the mathematical collaboration graph. If two people have written a song together the distance between them is one. If two people have not written a song together, but each of them has written a song with another (shared) person, then their distance is two. In other words, the distance between two people is defined to be the minimum number of edges that have to be traversed in order to get from one person to the other.

Using the same Java code that we used to construct our graph, (see appendix 1) we calculated the distance between every pair of people. This was necessary in order to calculate our centrality metric, and it allowed us to explore some interesting properties of the connections between songwriters. For instance, in order to get some idea about the size of the graph, we may want to calculate the distance between the two people who are farthest apart. When we do this, we find that it takes 18 steps to get from Amos Boyd to Ray Anthony & His Orchestra.

Before we reveal the most central figure in the music industry, we want to have an understanding of how much this average distance varies from person to person. Is this central figure one of only a few well-connected individuals, or do most people reside in the center of our graph? To get an idea about the distribution of average distances, we have included a histogram.



From this figure we can see that the most central figures have an average distance between three and four, while the majority of people have average distances between four and seven. Only those at the far edges of the graph have average distances over seven or eight. The person with the largest average distance is located an average of 11.85 steps from his fellow songwriters. How does this compare to other collaboration graphs? If we look at the central figure of the mathematical collaboration graph, Paul Erdős, we find that the average distance between Erdős and every other published mathematician is 4.65.[2] It makes sense that the central figures in our graph have a smaller average distance than Paul Erdős due to the relative sizes of the graphs. Whereas our musical collaboration graph contains approximately 20,000 vertices, the mathematical collaboration graph contains approximately twenty times that number of people, with just over 400,000 vertices.

At last we are in a position to answer our original question: who is the most central to our musical collaboration graph? Perhaps unsurprisingly, the answer is the rapper, songwriter, record producer, fashion designer, and entrepreneur: Kanye West. The top ten most central songwriters along with their average distances are displayed in the table below.

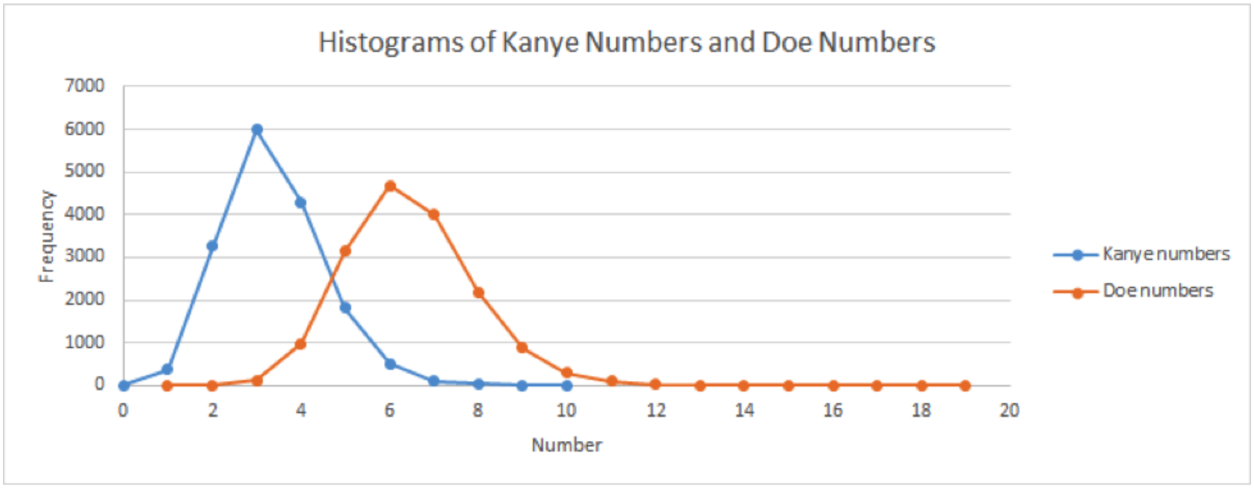
Rank	Writer's Name	Average Unweighted Distance
1	Kanye West	3.372
2	Sean Combs (P. Diddy)	3.483
3	Mariah Carey	3.590
4	Pharrell Williams	3.617
5	Christopher Wallace (Biggie Smalls)	3.630
6	Shawn Carter (Jay Z)	3.653
7	Mary J. Blige	3.657
8	Onika Maraj (Nicki Minaj)	3.672
9	Marshall Mathers (Eminem)	3.695
10	Lukasz Gottwald (Dr. Luke)	3.702

What we see from this data is that the most central part of the graph is dominated by modern pop, hip hop, rap and R&B music. This should not be surprising. These genres of music dominate the Billboard Hot 100 and frequently include collaborations between several artists. What one might find surprising is that modern songwriters seem to be more central to the graph than those who wrote songs at any other time since the data was first collected in 1959. One might have thought that those who have been in the music industry the longest would be most central to the graph. After all, it is reasonable to expect that they would have coauthored the greatest number of songs with the greatest number of people. It would also be reasonable to expect that songwriters who worked during the time directly between 1959 and 2017 (around 1990) would be the most central to the graph because they are positioned most centrally in time. However, this is not what we find. Of the top ten most centrally located figures, almost all are in the prime of their careers. One possible explanation for this phenomenon is that musicians tend to collaborate more often than they used to. It would explain why we see such a skew toward modern artists. Interestingly, a similar phenomenon has been documented in the case of the mathematical collaboration graph. In mathematics as well as music, it would seem the rate of collaboration has been increasing with time.[2]

Kanye Numbers

As we know, the concept of Erdos and Bacon numbers have provided interesting tools with which we can study the mathematical collaboration graph and the film collaboration graph. Since Kanye West is the central figure of our collaboration graph, we will define a songwriter's Kanye number to be the distance between them and Kanye West. Although Kanye West is at the center of our graph, we could choose to define similar numbers for other artists. This is where the concept of Doe numbers (as in "Jane Doe") comes in. Performing an analysis similar to that of Jerrold Grossman,[3] we will define a Doe number to be the distance from one writer to some other arbitrary writer (i.e. Jane Doe). We can compute the Doe number distribution for each writer in the graph, and compute the average of all such distributions. This average Doe distribution will give us a metric against which we can compare the distribution of Kanye numbers.

The mean Kanye number is 3.372 with a standard deviation of 1.147. The mean of our average Doe distribution is 5.455, with a standard deviation of 0.904. We find that the standard deviation for each of the individual Doe distributions is fairly small, so while the mean distance varies significantly from writer-to-writer, the average spread of those distances is consistent no matter who the writer is. The following figure compares the distribution of Kanye numbers as well as Doe numbers. As we can see, the average Kanye number is significantly lower than the average Doe number. This is a testament to Kanye West's centrality in our graph.



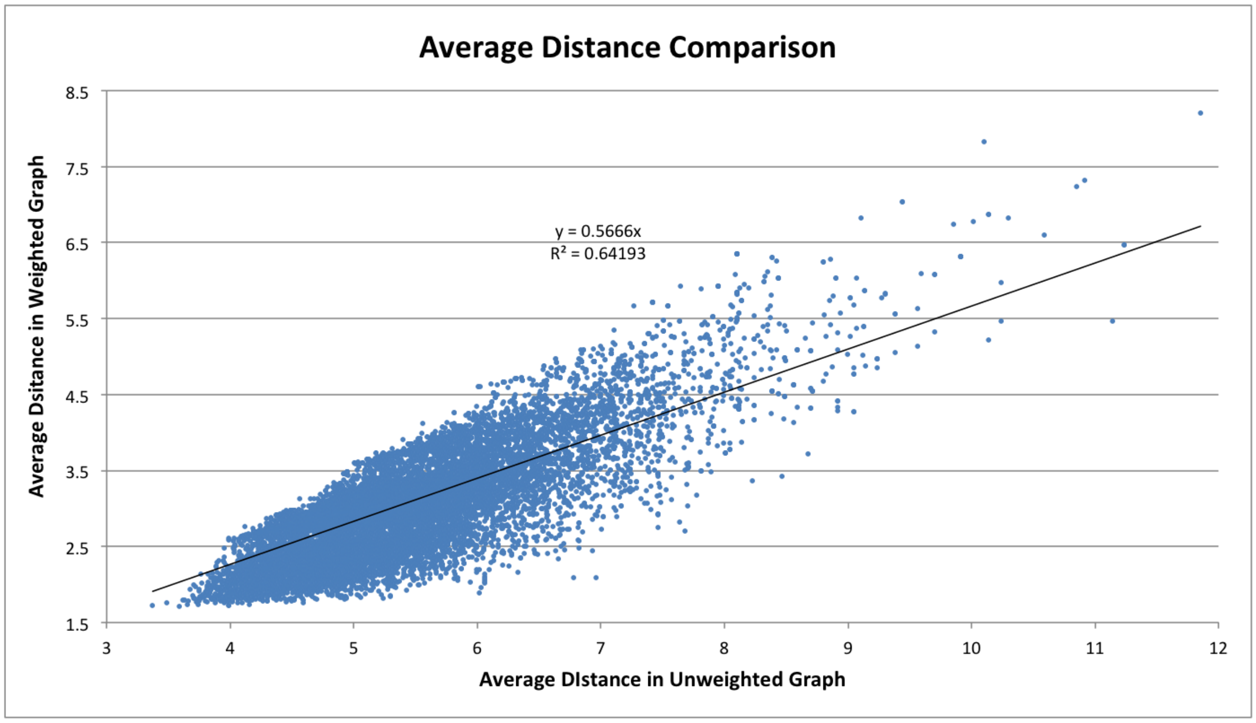
Redefining Distance

So far we have investigated the graph using a single measure of distance. Under the assumptions of our current model, two people are the same distance apart regardless of whether they have written one song together or one hundred songs together. This does not have to be the case, however. What if we want a metric that diminishes the distance between people the more often they collaborate? One way of doing this would be to assign a distance of 1/*n* between two people who have collaborated *n* times. Clearly, this will diminish the average distance from each person to every other person, because edge lengths can decrease or stay the same, but they cannot increase. This raises the question, how does this change in our definition of distance affect the rankings of the most central figures? Unlike with our previous distance metric, with our new model people will become more central the more times they have collaborated, even if they are collaborating multiple times with the same person. If we found the same people at the center of the graph using both methods of calculating distance, it would reinforce the idea that they really are central to the music industry. If we found entirely different people, it would call into question just how significant our measure of centrality really is. When we recalculate the top ten most central people in our graph, we get the following list.

Rank	Writer’s Name	Average Weighted Distance
1	Mariah Carey	1.712
2 (tie)	Walter Afanasieff (Baby Love)	1.723
2 (tie)	Kanye West	1.723
4	Christopher "Tricky" Stewart	1.726
5	Jermaine Dupri	1.738
6	The-Dream	1.739
7	Shawn Carter (Jay Z)	1.742
8	Sean Combs (P. Diddy)	1.756
9	Diane Warren	1.757
10	Desmond Child	1.759

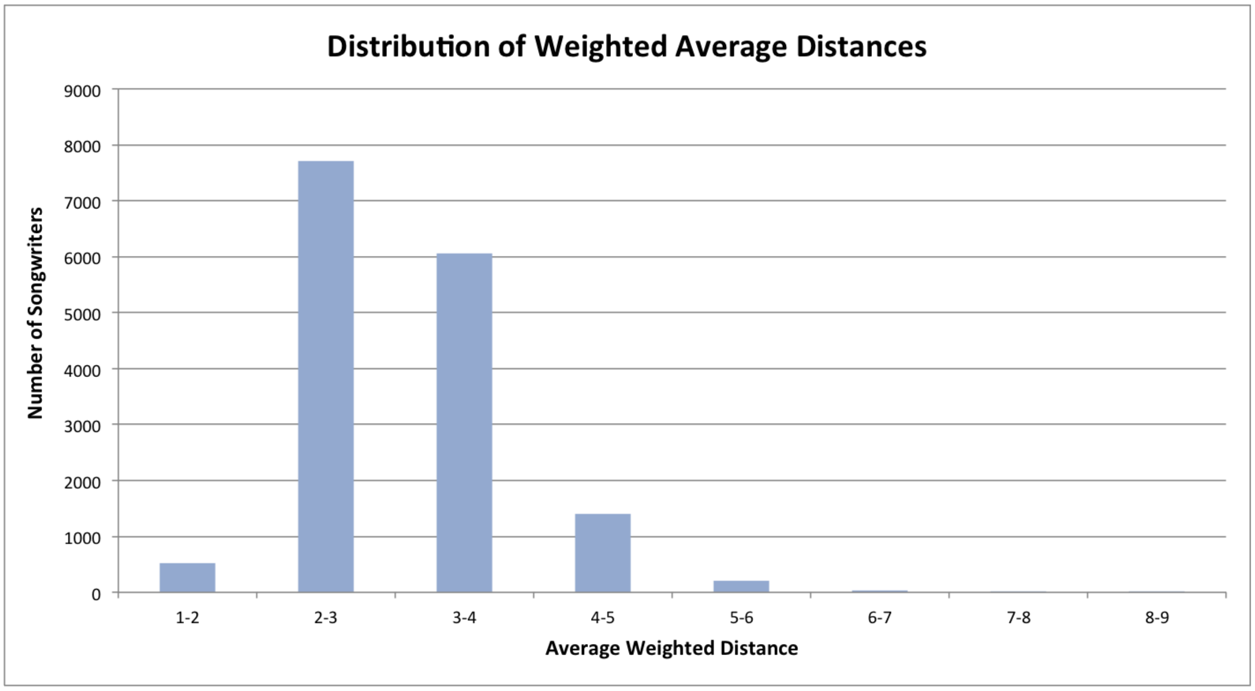
Although we see four of the same people in both top ten lists, six people are in the top ten most central figures of the weighted graph, but not the top ten for the unweighted graph. It is interesting to note that the center of the weighted graph does not seem to be completely dominated by pop, hip hop, and rap music as it was for the unweighted graph. The new additions to the list such as Diane Warren, Desmond Child, and Walter Afanasieff have all written a diverse collection of songs from a variety of genres including pop, rock, R&B, and soul. The new additions to the top ten list also seem to have been in the music industry for longer than the others. For instance Diane Warren has been active since 1983, Walter Afanasieff has been writing music since 1980, and Desmond Child has been active since 1975. This suggests that when the total number of collaborations is taken into account (even if many are between the same people), those who have been in the music industry the longest become centralized.

Although it is interesting to look at individuals, instead of tracking the movement of just the top ten most central people, we want to see how the average distances change for everyone in the graph. The following plot shows each individual's average distance using our unweighted graph on the x-axis compared to the same individual's average distance using the weighted graph on the y-axis.



Although we can see that the average distance from one person to all other people is smaller using the weighted graph, the people that are most centrally located according to one metric still tend to be centrally located according to the other. Similarly, the people that are farthest from the center of our unweighted graph tend to be the farthest from the center of our weighted graph. If we calculate a line of best fit through this data, setting the y-intercept to zero, we find that the slope is approximately 0.57. The R-squared of the fit is 0.64.

Again, we can plot a histogram of the average distances for each person using our weighted graph.



When we do this, we find that the shape of the histogram is approximately the same as it was for the unweighted graph, however the average distances are smaller, and so is the range of distances. Using this new metric, we find that the most central individuals have average distances just above 1.7, while the majority of songwriters have average distances between two and four. Incidentally, the least central figure is the same in both the weighted and unweighted graphs. Using the unweighted graph, the least central figure had an average distance of 11.852, while using the new weighted graph, he has an average distance of 8.206.

Undoubtedly, there are many other measures of distance we could explore. One we were particularly interested in investigating uses a weighting system exactly like the one we have just described in which edges have a weight of $1/n$ where n is the number of collaborations between the two people in question. The difference is that instead of only considering the shortest path between people, this method takes into account all possible paths. The more different ways there are to get from one person to another, the closer the people are. This model, proposed by Michael Barr, treats the collaboration graph exactly like an electrical circuit, where the edges represent resistors and the total distance between two people is the resistance of the circuit that separates them.[4] Unfortunately, due to limited computing power and time constraints we were unable to analyze our graph using this distance metric.

Small World Phenomena

A common analysis of real-world graphs is the degree to which a certain graph exhibits small-world phenomena. This is a measure of how connected the graph is. A small-world graph will have a small average path length across all pairs of vertices, sometimes called the characteristic path length of the graph, meaning that it will be possible to travel between any two vertices in a relatively small number of steps compared to the size of the graph. Small-world graphs also have a high degree of clustering and transistivity, meaning that on average a vertex and its neighbors will be well-connected to each other.[8]

Measuring the amount of connectivity across vertices in the graph can be done by defining a local clustering coefficient for each vertex, which measures how close a node and its neighbors are to forming a clique, that is, a subgraph in which a node and its neighbors form a complete graph. A global measure of the graph's connectivity can be defined as the average clustering coefficient of its vertices.

A graph with *n* nodes and *m* edges may be tested for the degree to which it exhibits small-world characteristics by comparing the average clustering coefficient and the characteristic path length to those of a similar random graph with about the same average degree. These "null" models of the graph can be constructed as an Erdős-Rényi (E-R) graph with the same *n* and *m*, where each edge from the original graph is randomly and uniformly reassigned to a node pair.[9] A graph may be considered small-world if, compared to equivalent random graphs, its characteristic path length is similar while its average clustering coefficient is much larger. The "small world" coefficient σ can be defined to be the ratio of the clustering coefficient to the characteristic path length, each normalized by the corresponding measure in the equivalent random graph. That is:

$$\sigma = \frac{C_g/C_r}{L_g/L_r}$$

Where C_g is the clustering coefficient of the graph under consideration, C_r is the average of the clustering coefficient across many equivalent random graphs, L_g is the characteristic path length of the graph under consideration, and L_r is the average path length across many equivalent random graphs.

Watts and Stogatz collected the same data about small-world phenomena in several different real-life networks, including the classic graph of connections between film actors, in addition to graphs representing power grid components and transmission lines, as well as neurons and synapses in *C. elegans*.[10] We compute C_r and L_r as the average of these measures across 100 random graphs. Below, we tabulate the small-world coefficient σ for this data as well as for our songwriter data.

	L_g	L_r	C_g	C_r	σ
Film actors	3.65	2.99	0.79	0.00027	2396.854
Power grid	18.7	12.4	0.080	0.005	10.610
C. elegans	2.65	2.25	0.28	0.05	4.755
Songwriters	5.455	5.249	0.554	0.000414	1288.060

Our graph of songwriters does not display small-world phenomena to the same degree as the film actors graph, but it displays more small-world phenomena than the other networks. One reason this could be the case is that songs are typically written by only a few people, rarely more than 5, while most movies include dozens of actors. This will necessarily create many more connections among those in the film industry.

Graph Clustering: Markov Cluster Algorithm

Another interesting concept that has been explored in collaboration graphs is the idea of clustering. The analysis of graph clusters is an important tool used to understand the behavior of graphs, especially those that are too large to be easily visualized. If given an arbitrary graph, investigating its clusters can be a useful method of gaining an intuition about the connections within it.[7]

To explore clustering within our graph we used an algorithm called Markov Cluster Algorithm (MCL). The algorithm was first discovered/invented by Stijn van Dongen at the Centre for Mathematics and Computer Science. The theory was proposed in his PhD thesis during his time at the University of Utrecht. MCL is a process consisting of multiple iterations of matrix expansion and inflation until it reaches convergence.[7]

The following are the definitions and purposes of expansion and inflation in MCL:

Expansion is simply the multiplication of a matrix. Since the matrix is a Markov chain transition matrix, the expansion step is responsible for providing the probability of transitioning from one vertex to another. The multiplication of the transition matrix can also provide information about connections within a graph, i.e. whether it is possible to go from one vertex to another.

Inflation is characterized by a parameter, r, which is used to strengthen or weaken the matrix. To strengthen a vertex means to increase the probability of transition between it and its neighbors. To weaken a vertex means to decrease the probability of transition between it and its neighbors. Inflation tends to strengthen the vertices that are well-connected and weaken the vertices that are not. Inflation can be done in a simple matrix operation. The process of inflation takes in an inflation parameter, r, which determines the amount of strengthening or weakening. The following is a mathematical definition of inflation:

Given a matrix $M \in \mathbb{R}^{k*l}$, $M > 0$, and a real non-negative number *r*, the matrix resulting from rescaling each of the columns of M with power coefficient *r* is called $\Gamma_r M$. Γ_r is the inflation operator with power coefficient *r*. The equation is defined by

$$(\Gamma_r M)_{pq} = (M_{pq}) / \sum_{i=1}^{k} (M_{iq})^r$$

[7]

From this definition, we can see that the entire operation is just renormalizing each column of the matrix after taking the r^{th} power of M.

The MCL algorithm requires the matrix to go through multiple iterations of expansion and inflation until it converges. One limitation is that MCL works very well with small diameter graphs, where diameter is defined to be the average hop from one random vertex to another random vertex. The reason is that MCL calculates random walks, and if the diameter is too large the matrix will converge very slowly. However, Stijn van Dongen provides no definition of a “large” diameter, and therefore we do not know whether our graph is feasible for the algorithm.

To find clusters in the graph, we need to first find the attractor of each cluster. After the MCL algorithm has been completed, if the final matrix has nonzero entries in a row, the vertex that corresponds to that row is an attractor. The columns that have nonzero entries in this row correspond to the attracted vertices. It is possible that clusters have multiple attractors. For example, if we consider the following matrix:

$$\begin{matrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \end{matrix}$$

- The first row contains a nonzero entry so vertex one is an attractor. It attracts vertex two because (1, 2) is nonzero. So it forms a cluster {1, 2}.
- The second row does not have any nonzero entries so it is not an attractor.
- The third row has nonzero entries, which form a cluster {3, 4}.
- The fourth row has nonzero entries, which form a cluster {3, 4}.

Notice that rows 3 and 4 belong to the same cluster, indicating that both vertex three and vertex four are attractors of the cluster {3, 4}. The above matrix demonstrates that the associated graph contains 2 clusters: {1, 2} and {3, 4}.

Graph Clustering: Simulation

Performing the MCL algorithm on a transition matrix is straightforward. The following are the steps required to compute the resulting matrix using MCL:

1. Expand the matrix by taking the k^{th} power of the Matrix.
2. Inflate the matrix computed from step 1 by using the inflation equation with parameter r.
3. Repeat step 1 and 2 until the matrix converges.
4. Analyze the resulting matrix and find clusters.

We have written both Java and Python code to perform the algorithm. We first want to compute a Transition matrix to input into the algorithm. The MCL algorithm is implemented in Python because Python is better at handling matrix operations, but the graph is built in Java. So we perform the following steps to simulate and analyze the clusters:

Step 1:

We wrote a Java function called **formatMarkov** that outputs two text files. One file contains the writer information with its corresponding index and one file contains a matrix file with 1s and 0s as its entries. For example, Steve Burgh and Robert Lee McCoy are connected and are assigned the indices 4 and 7 respectively. Therefore, the entries at (4,7) and (7, 4) are equal to 1. Note that the matrix we want to generate represents a markov chain, so we are excluding songwriters that do not share any connections to other writers. The implementation of the function is provided in appendix 2.

Using our formatMarkov function with a WriterGraph object as a parameter, (see appendix 1) we output two text files: *allWriters2.txt* and *matrix2.txt*. The *allWriters2.txt* file contains each writer along with their associated index. The code took about 30 minutes to run on a Mac Retina. We have include a shortened version of the file below:

```
0, Steve Burgh
1, Robert Lee McCoy
2, Earl Nelson
3, Vixen
4, Charles Watkins
5, Ashton Thomas
        ……
(19881 other similar lines)
```

From the file *allWriters2.txt,* we observe that there are a total of 19887 writers, each of whom has at least one connection within the graph. As mentioned before, *matrix2.txt* contains a 19887 by 19887 matrix with all 0s and 1s as its entries. We also include the shortened version of the file below:

```
0,0,0,0,0,0,0,0, ... (19979 other similar entries)
0,0,0,0,0,0,0,1, ... (19979 other similar entries)
0,0,0,0,0,0,1,0, ... (19979 other similar entries)
0,0,0,0,1,1,0,0, ... (19979 other similar entries)
        ……
(19983 other similar lines)
```

We have two separate files because, to perform the MCL algorithm, we are switching between two languages. So when evaluating the matrix we need to have a record of which writer corresponds to each index in the matrix.

Step 2:

After creating the two files we wrote Python code that reads the file *matrix2.txt* and runs the MCL algorithm using the matrix. For the inflation step we are using 2 as the inflation rate. The Python code outputs a text file named *rowToCol.txt* which contains all the nonzero entries of the final matrix after running it through MCL algorithm. Each line of output contains 3 entries: the row index of the matrix, the column index of the matrix, and the value of the matrix in that specific row and column. For example, one line in the text file contains the entries 3, 5, 0.5 meaning that the final matrix contains 0.5 in its 3rd row and 5th column.

The Python code is provided in appendix 3. The Python code took about 4 hours and 30 minute to run on a Mac Retina. The code also printed out the difference in norm, comparing the matrix before and after each iteration of MCL. By observing this difference we are able to tell whether or not the matrix is converging. If the difference decreases, the algorithm is converging to some value. From the output we can see that the matrix eventually converges:

```
113.587097168
40.6211357117
42.9465026855
27.552778244
      ……
0.00965067092329
3.14915887429e-05
3.3733488225e-10
3.92523207215e-17
```

The entire MCL algorithm took 66 iterations of expansion and inflation until convergence. The Python code computed a *rowToCol.txt* file, which has been shortened and provided below.

```
row col val
2 2 1
4 4 1
4 3074 1
4 3413 1
10 10 1
11 11 1
  ……
(21680 other similar lines)
```

Step 3:

After running the clustering algorithm and obtaining the finalized matrix, we again use Java code to analyze the matrix. In this step we first want to see how many clusters there are in the entire graph. So as mentioned earlier, we need to first find the attractor of the clusters. Then we find the associated clusters. We have written Java code that reads in the files *rowToCol.txt* and *allWriters2.txt* and computes the resulting clusters. The Java code is provided in appendix 4.

The code took about 3 minutes to run on a Mac Retina. We find that there are 5210 clusters with 5758 attractors in total. The top 10 biggest clusters are shown in the following table.

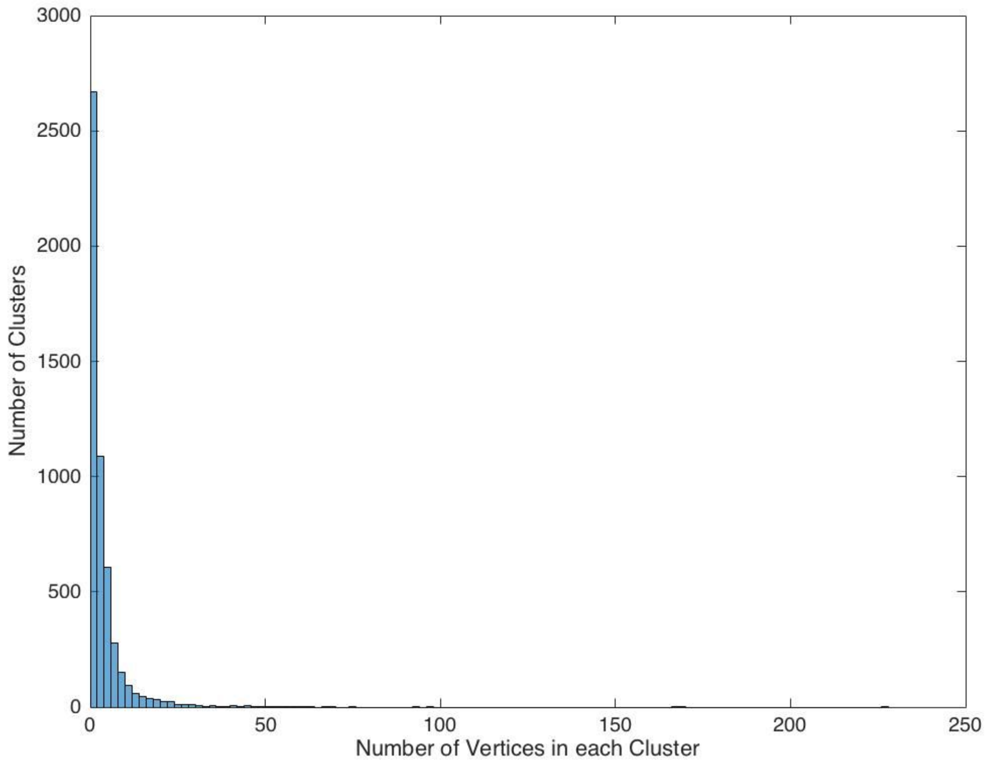
Rank	**Attractor**	**Cluster Size**
1	Kanye West	227
2	Sean Combs	169
3	Chris Brown	166
4	Traditional	97
5	Tramar Dillard	92
6	James Brown	75
7	Aubrey Graham	68
8	David Guetta	66
9 (tie)	Justin Bieber Jermaine Dupri	63

We researched the three largest clusters to see if we could find a connection (such as a common genre) for the writers in each. We found the following results:

- **Kanye West:** most writers in the cluster write Hip-hop, Rock and R&B music.
- **Sean Combs:** most writers in the cluster write Hip-hop and R&B music.
- **Chris Brown:** most writers in the cluster write Hip-hop, Rock, Soul and Pop music. There are also some songwriters that write Jazz, Blues and Reggae music. Since Chris

Brown is known for writing music in a variety of genres it is not a surprise that writers from a number of different genres are in this cluster.

The following is a histogram computed in Matlab. In analyzing our data we found that more than 50% of the clusters contain only 1 writer. This means that there are many writers that are neither attracted by other writers nor attracting to other writers.



Random Walks

In this section, we use Markov chains to find the people that a random walk through the graph are most likely to reach. We consider two different cases: one in which each edge is represented with either a 0 or 1 depending on whether or not the writers are connected, and one in which edges are weighted depending on the number of songs people have written together.

Since we are not interested in random walks that will end up in a subgraph that is disjoint from the main graph, we excluded all disjoint graphs from the Markov chain. After this process, we created a 16399 by 16399 transition matrix containing the data from our main graph using a Java program (Appendix 1). Then, we stored the matrix in a txt file, imported it into Python, and normalized the rows of our matrix (Appendix 2, 3). The process up to here took about 2 hours to run on Lenovo Yoga 910. Because we want to find the probabilities of random walks reaching various nodes in our graph, we are going to simulate a long random walk by raising the matrix to a large power.

To speed up the multiplication of this large matrix, we exported it from Python to Matlab using the following command.

```
scipy.io.savemat('matrix1.mat', mdict={'matrix':normalizedMat})
```

Using Matlab, we calculated this matrix raised to a variety of large powers (100, 200, 300, and 600) (Appendix 5). Running the Matlab code for matrix multiplication with power raised to the 100th took an hour and a half per matrix. Then, raising the power from the 100th to 200th took only 30 minutes. To get an idea of the probability of reaching various artists at the end of our random walk, regardless of the starting point, we added up the values in each column. Using this data we created a 16399 x 2 matrix which consists of the index given to each writer and the sum of the probabilities of each column of our big matrix.

First, we considered our edge weight to be one if a pair of writers ever wrote a song in the top 50 together, and zero otherwise. We started by calculating 100-step transition probabilities. Raising our matrix to the 100th power we get the following results.

Rank	Writer's Name	Sum of Prob. (100th)
1	Kanye West	54.1991
2	Chris Brown	46.1670
3	Sean Combs	44.5573
4	Dwayne Carter	24.5844
5	Mike Dean	23.8837
6	Onika Maraj	23.1429
7	Aubrey Graham	22.8542

8	Christopher Wallace	22.1939
9	Lukasz Gottwald	22.1353
10	Elliott Hayes	21.9908

When we raised the transition matrix to the 200th power, there was no difference in the writers' rankings as compared to the previous calculation, but the sum of the probabilities for every writer slightly increased.

Rank	Writer's Name	Sum of Prob. (200th)
1	Kanye West	54.7320
2	Chris Brown	46.7181
3	Sean Combs	44.9808
4	Dwayne Carter	24.8869
5	Mike Dean	24.1604
6	Onika Maraj	23.4311
7	Aubrey Graham	23.1398
8	Christopher Wallace	22.4164
9	Lukasz Gottwald	22.4119
10	Elliott Hayes	22.2662

Increasing the exponent further, we calculated the probabilities of the transition matrix when raised to the 300th and 600th powers. We noticed that changing the exponent from 200 to 300 changed the tenth ranked writer from Elliott Hayes to Tramar Dillard. The rankings stayed the same when we raised the matrix to the 600th power. The sum of the probabilities for each writer went up again, but the difference in the sum of the probabilities was greater when transitioning from 200 to 300 than from 100 to 200.

Rank	Writer's Name	Sum of Prob. (300th)	Sum of Prob. (600th)
1	Kanye West	54.7806	54.7910
2	Chris Brown	46.7662	46.7763
3	Sean Combs	45.0193	45.0277
4	Dwayne Carter	24.9129	24.9182
5	Mike Dean	24.1846	24.1896
6	Onika Maraj	23.4559	23.4610
7	Aubrey Graham	23.1645	23.1696
8	Christopher Wallace	22.4366	22.4410
9	Lukasz Gottwald	22.4359	22.4409
10	Tramar Dillard	22.2902	22.2952

Next, we weight the edges of the graph with the number of collaborations between each pair of songwriters. This means that we are more likely to transition from one node to another if the artists have written multiple songs together. The result shows a pretty big difference from the unweighted Markov chain. When we raised our transition matrix to the 100th power, none of the writers but Sean Combs (P. Diddy) appeared in the top 10 of both our weighted and unweighted rankings.

Rank	Writer's Name	Sum of Prob. (100th)
1	Eddie Holland	65.4995
2	Brian Holland	64.3993
3	Lamont Dozier	59.9170
4	Paul McCartney	57.0808
5	John Lennon	49.8077
6	Sean Combs	46.9177

7	Marshall Mathers	39.9270
8	Jerry Leiber	38.5446
9	Benny Andersson	38.5321
10	Björn Ulvaeus	38.1174

When we raised the matrix to the 200th power, the order of the writers in ranks 7 to 10 changed, but no writers dropped out of the top ten entirely. The writers with the top 10 largest probabilities stayed unchanged as the matrix power increased from 200 to 300 and then to 600.

Rank	Writer's Name	Sum of Prob. (200th)	Sum of Prob. (300th)	Sum of Prob. (600th)
1	Eddie Holland	64.8272	64.3842	63.9688
2	Brian Holland	63.7356	63.2962	62.8856
3	Lamont Dozier	59.3016	58.8929	58.5112
4	Paul McCartney	57.8507	57.6619	57.3428
5	John Lennon	50.4815	50.3175	50.0394
6	Sean Combs	46.7520	46.6238	46.4198
7	Benny Andersson	41.1841	41.2044	40.9376
8	Björn Ulvaeus	40.7400	40.7600	40.4961
9	Marshall Mathers	39.7370	39.5989	39.4036
10	Jerry Leiber	38.0732	37.8493	37.6335

Using a random walk to calculate the probability of passing through a node provides an interesting measure of the centrality of the vertex. Using the unweighted matrix, we still see current rap and hip hop artists dominating the top ten list. When we weight the graph we find a more diverse group of artists, however. We saw a similar effect when we added weighting to the average distance calculations we performed earlier. Again, this suggests that when the total number of collaborations is taken into account (even if many are between the same people), those who have been in the music industry the longest become centralized.

V. Conclusion

In our attempt to understand the connections that exist between songwriters in the popular music industry, we have employed a handful of tools used by mathematicians to model real-world social networks. We have used a variety of different metrics to determine the size and connectivity of our collaboration graph; we have explored several different methods of finding the most central figures; and we have analyzed the phenomenon of clustering within the songwriting community. Although we have gone to great lengths to measure our graph using a diverse array of tools, we cannot help but feel that we have only scratched the surface. There are so many interesting questions that can be asked of collaboration graphs, and there are so many different tools that can be used to answer them. One of the most important lessons we learned is that there is always more than one way to address a problem. Although we first planned to calculate the average distances between songwriters to determine their centrality, we realized that there could be different definitions of distance. We even found that we could generate similar data using a significantly different method: namely, using a Markov chain. One of our most interesting findings was that depending on the tools used to investigate centrality, we found that the most central figures had slightly different characteristics. For unweighted graphs that do not take into account the number of times two people have collaborated, we find that current hip hop and rap artists dominate the center. However, for weighted graphs we find that the center of the graph is more diverse, including artists from a variety of different times and genres. If we had not tried to find the most central figures using a variety of different techniques, we would not have noticed this. This is an excellent demonstration that mathematical modeling relies on exploration and human judgment. In contrast to the stereotypical perception of math in which there is a single correct answer, we have found with this project the opposite is true. There are so many answers to find. We just have to take the time to look for them.

Appendix 1

```java
import com.google.gson.JsonArray;
import com.google.gson.JsonDeserializationContext;
import com.google.gson.JsonDeserializer;
import com.google.gson.JsonElement;
import com.google.gson.JsonObject;
import com.google.gson.JsonParseException;
import com.google.gson.JsonParser;
import com.google.gson.GsonBuilder;
import com.google.gson.Gson;

import java.io.*;
import java.util.*;
import java.lang.reflect.*;

/*
    Deserializes an album object from JSON, in particular from the dataset
    it will deserialize each album from the "albums" property of the root object.

    root object:
    {
        "albums": [ album list ],
        "songs": [ song list ]
    }
*/
class AlbumDeserializer implements JsonDeserializer<Album> {
    public Album deserialize(final JsonElement json, final Type typeOfT,
            final JsonDeserializationContext context) throws JsonParseException {

        JsonObject jsonObject = json.getAsJsonObject();
        JsonObject albumObj = jsonObject.get("album").getAsJsonObject();

        Album album = new Album();
        Person artist = context.deserialize(albumObj.get("artist"), Person.class);
        album.artist = artist;
        Song[] tracks = context.deserialize(jsonObject.get("tracks"), Song[].class);

        // assign to each song the album in which it appears
        for (Song s : tracks) {
            s.album = album;
        }
        album.songs = tracks;

        String title = albumObj.get("title").getAsString();
        String year = albumObj.get("year").getAsString();
        String url = albumObj.get("url").getAsString();

        album.title = title;
        album.year = year;
        album.url = url;

        return album;
    }
}
/*
    Deserializes a song object from JSON. Songs appear as a list of tracks
    inside each album object.
    {
        "albums": [
            {
                "artist": {}
                "tracks": [ list of songs ]
            }
        ],
        "songs": []
    }
*/
class SongDeserializer implements JsonDeserializer<Song> {
    public Song deserialize(final JsonElement json, final Type typeOfT,
            final JsonDeserializationContext context) throws JsonParseException {

        JsonObject jsonObj = json.getAsJsonObject();
        JsonObject titleObj = jsonObj.get("title").getAsJsonObject();

        Song song = new Song();
        song.title = titleObj.get("name").getAsString();
        song.url = titleObj.get("url").getAsString();

        Person[] writers = context.deserialize(jsonObj.get("writers"), Person[].class);
        song.writers = writers;
        Person[] performers = context.deserialize(jsonObj.get("performers"), Person[].class);
        song.performers = performers;

        return song;
    }
}

/*
    Deserializes a person from JSON -- people appear in several places as either
    artists, performers, or writers, but each will at least have a "name"
    and a "url" property.
*/
class PersonDeserializer implements JsonDeserializer<Person> {
    public Person deserialize(final JsonElement json, final Type typeOfT,
            final JsonDeserializationContext context) throws JsonParseException {

        final JsonObject jsonObj = json.getAsJsonObject();
        final Person person = new Person();
        person.name = jsonObj.get("name").getAsString();
        person.url = jsonObj.get("url").getAsString();
        return person;
    }
}

/*
    Represents a person (artist, writer, or performer). Each has a name,
    as well as a url for that person's page on allmusic.com. The URL can
    sort of be used to test for uniqueness for people, although unfortunately
    unique people may have multiple URLs on allmusic.com, usually one page
    for them as a performer and one for them as a writer.
*/
class Person {
    public String name;
    public String url;

    public Person() {}

    public Person(String name, String url) {
        this.name = name;
        this.url = url;
    }

    public String toString() {
        return name;
    }

    @Override
```

```java
    public boolean equals(Object o) {
        if (!(o instanceof Person)) {
            return false;
        }
        Person other = (Person)o;
        return this.name.equalsIgnoreCase(other.name) && this.url.equals(other.url);
    }

    @Override
    public int hashCode() {
        return name.hashCode() + url.hashCode();
    }
}

/*
    Represents an album as albums appear in the JSON data. Each has the
    primary artist for the album, title, allmusic.com URL, and a list of
    all the songs appearing on the album
*/
class Album {
    public Person artist;
    public String title;
    public String url;
    public String year;
    public Song[] songs;

    @Override
    public String toString() {
        return title + " by " + artist.name;
    }
}

/*
    Represents a song as they appear as a list under each album. Each has
    the album it comes from, and a the performers and writers for the song.
*/
class Song {
    public Album album;
    public String title;
    public String url;
    public Person[] performers;
    public Person[] writers;

    @Override
    public String toString() {
        return title + " by " + Arrays.toString(performers);
    }
}

/*
    An edge in the writer graph. An edge has the two people it connects as
    endpoints, and the data in the edge will be a list of all the songs in
    which both people appear as writers.

    The graph will be undirected, but there will need to be two edge objects
    between a pair of people in the graph. However, they can each share the
    same list of songs.
*/
class Edge {
    public Person start;
    public Person end;
    public List<Song> collaborations;

    public Edge(Person start, Person end) {
        this.start = start;
        this.end = end;
        this.collaborations = new ArrayList<Song>();
    }

    // Adds a collaboration between these two people
    public void addCollab(Song song) {
        this.collaborations.add(song);
    }
}

/*
    Represents our writer graph! Map is represented by an adjacency list,
    basically a list of edges for each person in the graph, where each edge
    will have the people that a certain person has written songs with, and
    a list of those songs.
*/
public class WriterGraph {

    public Map<Person, List<Edge>> vertices;

    /*
        Creates a graph from the given JSON file name, and the given Gson
        object which will deserialize the JSON from the file.
    */
    public WriterGraph(Gson gson, String dataFile) throws FileNotFoundException {
        vertices = new HashMap<Person, List<Edge>>();

        Reader inReader = new BufferedReader(new FileReader(dataFile));
        JsonElement json = new JsonParser().parse(inReader);
        JsonObject root = json.getAsJsonObject();
        Album[] albums = gson.fromJson(root.get("albums"), Album[].class);
        for (Album a : albums) {
            for (Song song : a.songs) {
                Person[] writers = song.writers;
                for (int i = 0; i < song.writers.length; i++) {
                    for (int j = i + 1; j < song.writers.length; j++) {
                        Person writer1 = writers[i];
                        Person writer2 = writers[j];
                        if (!hasVertex(writer1)) {
                            addVertex(writer1);
                        }
                        if (!hasVertex(writer2)) {
                            addVertex(writer2);
                        }
                        addCollab(writer1, writer2, song);
                    }
                }
            }
        }
    }

    // Adds a person to the graph
    public void addVertex(Person p) {
        vertices.put(p, new ArrayList<Edge>());
    }

    // Returns true if the person is in the graph, and false otherwise
    public boolean hasVertex(Person p) {
        return vertices.containsKey(p);
    }

    // Adds a song that the two given people have both written. The graph is
    // undirected, so both directions are added to the graph.
    public void addCollab(Person p1, Person p2, Song collab) {
        Edge edge = getEdgeBetween(p1, p2);
        if (edge == null) {
            // If there's no existing connection between two people, create
            // two edge objects to go between the people, and add the given
            // song to the new edges
            Edge edge1 = new Edge(p1, p2);
            Edge edge2 = new Edge(p2, p1);
            edge1.collaborations = edge2.collaborations; // use the same list for both
            edge1.addCollab(collab);
            vertices.get(p1).add(edge1);
            vertices.get(p2).add(edge2);
        } else {
            // There is already a connection, just add the song to that edge
            edge.addCollab(collab);
        }
    }

    // Gets a list of all the collaborations that a given person has.
    public List<Edge> getCollabs(Person p) {
        return vertices.get(p);
    }

    // Gets a list of all the collaborations that a given person has with
    // another person.
    public List<Edge> getCollabs(Person p1, Person p2) {
        List<Edge> collabs = new ArrayList<Edge>();
        for (Edge edge : vertices.get(p1)) {
            if (edge.end.equals(p2)) {
                collabs.add(edge);
            }
        }
        return collabs;
    }

    // Gets the minimum distances (in edge "hops") from this person to all the other
    // people in the graph.
    //
    // Returns a map, such that the keys are people in the graph, and the values
    // are the distances from the given person to the corresponding key
    public Map<Person, Integer> getDistances(Person p) {
        Map<Person, Integer> distances = new HashMap<Person, Integer>();
        Queue<Person> curr = new LinkedList<Person>();
        curr.add(p);
        distances.put(p, 0); // people have distance of zero from themselves
        int currDist = 1;

        // breadth first search of the graph will give minimum distances
        while(!curr.isEmpty()) {
            int currQueueSize = curr.size();
            for (int i = 0; i < currQueueSize; i++) {
                Person currPerson = curr.remove();
                for (Edge e : getCollabs(currPerson)) {
                    Person end = e.end;
                    if (!distances.containsKey(end)) {
                        distances.put(end, currDist);
                        curr.add(end);
                    }
                }
            }
            currDist++;
        }
        return distances;
    }

    // Gets the shortest path between two people in the graph.
    //
    // Returns a LinkedHashMap, such that the keys are intermediate people on
    // the path between the two people, and the values are the edges that lead
    // "to" the intermediate people, if the path is viewed as going from the
    // first person to the second person.
    //
    // LinkedHashMap is used so that the keys will be in order of the path. To
    // use it, the first person in the key set will be some person p, and calling
    // map.get(p) will give the edge that goes to that person. The next person in
    // the key set will be the next person in the path, and calling map.get() on
    // that person will give the edge that leads to that person, from the previous
    // person in the path.
    //
    // If no path is found between the two people, returns null.
    public LinkedHashMap<Person, Edge> getPath(Person p1, Person p2) {
        Map<Person, Person> prevMap = new HashMap<Person, Person>();
        Map<Person, Edge> prevEdgeMap = new HashMap<Person, Edge>();
        Queue<Person> curr = new LinkedList<Person>();
        curr.add(p1);
        boolean found = false;
        while(!curr.isEmpty() && !found) {
            Person currPerson = curr.remove();
            if (currPerson.equals(p2)) {
                found = true;
            }
            for (Edge e : getCollabs(currPerson)) {
                Person end = e.end;
                if (!prevMap.containsKey(end)) {
                    curr.add(end);
                    prevMap.put(end, currPerson);
                    prevEdgeMap.put(end, e);
                }
            }
        }
        if (!found) {
            return null;
        }
        LinkedHashMap<Person, Edge> path = new LinkedHashMap<Person, Edge>();
        LinkedList<Person> people = new LinkedList<Person>();
        Person endPerson = p2;
        while (!endPerson.equals(p1)) {
            people.add(0, endPerson);
            endPerson = prevMap.get(endPerson);
        }

        for (Person p : people) {
            path.put(p, prevEdgeMap.get(p));
        }
        return path;
    }

    // Gets an edge between two people in the graph. If no edge exists, returns
    // null.
    private Edge getEdgeBetween(Person p1, Person p2) {
        for (Edge e : vertices.get(p1)) {
            if (e.end.equals(p2)) {
                return e;
            }
        }
        return null;
    }

    public static void main(String[] args) throws IOException {
        // Setup for deserializing the JSON data
        final GsonBuilder gsonBuilder = new GsonBuilder();
        gsonBuilder.registerTypeAdapter(Person.class, new PersonDeserializer());
        gsonBuilder.registerTypeAdapter(Song.class, new SongDeserializer());
        gsonBuilder.registerTypeAdapter(Album.class, new AlbumDeserializer());
        final Gson gson = gsonBuilder.create();

        String dataFile = "huge_writers2.json";
        WriterGraph graph = new WriterGraph(gson, dataFile);

        System.out.println("Num writers: " + graph.vertices.size());
        // for (Person p : graph.vertices.keySet()) {
        //  System.out.println(p);
        //  List<Edge> collabs = graph.getCollabs(p);
        //  System.out.println(p + " has " + collabs.size() + " collabs");
        //  for (Edge e : collabs) {
        //      System.out.println("collabs with " + e.end.name);
        //      for (Song s : e.collaborations) {
        //          System.out.println(s);
        //      }
        //      System.out.println();
        //  }
        // }

        // Finds all the separate, disjoint graphs across all the writers.
        // For each writer, if they aren't already part of a subgraph, start
        // building a new subgraph with that person. Explore the edges outward,
        // adding everyone connected to that one person into the subgraph. When
        // there's no one left to explore, then we've found a connected subgraph.
        //
        // Going on to the next person, they will either be part of the already
        // found subgraphs, or they will be part of a new subgraph.
        Set<Person> consideredPeople = new HashSet<Person>();
        List<Set<Person>> disjointGraphs = new ArrayList<>();
        for (Person p : graph.vertices.keySet()) {
            if (!consideredPeople.contains(p)) {
                Set<Person> subgraph = new HashSet<Person>();
                Queue<Person> currentPeople = new LinkedList<Person>();
                currentPeople.add(p);
                while(!currentPeople.isEmpty()) {
                    Person curr = currentPeople.remove();
                    consideredPeople.add(curr);
                    subgraph.add(curr);
                    for (Edge e : graph.getCollabs(curr)) {
                        if (!consideredPeople.contains(e.end)) {
                            currentPeople.add(e.end);
                        }
                    }
                }
                disjointGraphs.add(subgraph);
            }
        }
        System.out.println("num disjoint graphs: " + disjointGraphs.size());
        // for (Set<Person> set : disjointGraphs) {
        //  System.out.println("subgraph with size " + set.size());
        //  for (Person p : set) {
        //      System.out.println(p);
        //  }
        //  System.out.println();
        // }

        Set<Person> mainGraph = disjointGraphs.iterator().next();
        System.out.println(mainGraph.size());
        // Person start = new Person("Taylor Swift", "http://www.allmusic.com/artist/taylor-swift-mn00
        // Person start = new Person("Onika Maraj", "http://www.allmusic.com/artist/onika-maraj-mn0002
        // Person end = new Person("Ed Sheeran", "http://www.allmusic.com/artist/ed-sheeran-mn00026396
        // Person end = new Person("Michael Jackson", "http://www.allmusic.com/artist/michael-jackson-
        // Person end = new Person("Tim McGraw", "http://www.allmusic.com/artist/tim-mcgraw-mn00005929
        // Person end = new Person("Toby Keith", "http://www.allmusic.com/artist/toby-keith-mn00005108
        Person end = new Person("Jimmy Paterson", "http://www.allmusic.com/artist/jimmy-paterson-mn000
        Person start = new Person("Roger Cook", "http://www.allmusic.com/artist/roger-cook-mn000201318
        // Person end = start;

        System.out.println("finding path between " + start + " and " + end);
        LinkedHashMap<Person, Edge> path = graph.getPath(start, end);
        if (path == null) {
            System.out.println("no path.");
        } else {
            Person prev = start;
            System.out.println("path:");
            for (Person p : path.keySet()) {
                System.out.println(prev + " to " + p + " through the collaborations:");
                for (Song s : path.get(p).collaborations) {
                    System.out.println("\t" + s);
                    break;
                }
                prev = p;
            }
        }

        int maxDistance = 0;
        Person p1Max = null;
        Person p2Max = null;
        Map<Person, Double> meanDists = new HashMap<Person, Double>();
        for (Person p : mainGraph) {
            // System.out.println("distances for " + p);
            Map<Person, Integer> distances = graph.getDistances(p);
            int totalDist = 0;
            for (Map.Entry<Person, Integer> entry : distances.entrySet()) {
                // System.out.printf("%40s%3d\n", entry.getKey(), entry.getValue());
                totalDist += entry.getValue();
                if (entry.getValue() > maxDistance) {
                    maxDistance = entry.getValue();
                    p1Max = p;
                    p2Max = entry.getKey();
                }
            }
            meanDists.put(p, 1.0 * totalDist / mainGraph.size());
        }

        System.out.println("max distance: " + maxDistance + " between " + p1Max + " and " + p2Max);
        System.out.println("mean distances...");
        List<Map.Entry<Person, Double>> list = new LinkedList<>(meanDists.entrySet());
        Collections.sort(list, new Comparator<Map.Entry<Person, Double>>()
        {
            public int compare(Map.Entry<Person, Double> o1, Map.Entry<Person, Double> o2)
            {
                return (o1.getValue()).compareTo( o2.getValue() );
            }
        } );

        for (Map.Entry<Person, Double> entry : list) {
            System.out.printf("%-40s%.3f\n", entry.getKey(), entry.getValue());
        }
    }
}
```

Appendix 2

```java
/*
   This is a private function that takes in two Strings a file name for the
   writer file and a file name for the matrix. The writer file contains all the
   writer's name and a assigned index that correspond to the indexing in
   the output matrix. The matrix file contains only 0s and 1s in its entry.
 */
private static void formatMarkov(String writerFile,
    String matrixFile,
    WriterGraph graph) throws IOException {

    // initialize a buffer writer and a file writer
    // for writing writers
    BufferedWriter bwWriter = null;
    FileWriter fwWriter = null;
    fwWriter = new FileWriter(writerFile);
    bwWriter = new BufferedWriter(fwWriter);

    // initialize an array list that store all the writers
    ArrayList<Person> peopleIndex = new ArrayList<>();

    // initialize a 0 index
    int i = 0;
    for (Person p: graph.vertices.keySet()) {
        if(!graph.getCollabs(p).isEmpty()) {
            // choose only the writer that has previously collaborated
            // with other people and write its name and the index to
            // the file
            peopleIndex.add(p);
            bwWriter.write(i + ", " + p.name + "\n");
            // increment index
            i++;
        }
    }

    // close the buffer writer and file writer
    bwWriter.close();
    fwWriter.close();

    // initialize a buffer writer and a file writer for
    // writing matrix
    BufferedWriter bwMatrix = null;
    FileWriter fwMatrix = null;
    fwMatrix = new FileWriter(matrixFile);
    bwMatrix = new BufferedWriter(fwMatrix);

    // for loop through all writer and write their connection
    // with other writers
    for (int k = 0; k < peopleIndex.size(); k++) {

        Person p = peopleIndex.get(k);
        System.out.println(p.name);
        // initialize a array list that contains all the entry
        // for the matrix row
        ArrayList<Integer> m = new ArrayList<Integer>();

        // initialize the array with all 0 in its entry
        for (int z = 0; z < peopleIndex.size(); z++) {
            m.add(0);
        }

        // get all the connection the writer has
        for (Edge e: graph.getCollabs(p)) {
            Person node1 = e.start;
            Person node2 = e.end;
            if (!node1.equals(p)) {
                // current writer is node 2 then get the
                // index of the other writer and set the
                // entry of the array m of such index to 1
                int index = peopleIndex.indexOf(node1);
                m.set(index, 1);
            } else {
                // current writer is node 1 then get the
                // index of the other writer and set the
                // entry of the array m of such index to 1
                int index = peopleIndex.indexOf(node2);
                m.set(index, 1);
            }
        }

        // write the matrix row to file
        String toWrite = m.get(0).toString();
        for (int t = 1; t < m.size(); t++) {
            toWrite += "," + m.get(t);
        }
        toWrite += "\n";
        bwMatrix.write(toWrite);
    }

    // close the buffer writer and file writer
    bwMatrix.close();
    fwMatrix.close();
}

```

```
                                Appendix 3

import numpy as np
import scipy.linalg as la


def mcl(tempMat):
    # set the initial tolerance to be greater than 1e-15
    tol = 10
    while tol > 10**-15:
        # expansion
        # compute the square of the matrix
        intMat = la.blas.dgemm(1.0, tempMat, tempMat)

        # inflation
        # compute the square of the matrix
        squareMtr = intMat**2
        # normalize the matrix column
        newMat = squareMtr / squareMtr.sum(axis=0)
        # compare norm of old and new matrix
        tol = la.blas.snrm2(tempMat - newMat)
        tempMat = newMat
        print tol

    # return result matrix
    return intMat

# read in the file
print 'reading...'
# create a zero matrix of 19887 by 19887
mat = np.zeros([19887, 19887], dtype=float)
f = open("matrix2.txt", "rb")
line = f.readline()
i = 0
# set the value of matrix by each line read from file
while line:
    y = [value for value in line.split(',')]
    mat[i] = y
    line = f.readline()
    i = i+1

# normalize matrix
print 'normalizing...'
normalizedMat = mat / mat.sum(axis=0)

# call mcl function
print 'computing mcl...'
finalMat = mcl(normalizedMat)
```

Appendix 4

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

/*
    This is a class that contains all the functions to analyse the cluster
    of a graph by reading the matrix computed by MCL algorithm
**/
public class Cluster {

    // a map that map the cluster to its attractor
    public static Map<Set<String>, Set<String>> clusters = new HashMap<>();
    // a array contain all writers
    public static ArrayList<String> allWriter = new ArrayList<>();

    /*
        This is the main function
    **/
    public static void main(String[] args) throws IOException {

        buildWriters("allWriters2.txt");
        buildCluster("rowToCol.txt");

        // print the total number of clusters
        System.out.println("Number of Clusters: " + clusters.size());

        // get the largest 10 clusters in graph
//      ArrayList<Set<String>> topK = getTopK(10);
//      System.out.println();
//      for (Set<String> k: topK) {
//          // print attractor and cluster size
//          System.out.println(clusters.get(k) + " " + k.size());
//          // print cluster
//          System.out.println(k);
//          System.out.println();
//      }

        // print out size of all clusters
        printClusterSize();
    }

    /*
        This is a private function that takes in a string file name and
        put all the writers into the allWriter array
    **/
    private static void buildWriters(String filename) throws IOException {

        // initialize file reader and buffer reader
        FileReader fr = null;
        BufferedReader br = null;
        fr = new FileReader(filename);
        br = new BufferedReader(fr);

        String currentLine = "";
        while ((currentLine = br.readLine()) != null) {
            String [] tokens = currentLine.split(",");
            allWriter.add(tokens[1]);
        }

        if (br != null)
            br.close();

        if (fr != null)
            fr.close();
    }

    /*
        This is a private function that takes in a string file name and
        build the cluster of the graph
    **/
    private static void buildCluster(String filename) throws IOException {

        // initialize a map that maps a attractor to its clusters
        Map<String, Set<String>> oneAttractor = new HashMap<>();

        // initialize file reader and buffer reader
        FileReader fr = null;
        BufferedReader br = null;
        fr = new FileReader(filename);
        br = new BufferedReader(fr);

        // throw away the header (first line)
        String currentLine = br.readLine();

        while ((currentLine = br.readLine()) != null) {
            // get the associated writer using the row column index from file
            String [] tokens = currentLine.split(" ");
            String attractor = allWriter.get(Integer.parseInt(tokens[0])).trim();
            String other = allWriter.get(Integer.parseInt(tokens[1])).trim();

            if (!oneAttractor.containsKey(attractor)) {
                // if attractor not in map add a key to map
                oneAttractor.put(attractor, new HashSet<String>());
            }

            // add writer being attracted to cluster
            oneAttractor.get(attractor).add(other);
        }

        if (br != null)
            br.close();

        if (fr != null)
            fr.close();

        // print the total number of attractors
        System.out.println("Number of Attractors: " + oneAttractor.size());

        // go through all element in oneAttractor list
        for (String attr: oneAttractor.keySet()) {
            // get the cluster by the attractor
            Set<String> others = oneAttractor.get(attr);

            if (!clusters.containsKey(others)) {
                // if cluster not exist in clusters add key
                clusters.put(others, new HashSet<String>());
            }

            // add attractor as value in map
            clusters.get(others).add(attr);
        }

    }

    /*
```

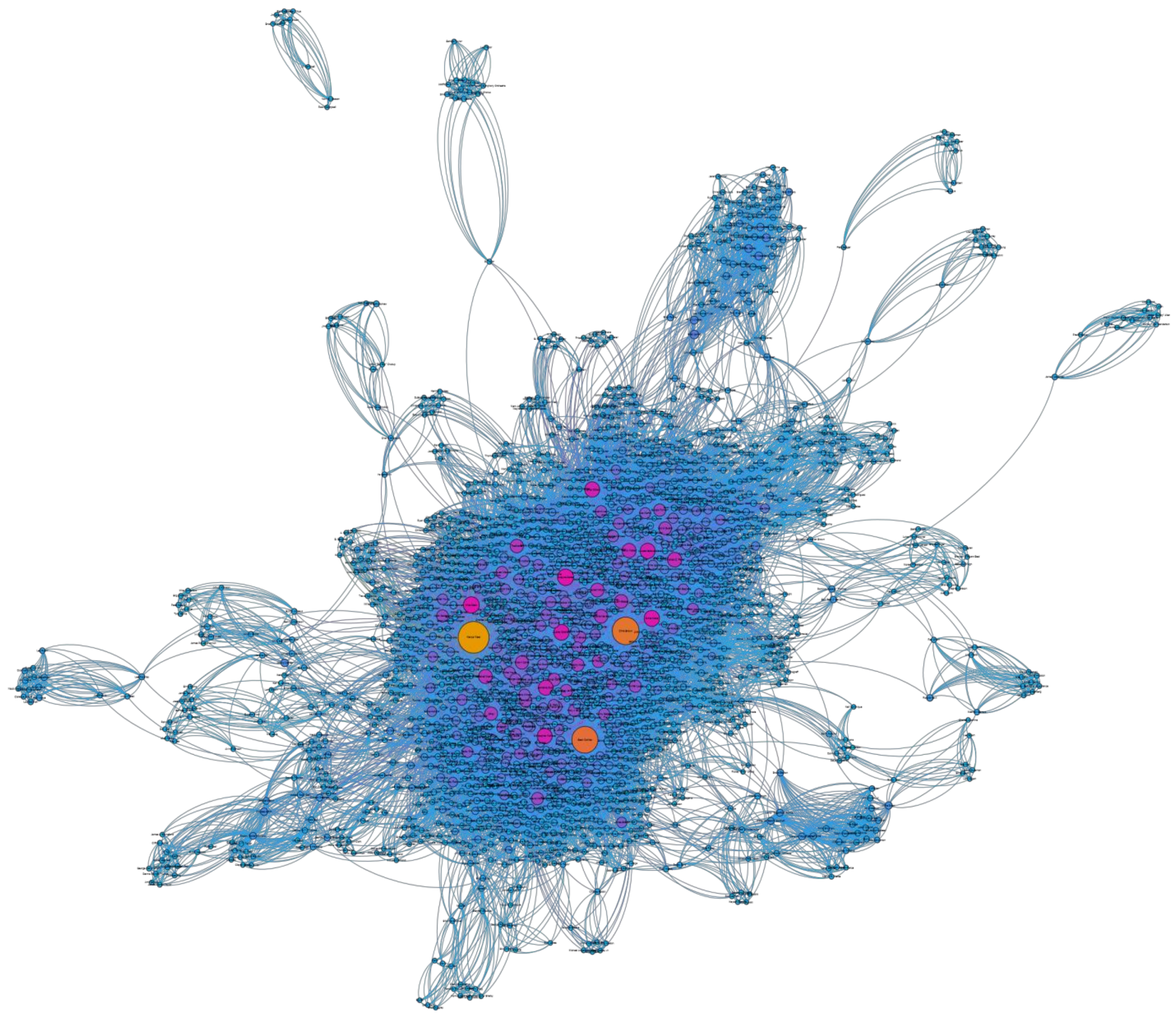
```java
        This is a private function that takes in a number k
        and return a list of biggest k clusters
    **/
    private static ArrayList<Set<String>> getTopK (int k) {

        // the index where element should be removed from top k
        int minIndex = -1;
        int minSize = Integer.MAX_VALUE;

        // create an empty array list contain largest k cluster
        ArrayList<Set<String>> topK = new ArrayList<>();

        for (Set<String> cluster: clusters.keySet()) {

            if (topK.size() != k) {
                // if top k array does not have enough element
                topK.add(cluster);

                if (cluster.size() < minSize) {
                    // if the add in cluster size is smaller
                    // than the min size set the new min size
                    // and set the index for next remove
                    minIndex = topK.size() - 1;
                    minSize = cluster.size();
                }

            } else {
                // the array contains k elements already
                int currSize = cluster.size();
                if (currSize > minSize) {
                    // if the current evaluating cluster has
                    // size bigger than the smallest cluster s
                    // in list, remove s from list and add the
                    // current cluster
                    topK.remove(minIndex);
                    topK.add(cluster);

                    // for loop through the new list and find
                    // the minimum cluster size in list and find
                    // its index
                    minSize = Integer.MAX_VALUE;
                    for (int i = 0; i < topK.size(); i++) {
                        int currKSize = topK.get(i).size();
                        if (currKSize < minSize) {
                            minIndex = i;
                            minSize = currKSize;
                        }
                    }
                }
            }
        }

        // return the largest k cluster
        return topK;
    }

    private static void printClusterSize() {
        int[] clusterSize = new int[clusters.size()];

        int i = 0;
        for(Set<String> cluster: clusters.keySet()) {
            clusterSize[i] = cluster.size();
            i++;
        }
        System.out.println(Arrays.toString(clusterSize));
    }

}
```

Appendix 5

```
r = load('matrix1.mat');
markovChain = r.matrix;
qwer = mpower(markovChain, 100);
save('fianl_matrix1.mat','qwer');


r = load('fianl_matrix1.mat');
finalM = r.qwer;
M200th = finalM * finalM;


r = load('final_matrix1_200.mat');
finalM = r.qwer;
sumlit = [0:16398]';
sumlit(:,2) = 0;

max = 0;
column = 0;
index = 1;
for i = 1:1:16399
    column = sum(finalM(:, i)); %colum sum
    sumlit(i, 2) = column;
    if max < column
        max = column;
        index = i;
    end
end
```

Appendix 6: A visualization of the songwriter graph produced using Gephi (https://gephi.org/)

To reduce the density of the graph somewhat, the 8-core of the original graph is shown below, which is the largest subgraph of the original such that each vertex has a degree of at least 8.



Nodes are colored and sized according to their degree. Nodes with small degrees are blue, with a transition to purple and then orange. Larger nodes have larger degrees. Kanye West is the yellow-orange vertex in the left center of the large central cluster, and Sean Combs and Chris Brown are the two other prominent orange nodes.

The central cluster contains writers of rap, hip-hop, and pop songs, with pop song writers appearing in the top right section, with a transition to hip-hop and rap moving down and to the right.

The cluster towards the top right contains country songwriters, and there are visible connections between country songwriters and pop songwriters.

References

1. Brian Hayes, Graph Theory in Practice: Part I, American Scientist, (2000) <http://www.americanscientist.org/issues/pub/graph-theory-in-practice-part-i/1>

2. Jerrold W. Grossman, The Erdős Number Project, Last modified November 26, 2016, <http://www.oakland.edu/~grossman/erdoshp.html.>

3. Grossman, Jerrold W. "The Evolution of the Mathematical Research Collaboration Graph."Congressus Numerantium (n.d.): 201-12. Winnipeg, Utilitas Mathematica, 1998. Web. 9 Mar. 2017. <https://oakland.edu/Assets/upload/docs/Erdos-Number-Project/eddie.pdf>.

4. Michael Barr, Rational Erdős Numbers (2001) <https://oakland.edu/Assets/upload/docs/Erdos-Number-Project/barr.pdf>

5. Patrick Reynolds, The Oracle of Bacon, Accessed March 4, 2017. <http://oracleofbacon.org/how.php>

6. Teo Paoletti, Leonard Euler's Solution to the Königsberg Bridge Problem, (2011) <http://www.maa.org/press/periodicals/convergence/leonard-eulers-solution-to-the-konigsberg-bridge-problem>

7. Dongen, Stijn Van. "MCL - a Cluster Algorithm for Graphs." MCL - a Cluster Algorithm for Graphs. N.p., May 2000. Web. Accessed March 5, 2017. <http://micans.org/mcl/>

8. Nykamp DQ, "Small world networks." From Math Insight. <http://mathinsight.org/small_world_network>

9. Humphries MD, Gurney K (2008) Network 'Small-World-Ness': A Quantitative Method for Determining Canonical Network Equivalence. PLOS ONE 3(4): e0002051. <http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0002051>

10. Watts DJ, Strogatz SH (1998) Collective dynamics of 'small-world' networks. Nature 393: 440–442.

<https://static.squarespace.com/static/5436e695e4b07f1e91b30155/t/54452561e4b08d9eb2170909/1413817697054/collective-dynamics-of-small-world-networks.pdf>

11. Barabasi, Albert-Laszlo; Bonabeau, Eric: Scale-Free Networks. Scientific American 2003. <http://barabasi.com/f/124.pdf>

12. Preiss, Bruno R. “Sparse vs Dense Graph.” Data Structures and Algorithms with Object-Oriented Design Patterns in C++. Accessed March 7, 2017. <http://www.brpreiss.com/books/opus4/html/page534.html>

13. Marr, Paul. “Transportation Network Analysis Connectivity Index.” Transportation Methods. Shippensburg University. Accessed March 8, 2017. <http://webspace.ship.edu/pgmarr/TransMeth/Lec%202-Connectivity.pdf>

14. Ducruet, Cesar; Rodrigue, Jean-Paul. “Graph Theory: Measures and Indices.” The Geography of Transport Systems. Accessed March 8, 2017 <https://people.hofstra.edu/geotrans/eng/methods/ch1m3en.html>

15. Polyhedra and Graphs: Planar Graphs and Euler's Formula /Counterexamples. (n.d.). Accessed March 10, 2017 <http://www.eprisner.de/MAT107/Polyhedra/Eulers.html>