Project Number: MOW-0901

Computer Visualization of Song Lyrics

A Major Qualifying Project Report
submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by

____
Andrew Labrecque

____
Jason Stasik

Date: December 17, 2009

Approved:

____

Professor Matthew O. Ward, Major Advisor

Abstract

Currently, there are several standard methods for recommending music to an individual. These methods vary from word-of-mouth to large recommendation databases by retail websites such as Amazon.com. This project attempts to visualize the lyrical relationships found in a large collection of songs. We discover these relationships by examining both the structural and contextual components of a song's lyrics, and using those components in a broad comparison algorithm. We gathered a set of 5605 song lyrics and associated song data in order to establish a large network of relationships to navigate and explore. The resulting visualizations allow users to graphically explore the collection in order to discover trends, interesting data, or simply to find new songs to listen to. We performed user testing in order to discover the ease-of-use of our system as well as the subjective accuracy of its recommendations.

Acknowledgements

We would like to thank Professor Matthew Ward for advising this project, as well as for the exceedingly helpful advice he has offered us over these past three terms.

We would also like to thank Jeffrey Heer for the Prefuse Visualization Toolkit, Greg Fast for the Syllable class, Martin Dudek for the ForceDirectedLayoutMagic class, and Leonard Richardson for the very useful BeautifulSoup HTML parser.

Table of Contents

Table of Figures

1 Introduction

Accurate comparisons of songs are an area of great interest to many people today. Record labels, retail stores, and individuals all have various reasons to want to learn how similar two songs are to each other. For record labels and retail stores this information can be useful for discovering interesting trends, detecting copyright infringement, and deciding what albums to support and promote. For normal listeners, it can be used to find new music and genres to enjoy. Additionally, music enthusiasts could find satisfaction in simply looking upon their own personal collections in new and interesting ways, discovering connections that they were previously unaware of.

Currently, the main method of achieving these comparisons is done with retail databases utilizing sale histories, which tend to recommend entire albums or artists, rather than individual songs. Another method involves analyzing MP3 data in an effort to decode a song's rhythm, instruments, and genre. Our intention with this project is to investigate the feasibility of making similar types of comparisons based upon the lyrics of a song.

The first part of our project attempts to analyze songs through their lyrics. With the lyrics, we attempt to discover two categories of information about each song – the meaning and the structure. The meaning involves the actual words themselves and what each means. By comparing the similarities between words in two songs, we can, to some extent, determine if the topics of the songs are similar. The structure of a song involves how the song is presented. Structural metrics include the number of lines per song, the number of syllables per line, the amount of line repetition, and more.

The second part of this project is visualizing these comparisons. Fitting all of our 5605 songs with all of their relationships on a single screen, and still having the visualization be useful, is impossible. Our visualization restricts what songs are displayed to only show the most relevant comparisons. In addition, we provide several capabilities such as searching, filtering, and multiple visualization types to allow users to easily find what they are looking for. Once a user finds the songs he is searching for, he then has additional tools for finding out to what extent they are related and why that determination was made by our program.

1.1 Goals

The main ambition of this project was to design and implement a proof of concept application for visualizing the relationships between items in a collection of text documents, based upon the comparisons of certain calculated metrics. While we focused on songs and their lyrics over the course of the project, one goal throughout was to not require the input dataset be song lyrics in future incarnations. Theoretically, this same application could be used to compare entire albums, court documents, or any other type of collection with only minimal alterations to the code.

In addition to the goal of keeping the target collection general, the application was designed to run smoothly and be usable on both large and small collections. This means that the data should load quickly and interactive components should be responsive to user input regardless of how large the collection is. Additionally, the application should take full advantage of screen space with smaller collections.

A final goal of this project was for the application to be extensible so that it could be enhanced and modified by future efforts without considerable rework. This goal includes allowing the adding or changing of analysis metrics, parsing documents in new and different ways, and accepting multiple file types with arbitrary metadata tags. Future developers should be able to accomplish these tasks by altering—or replacing—as few source files as possible.

1.2 Objectives

In order to develop an application that would meet the goals set forth, we identified three important objectives that we would aim to accomplish over the course of the project. The completion of each key objective was used to gauge our progress towards the conclusion of the project. They also served as a basis for the tasks we established and strived to finish on a weekly basis. Rather than jumping right into the code on week one and making what could be major mistakes, we spent an appropriate amount of time researching, creating formal design guidelines, implementing our system, documenting our decisions, and evaluating the finished application. As a general guide, we attempted to complete one or two of these objectives every seven weeks.

The first objective of the project was to research and brainstorm ideas for the application we set forth to develop. Previous work in the areas of computer visualization, document and text processing, and information retrieval helped us to establish our ideas as we began laying out a preliminary high level design. Identifying different alternatives, each with strengths and weaknesses, allowed us to choose the best option and adapt as necessary during implementation.

The second objective was to implement the prototype application using the preliminary design as a guide. Strictly adhering to mockups and specific algorithms was not a requirement of this objective. If an aspect of the design needed to be reworked due to unforeseen circumstances it was redesigned prior to implementation. The purpose of this objective was to produce the best possible prototype in the time we would have available.

The final objective consisted of two parts: to document our work and to evaluate our system. This included identifying our original design (and its justifications), justifying any changes to the original design made during implementation, and evaluating the resulting prototype in the interest of guiding possible future work. Self evaluation occurred both alongside and after development, while more formal user evaluations were conducted only after we had produced a stable working prototype.

2 Background

In order to better prepare ourselves to implement a visualization program with a focus on similarity and text processing, we found it worthwhile to research a number of related topics. This section discusses some of the background as it applies to our work, including the concepts of Similarity Algorithms, Computer Visualizations, Fault Tolerance, and Natural Language Processing. While not all of the ideas we researched and presented here made it into our final application they were all important to keep in mind along the way and ought to be considerations for possible future work related to this project.

2.1 Computer Visualization

Computer Visualization, as it applies to this project, is the process of creating diagrams that convey a quality, relationship, or general summary of some data, or group of data. Visualizations should be intuitively understandable through basic visual cues such as size, distance, color, and movement. There are many great examples of Computer Visualizations that can be found in prior work by experts in the field. One example, the Baby Name Wizard developed by researcher Martin Wattenberg[1], shows the popularity of baby names from the 1880s to the present day as an easy to read searchable chart. As the user types a name in a search box the visualization narrows to names spelled similarly and displays the popularity of each as a shaded area of changing size along a time axis. The visualization helps people discover at-a-glance what could take much longer using other means. Some may argue that it has additional value as a form of entertainment.

There are many different areas of Computer Visualization and many different types of Visualizations under each. Many Eyes, a site run by IBM's Collaborative User Experience research group, has a number of different examples of interactive information visualizations including node-and-edge graphs, histograms, pie charts, and geographical territory maps – just to name a few. The site's various datasets help to illustrate the wide variety of possible applications for Computer Visualization. Users can visualize text, census data, murder trends, weather data, blog posts, and almost anything else imaginable. According to Wattenberg[2] and other members of the team that worked on the site, the purpose behind the design and implementation of Many Eyes was to reach as large an audience as possible. To this end, they provided tools for collaboration, sharing, and feedback on customizable visualizations that would draw upon different user-uploaded datasets[2].

Lots of the visualizations on Many Eyes are intended for numerical data. Because we were focusing on the textual data of song lyrics, we were especially drawn to the Tag Cloud visualizations on the site. This visualization displays words from "unstructured text" and "[gives] the user an overview of the most salient terms" based on their scaling[2]. This is not unlike what we see on many websites where popular tags are shown in a list scaled by frequency. We also sought to discover similar text visualizations in addition to this. One that was particularly interesting color-coded terms in document text based on their parts-of-speech, revealing the grammatical structure of a sentence or paragraph and maybe even revealing quality (or lack of it) in the writing[3]. It occurred to us that a similar system could be used to color-code parts of a song's lyrics (chorus, bridge, verse) to view those structural qualities at-a-glance.

Another type of visualization we encountered during our research was a graph-based Radial Layout. This algorithm puts "the focus of attention [on] a single [center] node" with other nodes "arranged on concentric rings" around it, and animates transitions between centered nodes to "maintain orientation" for the user[4]. We especially liked the exploratory interaction of this visualization, allowing users move around a collection by moving between center nodes.

2.2 Visualization Toolkits

A Visualization Toolkit is a software bundle used for creating visualizations without needing to agonize over low level system calls and deal with graphics libraries. Part of our research for this project was to investigate some of the available toolkits and make an informed decision about which to choose for use with our project. Some of the toolkits that we looked at and considered most closely were Prefuse, VTK, and the JUNG framework.

Prefuse is an information visualization toolkit created by Jeffrey Heer[5] and made available under the Berkeley Software Distribution license, allowing it to be redistributed with its original copyright notice with or without modification and free of charge. Features of Prefuse include built in data structures for tables, graphs, and trees; a query language for filters, searching, and selection; database connectivity with SQL for data persistence; and support for automated force directed layout. It supports a good variety of different visualization types and includes useful examples with the source code to demonstrate some its most useful aspects. Prefuse is written in Java.

VTK (The Visualization Toolkit) is another package with more advanced 3D graphics capabilities and multi-language support. It provides more complex types of visualizations than some of the other toolkits we examined and has applications in geology, biology, acoustics, and medicine. While the core of VTK is written in C++ it has wrappers for Python, Java, and Tcl as well. Like Prefuse, VTK is free under the BSD license.

The JUNG (Java Universal Network/Graph) framework is a toolkit with a focus on node-and-edge graph-based visualizations. Again licensed under the BSD license, JUNG is used often in applications for social network analysis, information visualization, and data mining. It supports the force directed spring graph layout and has routines for clustering, decomposition, random graph generation, statistical analysis, distance calculations, and centrality as well.

2.3 Similarity Algorithms

Calculating how related two items are and recommending items based on a user's selection of another item are similar concepts. In many ways this is what we set out to do with our project, so as part of our research we examined recommendation databases. A recommendation is usually just the item most related to the one chosen. Recommendation databases are widely used today for a variety of reasons. Most commonly, they are used by major online retail organizations, such as Amazon.com, as a way of enticing users into buying additional merchandise. There are many different ways to implement a recommendation database system. Systems that use recommendations from other users, Amazon.com's item-to-item collaborative filtering, and Pandora.com's vector-based distance algorithm rank among some of the most used and high profile implementations.

The simplest recommendation system does not involve an algorithm at all, but only some users with a little motivation. In this system, users are able to recommend items themselves. This is usually done in the form of user reviews, where an individual writes a textual appraisal of an item and suggests items other users might enjoy as well. This type of system is often seen on smaller websites that lack the resources required to generate, store, and process millions of transactions; however, most major websites allow users to leave comments or review items in order to utilize this type of system.

Amazon.com utilizes a method that they call item-to-item collaborative filtering[6], a modification off of traditional collaborative filtering. This algorithm works by first creating a similar-items table – a listing of all products and all products that are related to each product in some way – along with a numerical value indicating how related every pair of products is. Products are determined to be related if any customer who bought one product also bought the other; the more people buying both products indicates a higher numerical value for the relationship. This vast set of calculations is periodically performed offline, while the table is stored online. With the resulting similar-items table, Amazon.com's servers are able to efficiently use a customer's purchase history to look up similar items independent to the total number of products in the database.

Pandora.com has a massive song database called the Music Genome Project[7]. This database stores information, designated as genes, for every song. A gene is a characteristic of a song such as the gender of the vocalists or the amount of distortion on a guitar in the instrumentals. Every gene is stored in the database as a number from 0 to 5. In order to compare two songs, each song's genes are represented as a vector. The similarity relation between two songs is calculated generically using the distance formula: distance(S,T) = $\sum_{i=0}^{n} w_i * (s_i - t_i)^2$ such that S and T are the two song vectors to be compared, s_i and t_i represent the value of the ith gene in songs S and T respectively, and w_i is a special unique weighting applied to every gene value. When a recommendation request is made, the system returns the songs with the lowest distances to the requested song.

2.4 Fault Tolerance

The concept of fault tolerance is best explained with an example. In keeping with other examples throughout the rest of this report we will use music and song lyrics as members of our assumed document collection.

Imagine that our corpus contains music from the artist "Dream Theater" but the user of the application performs a search with the condition: Artist equal to "Dream Theatre" (note the alternate spelling of Theater). There is no way for a program to know with certainty whether "Dream Theater" and "Dream Theatre" are two different bands or actually two different spellings of the same band's name. The easiest solution is to assume they are two different bands and leave it up to the creator of the corpus to keep fields like this consistent in spelling. As an alternative the program could be made more accommodating by providing the option to include very similarly spelled terms in searches involving string matching conditions. This type of fault tolerance also comes in handy in searching through text containing words where the first or last letter have been omitted out for effect or due to unintentional misspelling.

The Levenshtein Distance Algorithm[8] is useful for providing fault tolerance in these types of situations. This algorithm considers the number of steps needed to transform one string into another. This number is the Levenshtein Distance.

Levenshtein Distance revolves around the operations of character insertion, deletion, and substitution. The goal is to determine the minimum number of these operations that are required to get from one word to another. The algorithm uses a matrix to accomplish this task and fills it in using a very specific set of steps. Once constructed, the value in the lower right hand corner of the matrix is assured to be the Levenshtein Distance between the two terms[8].

When a user performs a search, the Levenshtein Distance between the search string and each target string can be calculated to see if any are within a certain threshold. If it is, the search can be altered to include both the requested string and the fault-tolerant string.

We recognized advantages to making the above approach more interactive. If, for example, the application finds a term with a low Levenshtein Distance to the term being searched for, it could prompt the user to decide whether to treat the two words as the same. Storage of the various fields and text in the system could be a way of storing these words for future searches to be performed without the procedure. Following this system an application can effectively "remember" when to treat two similarly spelled words as the same. It can also allow the user to specifically tell the application if two similarly spelled bands are actually two separate groups.

A shortcoming of Levenshtein Distance may be efficiency, especially if it is calculated over and over. For example: if a user wishes to view only documents containing a certain term in their text, it is not desirable for the Levenshtein Distance to be calculated for every word in the text of every document in the collection.

2.5 Stop Words

A stop word is "a word (usually one of a set of the words most frequently occurring in a language or text) that is automatically omitted from or treated less fully in a computer-generated concordance or index."[9] In our application, common words between many songs, such as "I" and "and" are filtered out so that when we compare two songs, the unwanted words are not taken into account for the calculation. An algorithm could also be used to automate the process of finding the most common words and removing them from the data being processed.

3 Design and Preparation

In the design of our system—an application for visualizing the relationships between text documents—we often found there to be multiple ways of fulfilling certain requirements, each with advantages and disadvantages. Sometimes the most important of these trade-offs were easy to identify, allowing us to make decisions that carried through into the implementation phase of our project. Other times this was not the case, and certain decisions needed to be revised as we proceeded with our work. The main factors that we took into account in making these decisions were the efficiency and simplicity of the alternatives.

Efficiency was a key concern for us because we had set forth a goal to have our application be usable given a target collection with a possibly large number of documents. This opened up the potential for the program to have a particularly large memory footprint and very long CPU-intensive computations. We wanted our efforts to result in an application that would run on a typical personal computer and without long loading times on startup.

Simplicity was also a key factor for us in the context of time and ease-of-implementation. Given the seven weeks we would have available to implement our application it was clear that we would not have the liberty to implement every optimization that was available to us. For this reason we tried to seek a balance between the efficiency of targeted algorithms and the time it would take to code these algorithms. Our hope was that this consideration would result in a good proof-of-concept for an application that could be enhanced and extended by future work, per our goal.

This section defines the basic breakdown of our application and describes the various alternative decisions that were available to us as we designed and implemented it. The next section, Architecture and Implementation, provides additional details on the final implementation of the system.

3.1 Lyrics Gathering

The first task that had to be done at the start of our project was to gather a large collection of song lyrics with which to use and test our system. In addition to song lyrics, we wanted to also have information about each song available, such as genre or popularity. The process of lyrics gathering can be broken down into two sections: downloading and extracting information from the Internet and storing that data.

3.1.1 Downloading

In order to download our desired information, we required a set of artists and song titles. While researching, we discovered the uspop2002 dataset[10]. This data was a list of 8764 songs from 400 different artists. While it was possible instead to perhaps crawl a lyrics website and download the lyrics for every single song on the site, this list allowed us to achieve similar results in much less time as we did not have to deal with the issues of automatically navigating a large series of different web pages.

For additional information about songs, we immediately decided upon last.fm as it contained exactly what we wanted. Last.fm is a large music database where users can upload information about what songs they are currently listening to, and then see what songs or artists they end up listening to the most. Last.fm also provides information about artists, albums, and songs. The following is a list of song information that we originally intended to extract:

- Genre(s)
- Related songs
- Related artists
- Description of song
- Number of total track plays
- Number of unique listeners
- Link to last.fm page

Upon further reflection, we decided against storing related songs and artists, as that data would only really be useful to our application if the recommended songs were only songs in our dataset. Descriptions were also not included due to the fact that descriptions were generally about the history of a particular album or the band as a whole, rather than the story behind an individual song.

Instead of genres, last.fm used tags to categorize each song. We believed it to be fair to assume, for our purposes, that tags and genres are the same thing. In order to assure that we would get at least one useful genre, we chose to take the three most popular tags and use each of these as a different genre for the song. An example of a tag that is not useful would be a song by The Beatles being tagged with Beatles. As an alternative to last.fm, we identified allmusic.com as a potential backup site, which keeps track of similar information based on artists instead of individual songs. In the end, we did not end up needing to use allmusic.com.

While last.fm was being used to gather metadata about each song, we still needed the lyrics. We looked at seven popular lyrics websites, including: lyrics.com, songlyrics.com, azlyrics.com, elyrics.net, lyricsmode.com, last.fm, and metrolyrics.com. While evaluating the pros and cons of each website, we eventually decided upon metrolyrics.com for its extremely large selection, accuracy of lyrics, and ease of parsing. We also decided upon a backup website to use in case the lyrics could not be found at metrolyrics.com. For the backup site, we decided upon azlyrics.com for its accuracy of lyrics and ease of parsing. Any song that couldn't be found on last.fm, metrolyrics.com, or azlyrics.com would have to be discarded.

When deciding upon what language to use to do this, there were many options. Initially we intended to use Java, as that was what we were both most experienced with. However, after some research, we discovered a tool called BeautifulSoup[i] which was written in Python and would allow us to very easily extract information from a website. Python has many additional advantages over Java for a lightweight application like this one. Python tends to be shorter and

[i] BeautifulSoup is distributed under the BSD license and can be downloaded at http://www.crummy.com/software/BeautifulSoup/

easier to write, as well as having more advancing string-handling functions and much simpler I/O operations.

3.1.2 Storing

After downloading this information we had to store it in a way that would later be easily read in by our application. Because we stated in our original goals that we intended for our application to be generic and accept any type of document with any type of metadata, we opted for an XML format. XML would allow us to easily add or remove tags, as well as easily read tags and what their parent tags were. The following represents a sample XML file in our format for the song Mrs. Robinson by Simon & Garfunkel:

```
<document name="Simon & Garfunkel – Mrs. Robinson">
    <attributes>
        <title>Mrs. Robinson</title>
        <artist>Simon & Garfunkel</artist>
        <genre>classic rock</genre >
        <genre>60s</genre >
        <genre>folk</genre >
        <plays>2349059</plays>
        <listeners>342088</listeners>
        <url>http://last.fm/music/Simon%2B%2526%2BGarfunkel/_/Mrs.+Robinson</url>
    </attributes>
    <text >lyrics go here</text >
</document>
```

In this format, name would be whatever a user wanted to name the document. The attributes tag represents the metadata for a document, and a user would be able to list as many as they wanted, including duplicates such as genre. Finally, the text tag indicates the text content of the document to be analyzed and compared. In our case, the text of songs is their lyrics.

3.2 Document Loading and Persistence

As mentioned previously, we established a system to store all of our downloaded lyrics and song information to XML files. By using XML files we had an easy-to-use format for storing generalized information that could also be easily read into our application with a standard Java XML reader. Because we allowed for the use of any number of arbitrary metadata tags, we decided to use an options file to define the current format. This options file consists of a list of each metadata tag and what its associated data type is. We allowed strings, booleans, integers, and floats to be specified. Our application, for instance, consisted of:

```
title string
artist string
genre string
genre string
```

genre string
plays integer
listeners integer
url string

Due to the fact that we parse the XML files, read in all their data, and then parse the text field to extract the metrics data every single time our application loaded, it took considerable time and memory to first load our application. In addition, we also had to calculate relevance values every time the application loaded. With such long load times no one would even considering using such an application, so we began to look into methods to reduce load time.

In order to mitigate the cost of reading XML files, we developed a new file type, which we designated NXP (Non-XML Preprocessed file). The NXP file was a way to read in XML files which we had already been processed in a very fast and efficient way. The NXP files had the following format and data:

Line 1, name
Line 2, title
Line 3, artist
Lines 4-n, metadata
Line n+1, number of lines
Line n+2, number of unique lines
Line n+3, max repetitions
Line n+4, number of syllables
Line n+5, all words in the song separated by spaces
Line n+5, number of occurrences of each word on the above line separated by spaces
Line n+5-end, the text of the document

The options file defined previously allowed us to know exactly how many lines to read for metadata tags. Since we know what to expect on each line, we can easily read in line-by-line all of the data. This means we do not need to read or parse the XML data, which takes longer than a normal file due to the way XML is parsed into a tree structure. We also do not need to process the text field again, as all the data we need to calculate the metrics is stored in the NXP file. The downside to this method is that if the stop word list changes, all files must be re-processed; however, that does not happen often and it does not take more than 10 minutes on an average computer to convert all of our XML files to NXP. Before NXP files, our application took approximately 5-10 minutes to load with 5605 XML files. With NXP files, 5605 files load in approximately 10-20 seconds.

In order to prevent the recalculation of all comparisons, which was necessary because the calculations took approximately 6 hours to complete with all songs, we developed two additional file types to help store this information.

The first file type was called DLT, the Document LisT. Not surprisingly, this was a list of all documents used in the application. This was used because we wanted the user to be able to store all their NXP files in one location, but still be able to have different subsets of visualizations available without duplicate processed files.

The second file type was called REL, the RELevance Table. This stored a list of all the relationship information for every single song listed in the associated DLT file. The first 5 lines consist of the standard deviations of the first five metrics, while the remaining lines consist of relationship data listed in order by document and only containing data for documents with a lower ID than it. This means that line 6 corresponds to the first document in the related DLT file and contains no information because it has the lowest ID. Line 8 corresponds to the third document and contains the relevance values between it and documents one and two (but not the value between document one and two, which is stored on line 7). Relationship data was stored as integers, as discussed in section 4.2.1. REL and DLT files were created in pairs automatically and they shared the same name, with only their extension differing, so that we could easily match them together.

3.3 Document Processing and Storage

Given the large quantity of data we would be dealing with it seemed worthwhile to consider many different systems for processing and storing processed documents in our system. As we have outlined, important considerations for each option include its efficiency and ease-of-implementation. However, given that the rest of our system would be built entirely around how we processed and stored documents, it seemed necessary to place particular emphasis on efficiency to prevent a potential bottleneck in this important part of our system.

Processing involved two steps: analyzing the lyrics of a song and calculating how relevant they were. Extracting information from the lyrics is simple and achieved by reading every line and word and storing final value in an object. However, an important part of gathering the list of words in a song includes the stop word list.

3.3.1 Stop Word List

Because the input data collection could consist of any type of text document, the words we want to eliminate will vary greatly in different circumstances. For song lyrics, we extracted the words that we thought were the most relevant to this project from [11], with some additions tailored to songs specifically:

i	for	you
a	from	me
an	how	my
and	in	your
are	incomprehensible	of
as	instrumental	that
at	is	i'm
be	it	so
by	the	we
chorus	to	it's

with	don't	get
but	when	if
do	this	you're

However, after testing the results of using this list on our finished product, we found that most of the songs that were being declared similar were full of words that had no real significance associated with them. To attempt to counter this, we examined the 100 most common words and picked the following words to add to our list:

was	where	too
will	can	then
i'll	can't	than
who	they	its
that's	they're	
what	i've	

3.3.2 Relevance Algorithms

We had many ideas for how to calculate the degree of similarity between two songs. In the beginning, we envisioned having several simple algorithms that we could easily switch between. These were algorithms included:

- The number of words that are common between two documents.
- The number of occurrences of words common between two documents.
- The number of occurrences of words common between two documents with each word occurrence weighted by how unique that word is to the pair.
- The percentage of words in common between two documents to those that are not common.

There were a few more algorithms that were very similar to the four above. However, when we tested some of these out in our application with a test subset of 100-500 songs, we found the results to be unacceptable. In general, these algorithms were too simple or too general. When manually comparing the lyrics between a related pair, we did not believe that the two songs were actually similar in any way most of the time. In response we went back, revised our strategy, and came up with the idea to add structural analysis in addition to contextual. There were several different structural qualities we could analyze given a set of lyrics, some of which included:

- Number of lines in a document
- Number of words in a document
- Number of unique words in a document
- Number of unique lines in a document
- Number of syllables in a word
- Number of rhymes between words in a line, or lines in the whole document

- Scheme or pattern of rhymes that exist in a document

With this data, we also gained access to values such as syllables per line, syllables per word, amount of word repetition, amount of line repetition, and others through a simple calculation. Unfortunately we were forced to abandon the rhyming analysis. Though we did find an accurate rhyming dictionary[ii], we found that it was missing a substantial number of words from our entire collection of songs. Even after checking only the last word on each line for rhymes, there were still 10,000 needed words that were missing from the dictionary.

Ultimately we settled on using six metrics for comparisons, with room to add additional ones later on if we felt that it was necessary. The following is a list of the metrics we chose and what we hoped to gain from using them:

- Total Lines, the length or tempo of a song
- Lines Per Unique Line, how often line repetition occurred
- Syllables Per Line, the complexity of words or specific syllable patterns
- Words per Unique Word, how often word repetition occurred
- Maximum Times a Line Repeats, how repetitive a song could be
- Percent Words In Common, contextual similarity

We felt that these six metrics provided a broad overview of general patterns within a song, without providing unnecessary redundancy or complexity. The percent words in common algorithm was chosen over others in order to provide accurate results for two similar songs of unrelated lengths.

The next step was incorporating these metrics into our comparisons. First was percent of words in common. This had to be handled differently, as it was not a static value in each song that could be easily compared. The algorithm used was as follows:

```
int rel = 0
for each word w in song1:
        rel += Math.min(occurrences of w in song1/number of words in song1,
                        occurrences of w in song2/number of words in song2)
```

In order to determine if two songs were significantly related, we set a minimum required rel (short for relevance) value. Originally, our default rel value, with the original stop word list, was 25%, or 0.25. Given that there are so many words in the English language, and several words can be used to say the same thing, we did not want to restrict ourselves to an unrealistic value such as 50%, where nearly no connections would have occurred. After changing our stop word list to be more rigorous, we lowered the minimum required value to 20%, in order to accommodate for the fact that almost all relationships had a significant drop in their

[ii] The rhyming dictionary is called "phondic.english" and can be found at http://www.ic.arizona.edu/ic/hammond/mhj/phondic.english

percentages. This eliminated many of the false relations achieved from popular words such as "what" or "you're" but still required sufficient enough similarity to produce reasonable results.

The other five metric algorithms were different than the above. For these each document could be parsed and the metric value determined separately from any other document. Therefore, in order to compare these metrics, we needed to determine how similar the values needed to be in order to decide if they were significantly related or not. These metrics were much more likely to be similar than the value produced by the word comparison algorithm, so we had to have much stricter guidelines. Our goal was to set a minimum cutoff of around 70-90% similar to determine if two songs were related for a given metric. We weren't sure exactly how to measure the percentage at first, but we came up with two ideas.

The first method was to use a static number, different for each metric, and then calculate whether or not the difference between the metrics of the two songs was less than the static number. For instance, suppose two songs have line counts of 40 and 50, and our static number is 5; if the numbers were equal, or +/- 0*5, they would be 100% similar. If instead the numbers were within +/- 1*5 of each other, they would be 90% similar. Since they are within +/- 2*5, they are 80% similar. The methodology for determining this static number was difficult to pin down at first.

The second method was to use variable numbers, based off of percentages of the higher of the two numbers. Take the above example again. In this case, 100% similarity would be if the values were within +/- 0.0*50. 90% similarity is achieved with +/- 0.1*50, and so on.

As we did not really know how to determine our static number, we gave the variable method a trial first. The results were not accurate because of two main problems. First, incrementing the multiplier in tenths provided too large an adjustment and caused too many metrics to be labeled as similar. Second, this method did not take into account the relative value of a metric compared to the whole collection. In a collection of 100 songs with 99 songs having 80 lines each, and 1 song having 100 lines, the 100 line song should not necessarily be similar to the others – in fact it is the outlier.

To combat these issues, we attempted to use the first method of static values. We decided the standard deviation of the metric for the entire collection would be a good value to start with. However, this, too, gave us too many false similarities, so we changed the number to one tenth of the standard deviation. This provided the results we were looking for. By declaring any values not within one standard deviation of each other to be irrelevant, we gained a much more focused set of results that was still large yet very accurate. Through testing with our application, we determined the ideal spot that we wanted to use for a minimum cutoff was 80%, or std*0.2.

For all of these metrics, we were simply using a boolean value to determine if each metric was similar or not. We did consider multiple degrees of similarity for each metric, but in the end decided against it. We believe that having more than one degree would cause a few problems. First, it would complicate how we decide what to show and what to not show. With one degree, we were able to easily create a function that used weighting that allowed us or the user to determine exactly which metrics would cause a relationship to display. Multiple degrees would add a second weighting factor to each metric which would likely cause additional confusion to many users. In addition, determining the math for obtaining a similarly useful display with multiple degrees would be far more complicated than what was required for a single degree. As we settled upon 80% as our minimum, there was really nowhere to go with multiple degrees. If we assume 80% was "medium" that would mean that 90% or 100% would be high and 60% or 70% would be low. Since we were using one tenth of the standard deviation – a very small value – for our percentages, a difference of 10% is not significant enough to warrant the separation. If we were to use any values above or below our 80% threshold, it would simply be to slightly alter the length of an edge in the visualization, and would not have any bearing on determining whether or not two songs are similar or not. By comparing the minor benefit of slightly more accurate edge weights to the needless complicating factors, we chose to keep metric comparisons as a single degree.

3.3.3 Storage

There were many choices to consider in this area. The two that we weighed most heavily on were an approach with tables of numerical relevance data and a more object-oriented approach where each document would encapsulate only information relevant to itself. The following subsections explain the two competing systems and the possible optimizations that could be used for each.

When we originally began implementing this aspect of our system we went with the first option, tables, as described below. In our minds it seemed preferable because it did not rely upon the ordering between the list of documents and the word frequency data table being in sync. In the object-based version if the ordering in the list of documents was somehow changed any given document might no longer point to the correct relevance value when a comparison was made. In the table-based version, even if the list of documents was reordered each document itself stores an index to its row or column.

While we acknowledged that there would be wasted cells in these tables we did not foresee just how large they would be and how little of them would actually be truly utilized. It did not take us long to realize that the tables we were using would cause considerable problems down the line. A two-dimensional array of 32 bit ints with a width and height of 5605 is 16 * 5605 * 5605 bits, or around 60 megabytes per table. The word frequency table would be average length of word * 5605 * the total number of unique words, which was in the tens of thousands. Considering we would have one table for word frequency and (at least) one other for relevance storage. This would cost potentially over a gigabyte of a user's memory.

Consider a word that only occurs in a few documents. Instead of having a large number of cells in the frequency table denoting the many documents with no occurrences of the word, the object-based alternative would ensure that each document only stores the words it has occurrences of. If a requested word is not in this stored list it turns out to be very simple to just return a value of 0 programmatically. Because we knew it would lower the memory usage of our program we opted to have each document object store the frequencies of only those words it contained, per the latter alternative. At an abstract level many aspects of the table-based alternative also apply.

Using Tables

The first option is the table approach. In this we foresaw two types of tables. In the first table, each document is a row and each word is a column. The value in each cell of the table is the number of occurrences of the word from that column in the document from that row. This value may be zero if the word has no occurrences in the document, but can never be negative.

Filling this table would be a time consuming process depending upon the number of documents being added. The parsing algorithm would have to check every word in a document being parsed against all of the known words in the table. If the word already existed, the value at the corresponding location in the table would have to be incremented. If the word did not already exist, another column would have to be added to the table. However, if the word fell within our list of stop-words it would simply be omitted. The intersections of any newly added columns with previously parsed documents are simply be filled with zeros following this system.

In the second table, documents are listed across both rows and columns. The value in each cell of the table is a relevance value calculated between the document from that cell's row and the document from that cell's column. There are a number of different algorithms that can be used to calculate relevance between documents, and therefore multiple tables may be calculated and be at hand at any time.

With every document being compared against every other document in the collection, the process of calculating relevance values turned out to be the most time-consuming processing step, regardless of what system we use. Due to the bi-directionality of the relevance relationship, only half of the cells in the table would need to be calculated and filled in this manner.

Depending on the quantity of documents in the corpus, it could take time to find a desired cell in the word or relevancy tables, especially if the desired document or word is at the far end of a row or column of the table being accessed. A loop that checks each position for a desired object might suffice for small quantities of data, but the seeking time would be improved by grouping documents in some way to narrow searching to a smaller subset of the corpus. In a proposed algorithm for this type of operation, two lists (one for documents and one for words) would contain objects in alphabetical order. Each document and word object would contain an identifier to serve as an index into these tables. For a certain document or word, the algorithm for accessing a table entry would involve searching one of these lists beginning in the proper alphabetical group, and then using the identifier as an index for the desired row or column in the table. This operation would require that documents and words be maintained in these lists sorted alphabetically.

Using Objects
The second option is one that just stores all relevant data for documents inside objects – one for each document – including relevance values to all other documents. All documents objects themselves would be stored in an ordered list. A document could be assumed to have an identifier that is simply its index in this array-like structure. Internal to each document, in addition to any attribute data it may contain, is a list of all unique parsed words from the document text and the number of occurrences of each unique parsed word in the document. This could be stored with a hash table of sorts mapping tokens, as keys, to frequencies, as values. Also as a part of this system, each document contains a list of the relevance values between itself and every other document in the collection. This is a one dimensional array of values, where the index of each value matches the index of a document in the global list. This rule must be true in order to maintain the integrity of the collection.

One simple optimization to this system of document objects would be to only store relevance values between a document itself and documents with lower indices. This is based upon the fact that the relevance between two documents is bi-directional, meaning that if document A has a relevance of 32 to document B, then document B also has a relevance of 32 to document A. So, if we are seeking the relevance of document A to document B, given that B has a higher index than A, we use the object for document B to get the value we desire, which results in the data taking up half of the space of using a 2-dimensional array. The overall idea of how relevance is stored is not very different from the table-based approach; the main difference is that the table is split up and stored in separate objects for each row.

3.4 Filter Specification

Filters are important to our application for narrowing the overall visualization to some subset of the data. They can also be used to highlight data with specifically desired attributes. Both of these functions can help a user discover certain relationships that may not be as obvious when every node is being displayed in an identical manner. A filter, as we refer to it, consists of a condition and an action. The composition of these two parts is outlined in the Conditions and Actions subsection.

When we originally began thinking about filters we tried to shy away from implementation specific details and focus more closely on what exactly we wanted them to do and how they would work at a high level of abstraction. We decided early on that filters would be separate from selection. This meant that the application of filters would never select or deselect document nodes in the visualization. Filters would be used strictly for altering the appearance of the graph with some set of global rules. A use case for filters might entail only showing songs in the rock genre and highlight songs by the Rolling Stones.

We initially came up with a rather inventive way of specifying and structuring complex filters. The system – specified and demonstrated in the Structuring Filters subsection -- involved defining and storing filter conditions in a tree-like structure where conditions at the same level of the tree would be ORed together and descendent conditions would be ANDed together with their parent conditions. In order for the actions of a filter to be applied the resulting boolean expression would need to evaluate to true.

An alternative to structuring filters that we came up with was a more simplistic approach where each filter itself was just a single compound condition paired with some number of actions. In order for these actions to be applied, each of the component conditions in the compound conditions would need to be true – as if they were ANDed together.

While the latter method had the shortcoming of not being able to specify filters with OR operations in their conditions we decided to choose this alternative because it was more basic to implement and use and because the same outcomes could be accomplished by making multiple filters – as will be demonstrated below. While the other system could produce considerably more complex results with less repetition in condition creation, we surmised that this level of usage would be beyond the capabilities of most users.

For the exact details of our chosen implementation of filters refer to the Filters section of Architecture and Implementation. Otherwise refer to the following subsections for greater detail on the overall design of the filter system.

3.4.1 Conditions and Actions

Filters are made up of conditions and actions. Conditions may easily be defined by three parts: a Field, an Operation, and a Value.

For Fields that have numerical types (such as years, relevancies, ratings, etc) the Operation can be “less than”, “equal to”, “not equal to”, “greater than”, “less than equal to”, and “greater than equal to”. For numerical representations the Value is a user specified integer or floating point number. It would be possible to specify multiple values if the Operation is either “equal to” or “not equal to” but this is by no means a necessary feature. Either way the program would need to ensure that the number entered is valid for the type of Field being used. For example, a floating point number would not be valid as a year—it would need to be an integer. Optionally, floating point numbers could be truncated to integers when necessary.

For Fields that have string representations—one or more tokens separated by spaces—the Operation can be “equal to”, “not equal to”, “contains”, or “does not contain”. Any alphanumeric value may be valid input. It would also be helpful to provide a list of known strings from the corpus to display for guidance in creating applicable filters, but this should be considered an enhancement.

Actions are defined as functions and multiple actions may be possible for a given filter. They are fairly self explanatory:

1. Apply Color sets the color of any nodes meeting a condition to a specified color.

2. Apply Shape sets the shape of any nodes meeting a condition to a specified shape and image.

3. Apply Scale sets the scale of any nodes meeting a condition to a specified size.

4. Apply Visibility sets the nodes meeting a condition to be either invisible or visible. Alternatively, this action could allow for the setting of opacity along a sliding scale for different levels of visibility.

3.4.2 Structuring Filters

Ordered Filters

In a basic implementation each individual filter is simply a list of basic conditions (forming a single compound condition) with one to four of the actions specified above. Complex filtering schemes under this system are accomplished through the order in which these individual filters get applied. This is especially important for filters with overlapping conditions and different conflicting actions. In the user interface of the application, functionality would need to be provided for ordering filters to obtain different results.

To help explain this configuration, suppose that a user wishes to color all of the songs by the artist "Simon and Garfunkel", as well as the songs by "Dream Theater" on the album "Systematic Chaos" that have either of the titles "Constant Motion" or "The Dark Eternal Night", or the songs by "Dream Theater" on the album "Images and Words" that do not have the title "Pull Me Under". This could be accomplished with the following four filters, each of which triggers the same color action:

```
- Artist = "Simon and Garfunkel"
- Artist = "Dream Theater"
        AND Album = "Systematic Chaos"
        AND Title = "Constant Motion"
- Artist = "Dream Theater"
        AND Album = "Systematic Chaos"
        AND Title = "The Dark Eternal Night"
- Artist = "Dream Theater"
        AND Album = "Images and Words"
        AND Title != "Pull Me Under"
```

Compared to the alternative that we will introduce, this system requires a greater number of filters in order to obtain similarly complex results. Having this many filters with the same actions effectively ORs the conditions of each together. An advantage to this system is that for each individual filter, compound conditions can be explained by simply stating that in order for a filter's actions to be applied, all of its sub conditions must be satisfied.

Filter Trees

In a more robust implementation a filter condition takes the form of a tree; one to four filters actions can be paired with a single condition constructed this way. A user can choose what branch to add a new condition statement to, or they can create a new branch. In this tree system all filter conditions at equal depth under a single parent are ORed together, while a filter condition and its child filter are ANDed together.

The entire operation for the same example used above is accomplished with a single filter construct under this system. The filter condition tree for this construct looks as follows:

```
- Artist = “Simon and Garfunkel”
- Artist = “Dream Theater”
        - Album = “Systematic Chaos”
                - Title = “Constant Motion”
                - Title = “The Dark Eternal Night”
        - Album = “Images and Words”
                - Title != “Pull Me Under”
```

This tree generates the following Boolean expression:

```
artist.equals(“Simon and Garfunkel”) ||
(artist.equals(“Dream Theater”) &&
        ((album.equals(“Systematic Chaos”)  &&
                (title.equals(“Constant Motion) ||
                title.equals(“The Dark Eternal Night”))) ||
        (album.equals(“Images and Words”) &&
                !title.equals(“Pull Me Under”))))
```

Under this system there would be a greater learning curve associated with filters; however, there is much less repetition in specifying complex effects such as the one in our example. If used, special care would have to be taken to design an interface to convey the power of this mechanism without confusing and overwhelming inexperienced users.

3.4.3 Applying Filters

Regardless of how filters are structured, the application of filters involves checking each node in the visualization against a condition and using a function to apply the desired effect on each node where the condition is satisfied. If two or more filters conflict and are both being applied (trying to make a node both green and red), then the nodes are updated with the effects of the last filter to be applied.

Depending on the efficiency of the chosen routine, we knew that filters would either get applied after each incremental change, after all changes were complete, or after a user-made request was made to apply them. Filtering turned out to be a rather expensive process so fewer executions of the operation were more desirable.

3.5 Fault Tolerance

Fault tolerance is the act of accepting user input that may have mistakes present in it and attempting to return results close to what the user may have intended. In the context of our application fault tolerance applies mainly to searches and filter conditions. Our application, if fault tolerant, should recognize the similarity between a search term like “beetles” and the stored artist “beatles” in the collection and bring this to the user’s attention. The idea is similar to the “Did you mean?” feature provided by Google.

Through research we discovered an interesting way of implementing fault tolerance using the Levenshtein Distance algorithm. The proposed system would calculate the number of steps needed to change the search term into what might be a near match and compare that value to a threshold. If accepted by the threshold the user would be made aware of the similarity and could then manually indicate whether to treat the two terms as equal. While this system seems sound, we were unable to implement it due to time constraints. An algorithm similar to Levenshtein Distance was used for matching lines repetitions, which will be explained in the Implementation and Architecture Fault Tolerance section.

Case insensitivity was a concern that came up during the implementation of our application and could be considered a form of fault tolerance. Because most queries in well known applications and search engines are case insensitive we decided that this was a necessary feature. Luckily this was fairly simple to accomplish by storing strings in all lowercase and converting search terms to all lowercase before performing searches.

One last issue that came up was in calculating line repetition. Many songs have repeated lines that are slightly different every time they occur. For example, in Fooling Yourself by Styx, one line in the song goes “And you’re fooling yourself if you don’t believe it,” while later in the song the line is “You’re fooling yourself if you don’t believe it.” As far as our application is concerned, this is considered repetition. When analyzing line repetition in two songs, we allow for a tolerance of a one word difference between lines. This was done by compared words not only in the same location in each line, but also in the next and previous locations. Doing this lets us check for line differences not only in lines of different length, but also lines of the same length with one word changed.

3.6 Visualization Displays

The goal of computer visualization is to assist in discovering trends or other useful features in a dataset that would not have been apparent otherwise. In order to do this, we felt it was necessary to give the user as much customization as possible, while also making it very easy to achieve adequate results with minimal effort. As an example, consider the fact that while it is useful to provide zooming capabilities in certain types of programs, it is also very frustrating if the default level of zoom is completely ineffective in most cases. It might also be equally frustrating if the zoom feature is worthless in the context it is used in.

Many of the features of visualizations changed over time. Often, these changes were made after an initial implementation attempt due to resource constraints or unforeseen infeasibility. For this reason, it can be difficult in general to draw a line between a design phase and an implementation phase for visualization development.

3.6.1 Force Directed Graph

Our reading and prior experience helped familiarized us with a variety of visualization types, their strengths, and their weaknesses. As a result of this research we decided very early on to use a graph-based visualization as the primary focus for this project. The type of graph we were looking at, often referred to as a network diagram, is typically used for showing relationships (edges) between items (nodes). In our intended application the individuals would be the documents in our collection and the relationships would be the comparisons between documents. In order to automate the task of laying out this visualization we also decided to utilize a force-directed spring graph algorithm. The length and force of a spring would be related to the relevance value between the two connected nodes.

Our original design and implementation of the force directed visualization was unsuccessful. It involved connecting every node to every other node in the graph, which resulted in an extremely large number of edges. For the 100 documents we originally tested with there were 9900 edges in total; for the 5605 documents in our full collection there would have been 31,410,420 edges! Under the force directed layout algorithm our graph took a very long time to settle, or it would not settle at all. Not only was this impractical, it also resulted in a diagram that was too cluttered to be of any use – even with all of the edges made invisible.

Realizing that we had missed a major flaw in our original design, we determined that the only way to continue using this graph-based visualization was to only show edges between documents that were significantly related. This new design would be beneficial because it would result in a more readable graph that could be generated quicker using the force directed layout. It also meant that users would only see the relationships that they were interested in, which is why they would be using our application in the first place. When we implemented this revised design we found the outcome to be far more desirable. By revising and improving our relevance algorithm we were able to further improve the visualization's appearance and usefulness.

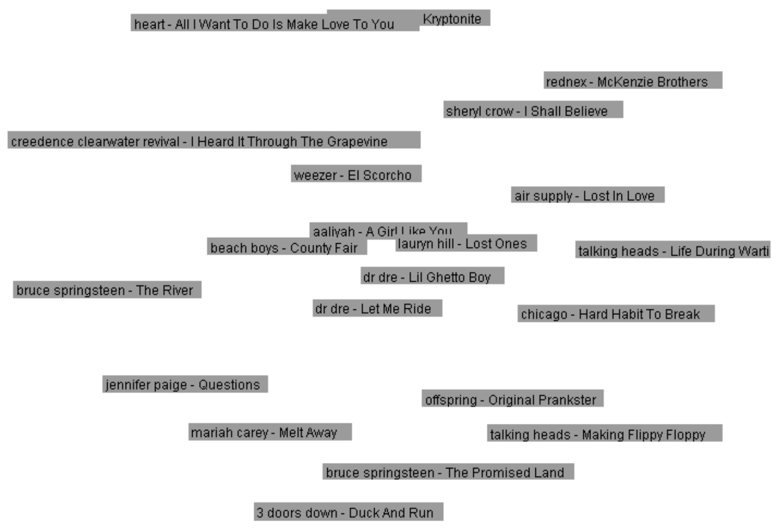

heart - All I Want To Do Is Make Love To You
Kryptonite
rednex - McKenzie Brothers
sheryl crow - I Shall Believe
creedence clearwater revival - I Heard It Through The Grapevine
weezer - El Scorcho
air supply - Lost In Love
aaliyah - A Girl Like You
beach boys - County Fair
lauryn hill - Lost Ones
talking heads - Life During Warti
dr dre - Lil Ghetto Boy
bruce springsteen - The River
dr dre - Let Me Ride
chicago - Hard Habit To Break
jennifer paige - Questions
offspring - Original Prankster
mariah carey - Melt Away
talking heads - Making Flippy Floppy
bruce springsteen - The Promised Land
3 doors down - Duck And Run

Figure 1: A screenshot of an early force-directed visualization with invisible edges and labels showing the artist and title of each song represented.

For the appearance of the force directed graph we originally intended for each node to be a small, gray circle with invisible edges connecting them. The user would have the option of turning edges on or off when desired. Our expectation was for nodes to end up close to one another as a result of the force directed layout. In practice – and as shown in Figure 1 – it was difficult to determine which nodes were intended to be related to one another with the edges turned off, so we removed this feature very early on and just made all edges visible by default.

We initially toyed with the idea of displaying the name of each document on the nodes themselves. This can also be seen in Figure 1. The result was too difficult to read when many nodes were clustered together, so we opted to retain the appearance of nodes as small circles and used hover-action tooltips to show additional details about each instead.

3.6.2 Node Centric Graph

Another type of visualization that we thought would be useful for our purposes was something along the lines of the radial graph view, which centers a specific node and arranges the remaining nodes around it in rings. Because our revised force directed graph could only display the connections above a certain relevance cutoff we became interested in an alternative view that would show more of the connections off of a given document's node. The radial graph layout we came across during our investigations[4] was the inspiration for what became the node centric graph visualization, shown in Figure 2.

We implemented our node centric graph view very much the same as we did the force directed visualization. In fact, the node centric view was also force directed and used the same relevance data for edge weights. To accomplish a radial layout we only added edges off of the central node to each other node around it. Different nodes could be centered this way by removing all existing edges and then adding new ones off of the new target to all others.

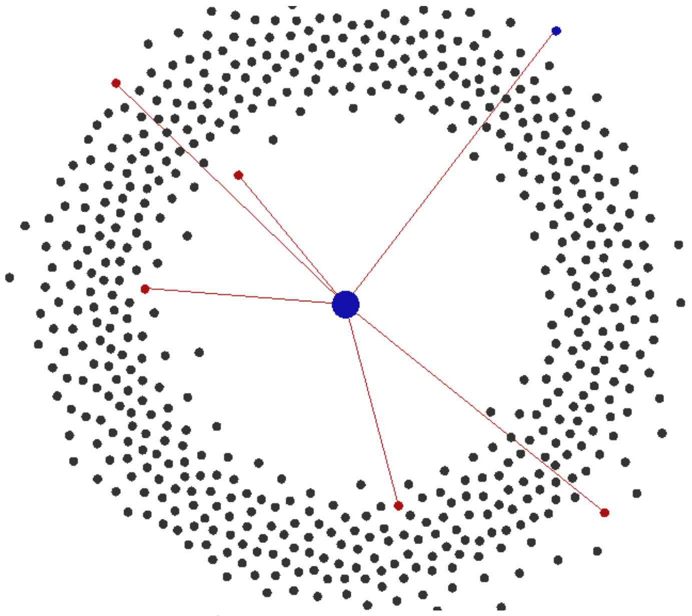


Figure 2: A screenshot of an early node-centric visualization with some nodes and their edges selected in red.

Our original node centric graph visualization displayed the entire collection of documents around the centered node. While this was fine for our initially small test collections, it became a problem when tested with larger datasets. This was resolved by only adding the top fifty related nodes and edges when centering a document. This turned out to be more than enough for this visualization to be useful without becoming too cluttered or unresponsive.

3.6.3 Other Visualization Types

In addition to the two visualization types explained above, which were generally chosen to display more collection-wide views of multiple documents, we also chose to include two other types of visualizations for users to explore documents and their relationships. These visualizations are focused more on specific documents.

The first of these document specific views was a tag cloud like we had seen and researched on Many Eyes[2], used for displaying all of the unique words in a document scaled by frequency. We wanted an interesting way of looking at a document's contents at-a-glance and knew that the tag cloud[iii] would be an effective way of accomplishing this task. The second type of document specific view we set out to create was a visualization that could aid in the comparison of two specific documents. We actually ended up prototyping two somewhat different versions of this during the implementation phase of our project. Other than color scheme and minor appearance adjustments, none of these visualizations changed too drastically over the course of the project. For additional details about these diagrams, refer to the Implementation and Architecture chapter.

3.6.4 Edge Weights

Rather than display all relationships as a static length, we opted to use edge weighting in order to better convey how related two songs were. The closer songs are, the more similar they are. We also wanted a way to specify weightings on each metric, plus a way to only show relationships that were very significant. That is, we would rather show relationships that had four or five metrics in common, since two songs having the same number of lines and nothing else are not very similar.

There are six metrics that are either true or false. By applying a weighting to each metric, we can get some variability in how much each metric counts, and which will result in a final edge weight. For example, assume all the weights are set to 2; the maximum possible edge weight would be 12. If four metrics were set to 2 and the other two set to 4, the maximum edge weight would be 16. Using this, we can set a minimum required edge weight to only display songs that meet certain requirements. For our project, we used the following weights:

- Total Lines: 3
- Lines Per Unique Line: 2
- Syllables Per Line: 4

[iii] Throughout the remainder of our report "word cloud" will refer to the tag cloud visualization type. We thought this name was more appropriate because in our case it is not used for tags.

- Words per Unique Word: 3
- Maximum Times a Line Repeats: 1
- Percent Words In Common: 6

These weights were specifically chosen after extensive testing. A default value of 3 was decided upon arbitrarily, however, as the scale of the weights did not really matter. The two line repetition metrics are related in such a way that both of them together are not necessarily any more important than any other single metric. Syllables per line proved to be a very interesting metric that often said a lot about the song structure. For instance, in White Room by Cream, every line has four syllables. Percent words in common, which is our only context-based metric and was the basis for the whole project, is weighted doubly.

These weights provide for a maximum edge weight of 19. Our goal in setting the minimum value was to ensure that the percent words in common metric was a requirement for displaying the relationship. This is why it was weighted doubly, and why our minimum edge weight requirement could not go below 14. Any value above 14 was guaranteed to have that metric. In its final form, the minimum value was set to 16 – this was the lowest value we could set it. If it was set any lower, too many songs would be displayed and the force-directed layout would crash due to a recursive function call. There are ways around this, which are described in the future works section, 6.3.

3.7 User Interface

The user interface of our application supports many types of user interactions, including alternate views for focusing attention at different levels of granularity, filters for altering the visualization appearance and visible content, and additional widgets for manipulating and displaying the underlying data of the visualization in various ways.

While it would do nothing without the “meat and bones” described in the previous sections, the outer shell of our application may be the most important aspect of the program from the user’s perspective; for this reason considerable time was spent in designing the layout of and connections between the various screens and components of the application.

This section describes the early design and layout of the graphical user interface (GUI) of our application and how each piece of this interface was initially perceived to connect together. It is important to note that all of this is a “first draft” created long before implementation and testing of the software. This is mostly just a general vision and guideline that we followed and is described here mainly for any comparisons that can be made to the final implementation. The Architecture and Implementation section will explain any major alterations to this initial design.

3.7.1 Main Visualization Window

When the application first starts the Main Visualization Window is displayed. The key purpose of the Main Window is to display the force-directed document visualization. In addition, all other functions of the application are accessible through buttons on this initial interface and options available under drop-down menus. Figure 3 shows the initial mockup of the Main Visualization Window.

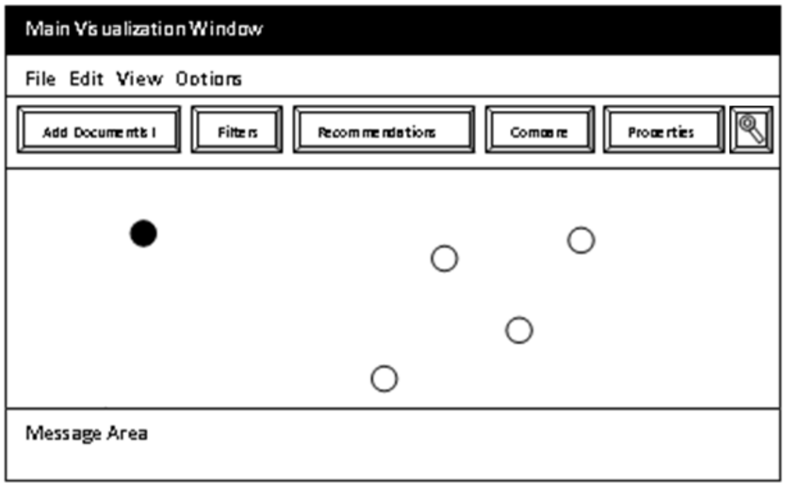


Figure 3: A Mockup of the Main Visualization Window.

The visualization is given as much space as possible. A narrow drop-down menu bar takes up around 20 pixels at the top of the Main Window. Below this is a toolbar with shortcut-buttons to common tasks; it has a maximum height of around 30 pixels. The exact dimensions of these features are not really as important as the general rule of keeping them as thin and readable as possible.

At the bottom of the Main Window there is a Message Area; this takes around 50 pixels from the height of the visualization space. It could also be possible for some of these components to be arranged on the sides of the Main Window to free up additional vertical space back to the visualization panel.

Message Area
The purpose of the Message Area is to update the user on the status of various tasks performed by the application without the need for intrusive dialog boxes. For example: when a user loads new documents into the corpus, the progress of parsing and processing the data is denoted by a stream of messages indicating the success or failure to load each. The panel is scrollable so that information does not get lost if multiple messages appear at the same time.

It may be that there are no significant status messages to send to the user (especially if individual document loading is omitted in the implementation). The message area can be used to show context-sensitive help messages to the user if this is the case.

Mouse: Selecting and Deselecting Nodes in the Visualization
To edit the selection of nodes in the visualization we use a system where clicking a node once selects it and clicking again (on a selected node) deselects it. A selection helper under the Edit Menu allows the selection of documents from a list or some custom widget as well, which can be useful if the visualization is packed with many tiny nodes.

Mouse: Zooming and Panning the Visualization
To zoom in and out on the visualization the user can either select the appropriate option in the View Dropdown Menu or right click and move in or out to zoom. To pan the visualization the user can left click on an empty location to move the view window of the visualization to center on that point.

Button: Add Document(s)
This button opens the Add/Remove Documents Window. See the appropriate subsection below for further details.

Button: Filters
This button opens the Filters List Window. See the appropriate subsection below for further details.

Button: Recommendations
This button opens the Recommendations Window. See the appropriate subsection below for further details.

Button: Compare
This button opens a Compare Documents Window for two of the documents selected. The purpose of the Compare Window is to show the text of each document with shared words highlighted in the same color. See below for further details.

Button: Properties
This button opens a Document Properties Window for each currently selected Document node. The purpose of the Properties Window is to display metadata for a selected document. See the appropriate subsection below for further details.

Button: Search
This button opens the Search Window, which allows users to focus their attention around a set of selected nodes. See the appropriate subsection below for further details.

Menu Bar
Rather than specifically naming the exact contents of each menu category it seems better to describe the rules used for assigning items to each category, along with some likely examples for each.

- The File Menu is used for actions that change the running state of the application in some way or involve saving and loading data. Some examples of these types of interaction are “Exit”, “Export Image”, “Export Text”, or “Add Documents”.

- The Edit Menu is used for actions that edit data already present in the application (without requiring additional file input), that manipulate selection, or that set certain preferences. Some examples of these types of interactions are “Clear Selection”, “Select All”, “Edit Edge Weights”, or “Edit Color Scheme”.

- The View Menu is used for actions that alter the user's view of the visualization in some way or display information without enabling any kind of editing. Some example interactions in this category are "Zoom", "Filter", "View Properties", and "View Comparison".

- The Options Menu is different from the others. Currently selected options have a check mark next to them. If the user wants to change the current visualization type or the current relevance algorithm used, they can simply pick another one. The location of the check mark is changed as a result and the visualization is updated.

3.7.2 Add/Remove Documents Window

The Add/Remove Documents Window is displayed, or would have been had we implemented it, when the user presses the Add Document(s) button or when the user selects a similar option under the File Menu on the Main Visualization Window. It displays two lists, one for documents that should be added to the active collection, or corpus, and another for documents that should be removed. A Mockup of this interface is shown in Figure 4.

Pressing the Add Document from XML File button opens a file browser through which the user can search for and load XML files. Loaded XML files are displayed in the list of documents to be added.

Pressing the Remove Document from Corpus button opens a list of the documents currently in the corpus. Selecting documents in this list adds them to the list of documents to be removed.

Pressing the Apply button makes the application perform all requested additions and then removals and then clears both lists. Closing the Add/Remove Documents Window discards any documents in either list and any pending additions and removals are not applied.

No changes to the corpus occur until the Apply button is pressed. This is essentially a "refresh" button. The user can add documents or remove documents in this interface, but until they hit Apply no changes are visible in any other parts of the application.

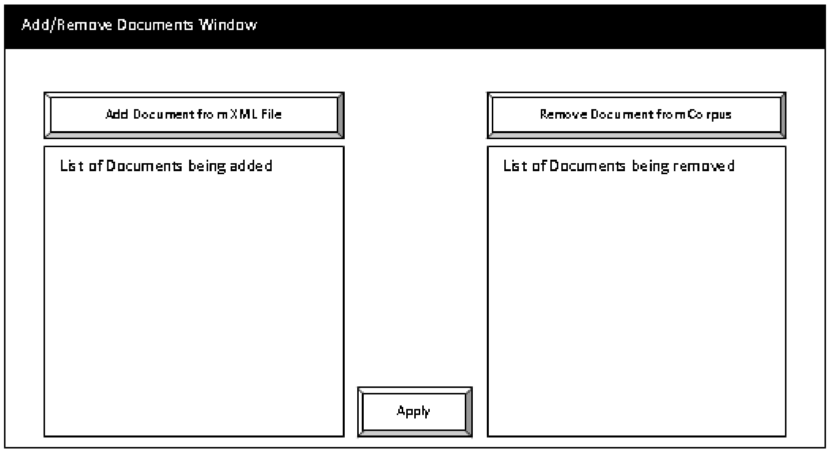
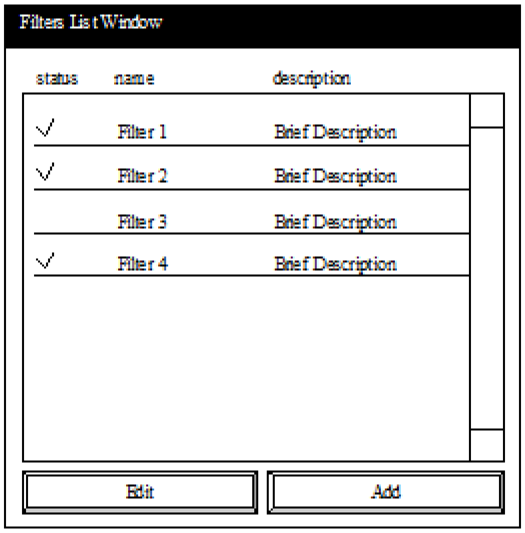

Add/Remove Documents Window
Add Document from XML File
Remove Document from Corpus
List of Documents being added
List of Documents being removed
Apply

Figure 4: A Mockup of the Add/Remove Documents Window.

3.7.3 Filters List Window

The Filters List Window is displayed when the user presses the Filters button or when the user selects an option under the View Menu on the Main Visualization Window. Figure 5 shows a Mockup of this interface element.



status	name	description
✓	Filter 1	Brief Description
✓	Filter 2	Brief Description
	Filter 3	Brief Description
✓	Filter 4	Brief Description

Figure 5: A Mockup of the Filters List Window.

The Filters List Window displays a roster of available filters with check boxes next to each to indicate whether or not it is active. Filters can be selected in this view as well. Pressing the Edit button with a filter selected opens an Edit Filter Window for that filter. Pressing the Add button opens an Edit Filter Window for a new filter.

3.7.4 Edit Filter Window

The Edit Filters Window is displayed when the user presses the Edit or Add Filter buttons in the Filter List Window or the Add Filter option under the Edit Menu on the Main Visualization Window. Figure 6 shows a Mockup of this window.

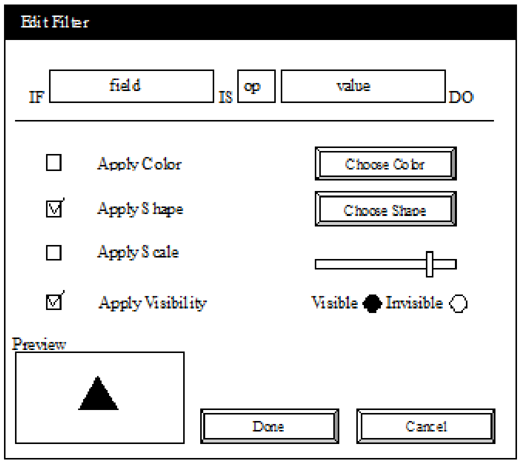


Figure 6: A Mockup of the Edit Filter Window.

The Edit Filter Window contains three fields for specifying the condition of the filter. Some of these fields are drop down menus to limit the user to only specifying one of a number of valid options. Other fields allow the user to type values such as strings, integers, or floating point numbers such as for a search query.

The window also contains four possible actions that can be applied using check boxes and customized in different ways. The Choose Color button opens a simple color picker dialog. The Choose Shape button has a similar function. Scaling is applied with a slider, and visibility with radio buttons or a slider as well for a range of opacity values.

An optional, although very useful, addition is to have a preview of what a node will look like if the filter is applied to it.

3.7.5 Document Properties Window

The Document Properties Window is displayed when the user presses the Properties button or when the user selects an option under the View Menu on the Main Visualization Window. One Document Properties Window opens for each document Node selected in the Visualization View. A Mockup of what a Properties Window looks like is shown in Figure 7.

This Window consists of two text areas. One displays the formatted metadata for the document. The other displays document text. Either or both text areas may be scrollable to allow reading of a long document's text or a long list of metadata.

The Display Document Visualization button switches the Document Properties Window to display a word cloud for the particular selected Document. This is the type of button that remains visually "pressed in" to convey the ability to toggle between these two views (on when pressed in, off when not).

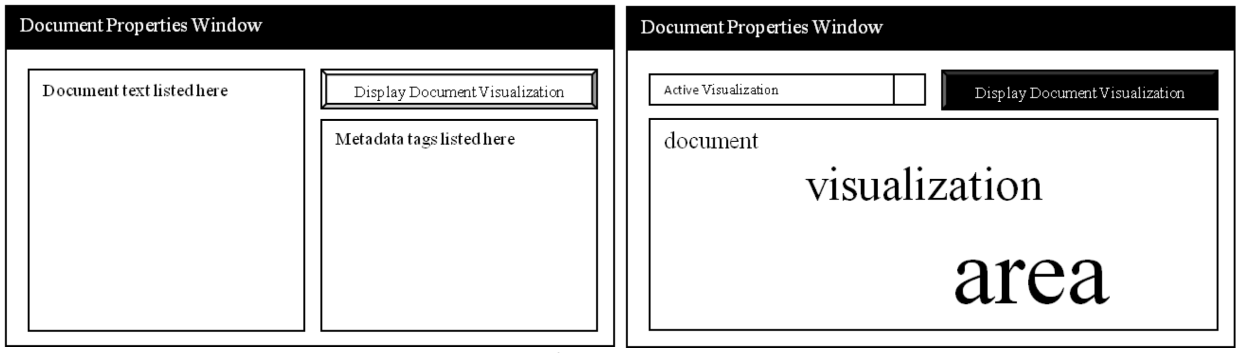

Document Properties Window
Document text listed here
Display Document Visualization
Metadata tags listed here
Document Properties Window
Active Visualization
Display Document Visualization
document
visualization
area

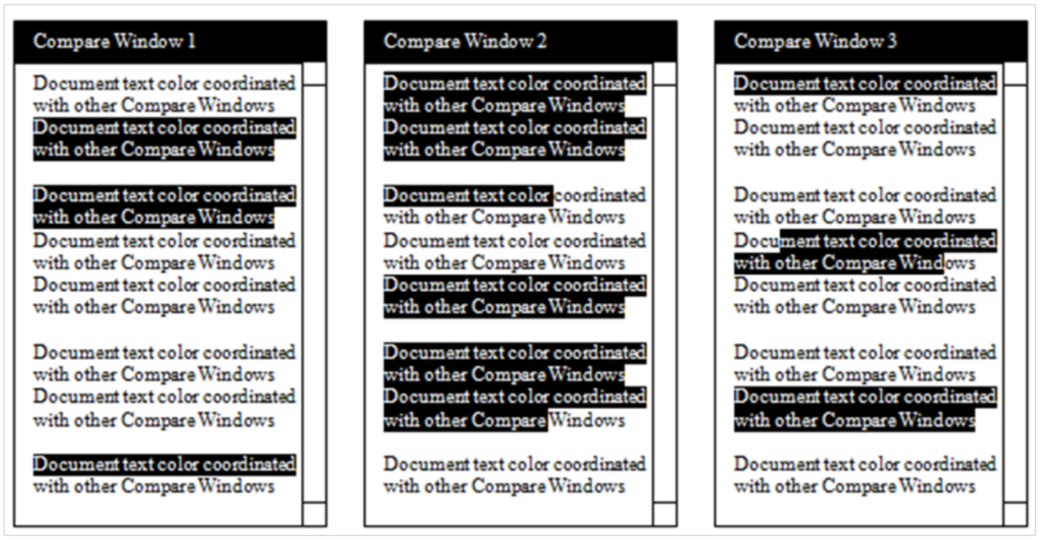
Figure 7: A Mockup of the Document Properties Window.

3.7.6 Document Comparison Window

The Compare Window is displayed when the user presses the Compare button or when the user selects the option under the View Menu on the Main Visualization Window. One Compare Window opens for each document Node selected in the Visualization View. This is shown in Figure 8.

This Window consists of a text box showing the text of a document, highlighted for comparison. A word that appears in each Compare Window is highlighted in the same color in the Compare Windows of the other documents it appears in.

Due to the constraints of screen size and resolution, only a finite number of Compare Window can be open at a time. These windows are opened side-by-side to facilitate ease of comparison between documents. If more than the allowed number of documents are selected when the comparison is requested, a dialog asks the user to choose which of the selected document Nodes to compare from a list of those selected.



Document text color coordinated with other Compare Windows
Document text color coordinated with other Compare Windows

Document text color coordinated with other Compare Windows
Document text color coordinated with other Compare Windows
Document text color coordinated with other Compare Windows

Document text color coordinated with other Compare Windows
Document text color coordinated with other Compare Windows

Document text color coordinated with other Compare Windows

Document text color coordinated with other Compare Windows
Document text color coordinated with other Compare Windows

Document text color coordinated with other Compare Windows
Document text color coordinated with other Compare Windows
Document text color coordinated with other Compare Windows

Document text color coordinated with other Compare Windows
Document text color coordinated with other Compare Windows

Document text color coordinated with other Compare Windows

Document text color coordinated with other Compare Windows
Document text color coordinated with other Compare Windows

Document text color coordinated with other Compare Windows
Document text color coordinated with other Compare Windows
Document text color coordinated with other Compare Windows

Document text color coordinated with other Compare Windows
Document text color coordinated with other Compare Windows

Document text color coordinated with other Compare Windows

Figure 8: A Mockup of three side-by-side Document Comparison Windows with highlighted text indicating shared lines between all three documents being compared.

3.7.7 Search Window

The search window allows the user to search for a document using a specific field, much like filters. This window is opened either with a button on the UI or through the View menu bar.

The window consists of several text boxes, similar to an advanced search system commonly found in a search engine. Every column for metadata has its own text box allowing the user to search based on the terms entered in each box. The application obtains search parameters from every text box that has been filled in by the user. When the search completes, all nodes that match the search are selected.

4 Architecture and Implementation

The main goals of our implementation, aside from the application working as intended, were usability and efficiency. Using our previous designs as a guide, we were able to implement most things as originally envisioned. Many components were heavily optimized yet still functioned the same way.

This section defines implementation-specific details on each component of our application. This includes in-depth discussion about data structures, libraries, algorithms, optimizations, and similar topics.

4.1 Lyrics Gathering

As mentioned in the design section, our lyrics gatherer was written in Python using BeautifulSoup for assistance. Speed was not an issue for this application, so using BeautifulSoup, which is not particularly fast, was good because it allowed very simple navigating and searching of a web page. This was also a standalone application, separate from the main visualization system, though in the future it would be possible to link them together to allow users to automatically add songs they want.

The code has four basic components. For every song listed in the input file (a list of songs and their artists), all four steps repeat. The first step is the last.fm check. If the song exists on last.fm, the statistics (listeners, plays, genres…) are extracted. BeautifulSoup makes this easy, allowing function calls such as:

```
soup = BeautifulSoup(urllib2.urlopen('www.last.fm/music/' + artist + '/_/' + song))
tempStats = soup.find("p", {"class" : "stats"}).contents[0]
```

This code downloads the last.fm page for the song, creates a BeautifulSoup object, finds the tag <p class="stats"> in the HTML document (which is the only <p> tag with that class), and extracts the contents. After some minor whitespace cleaning, we get the number of plays and unique listeners. If the song does not exist on last.fm, the rest of the steps are skipped and the next song is processed.

The second step is to download the lyrics from metrolyrics.com. This works the same way as getting the stats from last.fm, however much more cleaning is required. Additionally, some text, such as [SOLO] or (chorus), is removed to make data cleaning later on easier. However, it is possible metrolyrics.com is missing the current song, the lyrics of the song are missing, or the song's lyrics are licensed and metrolyrics.com is not allowed to display them, which leads to the third step.

If for some reason the lyrics cannot be downloaded, the backup site, azlyrics.com, is called. This goes through the same exact process as getting the lyrics from metrolyrics.com. If the lyrics cannot be found, the song is skipped and the next song is processed.

The last step is outputting the data to an XML file. This step only occurs if both the statistics and lyrics are successfully downloaded. The text is output to a file where the name is ‘artist – song.xml’ in the format previously described in the design section. After the file is written, the next song is processed until there are no songs left in the input file.

4.2 Starting the Application

When the main application first loads, a special launcher screen appears to the user. This screen allows the user to select which set of songs they want to load. From here, the application can progress two different ways.

If the chosen song collection has never been given to the application before, every song must be processed and compared to every other song before the visualization can be generated. The first step in accomplishing this is to create an object for every song in the list. Every object also requires an ID number in order to access the song later in the application – the IDs correspond to the order in which the songs are read. If an NXP file exists for the song, the NXP file is used and the already-processed values are just copied over to the new object. Since NXP files can exist without a relevance table and are independent of relevance calculations, checking for it can help speed up load time for collections without REL files. If only the XML file exists, the available information is stored in the object for processing later.

Next comes the processing of the lyrics for each song. If NXP files were used instead of XML, this step is skipped. Here is where all of the metric information for a song is extracted. First, lines are read as a whole. For the line count metric, it simply adds 1 to the total for any line that isn’t just whitespace. Line repetition is determined by storing unique lines in an array. Every time a new line is read, it is checked with all the previously stored lines in the array according to the algorithm defined in section 3.6 Fault Tolerance and added to the array if it is not already existent. The maximum number of times each line repeats is stored and used to determine number of unique lines and the maximum number of times a single line is repeated. Next, each word in the line is read individually. The number of syllables in each word is added to a running total and stored after all words have been added. The words themselves are then stored according to whether or not they are in the stop word list, and here total words and number of unique words in the song are computed. After all songs are completed, the standard deviation for each metric is calculated.

After processing, the relevance values must be calculated. This is the most time-intensive step, since the algorithm runs $n^2/2$ times where n is the total number of songs in the collection. Each of the six metrics are calculated exactly as described in section 3.2.2 Relevance Algorithms. The resulting relevance value is stored as a short, where each metric occupies a different bit using bitwise operations. Shorts were chosen instead of bytes to easily allow for the expansion of more than 8 metrics while still taking much less space than ints. Initially the relevance result is set to 0, and there are global constants for each metric. The value for these constants follow a 2^n enumeration, where n is from 0 to 5. For example, take syllables per line, the third metric. If this metric is determined to be significantly related between the two songs, the following code is called:

```
result = (short) (result | Global.METRIC3);
```

Assuming no other metrics were significantly related, this is the same as calling 0 | 4, or 0 | 100 in binary, which equals 4, or 100 in binary. Alternatively, assume the first two metrics were significantly related, and the result originally set to 3 (2^0 + 2^1). Then, the bitwise operation would be 3 | 4, or 11 | 100 in binary, which equals 7, or 111 in binary. Since the constants follow a 2^n pattern, every metric occupies a different bit. Later, when determining whether or not the two songs have that metric in common, the following code can be used:

```
if (result & Global.METRIC3) {
        ...
}
```

This format allows six values to be stored in a single one- or two-digit number, which is especially useful for keeping the file size down when creating the REL file. This value is stored in the song object whose ID is higher, such that every document stores the relevance value between it and all documents with a lower ID than it. When the relevance calculations finish, the NXP files for the input data are written as well as the REL and DLT files.

If the chosen song collection has already been processed by the application, the loading process is simple. The files have already been processed and the relevance table already calculated. So, the NXP files are read according to the list in the DLT file and stored as if they had just finished the processing step, and then the REL file is read and stored as if they had just finished the relevance calculation step. From here, the data is passed off to the visualization portion of the application, as described in the next section.

4.3 The Visualizations

This section discusses the implementation details of the various visualizations our application provides. The visualizations include the Global force directed layout, the Local node centric layout, the document specific Word Cloud, and the two types of Comparison Visualizations. All visualizations are graphs with nodes and sometimes edges, even if they do not appear so. In the architecture of our system, all visualizations extends the Prefuse Display class; this allows us to easily construct a visualization instance and add it as a component to a Java Swing container as if it was a JPanel or a JFrame.

4.3.1 Common Implementation Overview

Creating a Visualization

There are a number of steps that must be carried out before the visualization is ready to be displayed. The first step is to build the Graph object, consisting of nodes and edges. The structure of the Graph and the algorithm for building it is specific to the type of the visualization being constructed, but nodes and edges are always created using the Graph class's addNode() and addEdge() methods. In addition to defining a structure of nodes and edges, a Graph can store certain data about these objects in a table. It is important to have certain metadata in the Node tables of the Global and Local visualizations for use by Filter Predicates, as described in section 4.4. To define a field of data for a Node, the field must first be declared in the Graph via the addColumn(ColumnName, Class) method; ColumnName is a key for getting and setting data stored in that specific column; Class is the type—int, float, double, String—of data being stored in that specific column. Methods for getting and setting fields for all of Java's built in types are provided by Prefuse given a Node or Edge object.

Once the Graph is fully built up and filled with any necessary data it needs to be added to a Visualization instance. We always use the default constructor for Visualization and then add the Graph with add(GraphName, Graph). For each of our Visualizations there is only one Graph.

After the Visualization has a Graph additional Actions for layout and appearance can be specified using the putAction(ActionName, Action) method. Finally, the visualization itself gets added to the Display instance and the visualization gets painted to the screen, assuming it is placed in a visible container.

Common Visualization Interactions
Java Prefuse provides certain built-in interactions to Displays that we found very useful. These are known as ControlListeners and are added to a visualization Display via the addControlListener(ControlListener) method. All visualizations use the PanControl, the ZoomControl, and the ZoomToFitControl. The PanControl allows the user to left click on the empty background of a Display and slide the view left, right, up, and down as long the button is kept held down. Similarly, the ZoomControl allows the user to right click on the empty background of a Display and move the view in and out by moving the mouse forwards and backwards with the button held down. Finally, the ZoomToFitControl allows the user to right click and release on the empty background of a Display to perform an automatic zoom and pan that centers and fits the entire graph on the screen. Although the exact button mappings may not be the same, these are interactions that we identified in our original design. Other types of ControlListeners are utilized in some visualizations as well.

Global Color Schemes
All of the visualizations in our program utilize a global color scheme system. Global is a class filled with public static data fields and functions used for constants that need to be accessed at many places in our code. The color scheme system implemented in the Global class works as follows.

The Global class has a colors field, which is just an array of integers. At the start of the application the programmer can call the Global.setScheme(scheme) method in which scheme is the value of a constant representing the desired color scheme. Calling this method populates the Global.colors array with the values of the requested color scheme. To access a color we use the Global.colors array with a color field constant as the index.

For example, Global.setScheme(Global.COLOR_DEFAULT) populates the colors array with the color values for the default color scheme. Global.colors[Global.TEXT] gets the Text color of the current scheme out of the array. If a color scheme is changed, the changes are not applied until a redraw function is explicitly called.

Edge Weights

As described before, our application computes edge weight values that range from 0 to 19. However, a small conversion occurs to convert this calculated edge weight into what is used by the visualization. The edge weight is first multiplied by five, which we determined to be a good size that shows songs not too far apart and not too close together so that you cannot tell the difference between edge weights. Then, the value is inverted such that an edge weight of 19 is set to 0, and an edge weight of 0 is set to 95. Finally, a base value of 5 is added to all values.

In order to skip this calculation every time the visualization needed to be remade, a shortcut was implemented. At the start of the application, an array of size 64 is created, which is the maximum number of possible metric combinations, 2^6. In each element of the array, the adjusted edge weight is stored for that relevance value. For instance, the 13th element of the array contains the adjusted edge weight for any relationship that has metrics 4, 2, and 1 (8 + 4 + 1) set to true and all others false. According to the adjustment calculation, the value stored here, taking into account their respective weights 3, 4 and 3, is 5 + 5•19 - 5•10 = 50.

4.3.2 Global Force Directed Visualization

The Global Force Directed visualization was the focus of our project. It shows documents in the collection as nodes and the relations between documents as edges of differing length. We have discussed previously how this visualization has changed with time; in the beginning it displayed all of the documents in the collection and all of the relations between those documents; in the final implementation it only displays the most relevant documents and edges in the collection. This means that documents that are not significantly similar to any other documents do not get shown in the visualization.

The GlobalGraphView utilizes a force-directed layout algorithm to arrange nodes and edges on the screen. While nodes naturally repel one another, an edge keeps two nodes tethered together. The weight of an edge determines its length.

When an instance of GlobalGraphView is constructed it is passed the Corpus object containing all of the documents in the collection. Certain columns are always added to the visualization’s graph. These include a unique id, a field of the name of a document, a stroke color value for selection control, a text field for filters over a documents text, and a label field for hover-action tooltip control. A field for edge weights is also added to the graph, although only edges utilize this column.

Because our application is designed and implemented with future changes in mind, additional node attributes such as metadata cannot be hard-coded into the graph and must be added dynamically. The Corpus object that the GlobalGraphView obtains as a parameter through its constructor can be polled for the quantity, names, and types of the document attributes it contains using the getAttributes(), getAttributeName(index), and getAttributeClass(index) methods, respectively. Using these we iterate over each attribute and create a new column appropriate for each piece of metadata that will be added. While the values of these attributes are stored in the Corpus already, they must be added to the graph for our filters to have access to them.

A second graph, the purpose of which will be explained later, gets created following this same process. We will refer to this second graph as the inverse of the first.

Once the structure of the document is defined we can begin adding nodes and edges. To do this we must first identify only those nodes connected to edges that are above the global value. The algorithm we use for this process is shown below (in pseudo-code):

```
for every document i in corpus {
    for every other document j in corpus {
        if edge weight between i and j <= threshold value {
            if i is not already added {
                add node for i
                save mapping between i’s index in the corpus and i’s node id
            }
            if j is not already added {
                add node for j
                save mapping between j’s index in the corpus and j’s node id
                if j was added to inverse graph {
                    remove i from the inverse graph
                    remove i from the inverse graph mapping
                }
            }
            create an edge between node with i's id and node with j’s id
        }
    }
    if i was never added {
        add node for i to the inverse graph
        save mapping between i’s index in the corpus and i’s node id in the inverse graph
    }
}
```

To check every relevance relation in the corpus we use two for-loops, one nested inside the other. The first for-loop traverses every document in the collection. The second for-loop traverses every other document in the collection up to the current document of the first for loop. This cuts the number of comparisons in half, which can be done due to the fact that the relevance between A and B is the same as the relevance between B and A.

The Corpus method getRelevanceBetween(docIndex1, docIndex2) gets the relevance value, which is converted to an edge weight, and then compared to the global minimum display cutoff. The highest relevance values translate to the lowest edge weight values for the shortest edges in the visualization. If the weight between two documents is less than the threshold, both of the nodes are added and an edge gets created between them.

Following this reasoning so far, documents could be added twice. For this reason we store which documents have already been added by saving an entry in a mapping Hashtable. Basically, if a document has a mapping, a node already exists and a new one need not be added. The mappings themselves pair a corpus document index to a visualization document node id and are useful for selecting documents between the corpus and the visualization.

At the end of the outer for-loop is a condition that checks whether the current document was ever added at any point to the visualization based on the relevance values that were examined. If the document never was added to the graph, a node is created in the inverse graph mentioned earlier. The inverse graph exists for search functionality, which applies predicates to graphs in order to make selections. By searching both the visible graph and invisible inverse graph we can select and view any document in the collection, even if it does not show in the visualization.

The pseudo-code does not show it, but each time a new node is added for a document, all of the metadata from the document also gets entered into columns in the node table. At the completion of this algorithm, both graphs should be fully populated. The first graph can then be added to a new Visualization instance (which gets added to the display). The second is only saved to a local field; it never gets displayed.

Setting up the visualization's force directed layout is a simple matter of instantiating a new object of the layout's class and adding it in an ActionList to the visualization. Calling visualization.run("layout") executes the layout algorithm.

GlobalGraphView uses a number of control listeners in addition to the typical Zoom-, ZoomToFit-, and Pan- Controls that get added to all of our visualizations. The ToolTipControl is used to display hover-action tooltips for node and edge information. A custom SelectionControl is used to handle selection and deselection of nodes and edges in the visualization. A DragControl is used to grab and drag nodes in the visualization, which can be useful if many nodes are clustered on top of one another. Finally, a ControlAdapter for mouse events is added to trigger context sensitive help text and recognize right clicks on nodes for centering the Local View.

4.3.3 Local Node Centric Visualization

The Node Centric Visualization displays the local "neighborhood" of a specific document in the collection. While a node that displays in the Global Force Directed Visualization is only connected to nodes that are within a certain threshold of closeness, the Local Node Centric Visualization displays a number of the closest documents to a specific document. The node of the document being examined is in the center and the radial distance outward to another document's node is indicative of how similar it is to the center. The number of outer nodes that get added is defined by a constant in the NodeCentricGraphView class. Although this number could just as easily be dynamic and user-defined, we found a default value to be more than sufficient for most cases.

The steps of graph creation for NodeCentricGraphView are considerably simpler than those of the GlobalGraphView. First we obtain all of the documents from the Corpus that gets passed through the constructor. Next, we add each document that is not the intended center to a list of sortable objects along with a relevance value. Using the Collections.sort(ArrayList) method automatically performs an efficient sort on the list. Following this, we iterate over a set number of the top nodes from the list, adding a node and an edge to the center for each. Because the NodeCentricGraphView is filterable alongside the GLobalGraphView, each node is also populated with attribute information and metadata from the document it represents.

Adding the graph to the new Visualization instance and running the force-directed layout automatically arranges all nodes around the center node based on their edge weights. To prevent the graph from drifting off the screen, the center node is set to have a static position as well.

All of the controls listeners utilized in GlobalGraphView are also used in this Visualization. Rather than switching visualizations, the right click ControlAdapter generates a new NodeCentricGraphView for the document whose node was clicked on.

Some experimentation with the NodeCentricGraph view eventually brought the constant number of radial nodes down to a value of fifty. When we were working with earlier smaller collections we would display all nodes in this Visualization, but with the full 5605 of our full corpus we began encountering exceptions due to very deep recursive calls in the force directed layout function. We realized around this point that the any document past the first twenty most related are not very related at all and eventually settled upon our final value as a compromise between the interesting appearance of many nodes and the useful function of displaying the local “neighborhood” of a particular document.

4.3.4 Document Word Cloud Visualization

As we identified in our original design, the Word Cloud visualization is a diagram that shows all of the unique words from the text of a document, scaling those words to a size relative to their frequency in the document and arranging them so that they are all visible.

Our Word Cloud visualization, called DocuCloud, utilizes the same force-directed layout used by the Global and Local visualizations. The difference in this usage of the algorithm is that there are no edges in the Word Cloud’s graph. Without any connections, all of the nodes repel one another, causing the words being displayed to spread apart. Like the previous two visualizations, this can result in an “exploding” animation as the entire diagram expands outwards from a single center point.

When an instance of a DocuCloud is created, the constructor gets passed the document object the Word Cloud will represent. The getTokenSet() function implemented by the Doc class returns the Set of all unique words in that document.

To generate the graph we iterate over all of the words in the TokenSet, creating a new Node for each. In each node we store the token string and its in-document frequency. The frequency value determines the scale of the word. To apply this scale we use the setSize(size) function. The scale value passed as an argument to this method is always ten times the frequency value.

As each Node is added to the graph, its frequency is checked against global minimum and global maximum field values. If the frequency of the just-added word is a new minimum or maximum it replaces the current one. These values are used to calculate the brightness value of each word in the gradient color scheme (explained below).

Once the above steps are completed the Word Cloud is generated by adding the Graph to the Visualization instance for the DocuCloud and by then adding that Visualization to the Display. As mentioned earlier, the same ForceDirectedLayout is utilized to perform the arrangement. In order for the words to be visible a LabelRenderer is added; this class takes the contents of the "token" column for each node and displays that string instead of the default circular node. The end result looks similar to what is seen in Figure 9.

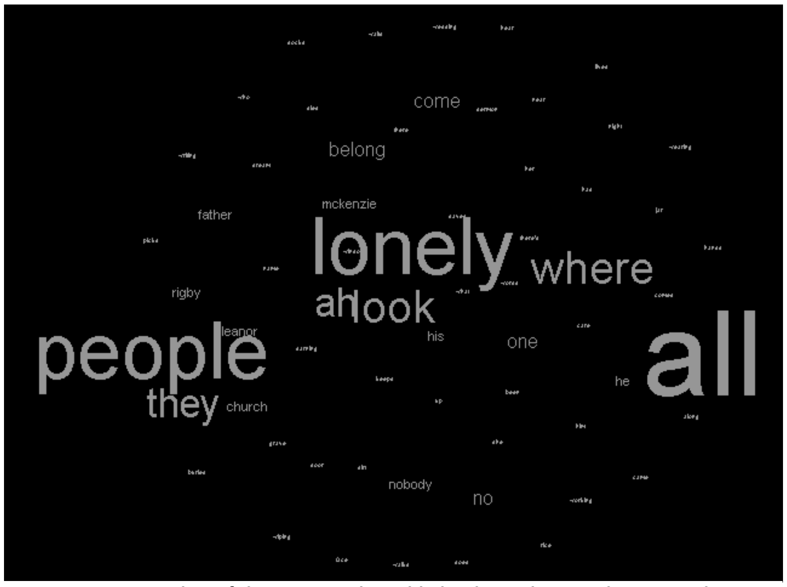


Figure 9: A screenshot of the DocuCoud World Cloud visualization; larger words occur more frequently in the song being visualized.

Instead of displaying all words in the document in the default gray shown in Figure 9 we implemented two alternative color schemes. The first of these is randomized and paints all of the words different colors. A private method in the DocuCloud class is used to generate these random color values using the HSBA (hue, saturation, brightness, and alpha) color model. In the randomColor() function only the hue is randomized. Saturation, brightness, and alpha are all fixed at 0.9, 0.85, and 0.75 respectively. The saturation and brightness were chosen to make the colors show well against the black background of our default global color scheme. The slightly transparent alpha value allows words to be seen through each other if they end up on top of one another. The outcome of the random color scheme is shown in Figure 10.

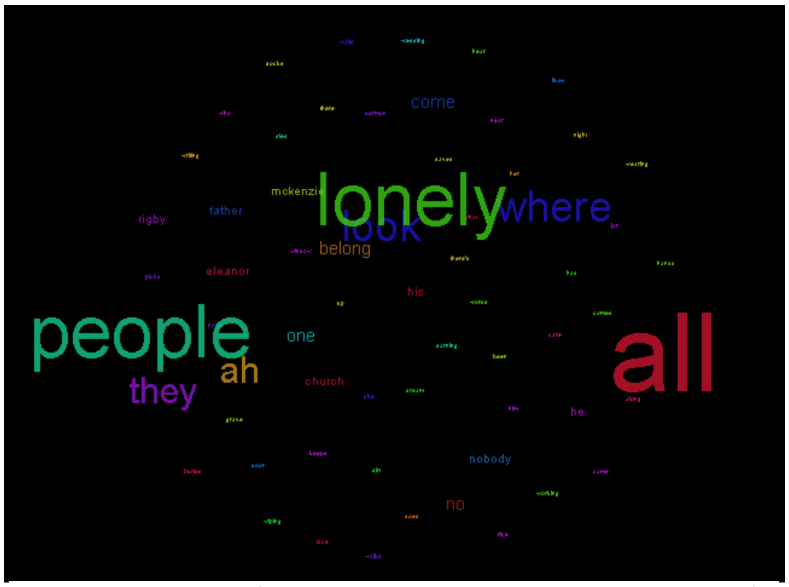


Figure 10: A screenshot of the DocuCoud with random color scheme; the colors of words have no purpose other than aesthetics.

While the random color scheme has some appeal, it is more of a novelty. Our alternative color scheme provides redundancy for the word scaling. The brighter the word, the larger the word, the more frequently it occurs in the document being visualized. Similar to the randomized scheme, a private function is used to generate the color value for a word based on two parameters: brightness and hue. The brightness argument is the ratio of the current word frequency over the maximum word frequency gathered during Graph creation. The hue argument is specified by the user through the user interface. The saturation and alpha values are fixed at 1.0 and 0.75, respectively. The result of this color scheme makes frequent words brightest and infrequent words darkest. The result can be seen in Figure 11; notice how the hue can be altered.

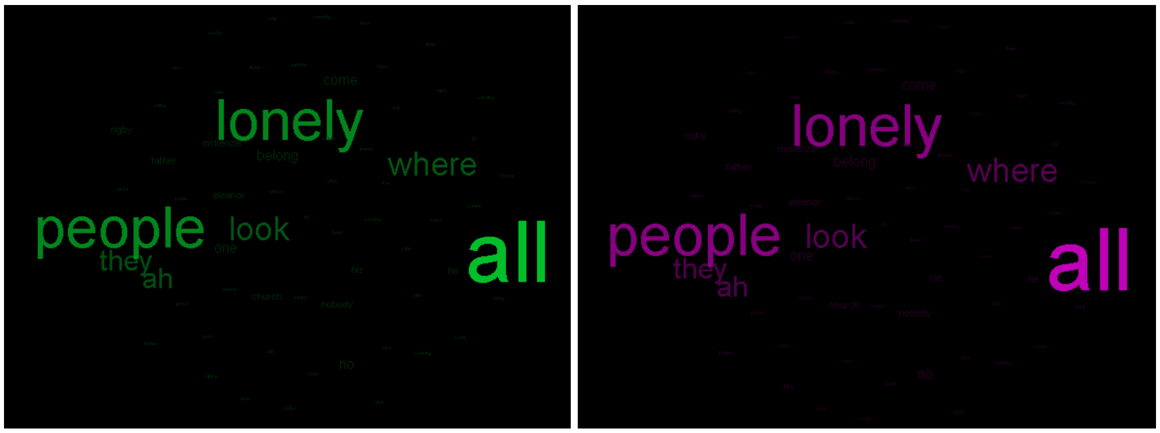


Figure 11: Two screenshots of the DocuCoud with the gradient brightness color scheme, where frequent words – in addition to being drawn larger – are colored brighter in the chosen hue.

We feel that the Word Cloud visualization provides a useful at-a-glance indication of the most frequent terms in a document. At one point we felt that this visualization would also be an adequate way of comparing two documents side by side but this turned out not to be true.

4.3.5 Document Comparison Visualization (Connections)

In order to address the shortcoming of the Word Cloud visualization as a tool for comparing two documents we set about implementing a visualization type that would perform this task. This type of visualization was not in our original design and was really implemented based on an idea.

The DocuCompare visualization shows two stacks of words. Each stack contains the unique words from one of two documents being compared. The words in a stack are scaled and ordered based on their frequency in the document they originate from, placing the most frequent words at the top and scaling them largest. Any words that appear in both documents are connected with a line.

Each word in this visualization is a Node. Each line in this visualization is an Edge connecting two Nodes. This is a static visualization so no layout algorithm is run and all component Nodes are placed programmatically.

To construct the Graph for this visualization we add a Node for each of the words from one document in descending order of frequency, and then do the same for the other document. When Nodes are being added for the second document a search is run against the Nodes added for the first document to identify and add Edges for words that appear in both. Resizing each Node by frequency and displaying labels to display the words on the visualization are both accomplished in the same manner as in the Word Cloud visualization.

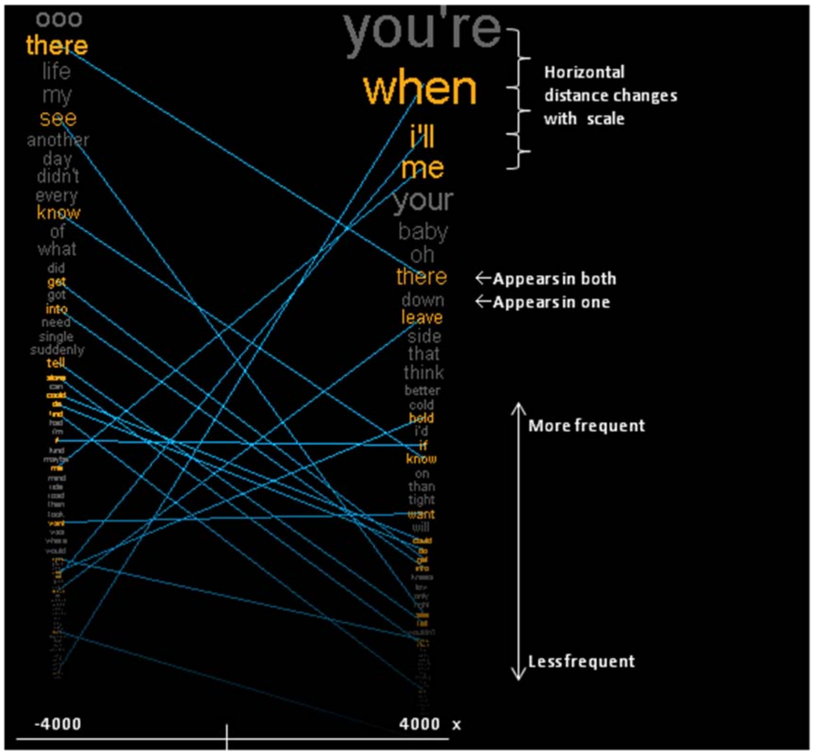


Figure 12: A diagram showing the DocuCompare visualization with a markup in white indicating higher frequency words and words that appear in both documents being compared.

After the Graph is added to the visualization, words can be arranged into two stacks by getting the VisualItem for each Node and using the setX(double x) and setY(double y) methods to perform manual positioning. The x position is fixed for each stack; words in the left are always placed at negative 4000 on the x axis and words in the right stack are always placed at positive 4000. The y position of a given Node is always offset from the y position of the previously placed Node by a value 100 times the frequency of the previous term. This relates vertical spacing of Nodes to their scaling values so that words of different sizes never overlap. This process, and an example of this visualization, is shown in Figure 12.

This visualization is effective for immediately seeing what words appear across both documents that are being compared. It also gives an indication of how frequently a word is in one document compared to the other. For example, "see" appears in both the left and right documents shown in Figure 12, but the word is much more frequent in the left document than it is in the right.

4.3.6 Document Comparison Visualization (Bars)

Later in the implementation of this project we added an alternative to the original comparison visualization, which could become difficult to read for documents of very different lengths.

This newer comparison visualization, DocuCompare2, lists all of the words from both documents being compared in a single stack, sorted alphabetically. Another change to the arrangement and appearance of these Nodes is that they are not scaled by frequency so they can be evenly spaced. As before, the words themselves are displayed as labels on Nodes using the LabelRenderer.

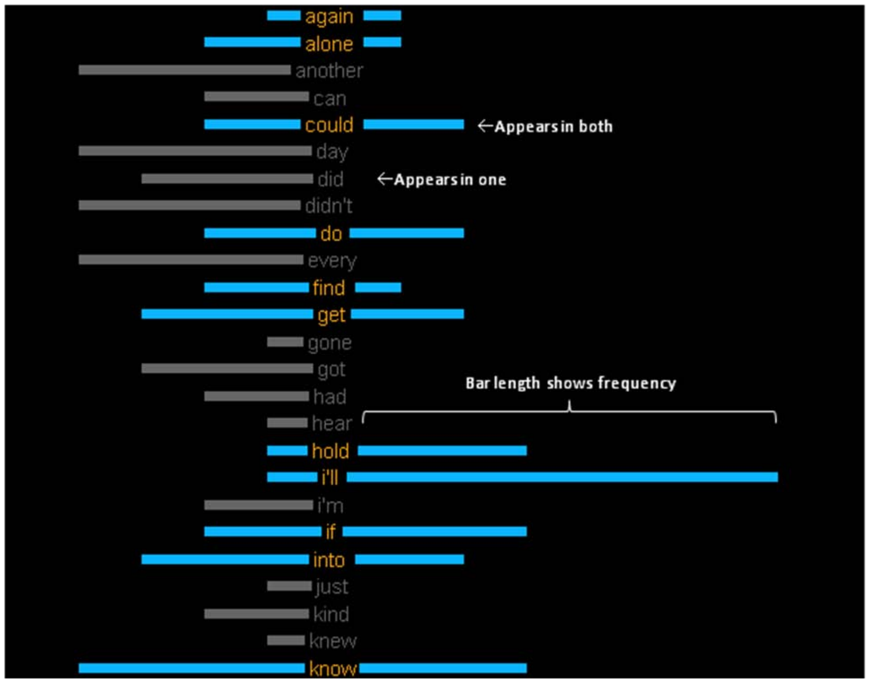


Figure 13: A diagram showing the DocuCompare2 visualization with a markup indicating higher frequency words and words that appear in both documents being compared.

To indicate frequency and whether or not a word is shared between both documents we use horizontal bars that originate from each word in the stack. The length of a bar is indicative of the frequency for the word it extrudes from. The side of the stack that a bar extrudes from tells which document the word occurs in. If a word has bars on both ends that means it appears in both documents. The resulting visualization is shown in Figure 13.

Bars are rendered as Edges that connect the Node of the word to an invisible Node placed programmatically at the same y Position but with an x offset that is 30 times the frequency value. To make the bars clearer, the Edges are given a thicker stroke width than the default by getting the VisualItem for the Edge from the visualization and using the setStroke(stroke) method.

4.4 Filters

The implementation of filters in our text visualization application has been built largely around the existing Prefuse Expression Language, Predicated Actions, and ActionLists. This infrastructure, provided by the Prefuse toolkit, allows lists of Actions (with conditions, or Predicates) to be attached to a given Prefuse Visualization under a single ActionList. Each ActionList added to the Visualization gets associated with a String token that can be used to run the ActionList or remove it at any time.

Given the underlying list organization of Prefuse Actions, it seemed appropriate to use this same structure for building filters. For this reason, we implemented the “simpler” alternative for filters described in the Design and Preparation section. In this system every filter would be a member of a list, with the ordering of the list denoting the order in which filters get applied.

Filters pair a condition (either singular or compound) and one to four of the previously described action functions, which include Apply Color, Apply Shape, Apply Scale, and Apply Visibility. Our application uses a Filter class that extends the Prefuse ActionList, acting as a wrapper that provides fields for the name and description of a filter (which our user interface displays). Because Filter extends ActionList, it inherits the add(Action) method and can be added to a visualization as described above.

4.4.1 Prefuse Expression Language

The Prefuse Expression Language is a database-like query language provided by the Prefuse toolkit. It can be used to obtain nodes and edges that match a certain criteria. A good reference of functions provided by the Prefuse Expression Language can be found in the Prefuse API Reference entry for the ExpressionParser class. What follows are the basic statements that we have used for implementing the conditions in our filter system.

For all field types:

Field equals value	“[field name] = [field value]”
Field does not equal value	“NOT ([field name] = [field value])”

For numerical field types:

Field greater than value	“[field name] > [field value]”
Field greater than or equal to value	“[field name] >= [field value]”
Field less than value	“[field name] < [field value]”
Field less than equal to value	“[field name] <= [field value]”

For string field types:

Field contains value	“NOT ([field name = REPLACE([field name], [field value], ‘_’))”
Field does not contain value	“([field name = REPLACE([field name], [field value], ‘_’)”

For queries that deal with numerical types the field value should be either an integer or a floating point number. For queries that deal with string types the field value is any alpha-numeric string enclosed in single quotes. Field names are not enclosed in quotes.

More complex queries can be obtained by concatenating any of the queries above with an “AND” statement between them. For example, “title = ‘let it be’ AND plays = 8845” returns only those nodes with the title “let it be” and 8845 plays. While the Prefuse Expression language also provides an “OR” operator, we do not provide compound filters with it in our filter system. The same result can be accomplished by creating separate filters with identical Actions.

4.4.2 Predicates

Unparsed Prefuse Expression Language queries are Java Strings. To use these queries as conditions for applying actions, the query must be converted into a Predicate. We accomplish this by using the Prefuse ExpressionParser class, which provides a static function parse(String) and a return value of the type Expression. This Expression can be cast as a Predicate and passed to any Action type constructor as a condition. We will refer to Actions that get constructed with Predicates as Predicated Actions.

Predicates can also be used in another way. Prefuse, like a database, stores information in tables; depending on the table, each row can be either a node or an edge. If we have access to either table we can use the get(Predicate) method to obtain an iterator of all of the row numbers that match the query. This is not the way the filter system applies filter actions, but other parts of the system (such as node selection) utilize this functionality.

4.4.3 Actions

To implement the action functions we described above, we utilized some of the Action subclasses provided by Prefuse. The aptly named ColorAction applies a color to any applicable nodes, allowing the programmer to specify what part of the node (stroke or fill) to affect. We decided to have filters only affect the fill color of a node, leaving the stroke color as an indicator of node and edge selection.

To change the shape of a node we used a ShapeAction, which takes as input one of the built in Prefuse shape constants (cross, diamond, star, hexagon, rectangle, and triangles for each of the four compass directions).

A similar SizeAction class exists for changing the scale of applicable nodes, given a numerical value.

While a VisibilityAction does exist for toggling visibility of a node on and off, Prefuse does not have a built in Action for applying opacity to a node. Because this functionality is far more robust than just switching visibility on and off, we decided to write our own Action subclass, extending the existing ColorAction to create our own custom VisAction. The VisAction stores an opacity value, or alpha, when it gets created. Any applicable items in the visualization will get passed to the process(VisualItem) method, which we overwrote to fetch the existing color of the node, change the alpha component of that color's numerical representation, and then apply the changed color to the node fill color.

Related to this, we also established a scheme for reflecting node visibility to adjacent edges. For each edge in the visualization we obtain the fill color of the two connected nodes. Next we extract the alpha value of the two colors and apply the lesser of the two alpha values to the edge being operated on. The result, shown in Figure 14, ensures that only edges connecting two visible nodes can be seen.

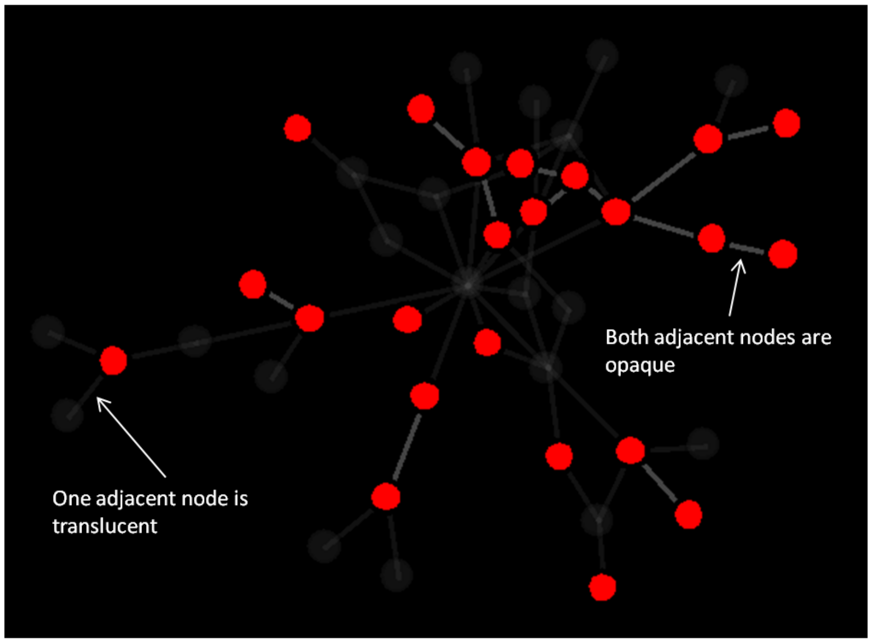

Both adjacent nodes are opaque
One adjacent node is translucent

Figure 14: A screenshot of a graph with a markup showing how edges take on the lesser opacity value of the two nodes they connect.

4.4.4 Filter Creation

As far as the user is concerned, filter creation is really more of a matter related to the user interface. This aspect of filter creation is therefore explained in the User Interface subsection. What the user may not know is how their input on the Filter Creation Screen translates into a complete Filter object in our system.

Filter creation is handled by a class called FilterFactory. In this class the getFilter() method grabs the filter specifications from the GUI and constructs a Filter object according to the following procedure.

First, raw unparsed query Strings are constructed. Before this Queries are broken down into zero or more QueryBlocks. A QueryBlock is representative of one of the basic statements described earlier. Creating query Strings from these blocks is a simple matter of inserting the user specified data in the place of bracketed text for “[field name]” and “[field value]”. The text for each query block is appended to the next using “AND” statements. If there are no QueryBlocks available, the Query is just “TRUE” and will match all of the nodes in the graph.

Once a single query String is built up from all of the existing QueryBlocks, it is passed to the ExpressionParser’s parse(String) method. The resulting Predicate is saved in a variable local to the getFilter() function.

The next step of filter creation is to construct the actions that will get applied to nodes matching the condition. Zero to four of the Action subclasses previously described can be used, but not duplicates of the same Action type. Each kind of Action is constructed by drawing data for options like color or opacity from the current state of the GUI. The locally saved Predicate is also added to each Action object, either through its constructor or a setPredicate(Predicate) function. When an action is properly constructed it gets added to the Filter being produced via the add(Action) function inherited from ActionList. Once all actions are built and added the Filter object is ready and can be returned. To summarize:

1. Build the raw query String
2. Get a Predicate by parsing query String with ExpressionParser
3. Create Actions and add the Predicate to each
4. Add each Action to the Filter object
5. Return the completed Filter object

4.4.5 Filter Management

As in filter creation, filter management is largely a user interface issue and will be discussed in that context in the appropriate section. It is worth noting here that the orderable filter list seen by the user is an accurate representation of how Filter objects are stored in the inner-workings of the application: using an ordered list of Filter objects with indices. The order of filters as shown to the user is always an accurate reflection of the order of filters in the internal data structure.

Order matters with filters because they can have overlapping conditions. For example, consider two filters, one of which colors songs containing the word “love” red and another that colors songs by the artist “Dave Matthews” yellow. If the “love” filter gets applied first, any songs containing love that also are by Dave Matthews will be yellow. If the “Dave Matthews” filter gets applied first, any such songs will be red. The order of filters can be important for obtaining the proper final result. As a rule, the last filter down the list will overwrite any previous filter effects.

4.4.6 Filter Application

In Prefuse, ActionLists and individual Actions can get applied to a visualization with the add(Action-Name, Action) method. Because ActionList inherits from the base Action class, ActionLists can also be added along with our own custom Filter objects.

When the user triggers the application of filters to the visualization, all of the currently selected filters are gathered up in the filter list and added, in their current ArrayList order, to a single ActionList. That ActionList then gets added to the visualization via the add(Action-Name, Action) method and is run by calling run(Action-Name).

4.5 User Interface

The user interface for the application we developed for this project is based largely on the designs and interactions described in the Design and Preparation section. There are, however, a number of key differences in the GUI, including the omission, alteration, and addition of certain features. The purpose of this section is to reveal the layout and design of the actual application GUI and to justify changes to the original design guidelines as appropriate.

4.5.1 Main Visualization Window

The Main Visualization Window is the first screen that the user will see after launching our application and choosing a collection of documents to visualize. It is still primarily used to display the force directed document visualization that our project revolves around. It also provides a number of options and tools for aiding in interacting with the visualization. Figure 15 shows a screenshot of what this screen looks like in our final application.

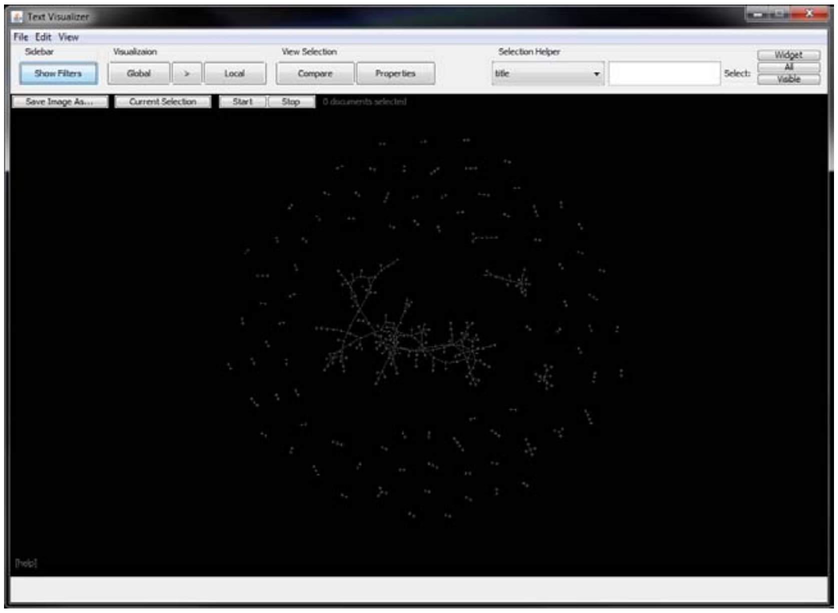


Figure 15: A screenshot of the finished Main Visualization Window from the running application.

The general layout of this screen is basically what we originally envisioned it to be. It consists most prominently of the current visualization's Display panel above which are various controls and menu options and below which is a thin area for text messages.

The size of the Main Visualization Window is actually somewhat larger than we had originally planned. Early prototypes of the application used a window size of 800 by 600 pixels, and while this space was adequate for viewing the visualization, the number of buttons we would end up positioning in the strip above the Display panel caused us to increase the width. For this reason the dimensions of the Main Window ended up being about 1030 by 750 pixels (including the window border decoration of the shell). The menu bar at the top of the screen takes up 20 pixels of height and the button strip below this has a height of 60 pixels. This leaves 600 pixels for the visualization and 32 for the message area at the base of the frame.

The Main Window (and any other window for that matter) is simply a class that extends JFrame. In the architecture of our application each of our visualization classes extends the Display class of Prefuse, which itself is a JComponent. This means that an instance of a visualization can be added, resized, and positioned within a JFrame container using just the standard add(component), setSize(width, height), setLocation(x, y), and setBounds(x, y, width height) methods all components and containers seem to share. Other than the Prefuse visualizations Displays, the GUI of the application has been made up entirely of Swing standard components.

Menu Bar
The menu bar at the top of the Main Window is a custom JMenuBar. The structuring of the menu items into each drop-down category was guided by the rules identified in our original design. We also included separators to group similar or related tasks within each category. Figure 16 shows the menu items included under File, Edit, and View. The Options category was removed during development and integrated with View.

File	**Exit:**	Closes the application
Edit	**Selection Widget:**	Opens a tool for selecting specific documents
	Echo Selection:	Duplicates current selection to other visualization
	Select All:	Selects all nodes in the current visualization
	Clear Selection:	Clears the current selection
	Edge Weights:	Opens a tool for editing edge weight properties
View	**Filters:**	Toggles whether the Filter panel is active
	Global Graph:	Checked if the Global visualization is active
	Local Graph:	Checked if the Local visualization is active

Figure 16: A table containing options included under each menu bar category (File, Edit, View) in the final application.

The number of items included in these menus is somewhat small because we tried to shy away from hiding features under menus and use easily accessible buttons whenever possible.

Button Strip and Selection Helper

The buttons above the visualization display are, for the most part, the original set that we planned on including with some additions. The key differences are the removal of the Add Document button, the addition of the arrow button (a shortcut to the echo selection function) and visualization buttons, and the incorporation of the search GUI (renamed selection helper) in the upper right of the Main Window. Figure 17 shows these components.

The omission of the Add Document button is based simply on the fact that in our current implementation, document addition and removal is not available. The adding and removing of documents was not one of the most important features and there was more to it than we had originally anticipated. Since relevance values were calculated using the standard deviation of a metric for the entire collection, adding or removing documents would alter that value and additional design would have had to go into making a system that would display accurate relevance values upon adding or removing a node.

The Echo Selection button was an addition later in the implementation of the application. It takes all of the documents that are currently selected in the active visualization and matches their selection in the other. The two visualizations that this operation can be performed between are the Global and Local displays. To convey this, the button is positioned between the two for switching visualizations and the arrow always points away from the currently active visualization. Similar to Echo Selection, the inclusion of the Global and Local visualization buttons was made later in development of the application. We found ourselves switching active visualizations so often that it seemed worthwhile for there to be a single-click alternative to the items present under the View menu.

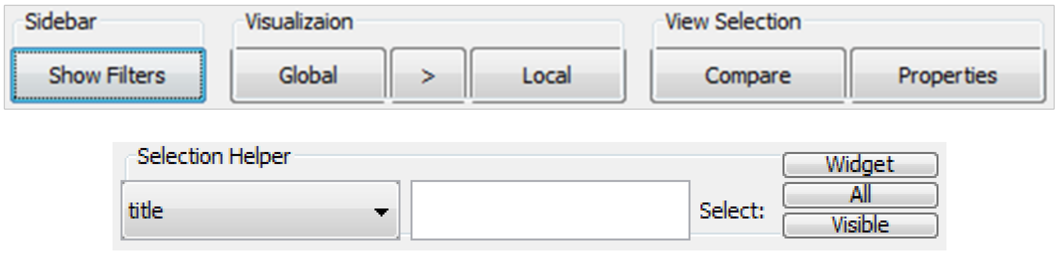


Figure 17: A close-up of the buttons included in the strip and of the Selection Helper, which can be found above the visualization on the Main Window.

The integrated Selection Helper was an earlier alteration to the design. Rather than having a separate widget for performing searches for documents (as we had originally planned in our design) we decided it would be better to include this functionality in a more easy-to-access location that would prevent having to switch windows. We considered the difference between the "ctrl-f" search features of web browsers when making this decision. Microsoft Internet Explorer (7 and earlier) opens a separate dialog for searching pages. Browsers such as Google Chrome and Mozilla Firefox have a search box attached to the browser window itself. We found the latter method to be more desirable than the first.

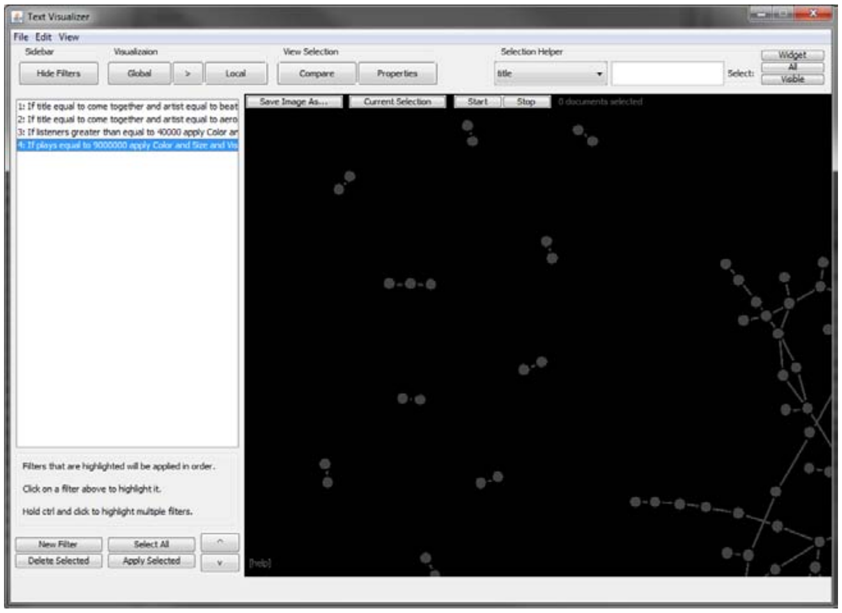
The evolution of Search to Selection Helper followed naturally from this streamlining. Predicting that users would typically be using Search to find documents that they would immediately want to select, we took out the middle step. The Selection Helper is essentially a Search tool that selects documents matching the provided “attribute-equals-value” criteria. The uses for the Select All, Select Visible, and Widget buttons are explained in a later subsection detailing Enhanced Selection Tools.

Buttons in the Visualization Display
Because our visualization classes all inherit the functionality of the Swing JComponent object, we can place Swing GUI components such as buttons and labels directly alongside the visualization being drawn. While it was not our desire to clutter the view of the visualization, some functions that are strongly related to a given visualization have been incorporated into the Display panels using reduced size JButtons. These tasks include exporting the contents of the display as an image, viewing a pop-out list of the current selection, and controls for visualization animation.

Message Area: A Help Bar
We originally intended to use space below the visualization as a console for status messages on document loading and parsing. With the omission of these features the Message Area went unused until we decided to use it as a context-sensitive Help Bar. This implementation of this feature is covered in a later subsection.

4.5.2 Filters List

The Filters List Sidebar shows all of the Filters that have been created by the user and can be used to create additional filters, manage existing filters, and apply a selection of filters to the currently active visualization. Figure 18 shows a screenshot of the Filters Sidebar on the Main Window.



1: If title equal to come together and artist equal to beat
2: If title equal to come together and artist equal to aero
3: If listeners greater than equal to 40000 apply Color ar
4: If plays equal to 9000000 apply Color and Size and Vis

0 documents selected

Filters that are highlighted will be applied in order.

Click on a filter above to highlight it.

Hold ctrl and click to highlight multiple filters.

Figure 18: A screenshot of the Filters Sidebar docked to the side of the Main Visualization Window.

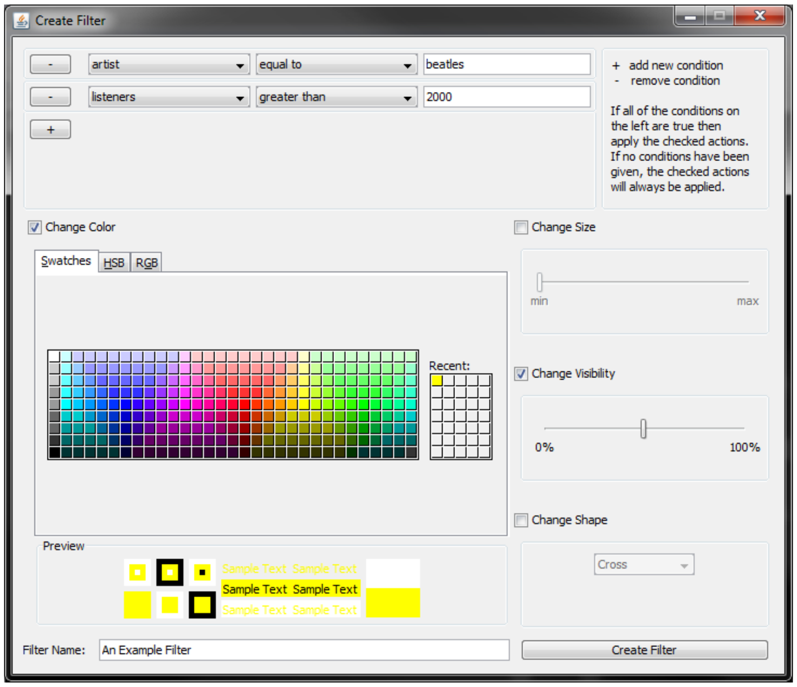
When the Filters Sidebar is shown it takes 300 pixels away from the width of the visualization panel. GUI elements within the visualization panel itself are shifted 300 pixels to the right to accommodate the Filters List. Our original design described the Filters List as a separate window instead of an integrated Sidebar. Other than our decision to incorporate the list into the Main Window its functionality is consistent with our design.

The underlying representation of the Filter List as a Java ArrayList of Filter objects is discussed primarily in the Filter implementation section. An important function of the Filters List GUI is to always display the same ordering as the internal data structure and to provide the capability to reorder the list.

Filters are listed by their name and description in a Scrollable JList Swing component. The list model displayed by this JList contains the ArrayList of Filter objects. Clicking Filters on the GUI list selects them. Multiple selections are possible by holding shift or ctrl. Selected filters can be moved up, moved down, or deleted using insertion and removal functions. When the user indicates to apply filters, the list model is polled for all selected Filter objects using the getSelectedIndices() method, and an ArrayList of Filter objects is returned to the FilterManager class for adding and running in the associated visualization. In Figure 18 only the last Filter, selected in blue, would be applied. As explained in the implementation of Filters section, all of the Filter objects are compiled into one single ActionList before being run.

4.5.3 Filter Creation Window

Filter creation opens a separate window with components designed to specify the condition and actions of a Filter. This interface was carefully constructed in an attempt to make the seemingly complex multi-step process of filter creation as intuitive as possible.


Create Filter

-	artist	equal to	beatles
-	listeners	greater than	2000
+			

+ add new condition
- remove condition

If all of the conditions on the left are true then apply the checked actions. If no conditions have been given, the checked actions will always be applied.

[✓] Change Color
[ ] Change Size
[✓] Change Visibility
[ ] Change Shape

Filter Name: An Example Filter

Figure 19: A screenshot of the Filter Creation Window specifying a simple Filter with two conditions and two active actions.

Figure 19 shows a screenshot of the Filter Creation Window specifying a filter that finds all songs by the Beatles with greater than 2000 listeners and colors those nodes yellow with roughly fifty percent opacity.

The two main parts of this window are the conditions panel at the top and the action panels below that. The action panels consist of objects for selecting the details of an action (such as the JColorChooser component, used for color specification) and a check box beside each action type indicating whether that action is active in the Filter being created. Checking a box enables the controls underneath and un-checking disables them. The difference is visible in the Change Size and Change Visibility panels in Figure 19. JSliders are used for sliding scales such as size and visibility. A drop-down selection box is used for choosing one of the built in Prefuse shapes.

The panel for specifying conditions allows up to five individual condition statements to be specified for a Filter. In order for the selected actions to be applied to any nodes, all of the conditions specified must met. The plus and minus buttons are used for adding new conditions and removing specific conditions, respectively. When a condition is added it is placed below all existing ones and the plus button is shifted down one space. When a condition is removed, all conditions below it (and the plus button) are shifted up one space.

Each individual condition line specifies an “if [field-name] [operation] [field-value]” statement. For example, the following (Figure 20) condition line in the GUI specifies the statement: “if artist equals beatles” where “artist” is the field-name, “equals” is the operation, and “beatles” is the field-value. Field-name is specified in a drop-down menu of available document attributes. Operation is specified in a second drop-down menu of operations appropriate for the type of the chosen field-name. Field-value is user specified and entered in a JTextField.

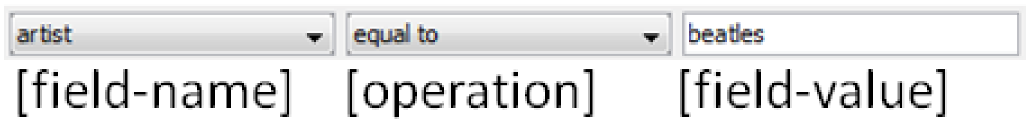
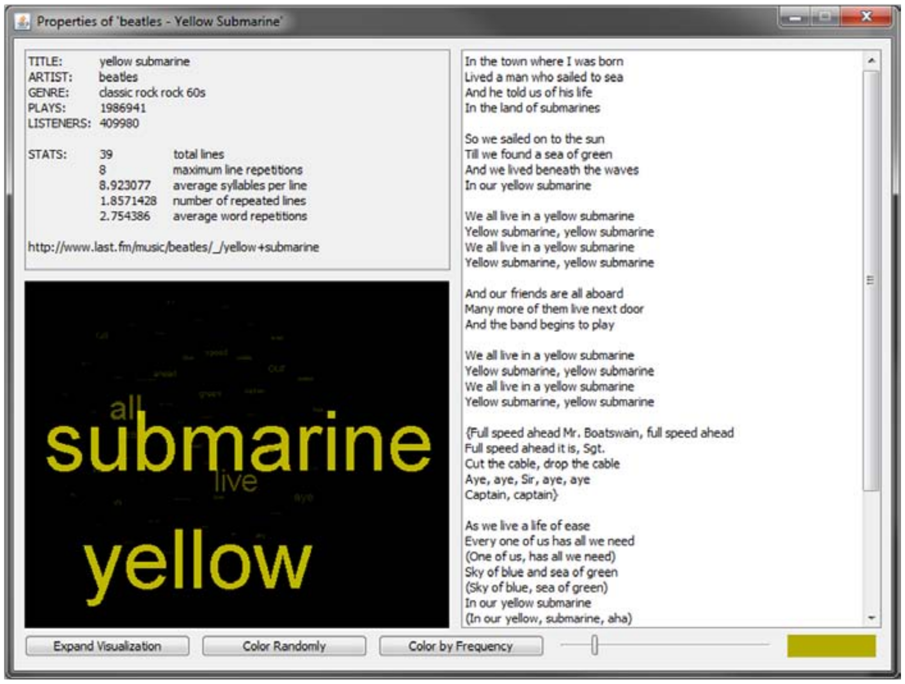


Figure 20: A diagram showing the three parts of a Filter condition on the user interface: field-name, operation, and field-value.

When the user is finished specifying both the condition and actions of the Filter they are creating they can click the Create Filter button. As a result the Filter Creation Window is hidden, the underlying Filter object is created (per the method described in the Filter implementation section), and a representation of that Filter is added to the Filter List in the Sidebar. If the field-value is invalid for the type of the chosen field-name (contains alphabetical characters when the type should be a number) the Filter will not be created.

4.5.4 Document Properties Window

The purpose of the Document Properties Window as described in our Design and Preparation section is still an accurate description of what it is in our finished application. It is used primarily to show document metadata, document text, and a document-only visualization. A screenshot of the finished Document Properties Window is shown in Figure 21.



```
TITLE:      yellow submarine
ARTIST:     beatles
GENRE:      classic rock rock 60s
PLAYS:      1986941
LISTENERS:  409980

STATS:      39          total lines
            8           maximum line repetitions
            8.923077    average syllables per line
            1.8571428   number of repeated lines
            2.754386    average word repetitions

http://www.last.fm/music/beatles/_/yellow+submarine
```

In the town where I was born
Lived a man who sailed to sea
And he told us of his life
In the land of submarines

So we sailed on to the sun
Till we found a sea of green
And we lived beneath the waves
In our yellow submarine

We all live in a yellow submarine
Yellow submarine, yellow submarine
We all live in a yellow submarine
Yellow submarine, yellow submarine

And our friends are all aboard
Many more of them live next door
And the band begins to play

We all live in a yellow submarine
Yellow submarine, yellow submarine
We all live in a yellow submarine
Yellow submarine, yellow submarine

{Full speed ahead Mr. Boatswain, full speed ahead
Full speed ahead it is, Sgt.
Cut the cable, drop the cable
Aye, aye, Sir, aye, aye
Captain, captain}

As we live a life of ease
Every one of us has all we need
(One of us, has all we need)
Sky of blue and sea of green
(Sky of blue, sea of green)
In our yellow submarine
(In our yellow, submarine, aha)

Figure 21: A screenshot of the Document Properties Window showing a documents attributes, metrics, text, and Word Cloud visualization.

The window consists of two Scrollable JEditorPane components. The right pane is the longer of the two and is used to display the text of the document being shown. By adding the JEditorPane to a JScrollPane we automatically obtain a scrollbar if the text is too long to be displayed all at once. The left side of the window consists of a second JEditorPane that shows the metadata of the document. For songs this includes the Title, Artist, Genre, number of Plays, and number of Listeners. We also found it useful to display the raw metrics calculated for each document here as well as a link to the song's last.fm page. All of this data is formatted for display with tabs and added to the JEditorPane. Both JEditorPanes are set to be static using the setEditable(Boolean) method.

Underneath the metadata pane is a small document-only visualization display. The visualization used here is the Word Cloud, which arranges each word that occurs in the document and scales the word occurring to its frequency in the document. Information on our implementation of the Word Cloud visualization can be found in the Visualization Display subsection. However, a few interactions with this display are facilitated through controls at the bottom of the Document Properties Window. First, the Word Cloud can be expanded to fill the entire Window allowing a closer view of the visualization. Second, the Word Cloud can be displayed in either random color or color by frequency mode. In the latter case the hue of the Word Cloud display is specifiable using the JSlider at the bottom left of the Window. Figure 22 shows some of these interactions.

Additional details on our implementation of the Word Cloud visualization is given in the Visualization Displays subsection of Architecture and Implementation.

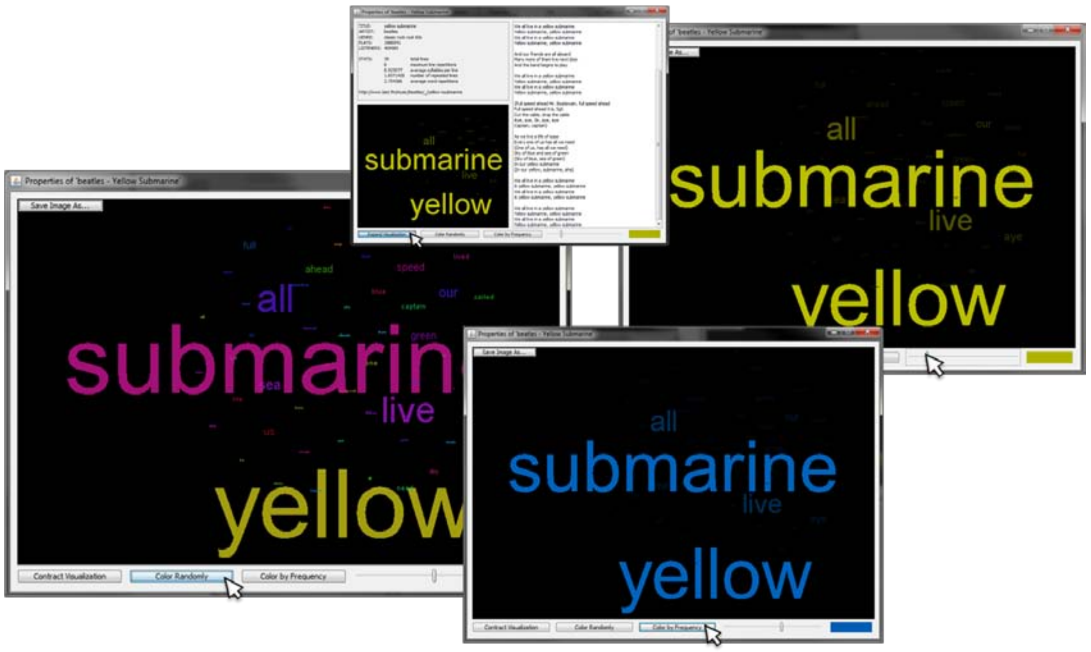
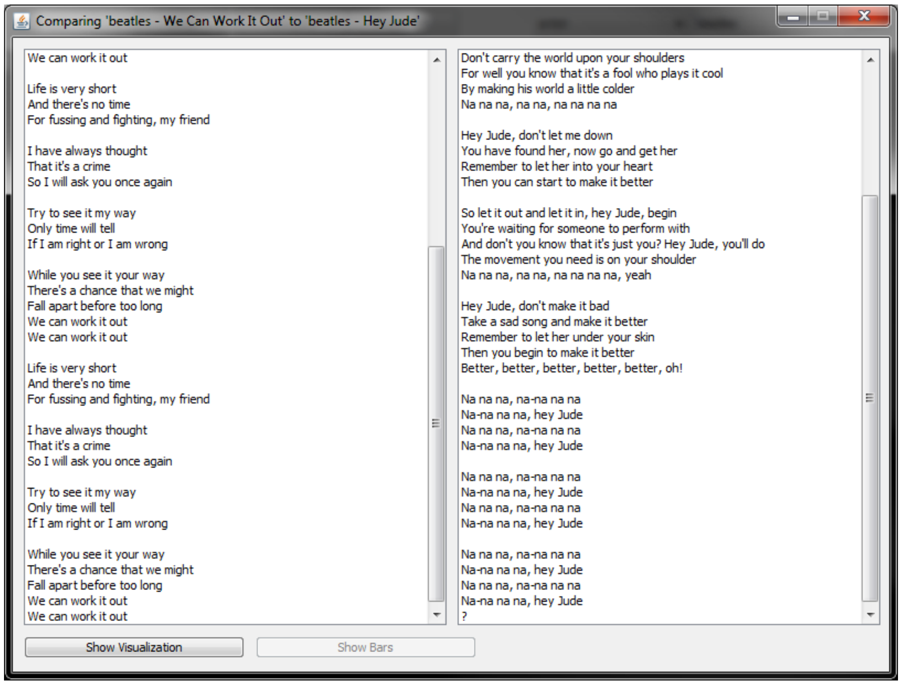


Figure 22: A diagram showing the Word Cloud interactions on the Properties Window, including expending the size of Word Cloud visualization and altering its color-scheme.

4.5.5 Document Comparison Window

The separate and synchronized Document Comparison Windows of our original design were scrapped during implementation in favor of a single Document Comparison Window meant to display just the comparison between two given documents. It is shown in Figure 23 and simply consists of two Scrollable JEditorPane displays, one for each document being compared.



Comparing 'beatles - We Can Work It Out' to 'beatles - Hey Jude'

We can work it out

Life is very short
And there's no time
For fussing and fighting, my friend

I have always thought
That it's a crime
So I will ask you once again

Try to see it my way
Only time will tell
If I am right or I am wrong

While you see it your way
There's a chance that we might
Fall apart before too long
We can work it out
We can work it out

Life is very short
And there's no time
For fussing and fighting, my friend

I have always thought
That it's a crime
So I will ask you once again

Try to see it my way
Only time will tell
If I am right or I am wrong

While you see it your way
There's a chance that we might
Fall apart before too long
We can work it out
We can work it out

Don't carry the world upon your shoulders
For well you know that it's a fool who plays it cool
By making his world a little colder
Na na na, na na, na na na na

Hey Jude, don't let me down
You have found her, now go and get her
Remember to let her into your heart
Then you can start to make it better

So let it out and let it in, hey Jude, begin
You're waiting for someone to perform with
And don't you know that it's just you? Hey Jude, you'll do
The movement you need is on your shoulder
Na na na, na na, na na na na, yeah

Hey Jude, don't make it bad
Take a sad song and make it better
Remember to let her under your skin
Then you begin to make it better
Better, better, better, better, better, oh!

Na na na, na-na na na
Na-na na na, hey Jude
Na na na, na-na na na
Na-na na na, hey Jude

Na na na, na-na na na
Na-na na na, hey Jude
Na na na, na-na na na
Na-na na na, hey Jude

Na na na, na-na na na
Na-na na na, hey Jude
Na na na, na-na na na
Na-na na na, hey Jude
?

Figure 23: A screenshot of the Document Comparison Window's default view, which shows the text of two documents side by side.

We had originally hoped to implement a feature in the comparison view that would allow the user to highlight shared and similar words in both of the document text areas. Multiple words could be highlighted in different colors between the two panels or all occurrences of a single word could be highlighted on mouse click or hover actions. While this feature never was implemented, we still feel that it would be a useful way of comparing words in context.

The two comparison visualizations that we implemented show what words occur in which of the two documents being compared. They also both show the relative frequency of each word in each document. There are two versions of the comparison visualization. The first version we came up with was a layout of two columns of unique words, one for each document. Words that occur most often in a document appear higher and larger in the stack. Words that appear in both documents are connected with edges. Figure 24 shows an example comparison using this visualization.

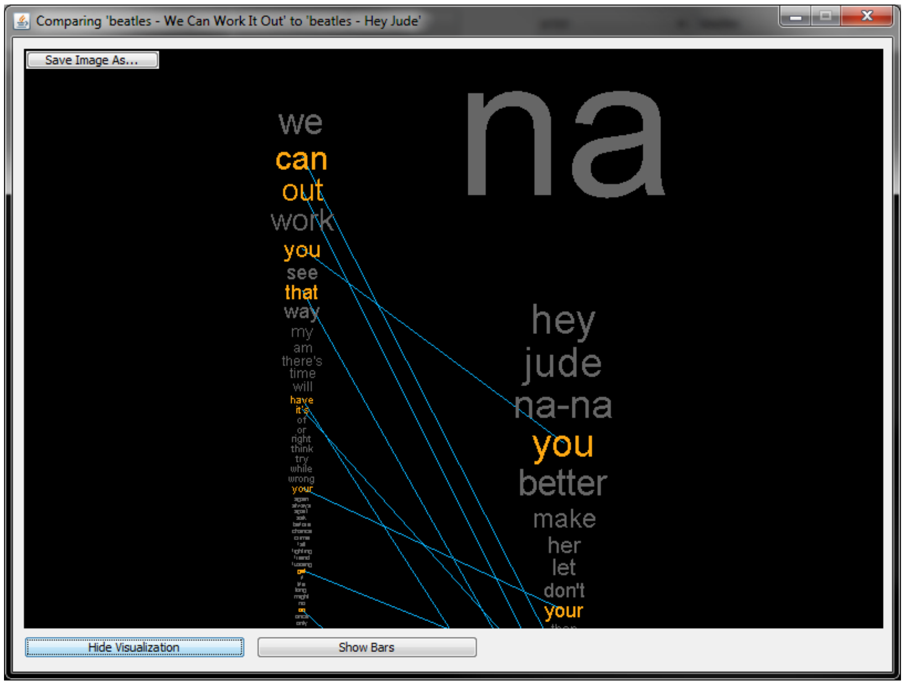


Figure 24: A screenshot of the Document Comparison Window with Visualization 1 – scaled words with connections – active.

We felt that this comparison visualization was a good way of viewing shared words and to compare the frequency of shared words across two documents. However with two documents of vastly different length one column would tend to hang lower than another; this could make it difficult and tedious to view in certain circumstances. To solve this we implemented a second comparison visualization that lists words alphabetically in one single column. Frequency and occurrence of a word in a song is indicated by the length of and side of the column a bar is shown on. If a word occurs in both documents, bars will occur on both sides. Figure 25 shows an example comparison using this visualization.

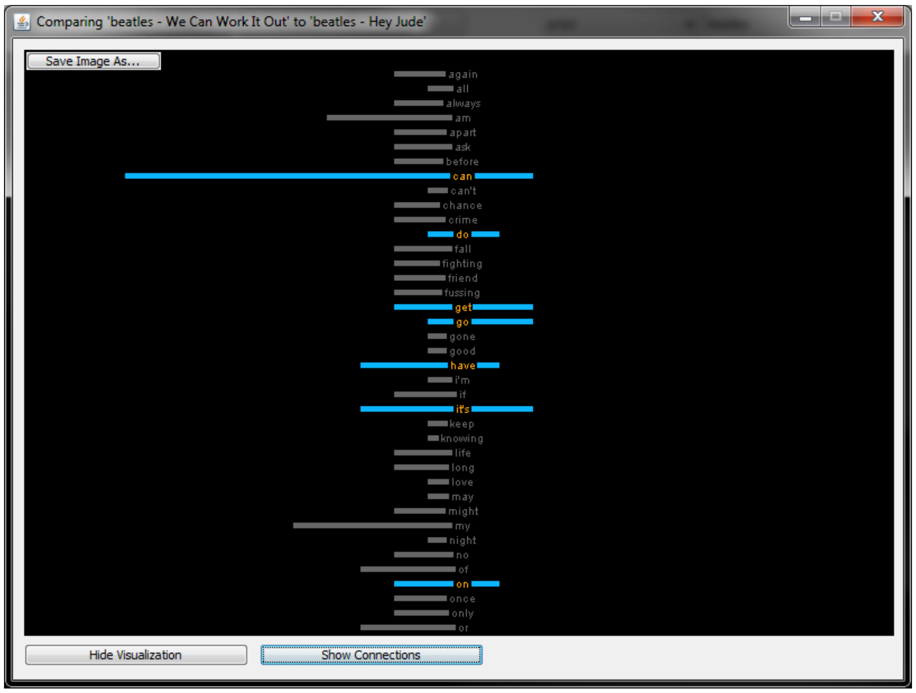
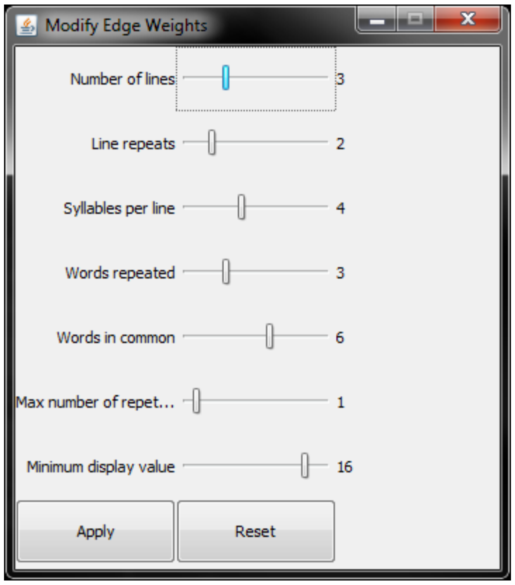


Figure 25: A screenshot of the Document Comparison Window with Visualization 2 – words with bars – active.

Additional details on the implementation of both comparison visualizations are discussed in the Visualization Displays subsection of Architecture and Implementation. Unlike the Word Cloud visualization in the Properties Window, interactions with these displays are limited to panning and zooming. The Comparison Window simply provides functionality for toggling between the visualization view and the side-by-side lyrics view and choosing between the two types of comparison visualizations.



Number of lines 3

Line repeats 2

Syllables per line 4

Words repeated 3

Words in common 6

Max number of repet... 1

Minimum display value 16

Figure 26: A screenshot of the Edge Weight Modification Window with sliders for metric weights and the minimum display value.

4.5.6 Edge Weight Modification Window

The Edge Weight Modification Window is a view we introduced during the development of the application to facilitate experimentation with the metrics algorithm used to determine the weights of edges. It provides sliders to adjust the importance of each specific metric and an additional slider to specify an edge weight threshold value. The threshold is simply the value required for an edge to be added between the nodes of two documents. This window is not intended for an end-user in its current form because adjusting the edge weights can have certain consequences, including having too many (or too few) nodes and edges in the visualization. Nonetheless, Figure 26 shows a screenshot of this window.

4.5.7 Enhanced Selection Tools

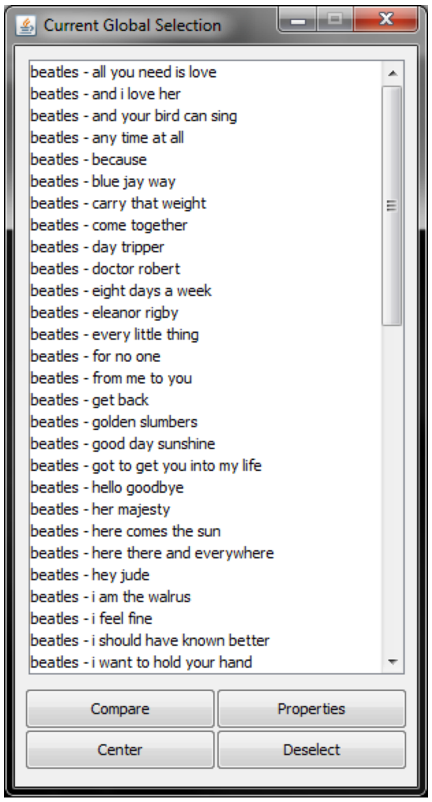
Initially the application we developed had two different methods for selecting nodes. First, the user could simply click on a node to select it (clicking again to deselect). The selection of a node was shown with a colored stroke, or outline, around the node. A second method of selecting nodes was through the Selection Helper. Using this, the user could easily select a node with a specific title or all nodes corresponding to a specific artist.

Consider, however, that both the Global (force-directed) and Local (node-centric) visualizations show only a subset of documents in a given collection. The global view filters out all nodes that have no connecting edge weights above the currently set threshold. The local view shows only the fifty closest nodes to the central node. This creates a problem: how is the user supposed to view properties of and compare these hidden invisible documents?

Later in the development of the application we added some features to help clarify what nodes a user currently has selected and where they are. The first of these features is a count of the total number of selected nodes with a breakdown of how many of them are visible and how many of them are not. This summary information can be seen in both Figure 15 and Figure 18.

The second enhanced selection tool is a list of all selected documents, accessed via a Current Selection button near the top border of the main visualization display. This list reveals the names of all invisible selected documents, something that was not possible previously. It also provides functionality for selecting documents, opening Properties and Comparison Windows, and centering documents in the local view. Figure 27 is a screenshot of the Current Selection Window. By default the list for the global view is opened on the left side of the screen while the list for the local is opened on the right.

The final selection tool we added to the application was the Selection Widget. This separate window provided a way of browsing through every document in the collection in alphabetical order. A JSlider allows users to quickly jump to the neighborhood of a document. Arrow buttons below the JSlider allow users to select a particular document with a higher level of granularity. In the example shown in Figure 28 a user scrolls until he sees the artist he is looking for and then uses the arrow buttons to focus in on a particular song. From there the user can add the selected document to the selection (whether or not it is visible in the current visualization) or center the document in the local view.



beatles - all you need is love
beatles - and i love her
beatles - and your bird can sing
beatles - any time at all
beatles - because
beatles - blue jay way
beatles - carry that weight
beatles - come together
beatles - day tripper
beatles - doctor robert
beatles - eight days a week
beatles - eleanor rigby
beatles - every little thing
beatles - for no one
beatles - from me to you
beatles - get back
beatles - golden slumbers
beatles - good day sunshine
beatles - got to get you into my life
beatles - hello goodbye
beatles - her majesty
beatles - here comes the sun
beatles - here there and everywhere
beatles - hey jude
beatles - i am the walrus
beatles - i feel fine
beatles - i should have known better
beatles - i want to hold your hand

Figure 27: A screenshot of the Current Selection Window which lists of the artist and title for every currently selected document.

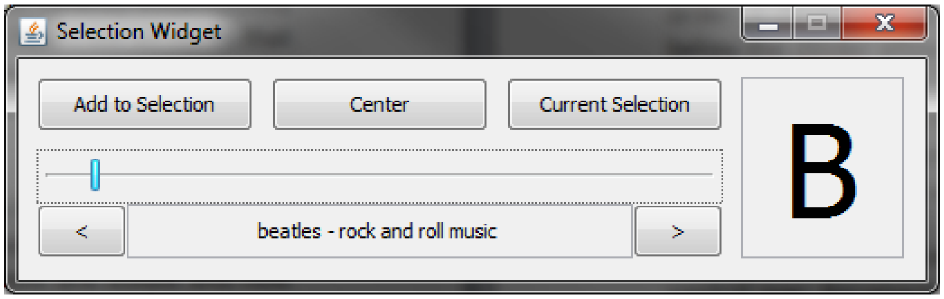


Figure 28: A screenshot of the Selection Widget Window with a slider for quickly scrolling to a document's place in the alphabet and arrow buttons for more precise selection.

These additional indicators and controls are meant to give the user a greater awareness and control over document selection.

5 Evaluation and Results

Our work during the implementation phase of this project resulted in an application with some clearly visible results. Very often we were impressed with the intricate designs and formations that the force directed layout algorithm yielded, but the true goal of this project was to make more than just “pretty pictures”. For this reason, in evaluating our application and our decisions in making it, we sought to answer two major questions.

The first question is in regards to our calculated metrics and the means we used them to arrive at the edge weights that display similarity:

1. **Are the results displayed by the application indicative of what users believe is similar among the text documents in the collection?**

The second question has to do with the usability of our program and is really a Human Computer Interaction concern:

2. **Does the application have a practical use as a professional or entertaining tool and is it appealing and intuitive to use?**

To answer these questions we used two primary methods. The first of these evaluation techniques was a self assessment of our work as we proceeded with implementation. Using the various tools of the application itself, including the comparison views and word cloud renderings, we spent time trying to determine the accuracy of similar songs, as well as identifying trends and interesting information. The second evaluation technique was a more traditional and rigorous usability test that we administered to willing participants. It targeted areas such as the effectiveness of the interface, interactions with the visualizations, and users’ perceptions of what they saw while using the application.

5.1 Self Evaluation

5.1.1 Procedure

Similarity Algorithm

Discovering the similarity between two songs is one of the main purposes of this application, and so we spent a lot of time trying to identify how accurate it was. This was a continuous process that occurred the entire time throughout development and shows itself in some of Design and Implementation sections in the guise of alterations and new features.

On a more final viewpoint, as the application neared completion we often would spend time looking at the lyrics of related songs and trying to identify why they were classified as related and if they really were. Key components to look out for here were essentially our metrics – number of lines, length of lines, etc. We also extensively used the comparison visualizations in order to view the words required to make the relationship form.

Trends and Interesting Data

There were many interesting visualizations we were able to create, and they were achieved mostly by making a large amount of filters and selectively applying subsets of them. Using filters that matched certain criteria, it was easy to color and size nodes with desired attributes. By looking at the visualization as a whole with attributes color-coded, identifying interesting sets was as simple as identifying colored clusters.

5.1.2 Results

Similarity Algorithm

There is much difficulty in determining if relationships are accurate or not; however, we tried as best we could. We did discover several trends within these relationships. Generally, the structural metrics worked well. Oftentimes related songs would have a very similar number of lines, lines were approximately the same length, and if one song had a lot of repetition so did the other. We felt that the structural metrics were a good indication of the structural relationship between songs, but we also felt that there could have been more metrics with more depth. Rhyming is an example that we wanted to use but could not. Others include the verse, bridge, and chorus structure (similar to a rhyme scheme structure) or the amount verse, bridge, and chorus repetition as a whole.

There were a number of problems when analyzing the words within songs. As a whole, we felt that our algorithm performed moderately well. When examining two songs, it seemed that about half of the time the words were significantly related. That means that half of the time we would see words like “river” or “kittens” a lot in each song, but the other half we would see words such as “go” or “about” often. Even for words such as “love” there is an ambiguity, since there are so many things the artist could be writing about that they love. And while “go” and “about” have meanings, they too are often ambiguous without context. If a song is about “rivers” or “kittens”, there is not as much ambiguity; still, it is possible those words are used in similes or metaphors. So, despite two songs being similar on the outside, they could have completely different underlying meanings.

There are many different aspects of this project that we could be changed in order to achieve better results that we discovered during this evaluation. First, more songs would have allowed us to place stricter requirements on determining whether two songs were significantly related. There might not have been more songs in the visualization, but they would have been more similar as a whole on average. If there were as many as 15,000 songs in the collection, it would not be too far out to consider requiring all six metrics to be related as well as boosting the percentage of words in common requirement up to 30%.

Other methods of achieving greater similarities that we identified lied mostly in the relevance algorithm. There are many non-structural components to lyrics that are more than simply words in common. Phrases, and what words mean in the context of which they are used, is a huge part of this. In addition to this, alternate or general meanings through the use of a thesaurus also would have helped. By using these tools it may have been possible to better analyze what a song means, at least on the outside. A Roget’s Thesaurus could theoretically be used to analyze every word and place it in its appropriate category of language, and then it would be possible to relate songs based upon their dominant categories. One issue we identified with this method is that currently the only version of Roget's Thesaurus in the public domain is from 1911[12]. While it may produce reasonable results, it is hard to say without actually testing it. Since nearly all of our songs are from the last 20-30 years, and most being within 10 years, it is also hard to identify how many words, such as modern slang, would not appear in the thesaurus as well as how meanings and slang words occur.

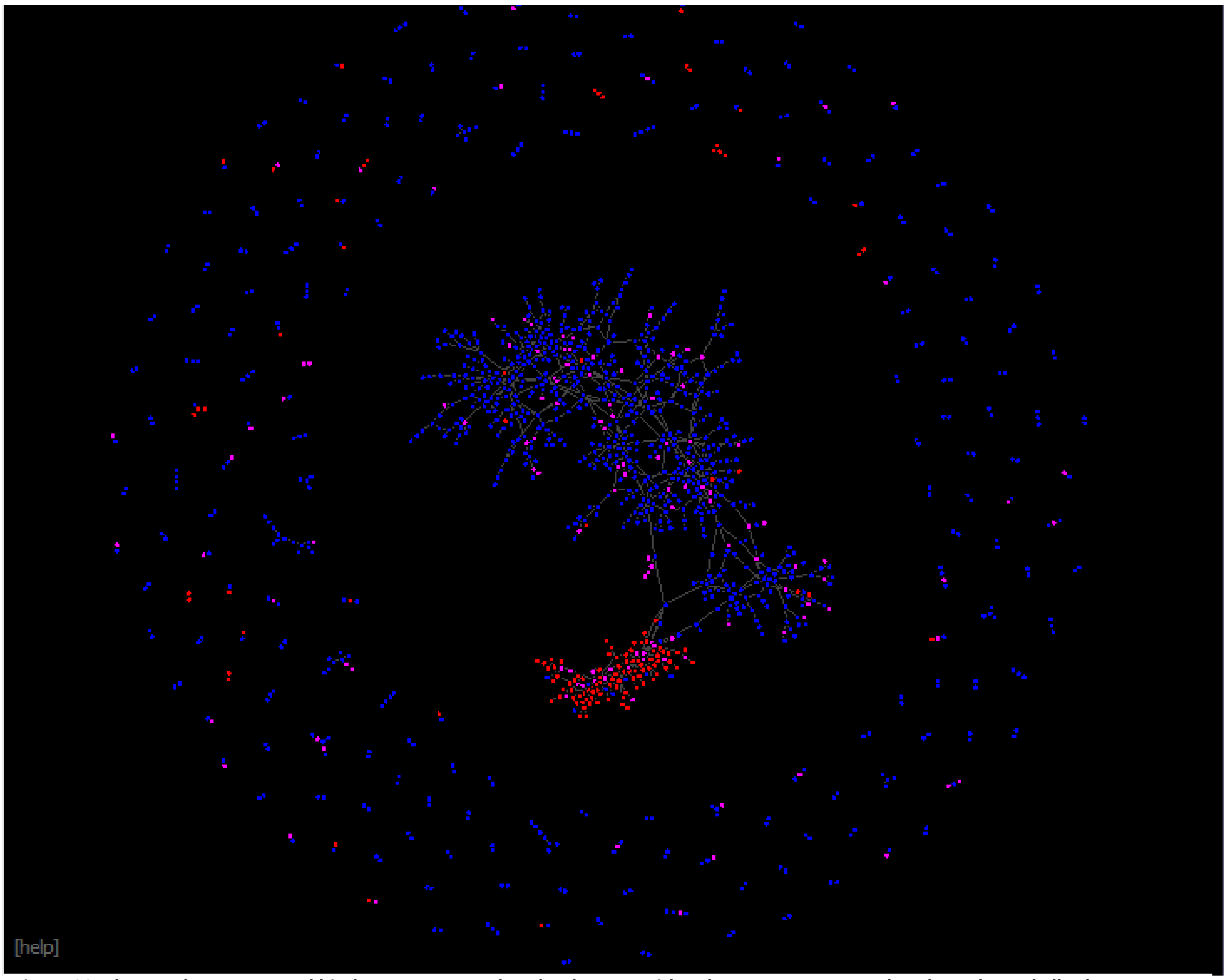


Figure 29: The rap cluster. Rap and hip-hop songs are colored red, songs with unknown genres are colored purple, and all other songs are colored blue.

Trends and Interesting Data

The most interesting thing our application was able to do is cluster rap songs together. While genres such as pop and rock had no specific layout, nearly every rap song was only related to other rap songs, forming a small subsection of the graph almost entirely of songs in the rap or hip-hop genre. Figure 29 shows a visualization with the cluster clearly visible by coloring nodes based on their genres. This pattern emerges for several different reasons. First, the lyrics in most of the songs in this group tend to be filled with profanities and racial slurs. This often ensures that the words in common metric is always applicable. In addition, these songs tend to have a high line count as well as a much higher average syllables per line compared to the rest of the songs in the collection.

The visualization was also explored for other patterns in the metadata; however, nothing else gave very good results visually. Other genres were examined, but no specific patterns emerged. This could possibly be due to the fact that a large majority of the songs in the collection are from the pop or rock genres. It may be that if other genres had a more respectable representation, there would have been clusters for country songs, or reggae songs, or ska songs as well. Another piece of metadata looked at was the number of unique listeners and the total number of plays, based upon last.fm's collected information. Filters were created to attempt to find clusters of songs with very high numbers of plays or listeners, which may have given insight into what words are often found in popular songs. Additionally, low values were examined to find out what words are found in unpopular songs. However, nothing of interest was found in these examinations.

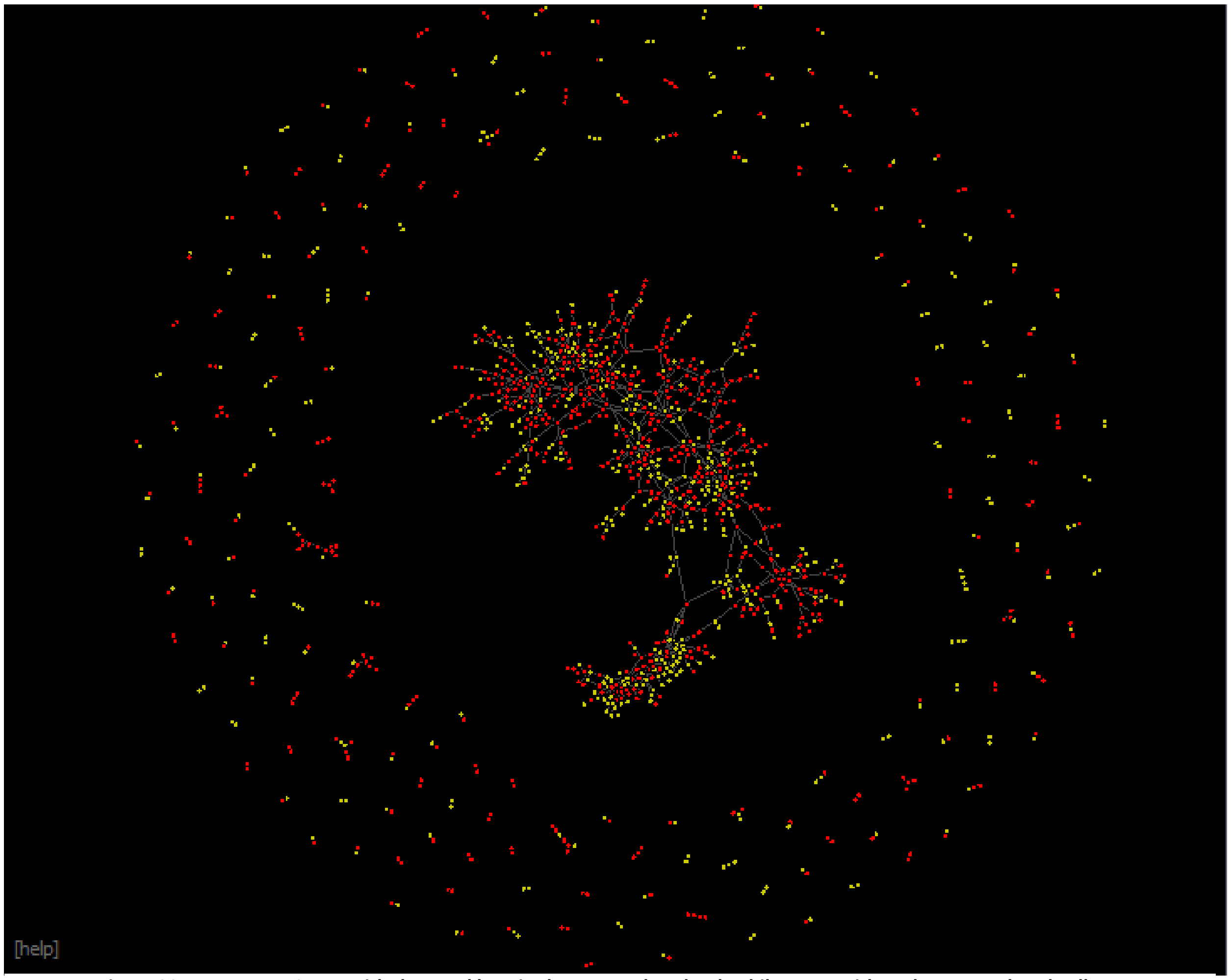


Figure 30: Love songs. Songs with the word love in them are colored red, while songs without love are colored yellow.

We then moved on to looking at word distribution in songs. It is possible to do this with nearly every word. For the sake of brevity, we examined the most popular word, excluding stop words, which was “love”. Figure 30 shows what the visualization looks like when comparing songs with the word love in them to songs without that word. It is clear that the word love appears at least once in approximately half of all the visible songs, which is quite astounding. There are also some small clusters that appear scattered throughout the entire visualization that are no doubt caused by that word being in common. An interesting follow-up to this would be to add love to the stop words list and see how radically different the visualization becomes.

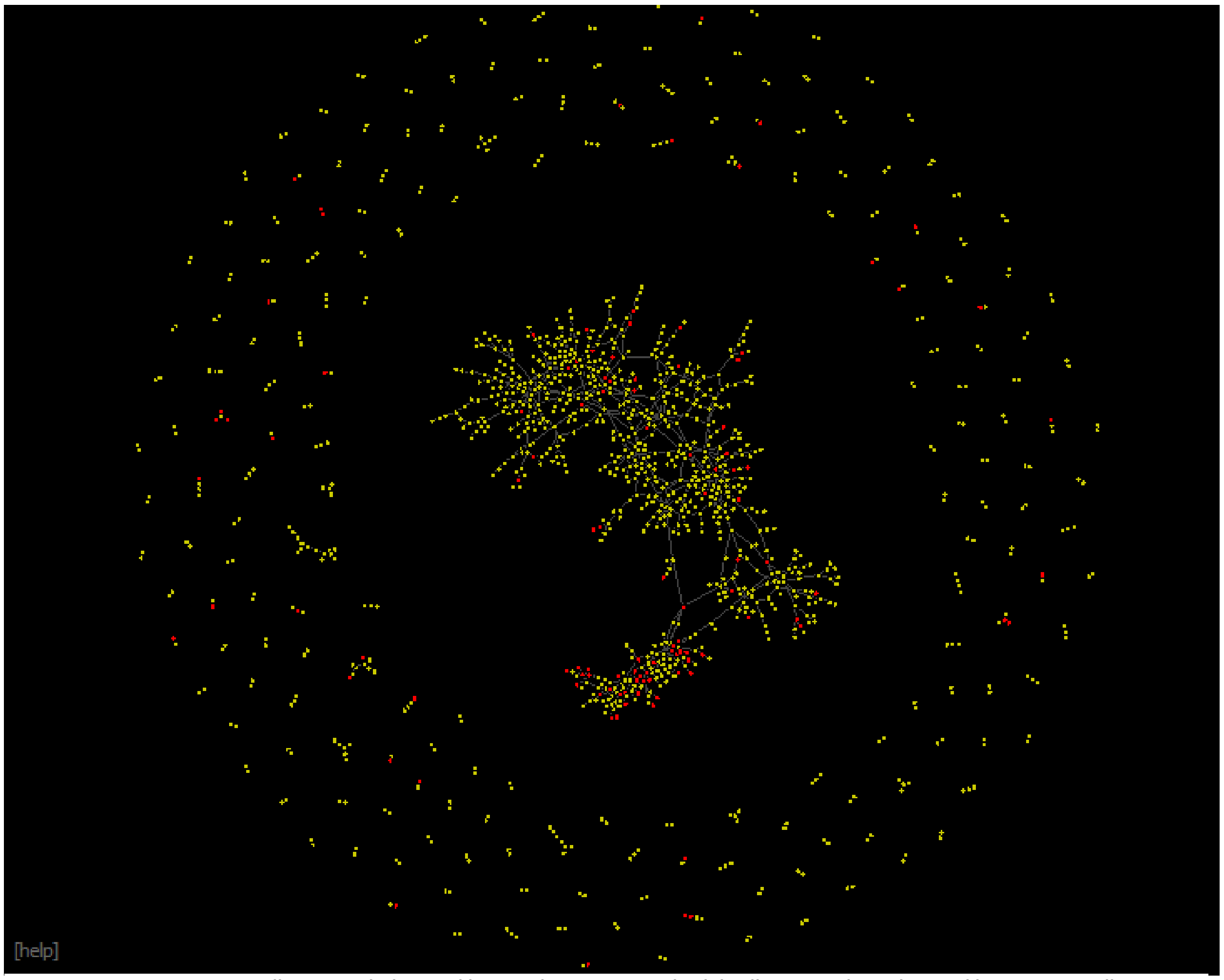


Figure 31: Hate songs. All songs with the word hate in them appear red, while all songs without the word hate appear yellow.

In addition to looking at all of the love songs, it seemed reasonable to also examine the word “hate”, the result of which can be seen in Figure 31. This visualization is radically different from its love counterpart, with hardly any songs containing the word hate. However, it is easy to see that approximately half of all songs containing the word hate appear in the rap cluster. To give the rap genre a little credit though (they’re not all about hatred), Figure 32 shows the same visualization but this time songs with both love and hate colored green. About half of the songs in the rap cluster change from red to green.

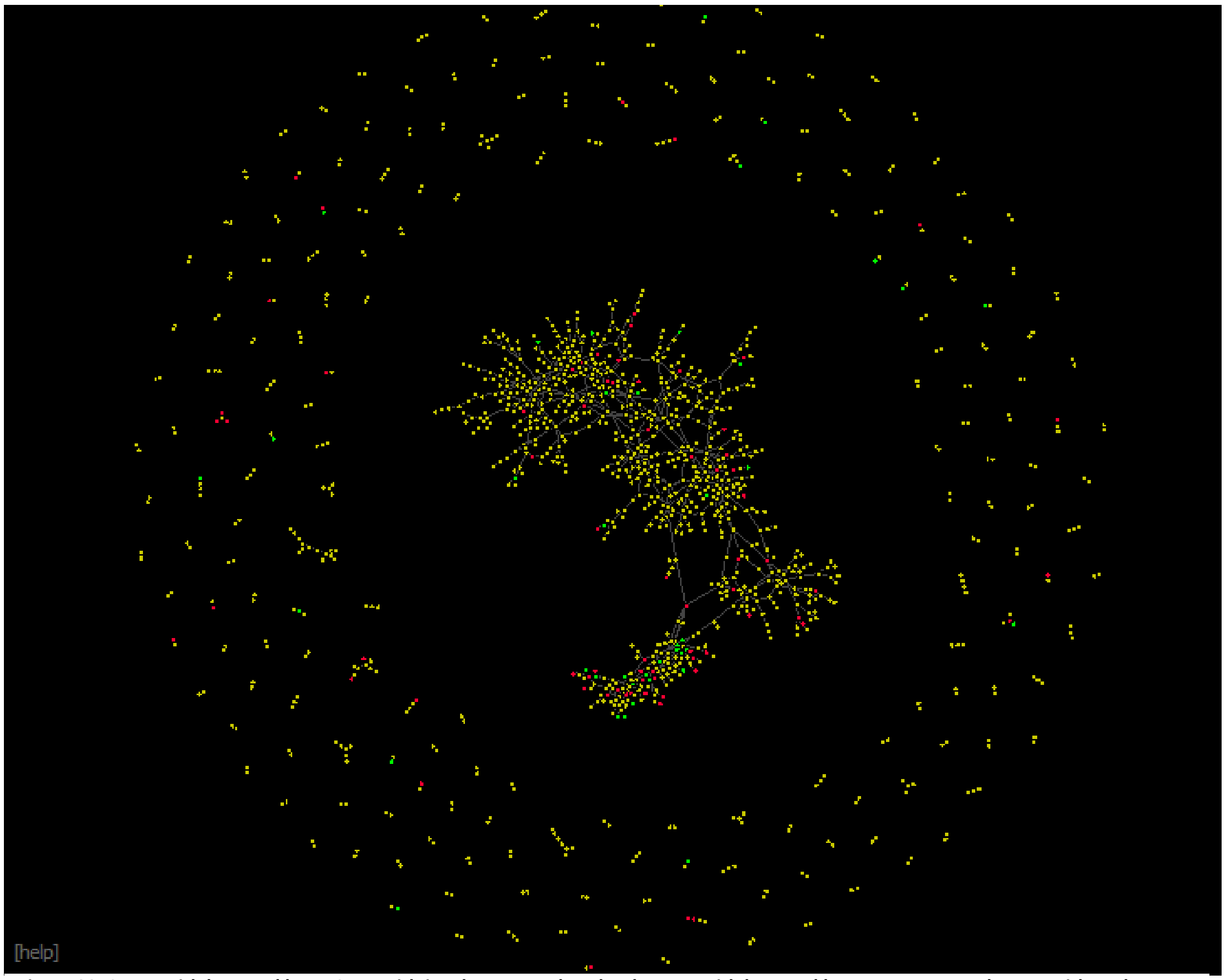


Figure 32: Songs with love and hate. Song with just hate are colored red, songs with love and hate are green, and songs without hate are yellow.

These kinds of comparisons could be done with nearly any set of words should a user be so inclined and have plenty of spare time. Given a larger collection of songs from more varied genres, interested researchers could almost certainly use this application to discover plenty of trends and interesting data hidden in the collection.

5.2 Usability Test

5.2.1 Procedure

Test Objectives

Our usability test had a number of key objectives. The first objective was to determine whether a user's assessment of documents similarity was in any way consistent with what our application reported. While our algorithms may be effective for grouping similar documents by the rules we have set forth, if those rules result in similarity relations that a majority of users strongly disagree with or cannot relate with, it may be sensible to change the system.

The second objective of our usability test was to understand how users exercise the application when they are given the freedom to experiment with it. Observations from this kind of "sandbox" period of evaluation would help us come to certain conclusions regarding how appealing this kind of application may be to a more mass audience. It might have also highlighted difficulties or frustrations encountered by new users that could be addressed with minor alterations to the user interface.

The third objective of our usability test was to assess whether users would understand what was being displayed to them by the application after following a brief series of instructions. To this end, it was also important to note whether users could grasp how the sequence of steps led to the outcome.

The final objective of our usability test was to determine how easily users could find the answers to some general questions about documents in the collection. This would require them to utilize what they may have learned from the rest of their time with the application to guide themselves to where they could find the answer.

Logistics

Initially, we planned to administer the usability test to our friends and family during the days of Thanksgiving break, Wednesday November 25th to Sunday November 30th. We targeted this audience because they had already shown an interest in our work by providing us with a set of song lyrics to bolster the original list we began with. Our hope was that the inclusion of recognizable songs would make our target users more receptive to what they would be testing. Thanksgiving break was the most ideal time for accessing this target audience. Following Thanksgiving break we also conducted an additional survey consisting just of the song-listening portion of our evaluation.

The overall geographical location of the test was not as important as the immediate environment. We decided that it would be most effective for the test to be performed on a reasonably current machine with a monitor and screen resolution large enough to accommodate the main window of the application, which is 1024 pixels wide by 740 pixels high. A quiet and comfortable workspace was also desirable, with a desk or a table set up so that any necessary instructions would be easily viewable when placed before the user. We also set forth the significance of the computer screen itself only displaying the running application, except when a web browser may be necessary for the listening portion. This restriction would prevent other factors from interfering with the experience of using the program.

Methods

Considering the fact that we had a stable and working prototype and a manageable number of subjects, we decided to adhere closely to the guidelines of a formal usability test. We foresaw three major parts to the test, the first and last consisting of a questionnaire and the middle consisting of both guided and self guided walkthroughs of the application. We decided that during any phase it would be appropriate for the test administrator to answer clarifying questions posed by the user, make observations of user actions, and pose impromptu questions to the user for clarification purposes.

Test Script

Usability testing followed the script below. The test administrator was equipped with an observation sheet as a guide for usability testing. This sheet contained brief instructions for the test administrator (like those below) as well as space for observational note taking. The detailed steps of each step of the evaluation are outlined below:

1. Explain the purpose of the project as a whole. For example: "I'm a member of a team from WPI working on a computer program that is meant to compare songs based on their lyrics and display those comparisons in a visual way." This ensures that the user understands the purpose of the program and the educational context in which it is being developed.

2. Explain the purpose of the test. For example: "This test is meant to evaluate our program. We are interested in learning how intuitive it is to pick up and use and how useful it is for the tasks it was designed to help accomplish. Your participation is entirely voluntary and anonymous. You may end your participation at any time. Also, all questions we ask, on the survey or verbally, are optional." This is meant to clarify the use of the study to evaluate the program and not the user. It also speaks some of the important consent-related issues required by WPI's Institutional Review Board.

3. Hand out the test sheet with pre-test questions. Ask the user to fill them out. A copy of this handout can be found in Appendix A. The questions ask the user about their prior computer experience as well as their taste in music, meant to provide a context to some of the observations and responses that follow.

4. Give the user some time to experiment with the application without any goals or objectives. Answer questions about the interface, the visualizations, or the project that come up. Observe how comfortable the user is with the application initially and over time. Allow the user to choose when to continue. The first few minutes with the application should reveal crucial information on how easy the program is to learn how to use.

5. Ask the user to perform specific tasks with the application. These tasks will include creating and applying a specific filter and finding a song and centering it in the local view. If the user has trouble accomplishing these tasks initially, provide minor assistance first, and then additional assistance as necessary. The idea is that these kinds of tasks should be easy to accomplish through the help features on the interface itself. Make any observations about what seems to be causing difficulty, and clarify any such issues with the user as necessary.

6. Ask the user to answer various questions based on song similarities. This will include finding and copying down relevance values and common words for a specific pair of songs and then determining which of two songs are most related to a single source node via their edge weight. The user may be asked to question their initial instincts and find

evidence for their choices. The point of this portion is to convey the difference between reliable concrete values reported by the program, and the less reliable dynamic layout.

7. Ask the user to indicate a song they enjoy listening to in the collection and then select any two closely related songs. Having the user listen to a related song and afterwards indicate whether they liked it or agree with the suggested similarity relation should help us evaluate our relevance algorithm impartially. There will be a space for these three songs with scales for rating them on the handout. The test administrator may find it easier to take control of the computer to locate these songs in the application and online for listening purposes.

8. Ask the user to answer the follow-up questions. A copy of this handout is included in Appendix A. The follow up questions ask the user what they have learned about the application during the test, how usable they felt the application was, and whether they would be interested in using such an application in the future. There is also a space for additional comments provided.

Questionnaires
Appendix A contains the questionnaire handout we gave to users during the usability test. We chose to use a combination of multiple choice, numerical scales (from one to five), and open answer questions where they were most applicable. We made sure to align positive and negative answers on the same sides of the page for every question and provided clear boundaries for question responses to speed the process of entering the data into a spreadsheet program. Analysis and compilation of this data led to the results explained in the following section.

5.2.2 Results

Our initial group of volunteers consisted of eleven individuals who were observed using the application and asked to perform an evaluation of similar songs. Altogether there were thirty seven such evaluations. A second group of participants were asked to perform the evaluation. There were ten individuals in this second group performing twenty seven evaluations altogether. They were not observed using the program because while we had learned enough through observing the initial group in terms of usability, we still required more numerical data to evaluate our relevance algorithm.

Observations
By observing users as they followed our instructions and requests, we gained valuable insight into the good and bad aspects of our design, mostly GUI-related, which we had not originally realized. The first thing users had to adapt to was the mouse controls. The mouse controls nearly all interactions with the visualization itself, and each button performs more than one function. Many users quickly adapted to this, as like any new application. Still, the adaption was usually not perfect and some parts of controlling of the mouse confused people more than other parts. The left click features of panning and selecting were features almost nobody had trouble with, but right click features like zooming and centering nodes in the local view often

caused trouble. Users tended to left click on a node when they meant to right click, or would entirely forget that right clicking allowed them to zoom in or out. These users would look at the visualization from very far away and try to differentiate nodes that were only two pixels wide, while other times they would have the visualization zoomed in very close and attempt to pan to find new nodes rather than zooming out and easily locating the desired node. Perhaps this is due to most applications that standard users are familiar with do not have right click functionality, aside from context-based menus. It is possible that if we had instead used a context-based menu, perhaps with choices such as “Zoom In”, “Zoom Out”, “Zoom To Fit”, and “Center Node Locally,” that came up every time a user right clicked, they would have had an easier time learning.

The local view was another feature that users seemed to like and take advantage of, but had some trouble using. The main trouble here was caused by the global view and figuring out how to center a node in the local view. Despite telling the user how to accomplish this, none seemed to be able to learn. Almost every user, at some point, tried to center a node by selecting it in the global view then clicking the local button, rather than right clicking the node. It seemed they thought that if only one node was selected and they switched to local view, it should automatically update. While this would be possible, it would have to be done with a pop-up menu that asked a user whether or not they wanted to center it, and it would not work if more than one node was selected. In fact, this confusion was a primary reason that we did not automatically center it; alas, it seems to have not worked in our favor.

There was less trouble with filters and searching, although there were still a few problems. With filters, about half of the users didn’t realize you had to both select the filter and click the apply button to get it to appear. They did not seem to realize that you could create multiple filters and only have certain ones active a time, since we only asked them to create a single one. For searching, many users requested that the search should be case insensitive and it should match partial queries, such as “beat” matching “beatles”. Only one user had trouble differentiating between the functionality of filters versus selections. Another user lamented over the fact that when searching for a song, the visualization did not automatically zoom and center the song. These are features we intended to have but did not implement as only basic searching was a top priority. However, if the application is to be used by the general public, these features would have to be updated.

Users seemed to have little trouble with the properties and comparison windows. When asking them for information about songs or words in common, no one had any difficulty. One thing in particular caused a lot of problems for users, though. When asking them to determine which of two songs was more similar to a third song, hardly anyone was able to answer correctly. The problem was that there are no metrics on the comparison windows, and rightly so there should be. Users alternatively attempted to try and manually compare how many words were in common or manually look at the metrics in the properties windows and try to compare them to no avail. Although the question could have been answered by using the local view and comparing distances – which only two users did – or by using the tooltips that appear over edges, there is no reason the compare window shouldn’t have all the comparison information.

Numerical Results

The numerical results from our user study fall into two categories: ratings of song similarity and exit-survey results. The first category of numerical results helped to indicate how successful our relevance algorithm was in the minds of users and listeners. The second category was more of an indication of the effectiveness of our user interface.

For song similarity users were asked to select a song in our collection and then listen to one or two related songs reported by the application. After listening to each related song the user would indicate on a scale from one to five an assessment of (1) whether they enjoyed listening to it and (2) whether they felt it was similar to the song they chose originally. A value of one would indicate minimal enjoyment or perceived similarity. A value of five would indicate the positive end of the spectrum.

For each possible similarity relation we also reported the edge weight value. These edge weights ranged from very similar, with a value of five, to moderately similar, with a value of thirty, with increments of five in between. The histogram in Figure 33 shows the distribution of edge weights under scrutiny. It can be seen that the mode edge weight value was 20, at the upper middle end of the spectrum, but we also encountered a number of very high similarity relations with values of 5 and 10. If we were to perform this test again in the future we would try to target more comparisons with high similarity (and lower edge weights).

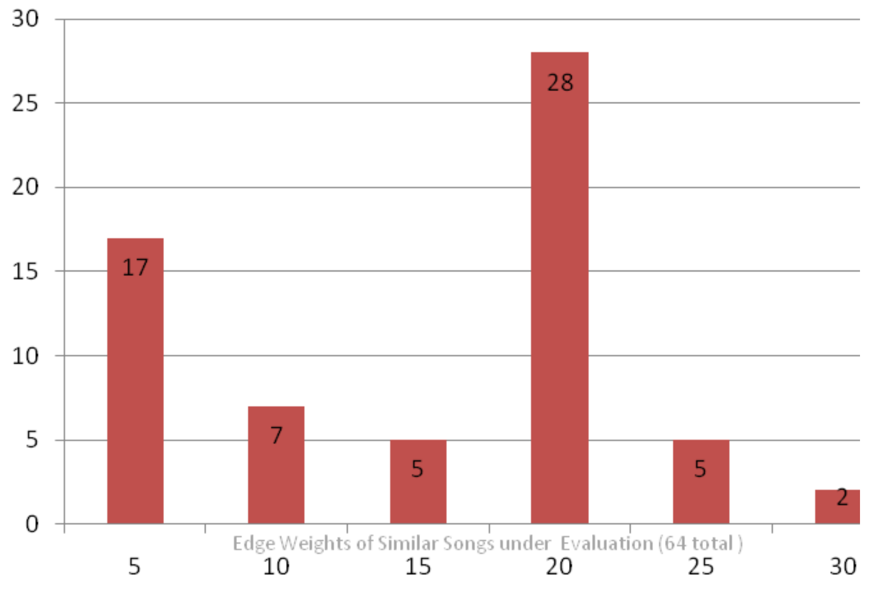


Figure 33: Chart showing the distribution of edge weights for comparisons under evaluation.

The histograms in Figure 34 and Figure 35 show the distributions of ratings for enjoyment and similarity evaluations of the sixty-four songs our participants listened to. The mode rating for enjoyment was in the middle of the five point scale at 3 with a consistent average and median. We can see that exactly 26 of the songs listened to were moderately liked by the listener. The rest of the distribution is actually very symmetric, falling off on both sides of 3. Almost an equal number of songs were highly enjoyed and highly disliked by listeners with counts of 7 and 6 respectively.

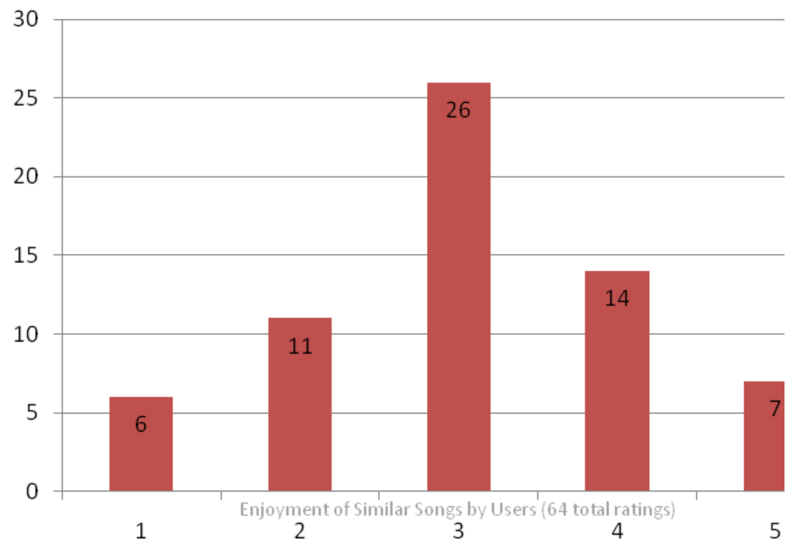


Figure 34: Chart showing the distribution of enjoyment ratings for evaluations of all edge weights.

We have thought about what a score 3 really means in terms of listener enjoyment. In the end we are classifying this result as a neutral stance indicating neither extreme like or dislike of a song. We could say that a 3 is the cutoff for a song that you would listen to on the radio (while a 2 might be on the border of making you change the station). Given that the distribution is symmetric (even slightly skewed towards the positive end) and has a standard deviation of about 1, we can surmise that our application seems to make good suggestions in terms of enjoyment a little more than half of the time. We see these as fairly favorable results, but the "enjoyment" factor is highly subjective.

Evaluations of similarity were less approving of our algorithm. The mode rating score was 2 with an equal median and a mean of 2.5. We can clearly see a skew towards the lower ratings in the histogram with a value of 1 being the second most frequent response. Overall, 37 of the songs listened to were indicated by users to have lower than average similarity to whatever song was chosen at the beginning. 16 of the songs were indicated to have higher than average similarity. While more data might be necessary to be certain, we can already begin to see a pattern emerging in this evaluation that suggests some problems with our similarity algorithm. There is some subjectivity in what users "feel" similarity between two songs to be, but we a confident this collection of ratings more directly addresses the question of whether or not our algorithm is effective more so than the enjoyment ratings. For the purposes of an application for a general audience, similarity should be something that users can see, hear, and agree with. In our Future Works, we propose an experiment related to this idea that could improve our results in the eyes of our users.

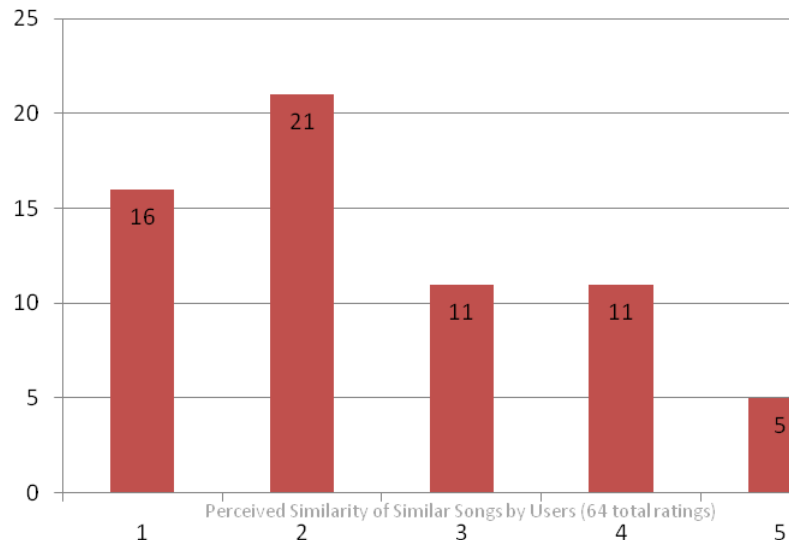


Figure 35: Chart showing the distribution of similarity ratings for evaluations of all edge weights.

There may be an inherent problem with exploratoring the ratings of similarity and enjoyment across songs with vastly differing edge weight values. To some extent, edge weights higher than 10 or 15 really do not suggest a very significant relation in terms of our calculated metrics. While the number of data points in each subcategory of edge weights is not enough to hold up to deep statistical inspection, the change in distribution indicates somewhat more favorable results. These results are included only under this disclaimer. The histograms in Figure 36 and Figure 37 show the same data limited to only comparisons with edge weights of 5.

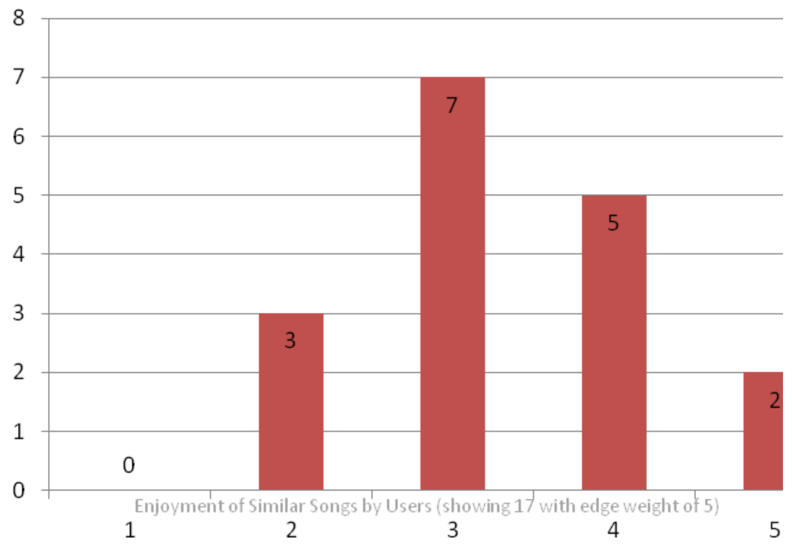


Figure 36: Chart showing the distribution of enjoyment ratings for evaluations of edge weights of 5 only.

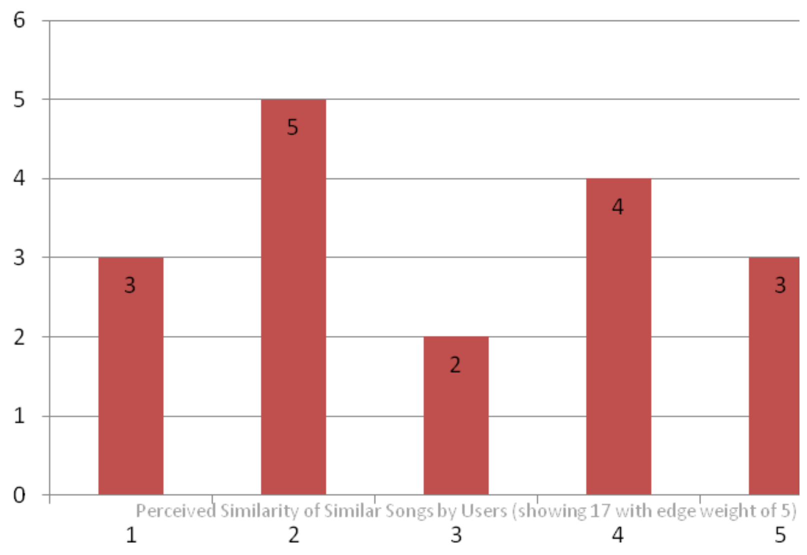


Figure 37: Chart showing the distribution of similarity ratings for evaluations of edge weights of 5 only.

We can see that the distributions are different with this focus. The graph showing song enjoyment by listeners is skewed further towards the higher values, and no one indicated a rating of 1. Most of the enjoyment ratings fell at 3 and 4, which we consider to be favorable scores. Also under this alternate focus, the histogram showing similarity ratings shows less of a negative outcome. While still slightly skewed towards the lower ratings with a mode value of 2, there are a greater percentage of higher ratings in this set of ratings than with all edge weights included.

Again, while there are not enough data points to make any serious conclusions, these alternative histograms may suggest additional testing be performed to target only higher similarity relations. This could be problematic with our current collection, but with a larger set of data, stricter cutoffs for edge weights could be used to limit visible relations to only the most similar pairs. It would be interesting to see if re evaluation under these circumstances yields better results.

The second category of numerical results includes the data compiled by user responses to our exit survey. We do take these responses with a grain of salt because they do not seem to coincide with the troubles and frustrations we observed during testing. Regardless, user response to the program was fairly positive and responses seemed to suggest that users were both confident and willing to do more in the future.

Our first question asked users to evaluate whether they thought they had completed all tasks adequately during the walkthrough and to indicate if they had help along the way. 9 users made it through with help. 2 were able to accomplish objectives independently. No users gave up.

When asked how confident users felt about being able to perform a task with the application after all they had learned and experience, 9 users indicated a high level of confidence and 2 were neutral.

Evaluation of how easy the application was to use and learn were mostly neutral with 7 middle ratings and 4 one step above. No one found the application very easy to use, but no one found it very difficult as well.

When users were asked if they thought menus and buttons on the interface were logically organized 5 responded neutrally, 3 responded positively, and 2 responded very positively. Oddly enough, there did not seem to be any definite correlation between this question and the one on overall ease of use, like we would have expected.

We were pleased to see that users liked using the program overall and would be looking to use a similar program in the future. We are somewhat afraid that these questions may have been biased in some way, but we do appreciate the positive reinforcement nonetheless.

One final response that was interesting in the exit survey was which of the two comparison visualizations users preferred. The choice of the “Words with Bars” version of the visualization was almost unanimous, which coincided with our predictions. This particular visualization was made because of usability problems we experienced using the original one. We are pleased that our newer implementation was well received.

Overall our numerical results were mixed, indicating positive results with room for improvement. We believe that future work, as described in the following section, will only improve the program, our algorithms, and the resulting visualizations.

6 Future Work

In this section we express a number of key areas for future work related to this project. While our results may have been somewhat mixed, we are confident in the future effectiveness of the application we have developed as a platform for future projects in the area of comparison based visualizations – especially with comparisons of different types of text documents.

What follows are summaries of what we feel needs to be addressed to improve upon the specific application we have developed for visualizing comparisons between member documents in a collection of songs. We also provide some instructions on how one could begin working with the code base in a brief developer's guide.

6.1 Improving the Relevance Algorithm

The area of this project that could benefit most from future work is the relevance algorithm that we used. There is much that we intended to do, but did not have time for in the end, including learning about and applying more advanced techniques in the area of natural language processing. There are many other aspects of the algorithm design that can be improved and added as well.

Many suggestions for future works have been discussed previously in this report, mostly in section 5.1.2. Suggested structural metric additions include rhyming, rhyme scheme, verse/chorus scheme, and verse/chorus repetition features. Suggested contextual metric additions and changes include analyzing word phrases, word meanings in context, and using Roget's thesaurus to categorize words.

6.2 Improving the User Interface

By observing users working with our application, we have identified a number of bugs and feature requests that could enhance the effectiveness of the user interface. Some of these future "tasks" were unintended mistakes we missed during testing while others were decisions or features we executed differently than most users expected.

There were a number of simple bugs that our users encountered that we never ran into ourselves. For example, our decision to make all text inputs case insensitive did not apply to filter inputs, which confused many users when they did a query for "Beatles" and obtained no results. Additionally, a bug with the Selection Widget made the Center function have no affect whenever the application was showing the Local visualization. Also, in terms of the Local visualization, whenever the center operation is performed the graph zooms out so that repeated centering without manual panning or zooming results in a smaller and smaller image.

One problem with the implementation of our feature to change edge weights is that instead of clearing the existing visualization and re-adding nodes, the program creates an all new visualization on top of the previous one. This is not ideal. Changing edge weights in this way also prevents Filters from working because the FilterManager still points to the original instance of the visualization.

An unusual user interface bug in the Filter Creation Window prevents the downward arrows from being shown on combo boxes. This is most likely an issue related to the Java Swing framework and the refreshing of components in a JPanel.

A number of users suggested possible features for the application that would make it easier to use overall. Some of our users felt comfortable pointing out these kinds of shortcomings, which we welcomed and encouraged. By observing other users we were able to infer certain features that would improve the overall experience as well.

One of the most requested features we encountered during testing was for the Selection Helper search bar to have the option for performing a "contains" search. This would be useful for searching for selecting all document nodes with the word "kittens" in their title. Another possible enhancement to search would be to search for both title and artist at the same time.

Many users had trouble with the interaction of right clicking a node to launch the Local View with that node centered. Users felt that if they selected a node in the Global View and pressed the Local View button in the toolbar that node should be centered. This could be done, but could be problematic when multiple nodes are selected.

While a context sensitive help bar currently indicates the use of the right click function, most users did not take note of these tips and messages at all. One user suggested that more traditional tooltips might be more noticeable, or a context-sensitive menu listing all possible actions upon a right click.

Some users found the information on comparison metrics in the Document Properties Window interesting and useful to look at. Many pointed out that this information could be helpful if also shown on the Comparison Window.

While users seemed to be able to understand how to use filters in the application, many would have liked for them to be automatically applied after being created. To make this enhancement possible, there would need to be an enhanced tool for enabling and disabling filters. Currently filters are enabled and disabled by holding the ctrl button and clicking them in a list with the mouse.

Some users were observed creating filters with no actions. A simple solution to prevent this mistake would be to alert users that their filter has no visible affect, or prevent creation of filters with no actions entirely.

Additional suggested features were minor and included requests for traditional scroll bars for panning and buttons for zooming in and out. Many users would have liked full screen support and changeable color schemes as well.

6.3 Additional Future Work

Aside from future works in updating the relevance algorithm and more user interaction features, the addition that would benefit this project the most is an enhanced collection. More songs would allow stricter requirements on determining which songs are relevant, which could improve results dramatically. While we have 5605 songs right now, we would have liked to have somewhere around 20,000.

In addition to simply having more songs, there are three related issues. The first is obtaining accurate lyrics for each song. An algorithm that could be used to assist in this is described in the paper "Multiple Lyrics Alignment: Automatic Retrieval of Song Lyrics."[13] The second is having a diverse set of songs. Most of the songs in our collection are from the pop and rock genres. It would be interesting to see how the visualization changes with larger representations for genres such as country or heavy metal. The third and final issue is the inclusion of additional metadata, most importantly the year of a particular song. Having the years for songs in the collection as a filterable attribute would provide a mechanism for examining shifts in song structure and content over time.

In order to accommodate so many songs, it would be necessary to change the layout algorithm as well. The force directed layout we use has recursive function calls which, when there are so many nodes and edges, loops countless times and causes the java stack to run out of memory. Some other layout algorithms we looked at to counter this were non-recursive force directed layouts and multi-dimensional scaling.

One final piece of future work that we want to propose is an experiment suggested by our advisor to try and adapt the results of our program to reflect what our potential users would appear to expect. Given time, we would survey a large sampling of individuals, asking them to rank pairs of songs by their perceived similarity. In the event that the survey group agrees on an ordering scheme for certain songs, the application's similarity metrics could be altered in an attempt to replicate the results as closely as possible. It would be interesting to note whether the outcome of this exercise would be more or less favorable than what we currently have.

7 Conclusion

Our project investigates the feasibility of comparing songs based solely upon the contents and structure of each song’s lyrics. We use these comparisons in order to create a visualization that users can interact with to try and discover interesting patterns and trends within the data. Additionally, users can locate songs that they enjoy listening to and use the application as a recommendation service.

The visualizations produced by the application can certainly be interesting to researchers looking for patterns within the data. Going by the examples discussed in 5.1.2, it is possible to surmise that there are a higher percentage of rap songs about hate than pop songs. Clearly, this is not definitive and a more formal study would have to be conducted to draw such a conclusion; nonetheless, this tool provides potential researchers with the power of visualizations – looking at data graphically to easily discover clusters and trends. Unfortunately, our 5605 songs are mostly inadequate for researching needs. A larger sample size from a wider variety of genres and artists, along with additional details about songs such as year, would provide a much more reliable foundation to make claims upon.

As a recommendation system, we had mixed results. When users were polled, the amount they liked songs recommended to them was slightly above average, while they felt the songs recommended to them were not very similar to each other. However, the goal of a recommendation system is not necessarily to recommend songs that are similar but to recommend songs that a user will enjoy. In this regard, we were successful to some extent. It is possible that with an updated similarity algorithm and a larger collection of songs, the quality of recommendations could greatly increase. No recommendation system is perfect and that we were not below average means this method of procedurally generating recommendations should not be ruled out. Some current recommendation systems, such as Pandora.com, rely on humans to manually input large amounts of information about each song in order to generate the recommendations. If this process could be automated, businesses could potentially save large amounts of money.

We believe that there is much to learn through analyzing our project’s methodology and implementation. Our application provides a strong foundation in comparing and visualizing text documents. Through future work, this application could be greatly improved as both a research tool and a recommendation service.

Appendix A Usability Questionnaire

Thank you for helping us with our project by volunteering to take part in this brief usability test. Your feedback is important to us. Before we begin please answer the following introductory questions.

1) Please rate your prior computer experience on the following scale.

1	2	3	4	5
(low)				(high)

2) Please list any genres of music you like listening to (list as many as apply).

3) Please list any genres you specifically do not like (list as many as apply).

We are now ready to begin. The questions on the following pages will be used later on during the evaluation of the program.

1) Write the title and artist of a song you enjoy listening to or have heard of that you were able to find in the application.

Title:	Artist:

2) Write the title and artist of the first related song that you will be listening to in the blank below. Rate how much you liked listening to the song and how similar your think it is to the song in 1.

Title:	Artist:	Edge weight:
		(We can help you find this value)

1	2	3	4	5
(disliked)				(liked)
1	2	3	4	5
(not similar)				(similar)

3) Write the title and artist of the second related song that you will be listening to in the blank below. Rate how much you liked listening to the song and how similar you think it is to the song in 1.

Title:	Artist:	Edge weight:
		(We can help you find this value)

1	2	3	4	5
(disliked)				(liked)
1	2	3	4	5
(not similar)				(similar)

4) Write the title and artist of a second song you enjoy listening to or have heard of that you were able to find in the application.

Title:	Artist:

5) Write the title and artist of the first related song that you will be listening to in the blank below. Rate how much you liked listening to the song and how similar your think it is to the song in 1.

Title:	Artist:	Edge weight:
		(We can help you find this value)

1	2	3	4	5
(disliked)				(liked)
1	2	3	4	5
(not similar)				(similar)

6) Write the title and artist of the second related song that you will be listening to in the blank below. Rate how much you liked listening to the song and how similar you think it is to the song in 1.

Title:	Artist	Edge weight:
		(We can help you find this value)

1	2	3	4	5
(disliked)				(liked)
1	2	3	4	5
(not similar)				(similar)

Before you go we would like to hear any feedback or opinions you have on your experience with our program. Your answers to these questions may help us steer future work related to this project.

1) Were you able to complete all tasks during the walkthrough?

A	B	C	E
(No)	(No, even with help)	(Yes, with help)	(Yes)

2) Please rate your confidence in being able to locate and view the properties of Come Together by Aerosmith in the program without any aid.

1	2	3	4	5
(not confident)				(confident)

3) Which comparison visualization did you find easiest to read and understand? Let us know if you would like to see them again before making a decision.

A	B
(Scaled words with connecting edges)	(Words with bars)

4) Compared to other software I have worked with I found this program to be…

1	2	3	4	5
(very difficult to use)				(very easy to use)

5) Please rate how strongly you agree or disagree with the statement in quotes.
"The menus and buttons in the program were logically organized and made it easy to find desired functionality."

1	2	3	4	5
(disagree strongly)				(agree strongly)

6) Please rate how strongly you agree or disagree with the statement in quotes.
"The name for each menu items and button made its function and purpose self explanatory."

1	2	3	4	5
(disagree strongly)				(agree strongly)

7) Overall, did you like using this program?

No	Yes

8) In the future would you use a program like this either for fun or for discovering new music?

No	Yes

Thank you for your feedback and participation. Feel free to use the space below for any additional comments you may have.

Appendix B Developer's Guide

This developer guide assumes that you are working with our application's code base and attempting to get your own custom data into the program. This custom data can consist of song lyrics, court proceedings, or any other type of text document collection. You could even use a collection of tagged items, with a tab delimited list of tags taking the place of a document's text. As long as you specify your collection in a consistent file format it should be relatively easy to obtain visible results in a short time.

Read Files

The first step to this process is to create your own DocFileReader class by implementing the provided DocFileReader interface. Because we expected our file format to change at some point during the course of the project, we used an interface to allow straightforward switching between file input algorithms. When implementing your own DocFileReader, override the fileToDoc(Path) method, which should produce a Doc object from the file specified by Path. Keep any assumptions about the file format within this class to keep file input separate from the rest of the implementation.

In your DocFileReader make sure to also implement the inherited getAttributes() and getAttributeClasses() methods. These should return the names of attributes in a Doc object produced by the fileToDoc(Path) method, as well as the class types for each respective attribute. This is how we handle dynamic attributes in our Doc object class, and how we load these attributes into our visualizations.

We encourage you to examine DocFileReader1 and DocFileReader2 for example implementations of the interface. DocFileReader1 reads an XML file format. DocFileReader2 reads a flat text file. You may actually find DocFileReader1 to be adequate for your purposes in the event you use a format similar to our own – just be sure to specific the attributes and attribute types in an options file, the location of which should be specified when constructing the DocFileReader1 object in Main.

Parse Files

The second step is to implement a text-parsing routine by altering the TextParser class. In particular, focus your attention on the parseString(Doc) method, which – given a Doc object – calculates base metrics and compiles unique words and frequencies from the document's text. This class may actually be adequate for most purposes in its current state, especially if you are just starting out and want to obtain visible results quickly. If you prefer a different set of metrics, however, this is the class you will want to copy and change. Note that stop words – words that you do not want included in frequency counts – should be specified in an options file, which gets passed to the TextParser when it is constructed.

Calculate Relevance

The RelevanceCalc interface is used to specify a class that can be used to compute the relevance values in a collection, or corpus, of documents. Expect the calculateBetween(Corpus, Doc1, Doc2) method to be called for every pair of Doc objects in the specified Corpus. When called, calculateBetween(Corpus, Doc1, Doc2) should use the metrics determined by TextParser to produce a value for the similarity relation between Doc1 and Doc2. RelCalc is an example implementation of the RelevanceCalc interface. Note that because the relevance between Doc1 and Doc2 is commutative (calculateBetween(Doc1, Doc2) = calculateBetween(Doc2, Doc1)) only half of the calculations are actually performed when the program is executed.

See What Happens

When you have made the above additions and modifications, change all instantiations of our implementations of these interfaces to your own in the Main class. We recommend testing with a small set of documents to begin with (less than 500) because the program can take some time to perform all preprocessing before showing the visualization. You may or may not need to disable the code that outputs processed data to fast-loading files, depending on what other changes you made to the system.

With any luck you should be able to see some visible results. You will probably have to alter certain values such as edge weight cutoffs and metric weights to obtain the exact outcome you may be looking for, but this should serve as an adequate starting point. If you only change the parts of the system that we have outlined above, all of the tools for evaluating and exploring comparisons should still function. Good luck!

Appendix C NSF Grants Side Project

Approaching the end of the implementation phase of our project, our advisor asked us if we could try to adapt our project to a different input collection. The goal of this side project would be to generate word clouds of popular terms in the abstracts of National Science Foundation (NSF) grants for the past five years.

The data we were provided was accessed through a website that allowed exporting of the abstracts in XML format. The resulting XML consisted of a single file, however, so the first step was to split the data into single XML files for each abstract. Data like year, primary investigator, institution, and grant amount were read in as document attributes; the abstracts themselves were used as a document's text. Overall we were very pleased with how simple it was to adapt our input schema to a different set of data in our existing implementation. The steps explained in the previous appendix are based upon this very process.

Once we had read in the abstracts we were then able to obtain visible results, although their worth was questionable. Considering that our previous work had resulted in metrics such as number of syllables, line and word repetitions, and number of lines – which were very specific to song lyrics – we decided to base similarities for our Global Force Directed View only on the percentage of words two abstracts had in common. Figure 38 shows an example of the Global Graph View visualization coloring abstracts by year and increasing the size of abstracts with the search term "software" in their text.

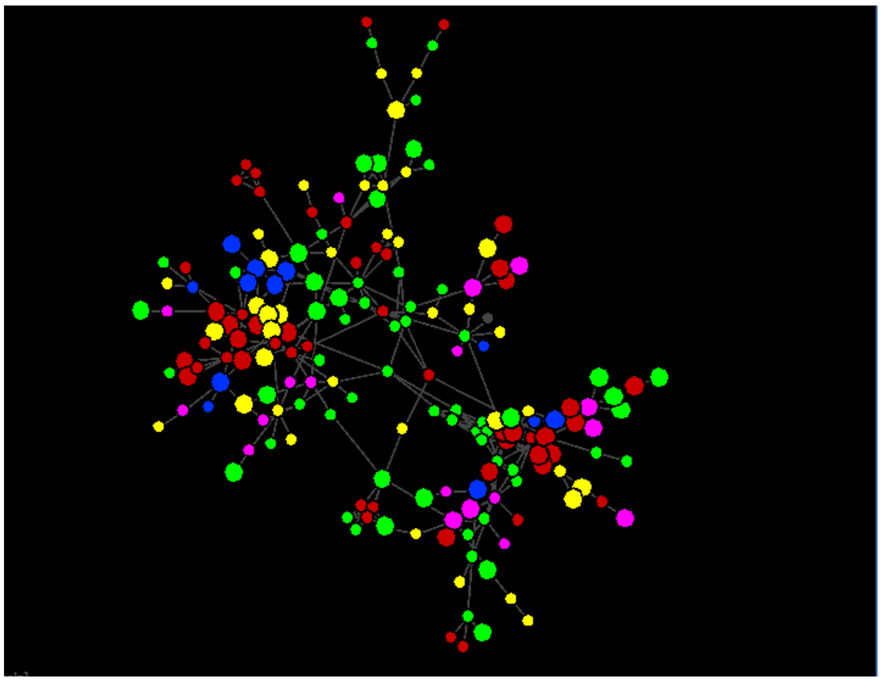


Figure 38: A screenshot of the Global Graph View of NSF abstracts with filters applied. Blue, purple, red, yellow, and green represent abstracts from 2005, 2006, 2007, 2008, and 2009 respectively. Larger nodes indicate occurrence of the "software" search term.

Our existing functionality allowed us to produce word clouds for individual abstracts via the Document Properties Window; however, the deliverables for this work would have to be word clouds for abstracts of a particular year. To accomplish this we implemented a new Word Cloud class which would take as input an ArrayList of documents. The Word Cloud class would then determine the overall frequencies of the unique words across all documents and produce the visualization as normal. An alternate Main class was also created to add functionality for launching Word Clouds by year without altering the work we had done previously (because year was not a valid attribute for a song). Figure 39 shows an example word cloud.

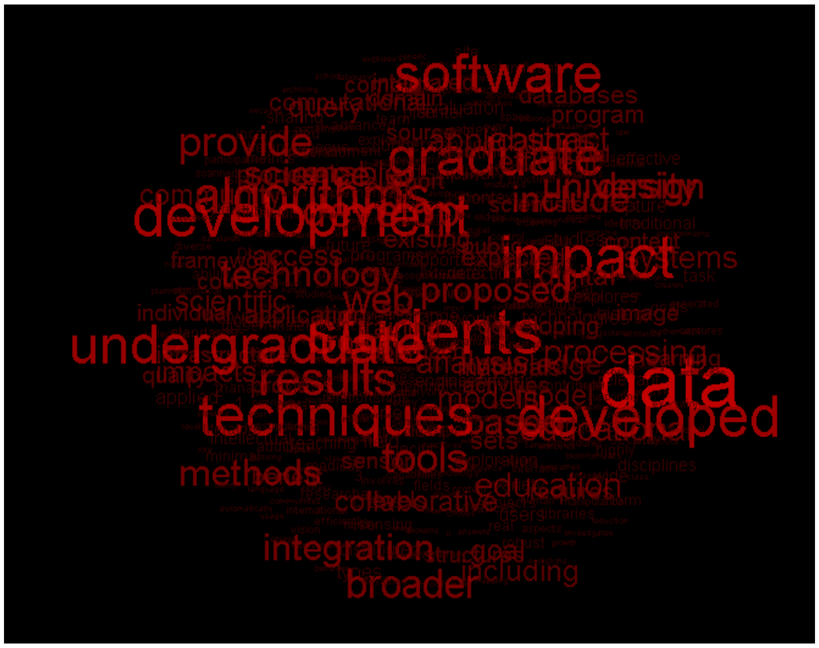


Figure 39: A screenshot of a Word Cloud of the most popular terms in abstracts from the year 2005.

The resulting Word Cloud visualizations were highly cluttered. The most popular terms were often unimportant ones like “impact” and “research” which appeared in virtually all abstracts. To eliminate extra words we created an updated stop word file that would eliminate a wider range of words than were being removed for songs. Additionally, because the resulting Word Cloud was still filled with unwanted words, we added functionality for selecting terms in the visualization and exporting them to a file that could be appended to the stop word list. This manual process took some time but was a way of eliminating only the undesired words.

The results of this side project were mixed. It was difficult to create useful Word Clouds in the time available due to the challenge of creating accurate stop word lists. We did, however, learn a number of things from this brief departure. The first is that our program is adaptable in the way it can read in new document collections. The second is that the difference between types of text documents can be very drastic and can require a complete rethinking of relevance measures and stop word lists. Third and finally we believe we can establish that with some additional work our program has the potential for use outside of our chosen area of focus.

Bibliography

[1] Wattenberg M. (2005). Baby Names, Visualization, and Social Data Analysis. *Proceedings of the IEEE Symposium on Information Visualization*. October, Minneapolis, p1-7.

[2] Viégas F, Wattenberg M, Ham F, Kriss J, & McKeon M. (2007). Many Eyes: A Site for Visualization at Internet Scale. *IEEE Transactions on Visualization and Computer Graphics*. Vol 13, Issue 6, p1121-1128.

[3] Weber W. (2007). Text Visualization – What Colors Tell About a Text. *Proceedings of the Information Visualization Conference.* Sacramento, July, p352-362.

[4] Yee K, Fisher D, Dhamija R, and Heast M. (2001). Animated Exploration of Dynamic Graphcs with Radial Layout. *Proceedings of the IEEE Symposium on Information Visualization*. October, San Diego, p43-50.

[5] Heer J, Card S, Landay J. (2005). Prefuse: a Toolkit for Interactive Information Visualization. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* April, Portland, p421-430.

[6] Linden G, Smith B, and York J. (2003). Amazon Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing*. Vol 7, Issue 1, p76-80.

[7] Glaser W, Westergren T, Stearns J, and Kraft J. (2006). *Patent No. 7003515.* United States.

[8] Navarro, G (2001). A Guided Tour to Approximate String Matching. *ACM Computing Surveys*. Vol 33, Issue 1, p32-88.

[9] "Stop Word." Oxford English Dictionary. 2nd ed. 1989.

[10] Berenzeig A, Logan B, Ellis D P W, and Whitman B. (2003). A large-scale evaluation of acoustic and subjective music similarity measures. *Proceedings International Conference on Music Information Retrieval (ISMIR)*. Vol 28, Issue 2, p63-76.

[11] Fox C. (1989/1990). A Stop List for General Text. *ACM SIGIR Forum*. Vol. 24, Issue 1-2, p19-35.

[12] Jamasz M, and Szpankowicz S. (2001). Roget's Thesaurus: A Lexical Resource to Treasure. *Proceedings of the NAACL Workshop WordNet and Other Lexical Resources workshop*. Pittsburgh, June, p186-188.

[13] Knees P, Schedl M, and Widmer G. (2005). Multiple Lyrics Alignment: Automatic Retrieval of Song Lyrics. *Proceedings International Conference on Music Information Retrieval (ISMIR)*. September, London, p564-569.