N12232709

Visualizing the Temporal Nature of Music: Extracting, Matching, Clustering, and Representing Chord Progressions

Zeeshan Lakhani

Submitted in partial fulfillment of the requirements for the
Master of Music in Music Technology in the
Department of Music and Performing Arts Professions in
The Steinhardt School New York University
Advisor: Dr. Juan P. Bello [DATE:2009/12/08]

"The point of visualization is to expose that fascinating aspect of the data and make it self-evident." (Fry, 2008)

Table of Contents

1. **Introduction**

1.1. **Motivation and Goals**

The current influx of musical information has become a pandemic. Whether that information emanates from an actual audio signal or is related via a determined system using metadata or tags, the quest for gathering, analyzing, and distributing data is at a frenzied pace. Furthermore, music has become more visual in the digital age. With a few clicks online, one can get a graphical layout of their listening history, a connected view of other people with similar listening trajectories, and a breakdown of song's musical or even lyrical structure. By attempting to retrieve musical information and then *see* it in some specific way, new relationships, patterns, correlations are formed.

The original motivation for this thesis was to create/program a graphical visualization that grouped shared musically-related patterns in a database (a collection of songs) over time. *Time*, as you will read, is the crucial element here. Many graphical music-based representations focus on playlist discovery, historical or genre-related similarity chains, and various sorts of categorization. Music as a time series is either clustered differently, erasing the time element in the view, or depicted in its more compositional form--notes. Additionally, these field-related visualizations, including those that focus on a musical piece's content and/or audio-driven features, tend to represent sequential musical similarity or *shape* as a whole item, one image or sketch that measures or equates to one or a series of tracks. What's being proposed here is something more linear and geared toward a clearer analysis of music.

After much deliberation, the overarching goals of the work developed for this paper were to visualize musical information, specifically harmonic content (e.g. chords and keys in Western notation) in a sequential/temporal way that provided both interactivity for the user and a better understanding of the structural relationship amongst a collection of harmonically related songs. To accomplish these goals, a prototype application was created that is comprised of three parts:

1) An offline database of chordal material (12 major and 12 minor chords) accumulated via the extraction of chroma features and the use of Hidden Markov Models from a set of remixes that pertain to one original track in this case. This database is linked to via:

2) a simple search engine that allows a user to call searches for specific chord progressions (one chord or multiple) based on either the comparison of the original track to its remixed offspring or on pure curiosity about the usage of certain progressions. Once a search is undergone,

3) the chosen pattern becomes the root for a visualization that *bifurcates* from the left and right of it, clustering the rest of the chords (textual strings) into lexicographically arranged subsequences/substrings. This visual representation aims to show the harmonic variations of the data with the user's *keychord* (like keyword) acting as the parent that all the other substrings (offspring) must share. The visual itself is text-based, depicting chords as their letter components. Parts 2 and 3 work in real-time.

The contribution of this work stresses the importance of music as a time series, visually representing chordal relationships as sequential text. Data is acquired, mined, grouped, and then displayed for interaction with the user. All the principal programming is done in Processing and Java (http://www.processing.org).

1.2. **The Structure of this Thesis**

This paper is structured in four sections. For the remainder of this introductory section, the focus will be the musical and visual foundations tied to this work. The second section concentrates on the details of the process outlined in the previous paragraph, beginning with the extraction and gathering of the musical database. This is then followed up with subsections discussing the search function and, more importantly, the clustering of the data to showcase basic sequential similarity. The third part of this thesis examines its most integral component, the visual approach, including a step-by-step run-through of the prototype application. The fourth section functions as a qualitative analysis of the end-work, including examples, conclusions/ thoughts, and possible leads toward future work.

1.3. **Time Series, Sequence, and Harmony**

Music is highly dynamic, constantly evolving over time, and shifts successively from sequence to sequence (in and out of key, repeating grouped sections) (Krumhansl, 2005). On a innate level, a musical piece is comprised of notes (tones). On a higher level, groups of notes form chords, from which then sequences and subsequences (movements) can be analytically defined. This is how musical structure takes shape, whether it be an abstract composition or something more heavily standardized (pop songs). This thesis focuses less on a song's self-evolving sequences and, instead, draws up comparisons amongst a set of songs on the chordal level. The chord-based relationships are composed of exact matches (one part exists in or shares another); measuring inexact similarity amongst the chords themselves, e.g. how close is *A minor* to *B major* is not in effect for this project, but serves as discussion for future work.

The reason for using harmonic content as a representational factor is threefold. Firstly, such features represent more of a mid-level approach that reaches a higher level of semantic complexity and are more definitive than lower-level features that deal with perceptual qualities like timbre (Bello, 2005). Secondly, as a string-based visual tool, chords translate better because they can be mapped and linked symbolically and literally as letters, which are recognizable to musicians and non-musicians alike, stressing similarity and distinct correlations.

One clarification should be noted. Music is considered a time series because it obviously takes place over a certain interval of time--start to end. The visual approach to this thesis doesn't attempt to showcase time as a hardline component, i.e. *A major* takes place at a given second or millisecond. Instead, time is used as a means of keeping order between chord changes, whether the initial chord begins at zero or ends after the two minute mark. A more theoretical model that describes time series as it is applied here exists in basic musical harmonic analysis.

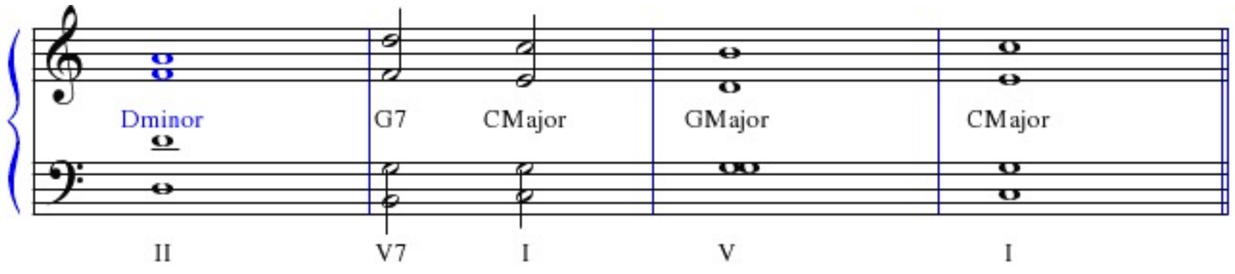


Figure 1.1: harmonic analysis with Roman numerals.

Time appropriates the music sequentially. Chord movement is stressed over all things else.

1.4. **A Brief Foray into Musical Data Visualization**

The most unique and compelling aspect of this thesis is that of the visualization. Musical data visualizations abound in both popular and information retrieval sectors. A basic overview of what exists for the facets of music discovery can be found in the presentation *Using Visualizations for Music Discovery* (Lamere, 2009). The presentation contains a breakdown segmenting types of data into a four quadrant representation with history/genealogy at the top and social/acoustic similarity at the bottom of the y-axis.

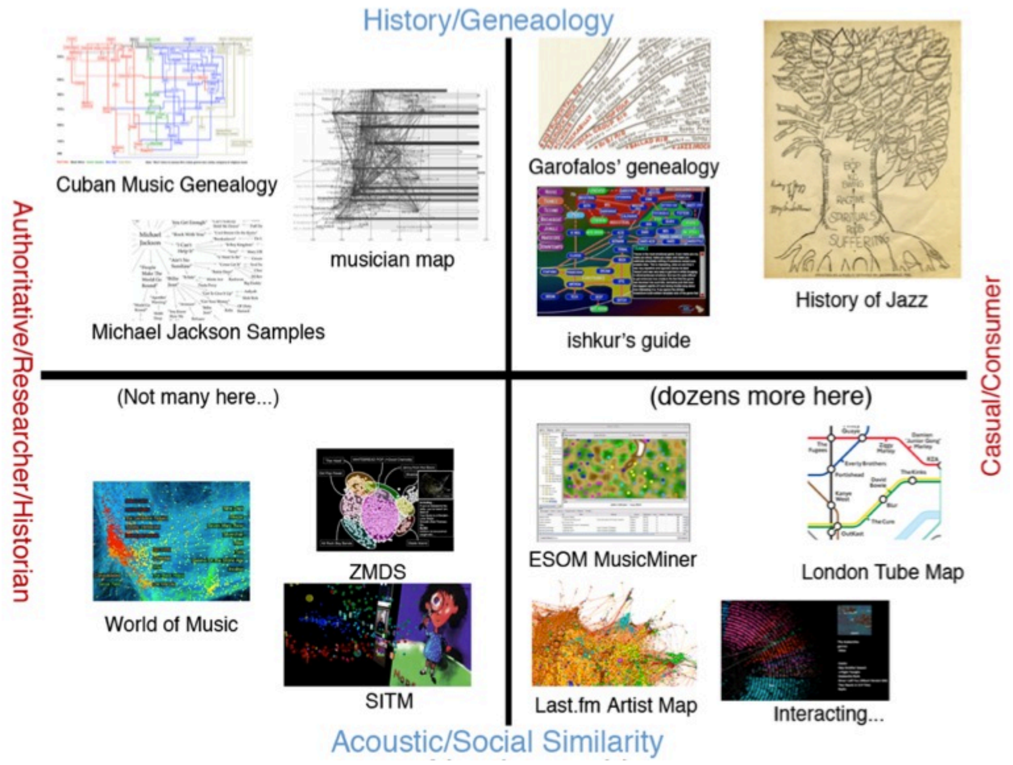


Figure 1.2: Visualization of Visualizations and where they fit in terms of informational content (Lamere, 2009).

Visual representations that protrude structural and signal related content are not as widespread as those covering social matches, playlist/genre searchers, and recommenders. Plenty of the playlist recommenders and similarity definitions are founded on algorithms and machine learning principles that revolve around audio signal features; however, they tend to only serve the user/listener (what's being or should be listened to) and not the essence of the music itself.

Many music-based visualizations that attempt to actually show song-related information (rhythm, harmony, etc...) tend to emphasize the whole sketch as a representation of an audio piece or a collection. Basically, these visualizations are without further analysis or details within the graphical elements themselves. *Figure 1.3*, below, serves as an example.

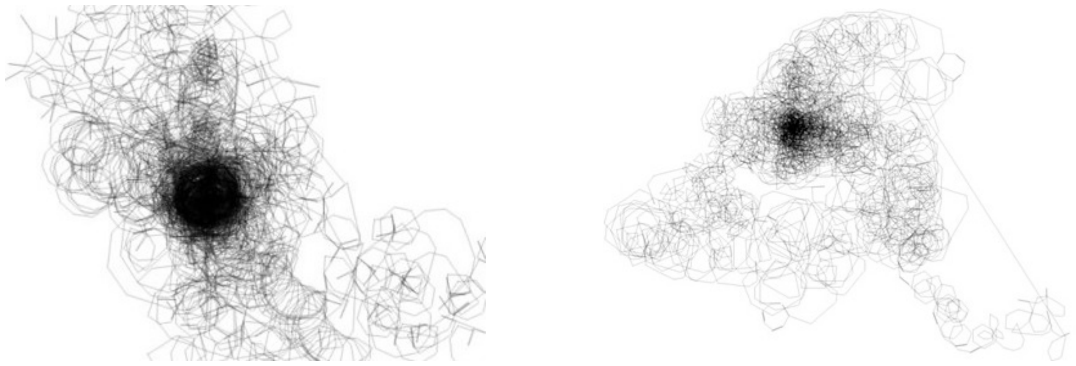


Figure 1.3: 2 Stills (songs) from Jake Elliot's *Pop Sketch Series.*

Elliot's *Pop Sketch Series* is aesthetically pleasing and that's his point. Sketches are *drawn* via changes in pitch (correlating to changes in angle) and volume (correlation to length) (Elliot, 2006). Even though musical information creates the visual, the information itself becomes lost in the paradigm of the aesthetic. The songs definitely look different, but they only provide a very general comparison between the structures of the songs themselves. The line that separates visualizations that are *good to look at and interact with*, and those that promote innate musical relationships, can be substantial.

In creating an information visualization, more detail usually leads to more navigation, which then leads to more use. When dealing with piles and piles of data, it's always difficult to avoid *information overload*, but now more than ever before, the ability to represent complexity is at an all-time high (Fry, 2008). Resolution is better. Computers are faster. And, for these reasons, the expansion and interaction of musical visualizations have truly become integrated, hybrid systems.

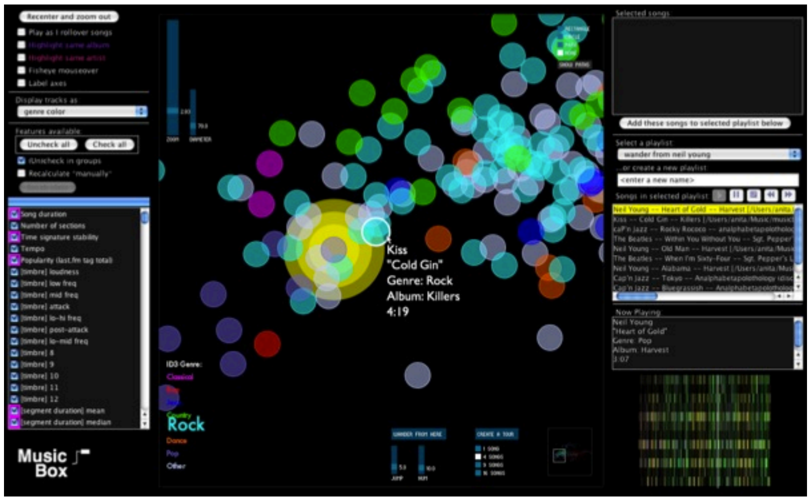


Figure 1.4: A still from Anita Lillie's *MusicBox.*

The mapping and scaling of symbols and/or nodes which represent musical data are the most integral features in the design process; it's how the content is expressed and made discernible to the viewer. Musical content can be linked, exchanged, clustered, and segmented via tree hierarchies, literal maps, graphs, scatter plots, radial networks, and a hybrid of other concoctions (Lamere, 2009). A good example of and argument for the importance of mapping, spacing, and the interactive affect of hybrid visualizers is exemplified in *Figure 1.4*. *MusicBox* is large in scope and is not specifically related to the concepts of the prototype created for this thesis; yet it provides an extensive landscape for the analysis of musical content, song similarity, and playlist recommendation. Its arrangement of music in space allows for dynamic feature selection, forming distances amongst song nodes (circles) in relation to metadata, harmony, timbre, tempo, and song duration (Lillie, 2008). While the application allows for the grouping of similarity based on audio signal information, you are not able to actually see the similarities within the song (each node represents the whole song itself). A key feature, however, is the song representation graph that exists in the lower right corner of the applet. It provides a look at the timbre and detectable notes of each track clicked. Providing both views only further enhances the breadth of such visualization applications and allows for the musical relationships to be formed from multiple perspectives. Clarity and detailed spatial information are definitely two of the most attractive principles in the field, and they will discussed further in section 2.

When it comes to representing music as a time series, the field is just not inundated with such representations of order. When time is displayed, it's more about time as a plotting point--a song or features distinguished *over* time--or as another type of genealogy or historical graph that is either void or limited in its visual exposition of music data, even if analysis and connections are being compiled *under the hood*. The reason for this thesis grew out of the fact that such temporal systems either have not been very successful or just haven't gained much popularity.

1.5. **Related Work**

1.5.1. **Martin Wattenberg's *The Shape of Song***

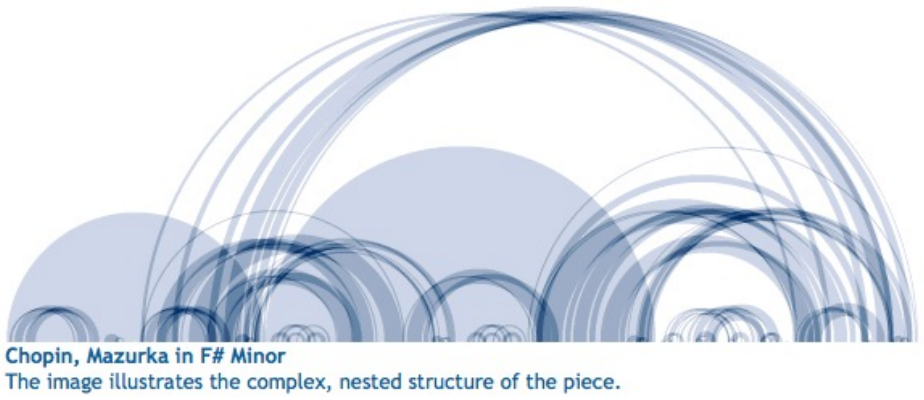

Chopin, Mazurka in F# Minor
The image illustrates the complex, nested structure of the piece.

Figure 1.5: Wattenberg's arc diagrams represent repeated subsequences.

This thesis has preliminary inspirations tied to Wattenberg's *The Shape of Song* software/ series, particularly its focus on time and visual coherence, and his work with arc diagrams in order to represent sequence structure by highlighting repeated subsequences (Wattenberg, 2002). *The Shape of Song* takes in MIDI files for analysis, which a user can upload to the site, and then creates an arc diagram that connects matching passages in the track; a *match* means that they contain the same sequence of pitches. An arc diagram connects only a subset of all the possible pairs of matching substrings by distinguishing essential matches on their relationship to repeated regions and if they are contained in the same fundamental substring of any repetition region that contains it. The connections are consecutive, focusing on the subsequence and not on every exact, note-to-note repeat. Translucency is used to make sure that no match is obscured by another.

The importance of Wattenberg's work is that it visually labels the repeated structures--harmonies--within a piece, which helps to describe musical complexity, order, and compositional form. Temporal order is kept intact and is a major force at play in the visualization. And, despite the fact that similarity is based simply on repeated note significance, "the resulting matching diagram reveals an intricate and beautiful structure" (Wattenberg, 2002). Though T*he Shape of Song* focuses on a piece's self-structure and not on a set of music, it has influenced the concepts of time series and musical pattern matches that exist in this work.

1.5.2. **Ben Fry's Work with Haplotypes and Bifurcation Plots**

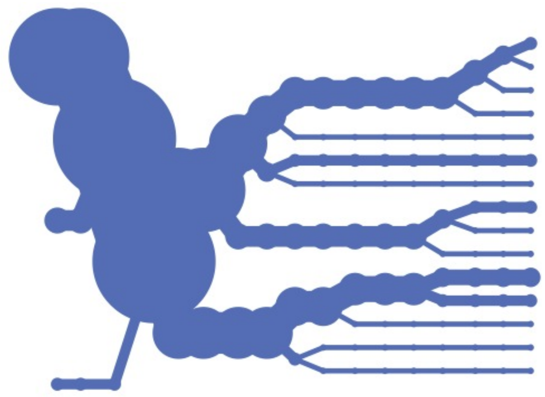


Figure 1.6: A SNP plot by *Sabeti et al* (2002) that Fry uses as his initial example about bifurcation.

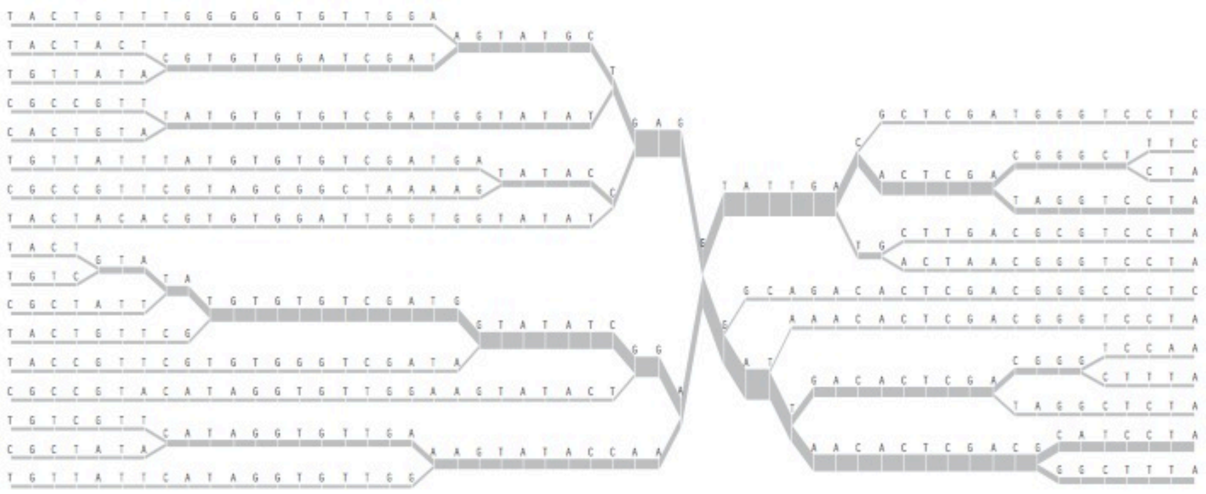


Figure 1.7: Fry's clearer interpretation of *Figure 1.6.*

Ben Fry's dissertation and emphasis as a whole deals with data visualization in its many facets and incarnations. The above examples refer to the research and diagrams that he has integrated in his work with human genome sequences and SNPs (single nucleotide polymorphisms) (Fry, 2004). *Figure 1.7* showcases how the work in *Figure 1.6* suffers from an obscurity of the exact data by elucidating where clusters and branches occur. The diagram bifurcates, forks outward, from a center node, and continues branching until it reaches an ending point in every string of information. Genome visualizations use such a device because it easily defines parent nodes and the offspring that follow it (*Figure 1.6* depicts regions where SNPs were preserved, regarded as a sign of higher natural selection). The thickness of the lines is determined by the number of offspring that a specific parent/node contains. The inherent visual structure and clustering of the patterns (like a tree diagram) served as the initial guide and foundation of the prototype created for this thesis. One caveat to mention is that Fry's example deals with strings that are of fixed length; our prototype does not (instead, focusing on song lengths).

1.5.3. **The Tag Cloud**

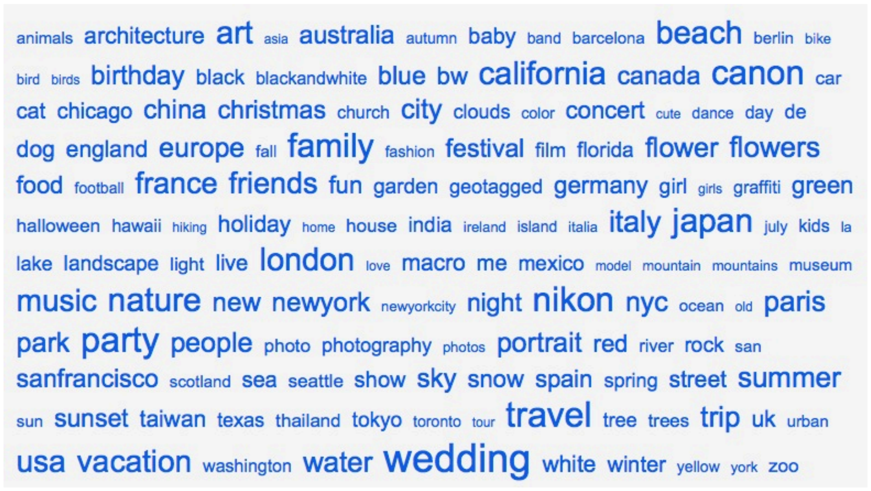

animals architecture art asia australia autumn baby band barcelona beach berlin bike
bird birds birthday black blackandwhite blue bw california canada canon car
cat chicago china christmas church city clouds color concert cute dance day de
dog england europe fall family fashion festival film florida flower flowers
food football france friends fun garden geotagged germany girl girls graffiti green
halloween hawaii hiking holiday home house india ireland island italia italy japan july kids la
lake landscape light live london love macro me mexico model mountain mountains museum
music nature new newyork newyorkcity night nikon nyc ocean old paris
park party people photo photography photos portrait red river rock san
sanfrancisco scotland sea seattle show sky snow spain spring street summer
sun sunset taiwan texas thailand tokyo toronto tour travel tree trees trip uk urban
usa vacation washington water wedding white winter yellow york zoo

Figure 1.8: Flickr Tag Cloud.

An significant part of the *look* of our proposed visualization is inspired by the tag cloud and its frequency-count-based approach. Additionally, since the prototype application displays chords literally as text, we decided to use the tag cloud's depiction and calculation of size/font as a way of showing how many offspring a specific substring holds (more on this later). The tag cloud is omnipresent, and its initial burst onto the visual scene is not definitively clear. *Flickr's* use of it as a search and organization system gave it prominence. Tag-clouds have been used to compare everything from word usage in historical Presidential speeches (Mehta, 2006) to shared bookmarks amongst individuals. They tend to be easy to understand, providing clarity.

2. **Process**

2.1. **Audio features, Data Extraction, Gathering and Reduction**

2.1.1. **Remixes as a Start**

At the onset of this thesis, the goal was to choose a musical database that would consist of harmonically similar content--i.e., similar chord progressions. In other work focusing on musical similarity and chord sequences (Bello, 2007), cover songs were used, but more as a means to measure a beneficial model for musical similarity. For this project, the purpose was to get a sense of sequential movement and the location of specific patterns. Therefore, it was initially decided to acquire a database that consisted of one track, *the original*, and remixes of that original track. Such an assemblage of material would mostly share the same set of chords and information, whereas covers can produce difficulty due to tempo variations and key changes, which would call for a key matching technique. Remixes, for the most part, stress the concept of editing and moving pieces--sequences--around. This is not always true, as the database itself contains tracks with additional elements, e.g. guitar, more noise. Still, elements like the chorus will show up more often than not and certain remixes will almost be identical to the original, except for the stripping out of a vocal section or drum loop.

We gathered two[1] collections (databases) for this project, each consisting of an original Nine Inch Nails track (1-*Echoplex,* 2-*The Hand That Feeds*) and thirty remixes of that track. All of the tracks, including each original's source files, are free for download at http://remix.nin.com. The duration of *Echoplex* is 4:45. Its remixes range in length between 56 seconds on the short

[1] The second collection was taken on as means to provide an emphasis on the generalizability of the approach discussed in this thesis. The Echoplex collection will serve as the main database for this paper. However, an example from the The Hand That Feeds database will be shown and discussed in section 4.1.

side and 8:17 on the long side. *The Hand That Feeds* clocks in at 3:31. Its remixes range between 2:37 and 7:55. Thirty is just an arbitrary number as an initial prototype set. The reasons for choosing *Echoplex* and *The Hand That Feeds* above other Nine Inch Nails' originals were because they are somewhat less noisy and more pop-oriented (chordal) than the others. The visualization application itself can read from any database of musical material, but similarities may or may not exist in certain collections of differing songs. Once I organized the material, the next step was to extract the chord sequences.

2.1.2. **Chroma Features and Hidden Markov Models (HMMs)**

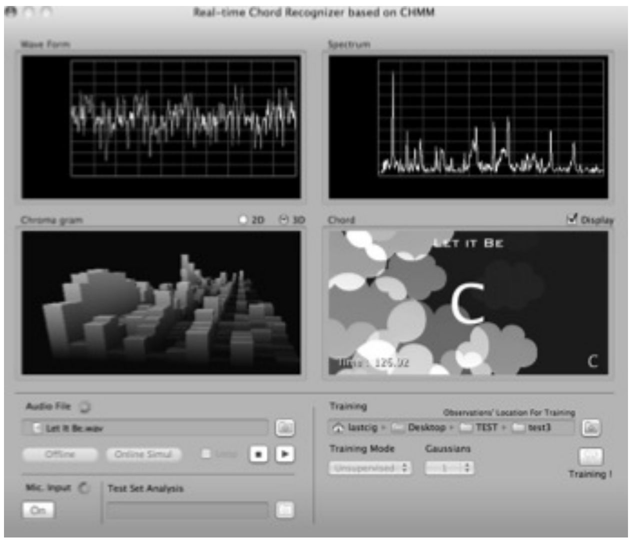


Figure 2.1: Realtime Chord Recognizer application.

Pitch Class Profiles (PCP), better known as chroma features, represent the distribution of a signal's frequency content (spectrum) across the 12 semitones of the chromatic scale (Fujishima, 1999). Being that these features/profiles deal with harmonic content, they are used frequently in modeling musical harmony.

To extract the chordal data needed for the visual application, I used the implementation and system (*Figure 2.1*) proposed in (Cho, 2009) and (Bello, 2005), which uses the chroma features as observations on a 24-state hidden Markov model (HMM), with each state corresponding to a major or minor triad. More information about the HMM can be found in (Rabiner, 1989). The process for computing the chroma features of the raw audio begins with the calculation of the Constant-Q transform for spectra analysis, where the frequency domain channels are logarithmically spaced (unlike the linearly spaced Fourier transform). Once the transform is solved for, it can be folded into chroma via this formula:

$$O_b = \sum_{m=0}^{M-1} \mid X_{cq}(b + m\beta) \mid \quad (2.1)$$

Where X_{cq} is the Constant Q transform, β is the numbers of bins per octave, b in $[1, \beta]$ are the chroma bin indices, M is the total number of octaves from a minimum frequency. The feature vectors are then quantized into 12 bins using a Gaussian filter bank. The process from a sequence of chroma vectors to actual chord labels can be summarized in *Figure 2.2*.

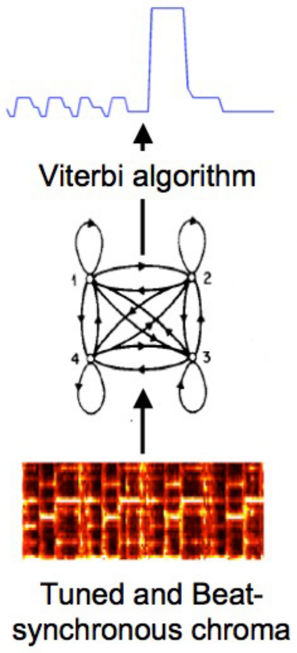


Figure 2.2: Chroma - HMM (calculation of parameters) - decoding through Viterbi algorithm, where the final sequence of triads is obtained (Bello, 2005).

The model parameters of the HMM discussed in (Bello, 2005) and (Cho, 2009) are initiated using musical knowledge, specifically chord templates and chordal relationships designated by the doubly-nested circle of fifths.

Using Cho's *Realtime Chord Recognizer*, a model was trained in an unsupervised approach for each collection (totaling two models). The labeled chordal data for each track was obtained via the decoding of the model(s) by way of the Viterbi Algorithm. Because the database used consists of Nine Inch Nails music--i.e., rock, industrial, and noisy--the recognition system probably suffered from poor accuracy (manuel transcriptions were not made for testing) as the chordal movement in the songs is sometimes masked by electronic and distortion-laden effects/processing. However, since each model uses the same parameters and setup and each track is from a similar domain, their inaccuracies are shared.

2.1.3.**Reduction of Data Information via Echonest API**

After obtaining the extracted data, some songs totaled hundreds, if not over a thousand chord labels, depending on the length of the track. This was due to the frame length determined to represent each chord in the *Recognizer's* system. Seeing that times between chords were in the milliseconds, many of them were repeats. From what was understood about visual scaling, we knew that displaying chord sequences comprised of thousands of letters in space (specifically along an x-axis) would cause obscurity and an inability to acquire any pertinent information from the visualization. We wanted to reduce the data in a musical way, i.e. not just eliminate all repeats or arbitrarily cut sections out. To accomplish this, the *Echonest Developer's API* (http://developer.echonest.com) was utilized to perform beat analysis (retrieve the set of times of beats in a track--in seconds) on each song in the database(s). The basis of the algorithm's approach is found in (Jehan, 2005, p. 73). More information about the tradition of beat-synchronous features and beat-segmented representations can be found in (Ellis, 2006) and (Bartsch, 2001).

To call the *get_beats* function of the API, a third-party wrapper for Processing was used, developed by Melka (http://www.melkaone.net/echonestp5). Once the beat times of each track was obtained, a Java *Hashmap* (a key-value based array) was implemented in order to take in all the extracted chords between each consecutive set of beat-based time frames. As a simple example, say that a section of *Echoplex* consists of five chord labels between beat 7 and beat 8 (maybe one-second apart): *A A C A A*. The map would take in those five chords between the beats and represent them in a reduced fashion, without duplicates: *{A = 4, C = 1}*. Then, majority rank would determine the *final* chord. In this example, *A* would win out, and it would represent beat 7--if the system was rounded downward. Once we reduced the data, chordal sequences ranged in length from 400 to 800 chord labels, a much more manageable number.

2.2. **Search Function**

2.2.1. **Exact-Match Chord Search**

In this stage, songs from the database(s) were just strings of characters. With the labeling complete, the next step concerned pattern matching and finding repeating sections (over the database(s)). There existed a need to measure the songs' harmonic similarity and create a system that would lend itself to the clustering and visualization of shared chord patterns--highlighting structural connections. We'd wanted to express such connections in a manner similar to what was mentioned in section 1.5.2. Tracks would need to share one specific node/chordal progression (pattern) and then bifurcate from that node, bifurcating further and further, until patterns were no longer shared and all the tracks reached their beginning (left-side) and ending (right-side). The original node could exist anywhere in time for each of the tracks--as long as it was contained in whole. Again, each song's time series, length, would be left intact.

Wattenberg had mentioned that despite the fact that the similarity of his system was limited in nature (exact matching), the resulting visuals relayed information about a musical piece's intricate structure. Defining musical similarity this way could be considered narrow in scope, being that *similar* music can be set in different keys and that approximate matches may hold more weight as a whole than exact ones. In contrast, (Orpen, 1992) makes the argument that approximate similarity is more important than pattern matching because musical experience appears to be dominated by "impressions of resemblance rather than impressions of discrete identity or equivalence." Additionally, the paper mentions that compositional methods place a greater emphasis on inexact variations rather than on exact formal transformations. With inexact conditions, a weighing of the similarity takes precedence. This pull, however, is also flawed, as the scaling, definition, and threshold of the weighing scheme could be biased or seemingly arbitrary depending on the smallest change in the value(s) of one of the factors.

For the purpose of this project, it was decided that the best route to take would involve exact pattern matching of the chord progressions. Firstly, this choice was made due to the nature of the material that comprises the database (Smith, 2001). Remixes are known for their rearrangement and stripping of sources from the original material. Many of the same harmonic patterns, the foundation, still exist; they may just be shortened, extended, shifted, or reindexed. Secondly, this thesis is aimed solely toward the visual domain, and, following Wattenberg's comment, even with a limited definition of musical similarity, exact connections can still be very informative, especially in regards to the fact that this project deals with visually describing a collection of songs.

In order to define the center node/progression of the diagram, an exact pattern search was implemented in the prototype application. Its result, as per the user, acts as the parent and determines how the visualization will be drawn up. More information about the implementation can be found in section 3 of this paper.

2.2.2. **Knuth–Morris–Pratt (KMP) String Searching Algorithm**

The results of the search are determined by the Knuth-Morris-Pratt algorithm, one of the two classical comparison-based matching algorithms for the linear-time exact matching problem. The various steps and procedures can be found in (Gusfield, 1997). Its original conception and documentation can be found in (Kunuth, 1977). To produce the visualization in real-time, as a bonafide application, this algorithm was used because it's much quicker than brute-force matching techniques, and though it's not as fast or as popular as the Boyer-Moore algorithm, it's more generalized for problems such as real-time string and pattern matching.

Complexity is the main issue at hand when KMP is compared to brute-force methods. Whereas a naive algorithm could take worst-case $O(nm)$ time, n: the length of overall text and m: the length of the pattern, the KMP search (with pre-processing to analyze the search space) can take $O(n)$ time. This is due to the fact that the algorithm makes use of the information gained by previous symbol comparisons; it avoids comparisons with elements in the text string that have been previously involved in a comparison with an element of the pattern string. Basically, it avoids backtracking, which is inherently redundant.

As a simple example, let's take four letter symbols: *A A B d*. If the user's search input was *B d*, the algorithm would initially start left to right and compare the pattern to *A A*, a quick mismatch. Instead of shifting over one slot and comparing *A B* to *B d*, the algorithm is aware that the final letter from the mismatch is *A*, and it shifts over two slots to *B d*. With previous knowledge, there is no reason to shift once over to the right because one *A* would automatically count as another mismatch. *Figure 2.3*, below, provides a visual example of the shifting that occurs. When a mismatch arises at position j, the widest border of the matching prefix of length j of the pattern is considered. $b[j]$ is the width of the border and is shifted along after the mismatch (mismatches are depicted in the grey segments).

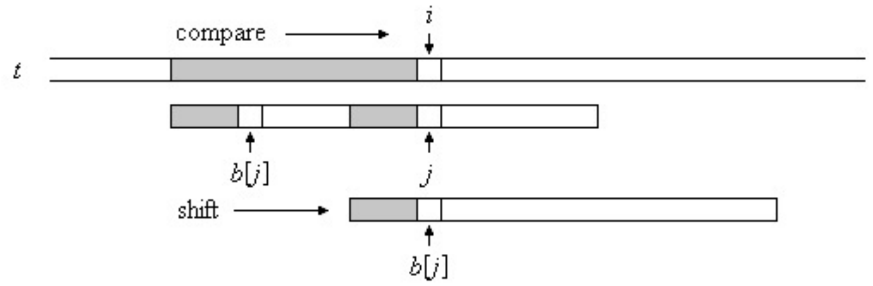
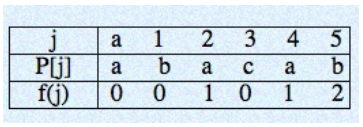


Figure 2.3: Shift of the pattern at position *j* when a mismatch occurs with the text string *(Lang, 2001).*

The preprocessing step pares down complexity by computing a failure function that indicates the largest possible shift in regards to previously run comparisons. It is defined as the length of the longest prefix of the pattern, P, that is a suffix of $P[i...j]$ and is calculated by determining the width of the widest border of each prefix, encoding repeated substrings inside the pattern itself. Basically, it is implemented in order to pre-search the searched pattern and compile a list of all the possible fallback positions that skip over a maximum number of mismatched characters without disrupting or bypassing potential matches. The example output of a failure function can be seen in *Figure 2.4.* After a pre-search, the suffix *ab* is the longest repeated and matched substring from the comparison of two strings. The failure function, $f[j]$, represents this knowledge with the ordinal *1 - 2*. As a preliminary run function, *ab* would seem like to the goto best match.



j	a	1	2	3	4	5
P[j]	a	b	a	c	a	b
f(j)	0	0	1	0	1	2

Figure 2.4: f(*j*) is the output failure function. The largest prefix is *ab* and it is also the suffix.

In terms of this thesis application, the KMP algorithm was used to match user searches, chord patterns, to each track's chordal sequence in the database(s). Mismatches were designated with a value of *-1*. After the first match is found, the algorithm stores the indices of that match in an array, *breaks* the iteration (loop), and then moves on to the next song. The reason for concluding each track's search at the first found match is a consequence of the fact that if multiple matches do occur between a pattern and track's chord sequence, then there would have to be a way to decide what the best or most important match is. Displaying all the matches would limit the ability of the clustering mechanism (next section) and distort a key point of the visualization: displaying structure from one main source common to a selection of songs, the essence of similarity in this application. Originally, we had wanted the search to operate on a broader and more interactive level, allowing the user to choose the location of the match (in songs that contained that match and multiple indexes of it) that they would like to see the diagram s*pring out* from. However, the concept, initially, would be tedious for a user and needs more fleshing out; it will be discussed further in section 4.3. Other factors in choosing the best match per track could be more probabilistic in nature, but such a decision in itself is another paper.

2.3. **Clustering Substrings using a Radix Tree**

2.3.1. **SubStrings and Branching Out from the Root**

Fry’s demonstration, discussed previously, shows shared substrings--strings or sequences of strings that stem from a parent source/string--that are clustered when they share with another string and then stem outward (unless they are always sharing with another string sequence). To *draw* the diagram, his system knew where the *ancestor* was (the main source in the center) and knew how to cluster and bifurcate over a defined space. The project described in this work is defined by a similar approach. To convey, *draw*, the data at hand, the system that best fit the concept presented here was a specific string-storing/clustering data structure.

A radix tree, also know as a Patricia trie/tree, is a specialized data structure that stores and clusters/merges a set of strings. Its origins and design are defined in (Morrison, 1968). It’s a structure defined by nodes and edges, where nodes are the objects and edges are the connections (the links). The radix tree is related to the more universally known trie data structure, which is a multi-way tree structure that is usually called upon for alphabetical tasks. The main difference between the structures is that a radix tree connects and derives edges with sequences of characters (substrings) by storing its position in the node, while a trie does so with single descendants or characters (using every key or string to find the next subtree), causing it to have a higher space-complexity.

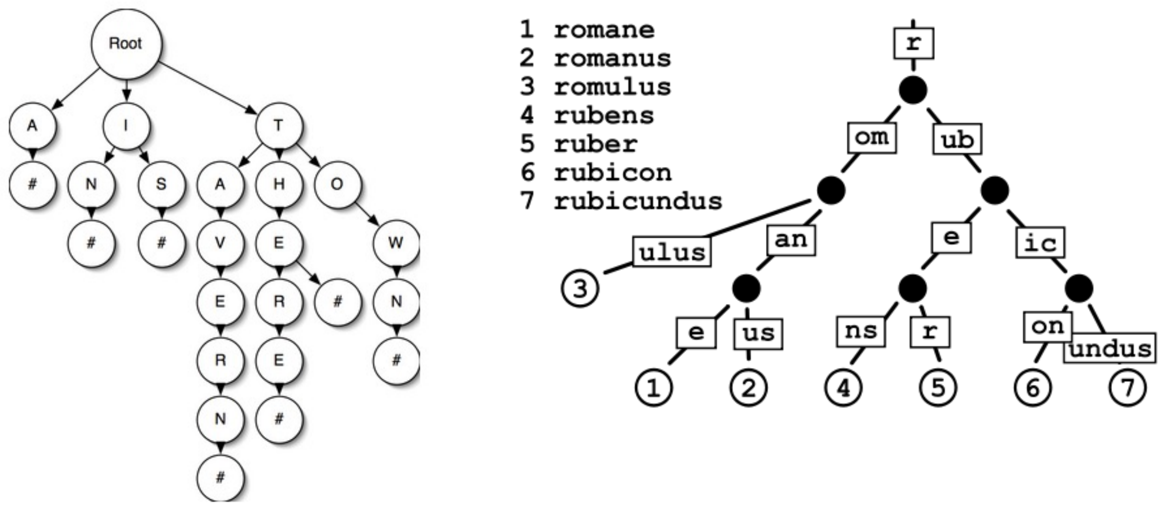


Figure 2.5: On the left is an example of a normal trie structure, which focuses on connections being found at the singular character level. On the right is an example of a radix tree, which connects substrings of characters. Both structures connect prefixes from common nodes.

While both structures are definitely applicable for storing and categorizing large dictionaries, the radix tree can be very useful for textual information retrieval, especially when dealing with larger strings of information and patterns of common substrings or subsequences. The set of operations that is supported by a radix tree (with a complexity of $O(k)$, where k is the number maximum length of all the strings in the set) includes the insertion and deletion of strings, lookup, prefix search, and the finding of a node's predecessor and successor.

Looking at the radix-tree-based diagram in *Figure 2.5,* the clustering of shared substrings heavily optimizes the diagram's spatial representation (it's much more compact than its counterpart to the left). In referencing a larger database, similar to the one used in this project, such a categorization of the strings goes a long way into the concepts of clarity and detailed spatial information that's been discussed previously. Additionally, even though the diagram above doesn't really compare the breadth of shared occurrences, it, as a data structure, shares much in common with and operates on the same system of *bifurcation* that is a vital building block of this work.

2.3.2. **Maintaing the Song Structure's Lexicographic Order**

```
|
 |-A
  |--A
   |---AACCA[AAAACCA]*
   |---BCC[AABCC]*
  |--cA[AcA]*
```

Figure 2.6: a hierarchical radix tree output, the strings inserted into the tree were *AAAACCA, AABCC,* and *AcA*.

Figure 2.6 provides a quick rundown that describes the data structure in action. A radix tree is a prefix-oriented tree where order plays a major part in how a string set is grouped and segmented. The breakdown of the sequences is said to follow suite *lexicographic,* alphabetic or dictionary, order. The inserted strings into the example above were (1) *AAAACCA*, (2) *AABCC*, and (3) *AcA*, with character case preserved. From left to right, the *bifurcation* of the strings begin. Seeing that all three strings begin with *A--A* then becomes the ancestor node from which the other branches grow. The next branch, two-dashes, is *A* again, which is the *A* shared between strings 1 and 2. Within and keeping true to lexicographic order, the rest of strings (the substrings) 1 and 2 bifurcate into the next tier, concluding there, as there are no more shares or clusters. The tree then moves back one tier to two-dashes to label the rest of string 3. Though a simple example, it stresses the concept of temporal order, as bifurcations can only occur in relation to the sequential ordering of the characters. The figure also emphasizes exact similarity and repetition, as each parent symbolizes an area of grouped substrings.

For this project, the radix tree data structure seemed like the best fit because in regards to chordal sequences, lexicographic ordering would serve as a means to easily show where changes occur. Again, chord progressions are a specific kind of alphabet. Additionally, this structure seemed like the best way to explain the work behind Fry's and Sabeti's bifurcation plots, which are tree-based in origin and founded on the same principles of parent/offspring relationships. And, since Fry's example was a major player in outlining the goals and overall feel for the visual display of this application, the radix tree made sense on various levels. To implement the data structure, we used the java implementation made available under the MIT license, located at http://code.google.com/p/radixtree.

3. **Visualization Approach and Implementation**

3.1. **From Tree to Visual Representation**

3.1.1. **Bifurcation Plot**

The operating procedure thus far--with more explanation in section 3.2--has consisted of *acquiring*, *parsing*, and *filtering* data, and then *mining* it to measure and describe patterns, similarities. The italicized actions are four of the seven tenets that Fry uses to explain the ins and outs of visualizing data (Fry, 2008). The other three: *represent*, *refine*, and *interact* make up the rest of this paper. We've described how the search function is wielded in order to create/become the ancestor for a radix tree, which then puts forth the work in grouping and bifurcating the rest of the string elements in the database via lexicographic determination. Thus far, everything has been explained on a theoretical level. Now, the visualization, the key to this thesis, takes shape.

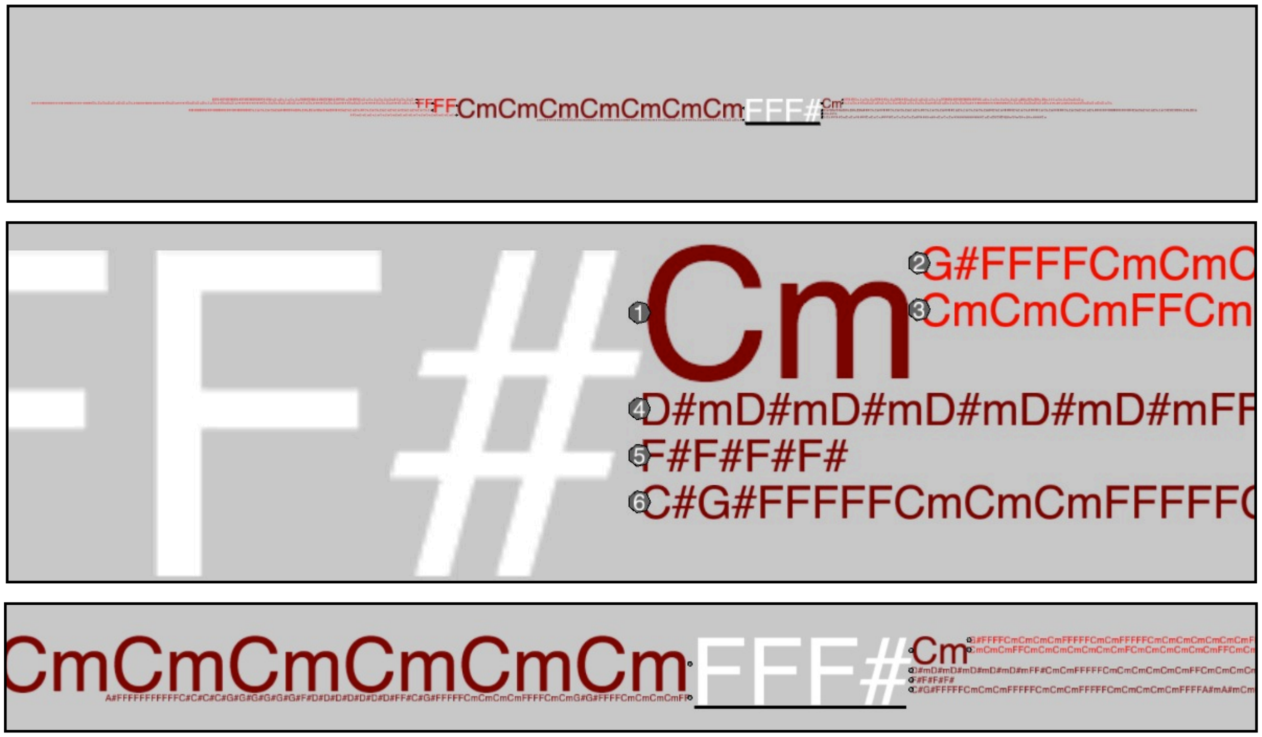
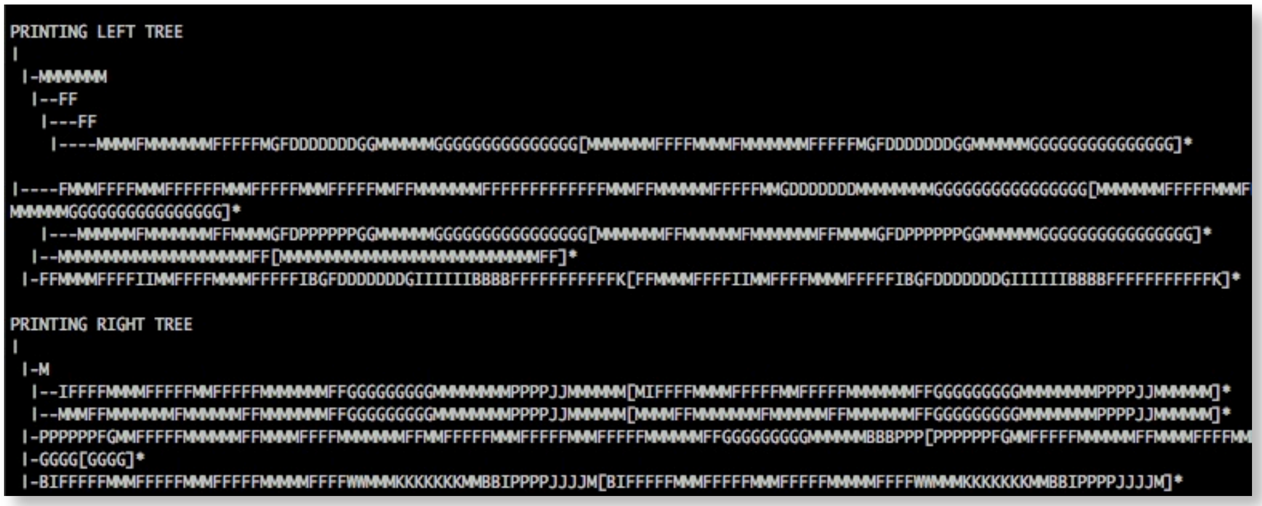


Figure 3.1: (Top) A whole-picture still of the prototype visualization, using *FFF#* as the main root (ancestor), (Middle) A scaled-in version of the same visual, focusing on the right side of the main node and showing the numbering of each string, (Bottom) A centered and somewhat scaled-in view of the same visual.



```
PRINTING LEFT TREE
|
 |-MMMMMM
  |--FF
   |---FF
    |----MMMFMMMMMMFFFFFMGFDDDDDDDGGMMMMMGGGGGGGGGGGGGGG[MMMMMMFFFFMMMMFMMMMMMFFFFFMGFDDDDDDDGGMMMMMGGGGGGGGGGGGGGG]*

|----FMMMFFFFMMMFFFFFFMMMFFFFFMMMFFFFFMMFFMMMMMMFFFFFFFFFFFFFMMMFFMMMMMFFFFFMMGDDDDDDDMMMMMMMGGGGGGGGGGGGGGGG[MMMMMMFFFFFMMMF
MMMMMGGGGGGGGGGGGGGG]*
   |---MMMMMFMMMMMMFFMMMMGFDPPPPPPGGMMMMMGGGGGGGGGGGGGGG[MMMMMMFFMMMMMFMMMMMMFFMMMMGFDPPPPPPGGMMMMMGGGGGGGGGGGGGGG]*
  |--MMMMMMMMMMMMMMMMMMFF[MMMMMMMMMMMMMMMMMMMMMMMMMFF]*
 |-FFMMMMFFFFIIMMFFFFMMMMFFFFFIBGFDDDDDDDGIIIIIIBBBBFFFFFFFFFFFK[FFMMMMFFFFIIMMFFFFMMMMFFFFFIBGFDDDDDDDGIIIIIIBBBBFFFFFFFFFFFK]*

PRINTING RIGHT TREE
|
 |-M
  |--IFFFFMMMMFFFFFMMFFFFFMMMMMMFFGGGGGGGGGMMMMMMMPPPPJJMMMMM[MIFFFFMMMMFFFFFMMFFFFFMMMMMMFFGGGGGGGGGMMMMMMMPPPPJJMMMMM]*
  |--MMMFFMMMMMMFMMMMMFFMMMMMMFFGGGGGGGGGMMMMMMMPPPPJJMMMMM[MMMFFMMMMMMFMMMMMFFMMMMMMFFGGGGGGGGGMMMMMMMPPPPJJMMMMM]*
 |-PPPPPPFGMMFFFFFMMMMMMFFMMMMFFFFMMMMMMFFMMFFFFFMMMFFFFFMMMFFFFFMMMMMMFFGGGGGGGGGMMMMMBBBPPP[PPPPPPFGMMFFFFFMMMMMMFFMMMMFFFFMM
 |-GGGG[GGGG]*
 |-BIFFFFFMMMFFFFFMMMFFFFFMMMMMFFFFWMMMMKKKKKKMMBBIPPPPJJJJM[BIFFFFFMMMFFFFFMMMFFFFFMMMMMFFFFWMMMMKKKKKKMMBBIPPPPJJJJM]*
```

Figure 3.2: Partial console output of radix tree for the visualization in *Figure 3.1.*[2]

[2] In attempting to correlate *Figure 3.1* to *Figure 3.2*, it may seem odd that unmusical characters like *M* and *P* exist in the latter but not the former. Well, as a quick aside, due to sharped (#) notes (flat and enharmonic notation is not used), it was best to remap the musical alphabet (12 major and minor chords) to a singleton alphabet (*A* through *X*). This allowed the search and tree structures to work from one character to the next. Once those methods are finished in the code, the alphabet is then mapped back to its counterpart for the actual visualization.

For the original bifurcation to occur, the visualization must sketch or draw itself left and right from the main node (labeled in white in *Figure 3.1*), to complete the beginnings and endings of the song string sequences. Therefore, the final visualization actually implements two radix trees, one for each side. *Figure 3.2* depicts the lexicographic path of the two separate trees.

Bridging the gap between the drawing algorithm and the radix trees involved a process of Java Array and Hashmap manipulations. We remapped the node and edge information from the trees to a linear Hashmap (e.g. the first element of a Hashmap could be something like *CmCmCm = 3*), which consisted of each substring and its branch value. The ancestor (common) node does not exist in any of the trees; it's visualized directly from the user-search. Its indices act to segment the visualization into two trees. In *Figure 3.1*, the ancestor is *FFF#*. So, for songs in the collection that contain that match--if they do, the first match would be used--the left tree starts at all the indices after the first *F* in the common pattern. The right tree, following the same criteria, would be composed of indices that occur after that particular *F#*.

A radix tree, like many other structures, inherently avoids duplicate keys (strings) and contains exceptions for them. Computationally, to allow duplicate keys, it usually takes a few extra algorithmic steps in order to actually store that data. In the case of this thesis, we not only needed to permit duplicate strings (simple rewrite), but we also needed to store which branches contained those duplicate strings. As a workaround, before a key and value are stored, the system searches for duplicate keys. If one is found, an asterisk and an integer are added to the end of that string. If another of the same kind is found, the integer counts upward. Later, once the elements need to be remapped to a musical alphabet, the system searches for all strings that include an asterisk and removes it and all the characters afterward.

The application in *Figure 3.1*, though different in aesthetic, represents data with the same global visual approach involved with the diagrams displayed in *Figures 1.6* and *1.7*. A comparison is shown below.

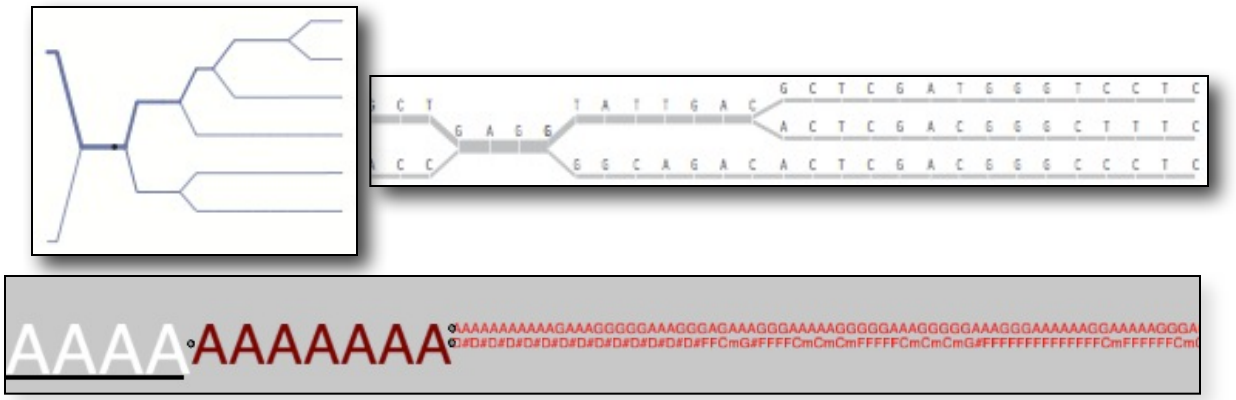


Figure 3.3: Showing all three bifurcation-based diagrams for a comparison view. The actual data examples are simpler in scope.

3.1.2. Scaling and Spacing: Keeping Sequence Structure Intact

Without a doubt, when developing *drawing* algorithms, the most non-trivial elements are scaling and space. The spatial component of a visual application is what determines how specific data and connections are read. The flowchart, for example, is an extremely controlled and didactic diagram that evenly spaces out paths and clearly defines direction and order via a set amount of space, i.e. the length of the connection line or arrow. Wattenberg's arc diagrams channel different levels of translucency to avoid spatial collisions--lines atop other lines. In scatter-plot oriented works, like *MusicBox*, spatial distance itself varies upon the numeric difference in the calculation of the similarity metric. Fry's bifurcation plot uses fixed space between nodes and edges. Unless the intention of the visualization is to display data as a shape that represents a bigger picture (*Figure 1.3*), a lack of space can cause distortion and drive comparisons into visual obscurity. On the other hand, if there is too much space, then relationships may not even be formed at all.

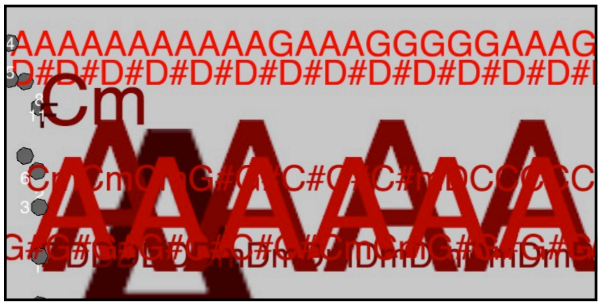
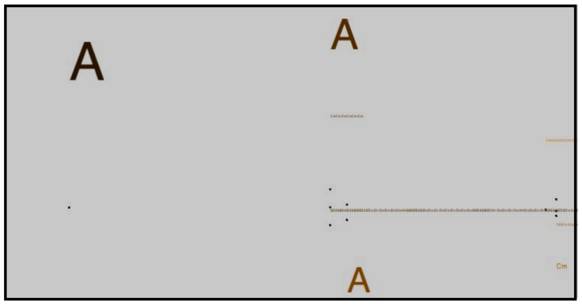



Figure 3.4: spatial variations on the prototype visualization. (Left) being too cluttered, (Right) being too distant.

Collisions that limit perceptibility were the most challenging issue that was faced in implementing this application. To limit branches from running into each other, each brach, or sequence of chords in this case, needed to be its own object--aware of its connections horizontally and its counterparts vertically. The distances between branches were to be set at a fixed length. However, when dealing with such long sequences (even though they were reduced) and attempting to visualize them in an interesting and informational way--not just a list of branches like in *Figure 3.2*--spatial design is key.

To implement such a system, a physics library developed for Processing called *TRAER.PHYSICS 3.0*, http://www.cs.princeton.edu/~traer/physics, was utilized. The library itself employs algorithms involving spring tension, attraction and repulsion. In this project, the library benefitted us with the ability to store each branch as a node, called a *particle*, at a specific position. Particles are aware of other particles within their own particle system. Regarding this application, each bifurcation tree (one left and one right) acts as its own particle system. I didn't make use of the attraction objects in the library, but I did call on spring objects to keep space between branches above and below each other and to keep a connection with each of their own, specified parents, or previous nodes.

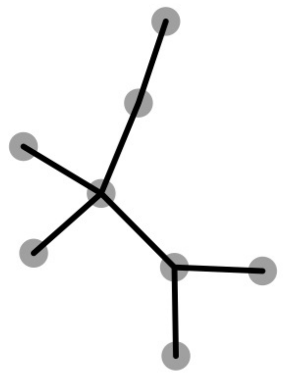
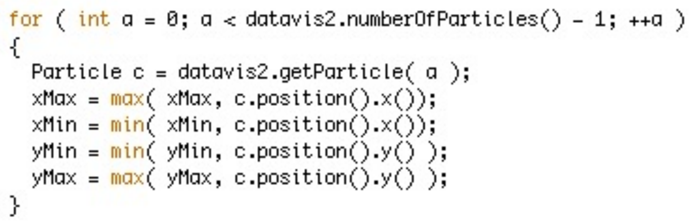


Figure 3.5: A simple example of the physics library in practice. Particles are the grey circles, or nodes. Springs, edges, connect the nodes. The visualization described in this thesis makes each branch a node and uses springs to create the initial separation.

Iteratively, the process of keeping track of space is done via each branch's connection to its parent's node. No matter where it moves in space or how lengthy its chordal sequence is, to hold temporal structure and series, the link between parent and offspring cannot be broken.

Scale was the second-most daunting challenge in completing this prototype. The immediate factor that made scale an issue was the length of the sequences. It's impossible to provide complete clarity of the information if that information in itself is complex and elongated. Compromises have to be made; interactivity may have to be established. To deal with scaling, at the outset, a function was created to do two things: update the centroid of the visual and scale outward accordingly, as *longer*, not more, branches are added to the mix. The function operates on tabulating the minimum (shortest) and maximum (longest) particle positions (x and y axes) that exist in both particle systems, and then finding the difference, change, between the smallest minimum and largest maximum. The visualization's window length and height are then divided by this difference (plus a small scalar) to determine the overall scaling for sure--maintaining that every part of the visual, (i.e. the longest string's letter) is visible.

The ability to zoom and pan have been added to the application in order to make up for issues where the original scaling renders the visualization too distant. Even though the visual is essentially 2-D, its space is 3-D. With the capacity to zoom and pan, users are able to focus in on details, instead of relying on the larger picture. An example of the zoom and pan is shown in *Figure 3.1*.



```
for ( int a = 0; a < datavis2.numberOfParticles() - 1; ++a )
{
  Particle c = datavis2.getParticle( a );
  xMax = max( xMax, c.position().x());
  xMin = min( xMin, c.position().x());
  yMin = min( yMin, c.position().y() );
  yMax = max( yMax, c.position().y() );
}
```

Figure 3.6: A block of the iterative process, finding the max and min of each particle in particle system 2.

3.1.3 **Color, Size and Look: Ordered Tag Clouds**

To actually *represent* the changes and shared progressions in harmonic, chordal structure amongst the set of data, the elements of color, size, and overall look take prominence. Without a sense of color or font change, the current project would still derive viable conclusions about the patterns shared; however, it would be lackluster. In order to *refine* the visualization and carry across the hypothesis, clarification is needed.

Three major methods were used to represent and refine the display. Various font sizes were used in order to show how many matching nodes (same string patterns) existed in another node, i.e. how many offspring does a parent have. Font size is used in the same way that Fry uses line thickness.

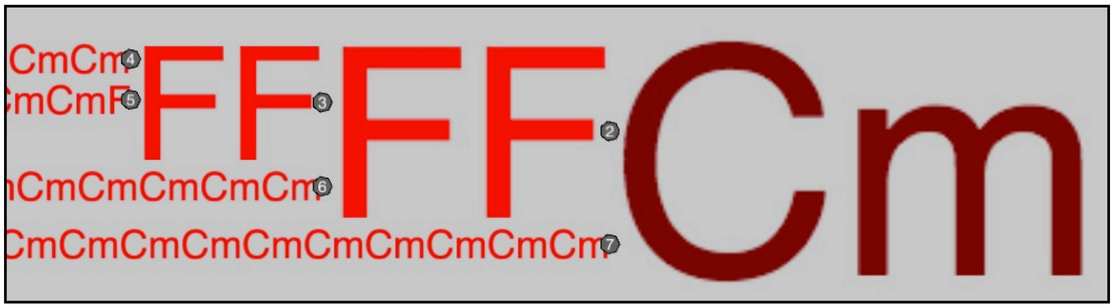


Figure 3.7: An example of the changes in font. The *Cm* chord contains all the other strings shown to the left, making the font larger and tall enough to *fit* the others.

The figure above exemplifies the role that font plays in the diagram. To analyze it visually, the *Cm* chord contains the substrings to the left of it, a total of 4. The next most grouped pattern, *FF*, contains 3 strings; the next *FF* contains the top 2. The smallest and shortest substrings, comprised of multiple *Cm's*, continue on until their length (time) reaches an end (actually beginning, as it is the left tree). These chordal subsequences do not contain shared patterns and, instead, just play out the rest of chords that exist in each track's whole sequence.

On another note, the use of font-size belies the need for lined connections because each the height of each font *fits* a parent node with its offspring, e.g. the height--top to bottom--of the large *Cm* string contains the positions of the strings that belong to it sequentially. The actual font-size is calculated by the number of nodes that each one holds, + 1. Looking at the excerpt above, the *Cm* node contains seven branches, setting the font-size at 8. From the note itself, only two substrings are actually connected. However, the system uses recursive programming to tally the *holdings* of all the branches that follow it, and then calculate the sum. All in all, a parent's size is determined by its offspring plus its offspring's offspring, etc...

Figure 3.7 also exemplifies the visualization's use of color to refine clarity. Colors change according to what branch tier each substring is a part of (branches immediately from the center are tier 1, branches that bifurcate from those are tier 2, etc...). Again, the reason for implementing the radix tree was to group similar patterns in a lexicographical context, like a dictionary. Once a chord is not a match, it then bifurcates into another branch. The path of each particular chord sequence, made up of subsequences, is--from top to bottom and right to left (for *Figure 3.7*):

CmFFFFCmCm....

CmFFFFFCmCm...

CmFFCmCmCm...

CmCmCmCmCmCm...

The color gets lighter from branch tier to tier; the first *Cm* is darkest; the *FF's* and the initial set of *Cm* characters on the bottom are next in line, etc... The variations can be subtle and the resolution of this paper is lacking; so, a close-up of the previous figure is provided.

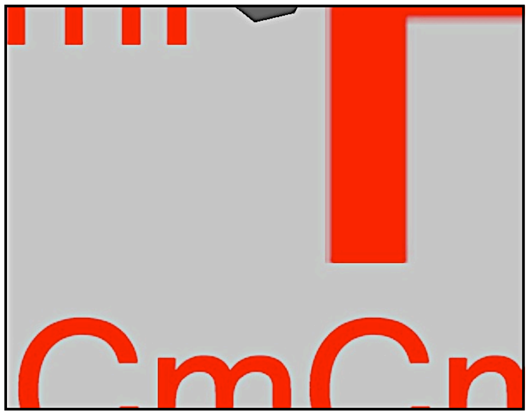


Figure: 3.8: Though this still's rendering lacks the resolution of the application, it provides a little more detail about color change.

The number of tiers equate to systems that are more similar, meaning that they contain more branch tiers and, therefore, more matching substrings. Within diagrams displaying only a few branches, dark to light is very distinguishable. In more shared systems, the entire color palette changes. The reason for the subtlety in color is the system's attempts to create color continuity for the various types of outputs it produces.

The visualization also contains numbered ellipses. Each number presents a different bifurcation in the diagram. Once a search is undergone and the visualization established, the application prints out two text files (left, right) that document which songs' substrings exist at each node (if the song makes the cut initially from the search). For example, when looking at the file is outputted from *Figure 3.7's* diagram, the original *Echoplex* track exists in the search, and its entire sequence to the left of the main node (the searched root) is made up of substrings **1** (not pictured, but further to the left), **2**, **3**, and **4**--the final bifurcation for that sequence.

The final *look* of the visualization acts as an *Ordered Tag Cloud*: the time series of each track is not scattered and is kept straight on the horizontal axis. The visualization consists of only text. And, font-size plays the major role of describing the information--shared chord progressions in this case--where bigger text associates with a higher count.

3.2. **A Run-through of the Prototype Application**

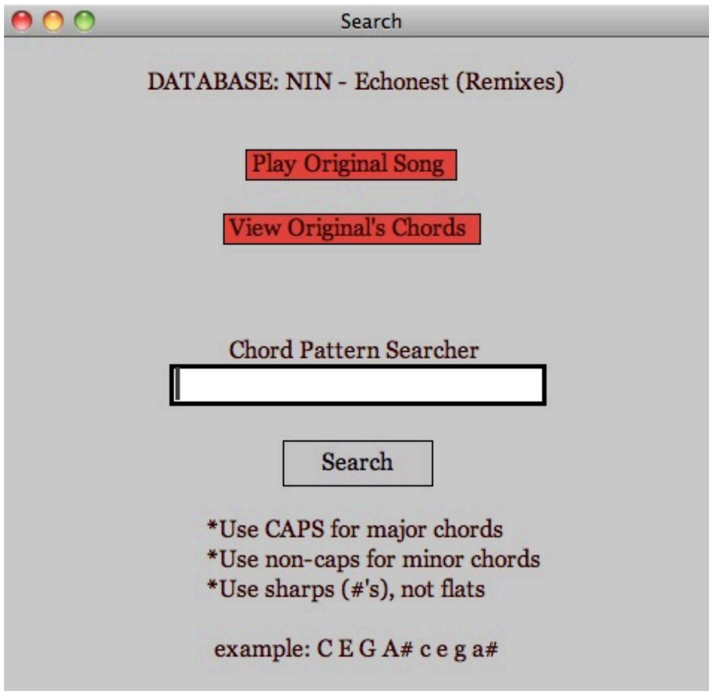
In section 1.1, the fundamental aspects of the application were laid out for how this thesis project was built. We've since discussed the various implementations and concepts behind the processes that actually makeup the application's functionality. This section should fill any and all gaps, as it will deal with an actual run-through of the application from start to finish[3].

Ninety percent of the implementations discussed throughout the thesis were programmed in and run on the open-source Processing platform/language. The other ten percent, specifically the radix tree, were completed in straightforward Java. Processing is a great framework for visualizing data because it's built on the Java platform, a universal computer language, and it is packaged with objects, references, and libraries whose sole purpose it is to visualize. On one

[3] This run-through will involve only the first collection (*Echoplex*). The other collection (*The Hand That Feeds*) will be mentioned in 4.1.

hand it could be a helpful facility to workshop ideas, prototypes, and tests; on the other, it could suffice as application module on its own. Ben Fry, quoted often in this paper, and Casey Reas are the creators of the language and environment. With ease, we were able to experiment with diverse libraries and reference similar situations. Also, being Java based, we were able to call on Java classes through processing (via the loading of a JAR file).

As fair warning, this application is a prototype, and, in many ways, it is still in its infancy. The main setup is comprised of four Java classes: KMP search algorithm (*sKMP*), the Radix Tree structure and filtering (*setTree*), the actual drawing class (*Drawer*), and the main run class (*text_search_yes*), which also provides the GUI that ties the whole process together. The code for the beat reduction is done in a separate package. Eventually, this application will be a stand-alone, but this will happen down the road.



DATABASE: NIN - Echonest (Remixes)

Chord Pattern Searcher

*Use CAPS for major chords
*Use non-caps for minor chords
*Use sharps (#'s), not flats

example: C E G A# c e g a#

Figure 3.9: The first window is a GUI for the chordal search engine. Here, a user can search for chords patterns, either based on the original or on curiosity.

Figure 3.9 depicts the initial window that opens up when running the application. The only process completed beforehand, offline, has been the acquiring, labeling, and reduction of the data--giving the system a musical database. 30 songs (1 original and 29 remixes) were used for this initial prototype, but more can be added. Additionally, to further enhance the clarity of the visualization, in this instance, the database only takes the first minute of every track (still reducing each sequence to one chord label per beat).

The GUI itself is pretty basic. The search box allows you to input major chords as capital letters (*C*-*G*) and minor chords as lowercase letters (*c* - *g*). Flats are not allowed; instead, sharps are used. If a non-existent chord is entered (e.g. *Z*), then the system reads it as an error and outputs *Music Chords Only* to the box. If there is not at least one match in the database for the entered chord progression, then *No Matches* is output. To obtain a sense of how the original track (*Echoplex*) relates to its remixes, a user can refer to its chord sequence to populate searches--a pdf automatically opens when *View Original's Chords* is clicked. A user may also listen to original track via the *Play Original Song* button.

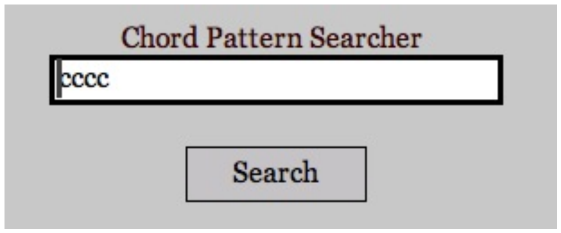
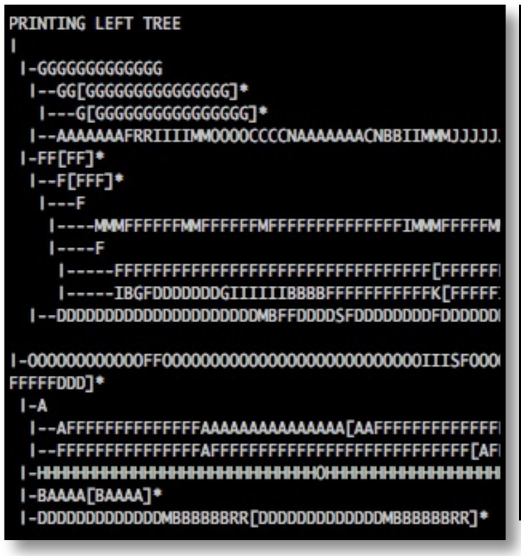


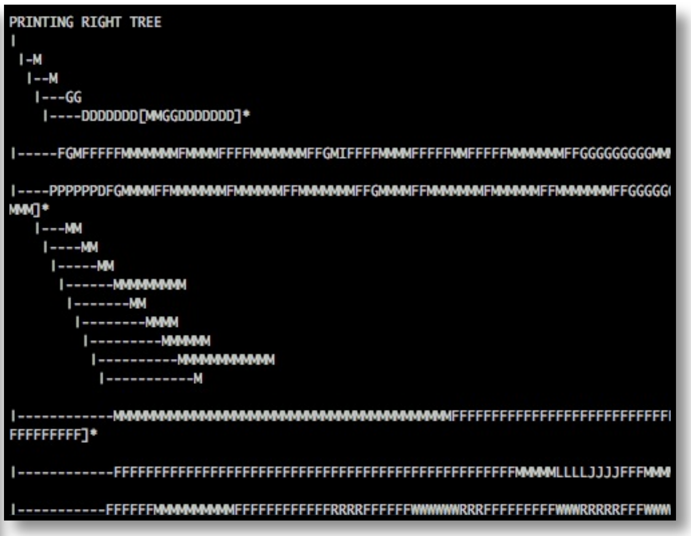
Figure 3.10: The user inputting a search for a pattern consisting of four *C minor* chords in a row.

Once the *Search* button is clicked or a user hits the *Enter* button on their keyboard, the visualization will display instantly. In the spurt between, the user-chosen pattern (four consecutive C minors as shown in *Figure 3.10*) will be searched against the database via the KMP algorithm. If a track contains a match to that pattern, then the first match will be indexed--where it begins, where it ends--and saved into two adaptable, expanding arrays (e.g. *aK_Left and aK_Right)*. The loop iteration will then move to another track. After the search function has been exhausted, the two radix trees will be computed, comparing the prefixes of everything to the left and to the right of the searched chord progression. The left sequences are segmented by each track's *aK_Left* position (the position where the KMP match starts); the right sequences are composed of sequences that occur after each one's *aK_Right* position (the position where the KMP match concludes). Chord-to-chord time structure is left intact, even though the lengths of the strings themselves will be different on either side of the root. On a side note, even though each song is a minute in length for this prototype, the strings lengths will be different due to the beat analysis discussed previously.



```
PRINTING LEFT TREE
|
 |-GGGGGGGGGGGGG
  |--GG[GGGGGGGGGGGGGGG]*
   |---G[GGGGGGGGGGGGGGGG]*
  |--AAAAAAAFRRIIIIMMOOOOCCCCNAAAAAAAACNBBIIMMMJJJJ.
 |-FF[FF]*
  |--F[FFF]*
   |---F
    |----MMMFFFFFFFMMFFFFFFFMFFFFFFFFFFFFFFFFIMMMFFFFFM
    |----F
     |-----FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF[FFFFFF
     |-----IBGFDDDDDDDGIIIIIIBBBBFFFFFFFFFFFFK[FFFFF.
  |--DDDDDDDDDDDDDDDDDDDDDMBFFDDDDSFDDDDDDDDFDDDDDD

|-000000000000FF0000000000000000000000000000IIISF000
FFFFFDDD]*
 |-A
  |--AFFFFFFFFFFFFFFAAAAAAAAAAAAAAAA[AAFFFFFFFFFFFFF
  |--FFFFFFFFFFFFFFAFFFFFFFFFFFFFFFFFFFFFFFFFFFF[AF
 |-HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHOHHHHHHHHHHHHHHHHH
 |-BAAAA[BAAAA]*
 |-DDDDDDDDDDDDDMBBBBBBBRR[DDDDDDDDDDDDDMBBBBBBBRR]*
```



```
PRINTING RIGHT TREE
|
 |-M
  |--M
   |---GG
    |----DDDDDDD[MMGGDDDDDDDD]*

|-----FGMFFFFFMMMMMMFMMMMFFFFMMMMMMFFGMIFFFFMMMMFFFFFMMFFFFFMMMMMMFFGGGGGGGGGMM

|----PPPPPPDFGMMMMFFMMMMMMFMMMMMFFMMMMMMFFGMMMMFFMMMMMMFMMMMMFFMMMMMMFFGGGGG
MMM]*
   |---MM
    |----MM
     |-----MM
      |------MMMMMMMM
       |-------MM
        |--------MMMM
         |---------MMMMM
          |----------MMMMMMMMMMM
           |-----------M

|-----------MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFF]*

|-----------FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFMMMMLLLLJJJJFFFMMM

|----------FFFFFFMMMMMMMMMFFFFFFFFFFFFRRRRFFFFFFWWWWWRRRFFFFFFFFFFWWWRRRRRFFFWWW
```

Figure 3.11: Partial console listings of the left and right radix trees for the search in Figure 3.10.

The searched root acts as the central clustering and filtering device, as it does determine, *off the bat*, which songs will exist in the graph, and which will not. Strings are bifurcated from offspring to offspring, depending on the amount of exact-shared similarity that exists amongst strings. The search and tree structuring steps happen quickly and are invisible to the user.

After the user enters their search, they initially see the view presented in *Figure 3.12*.

With the mouse, the user can zoom into (*Drag + Left Click*) and pan to the left or right

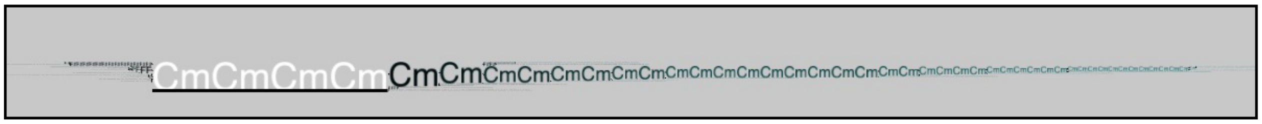


Figure 3.12: the initial output of the all the processes, data-search-tree-visual. It is scaled in order to fit all string lengths that exist in the search. It presents the overall structure and highlights shared patterns (bigger in font).

(*Drag + Right Click*) of the visual at a high resolution. The 3-D environment for the 2-D graph is made possible by OPENGL (Open Graphics Library) calls in processing to the graphics card. *Figure 3.13* showcases the zooming and panning features that are available. If a mouse button is held down, white lines surround each node (branch), exposing the bifurcation, see *Figure 3.14*.

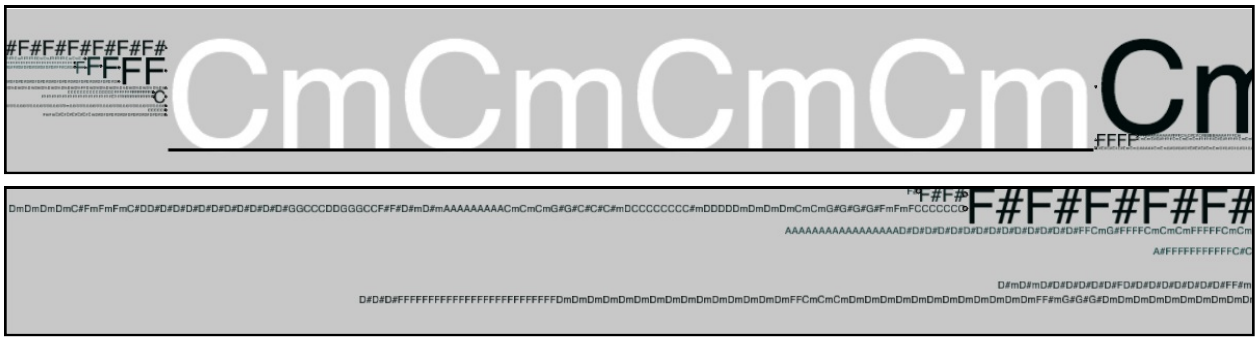


Figure 3.13: Zoom, panned images of the visualization in Figure 3.12.

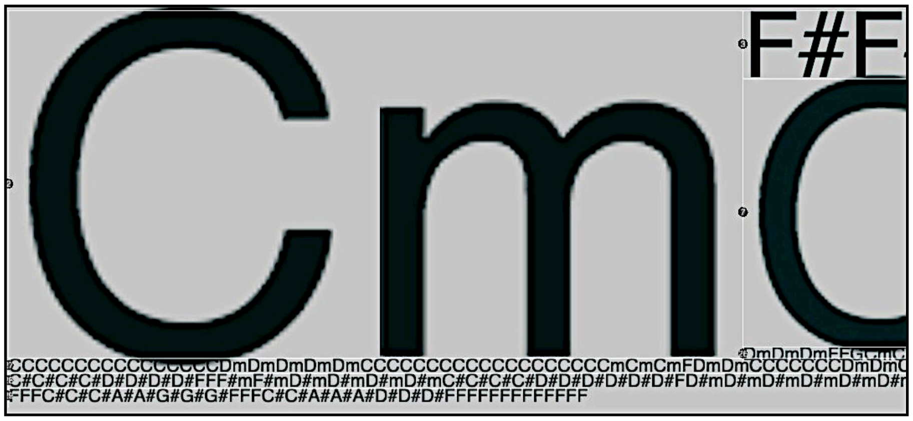
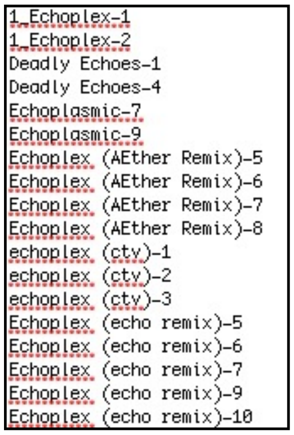


Figure 3.14: Thin white lines surround each branch.

The other characteristic that's useful is that of numbered circles that exist when a user closely zooms in on specific locations of substrings. We already discussed their meaning and function in 3.1.3. After the application is run, two text files are automatically printed out. They not only share the information about where song substrings are located, but they also define which songs actually ended up in the visualization from the original search.



```
1_Echoplex-1
1_Echoplex-2
Deadly Echoes-1
Deadly Echoes-4
Echoplasmic-7
Echoplasmic-9
Echoplex (AEther Remix)-5
Echoplex (AEther Remix)-6
Echoplex (AEther Remix)-7
Echoplex (AEther Remix)-8
echoplex (ctv)-1
echoplex (ctv)-2
echoplex (ctv)-3
Echoplex (echo remix)-5
Echoplex (echo remix)-6
Echoplex (echo remix)-7
Echoplex (echo remix)-9
Echoplex (echo remix)-10
```

Figure 3.15 (left): A partial list of which tracks share commonality with the search root and where each of their substrings are located (this is for the left tree).

4. **Qualitative Analysis**

4.1. **Examples and Discussion**

Overall, does the visualization aesthetic and structure serve to represent a sense of the harmonic relationships working within a collection of musically related material? To debate this, three examples will be discussed:

1) (*Echoplex* collection)

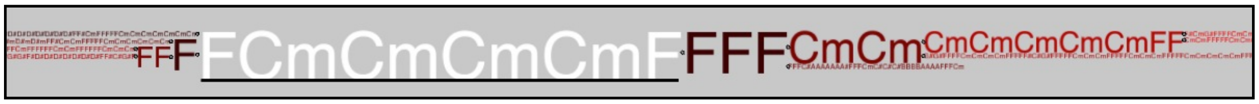


Figure 4.1: Searched chord progression: FccccF (zoomed in from original scaling).

The visualization shown above, produced by the prototype application, discernibly calls attention to the fact that this collection, consisting of four tracks, heavily favors the chords *C minor* and *F major*. The basis for this search was a pattern that exists in the original *Echoplex* track. Searching repetitions of just *C minor* turns up many more results (evidenced in the last section's diagram). However, in this case, only four tracks share this specific pattern--in this consecutive order. After listening to the matched tracks (*Echoplexaterrestrial, Echoplex (AEther Remix)*, *Echoplex-ctv*), we noticed that they all follow a similar approach to the remix in the first minute. The starts of each track vary pretty heavily--two of them without the percussion elements, focusing solely on vocals. Then, they bring in the main percussive beat and verse structure, with only slight changes between each track. Finally, toward the end of a minute, new additions are introduced, causing the nodes to contain less offspring, bifurcating into the standard font-text with no shares. The original track and *Echoplex (AEther Remix)* are definitely the most similar throughout minute one, and, as per the diagram, they are the ones that share the most substrings (separating at *FF* toward the right-hand side).

However, a major piece of this diagram is missing: a track entitled *Echoplex (Instrumental)*. This instrumental version of the original should share most, if not all of the same chords, but it's not here--in this search. Therefore, it's safe to say that the polyphony that makes up this collection of music, specifically the vocals, will skew matches, especially in areas where vocals have been stripped away. Exact matches make matters worse because it's possible that the vocal elements are causing only a trivial shift. Yet, in such a exact system, trivial can become extremely significant.

2) (*Echoplex* collection)

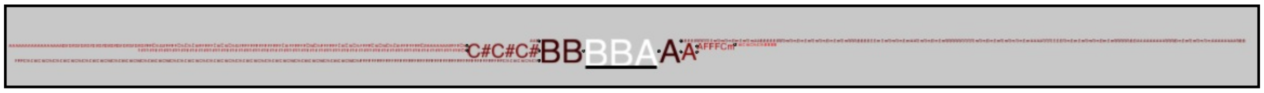


Figure 4.2: Searched chord progression: BBA (not zoomed in).

Though this system of visualization does not stress intricate structure quite like Wattenberg's arc diagrams (though he dealt with sections in individual songs), it does function to highlight a selection of music's chordal stress. And, it does perpetuate a harmonic shape for a collection. When comparing *Figures 4.1* and *4.2*, it is safe to say that the database as a whole contains more repeatability and more sequences containing the *F major* and *C minor* chords than those of *B major* and *A major*. One pivotal change to this application that would lead it toward new ground would be a larger and less biased dataset. This is not difficult to do and was supposed to be a part of this thesis; however, it just hasn't been implemented yet.

The photos of the visualizations in this paper do not do the actual application justice. Being able to push in toward particular areas of substrings and pan along the sequences horizontally enable a much clearer understanding of the matches, bifurcations, and continuation of time, from left to right, that these songs inhabit. Still, the original state of the diagram, as seen in *Figure 4.2*, sticks to these goals and measures the significance of the chords in a collection, whether that collection is large or small in scope. The visualizations focus on heavily shared/clustered progressions, which, as a whole (comparing different searches), help to define the various relationships in the database.

3) (*The Hand That Feeds* collection)

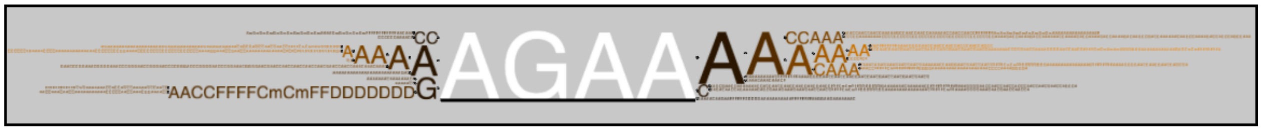


Figure 4.3: Searched chord progression: AGAA (slightly zoomed in).

The purpose of this example is to stress the generalizability of the system/approach. By being to able to analyze and search through another collection (still remixes, yet another set) in a uniform fashion (same look), the approach allows for more endeavors with various tracks, playlists, and genres.

In trials with *The Hand that Feeds* collection, the major point of comparison to the previous examples is that more tracks share longer patterns of chords, meaning that the collection as a whole is somewhat more similar. Searches containing a pattern of four to five characters in length return more results. Additionally, as seen in the case of *Figure 4.3*, the step-ladder affect is more consistent on both sides of an input-search--the number of bifurcations are more balanced--than what we've encountered with the *Echoplex* collection, which tends to have more bifurcations to the right of the searched pattern. Otherwise, this example, like the others, promotes the significant chordal relationships shared amongst a specific selection of the entire collection (12 of 30 in this case).

The entire visualization concept itself, which is the focus of this thesis, may only define harmonic similarity to a limited and possibly over-fitted (to the database) extent, but it does follow the tenets laid out by Fry that qualify as good data visualization research. The diagram, as an ordered tag-cloud, is pretty easily understood in regards to size and space. Also, it keeps true to this thesis's concentration on temporal and structural order, emphasizing that goal by using the chord symbols themselves to define changes and matches. The problems facing this thesis are mostly at the similarity end (the limited capabilities of exact matching schemes), and beyond the scope of this current work.

4.2. **Shortcomings and Limitations**

All in the all, the system/approach suffers from one main factor: exact similarity. The current setup doesn't always do justice to the overall similarity of each collection. However, the *best* alternative, one that configured inexact matching, is debatably obscure--though, there are a deluge of possible and interesting solutions out there in bioinformatics literature. Incorporating inexact clustering and matches into the current visual layout is also very complicated. The ordered tag-cloud look doesn't really provide a solution to the problem either. If similar, not exact, chords were represented, the ease-to-comprehend factor would be lost without some critical redesign. How do you cluster the character *A* and the character *C* and depict the measurement of the musical comparison's similarity without a large, complicated symbolic map/ legend for the user?

The secondary issues stem from using only the first match of a pattern in a search and the handling of repetitions by means of choosing the most repeated element between beats. The former could be solved by actually supplying the user with the ability to choose a match, more akin to web-based search engine. Both issues are in need of a more probabilistic approach with a higher importance placed on musical knowledge for *choosing*. Though the beat-related reduction does factor in musical importance, the major rank rule that ultimately chooses the chord label, does not.

Many of the shortcomings of this project's representation of musical similarity is derived from the fact that the approach is based on many text-based information retrieval comparisons and methods. The visualization is predicated on displaying the actual chord labels; the matching scheme and search algorithm emanate definitively from the text-comparison world. The entire plot-point of bifurcation is also truly textual in nature. To incorporate inexact bifurcations into the current system, the complexity would much, much higher--if even possible at all. To up the musical framework of this project, it might be safe to say that the bifurcation/tag-cloud approach is not completely worthwhile for the task. As the system is now, it provides an good impression of the chordal relationships and weight of a collection, but a more intricate representation of musical space would be difficult to coalesce.

4.3. **Future Work**

Still, all the facets of this work can be modified, reinterpreted, and added to, depending on the overarching goal. This visualization presents a new approach with historical significance, but it lacks adaptability for certain data-based problem sets. To be more useful and common, it needs to be more accepting of variations in results and views for change, i.e. more interactivity and more methods for analytical behavior, depending on the data set and goals at hand.

As mentioned earlier, the overall idea that a person can search for chord patterns and have their search be visualized as a means to depict prominence and relationships amongst other kinds of music is a fantastic challenge to approach. A chord search engine, specifically one that went beyond just harmony, and integrally involved beat or timbre in its estimations, would be a great tool for determining the importance of certain patterns in musical history. Imagine being able to search patterns in a database and retrieve a visual diagram that connected progressions from Bach to those of Miles Davis. The search engine itself, if user-based, would have to permit choice and *best results*. The problem would occur when deciding on how to rank the results of the search, which could prove more difficult than how semantic textual searches are setup.

Additionally, the visualization itself suffers from having a fixed-length distance between other sequences. To pull out more comparisons, it would be helpful to measures distances on calculations derived by different feature mechanisms, as in (Lillie, 2008). Measuring similarity horizontally and vertically would permit a better understanding of a collection as a whole.

Finally, the key crux in this thesis, as mentioned earlier, is how it deciphers harmonic similarity. Future advancements will take into account sequence alignment algorithms and bioinformatic principles. However, the scoring and value of such systems are debatable. Without a doubt, the next priority is the inclusion of more musical knowledge into the system. The degree of musical knowledge measured drives the visual navigation representation of any such approach. Jazz music's use of chord substitutions and other genres' use of playing the same melodies in different keys should affect the representation of similarity. With the current setup, those intrinsic musical elements are missing.

4.3 **Final Conclusions**

In this thesis, an approach to visualizing harmonic information has been proposed. The diagrams are created via a search engine that allows users to search for and see the similar patterns created from a specified chord progression. A data structure, called a radix tree, is used to cluster the data in lexicographic way, keeping the temporal order of each song intact, which is the key element of the visual outlay that is generated.

As a problem, visualizing similarity in time leaves a lot to be desired and is even more of a hassle to be reckoned with. Scaling, spacing, and avoiding obscurity and unnecessary complexity comprise only a handful of the roadblocks in creating a successful and pertinent visual application. The system presented in this paper, though only a prototype, attempts to deal with most of these roadblocks through the uses of bifurcation plotting, physics systems, font-sizes and color. It has limits, specifically in regards to its approach to matching and segmenting the harmonic elements; yet, it still displays chordal weight and substring connections that are perceivable on first glance. The next set of research and implementation will only make it more personable via a an integrated search engine and taken to new levels--and approached from different angles--in regards to the functionality of the visualization as a whole. In essence, the idea is for it to catch on with musicians and musicologists, two types of people who could benefit most from such a visual system.

5. **Bibliography**

Bartsch, M.A. and Wakefield, G.H. (2001). To Catch a Chorus: Using Chroma-Based Representations for Audio Thumbnailing, *In Proceedings IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, Mohonk, New York*.

Bello, Juan P. (2007). Audio-Based Cover Song Retrieval Using Approximate Chord Sequences: Testing Shifts, Gaps, Swaps, and Beats, *In Proceedings of ISMIR-07, Vienna, Austria*.

Bello, Juan P. and Pickens, Jeremy (2005). A Robust Mid-level Representation for Harmonic Content in Music Signals, *In Proceedings of ISMIR-05, London, UK*.

Cho, Taemin and Bello, Juan P. (2009). Real-Time Implementation of HMM-Based Chord Estimation In Musical Audio, *In Proceedings of ICMC-2009*.

Elliot, Jake (2006). Pop Sketch Series, http://www.visualcomplexity.com/vc/project.cfm?id=370.

Ellis, Daniel P.W. and Poliner, Graham E (2006). Identifying 'Cover Songs' with Chorma Features and Dynamic Programming Beat Tracking, *LabROSA, Dept. of Electrical Engineering Columbia University, New York, NY*.

Fry, Ben (2008). Visualizing Data, *United States of America, O'Reilly Media, Inc.*

Fry, Ben (2004). Computational Information Design, *Massachusetts Institute of Technology*.

Fujishima, T (1999). Realtime Chord Recognition of Musical Sound: A System Using Common Lisp Music, *Proceedings of the International Computer Music Conference*, *464 - 467*.

Gusfield, Dan (1997). Algorithms on Strings, Trees, and Sequences, *University of California, Davis: Cambridge University Press*, *23 - 29*.

Jehan, Tristan (2005). Creating Music by Listening, *Massachusetts Institute of Technology, 71-77*.

Krumhansl, Carol (2005). The Geometry of Musical Structure: A Brief Introduction and History, *ACM Computers in Entertainments, Vol. 3, No. 4, Article 3B*.

Knuth et al (1977). Fast Pattern Matching In Strings, *SIAM J. COMPUT. Vol. 6, No. 2, 323-350*.

Lamere, Paul and Donaldson, Justin (2009). Using Visualizations for Music Discovery, *ISMIR-2009*.

Lang, H.W. and Flensburg, FH (2001). String Matching: Knuth-Morris-Pratt algorithm, *http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/kmpen.htm*

Lillie, Anita (2008). MusicBox: Navigating the space of your music, *Massachusetts Institute of Technology*.

Mehta, Chirag (2006). US Presidential Speeches Tag Cloud, http://chir.ag/projects/preztags.

Morrison, Donald R. (1968). PATRICIA --Practical Algorithm To Retrieve Information Coded in Alphanumeric, *Journal of the Association for Computing Machinery, Vol. 15, No. 4, 514-534.*

Orpen, Keith and Huron, David (1992). Measurement of Similarity in Music: A Quantitative Approach for Non-parametric Representations, *Computers in Music Research, Vol. 4, 1-44*.

L. R. Rabiner. A tutorial on HMM and selected applications in speech recognition. *Proceedings of the IEEE, 77(2):257–286, 1989*.

Sabeti et al (2002). Detecting recent positive selection in the human genome from haplotype structure, *NATURE | VOL 419*, *832-837*.

Smith, Lloyd and Medina, Richard (2001). Discovering Themes by Exact Pattern Matching, *New Mexico Highlands University*.

Wattenberg, Martin (2002). Arc Diagrams: Visualizing Structure in Strings, *IBM Research*.