Aalborg University
Department of Computer Science

Fredrik Bajersvej 7 ▪ DK-9220 Aalborg East

Title

Novel Approaches in Audio Similarity
- NCD; Amplitude Pattern Mining

Project group:
KDE4 - d629a

Members:
Adesola Adegboye
Ruobing Li
Ruoran Zhou

Project period:
Feb-2006 to June-2006

Number of copies:
07

Number of pages:
54

Supervisor:
Bukauskas Linas

Abstract

With the increase of music objects available, there is a need for searching similar songs. This thesis proposes and investigates two novel approaches for audio similarity. One is a non-feature based algorithm called Normalized Compression Distance (NCD), which uses a distance to tell audio similarity using compression technique. The other is amplitude based algorithm, using Amplitude Pattern Mining (APM) to measure similarity. It is found that NCD is not applicable because of the incapability of real world compressors, amplitude is a promising feature and APM shows good performance. For both distance and APM approaches, models are built for playlist generation.

Aalborg, June 2006

This project is done by KDE4-group **d629a**

Adesola Adegboye _____

Ruobing Li _____

Ruoran Zhou _____

Acknowledgment

This thesis was completed in Aalborg University, under the guidance of our supervisor Bukauskas Linas, who is very supportive, and we are very thankful for fruitful discussions and comments he provides us. We also want to thank Albrecht Schmidt, who suggested the topic of our report in last semester, which is part of the base of this thesis. We gratefully acknowledge the hospitality of the people who support us during this master thesis work.

- Adesola Adegboye: I would like the acknowledge the support of my parents and siblings. Without you guys, I would not have completed this journey. Another person who had great contributions is Amanda, thanks for your ideas, encouragements and love.
- Ruobing Li: I would like to thank my dear parents far away in China. Their love supports me all the time during the two years studies in Denmark. I miss and love you!!
- Ruoran Zhou: I would like to thank my father, who has been supporting me all the time. I love you.

Contents

Chapter 1

Introduction

In recent years, the increase in quantity and variety of available music objects raises the needs for music recommendation. A common query is for similar music objects to the ones specified by the user. One difficult task is to measure likeness between music objects. In this thesis, we present two approaches to achieve audio similarity detection and measurement. One approach is NCD which is a non-feature based distance method. The other is Amplitude Pattern Mining, which uses amplitude in this thesis. For each approach, playlist generation algorithm is designed to return songs similar to query songs.

1.1 Related Work

1.1.1 Music Similarity

Music similarity is a topic that has attracted attention from both academic and commercial societies. Researchers have tried to find music similarity from all kinds of music information available. The main approaches that have been explored in music similarity fall into three distinct groups: collaborative filtering, meta-data similarity and audio content analysis.

Collaborative filtering techniques are based on publicly available data. With the growth of the Web, collaborative filtering techniques and text analysis are used to combine data from many individuals to determine similarity based on users feedback information. For example, collaborative filtering relies on comparing records of purchasing patterns among customers or the request records from a radio station. The disadvantage of these approaches is that they are only applicable to music for which a reasonable amount of reliable data is available. For some new or undiscovered artists, it does not work.

Metadata similarity focus on comparing metadata fields such as genre or subgenre. Pauws and Eggen in [1] employ dynamic clustering to group songs by metadata attribute. Different weights for the various attributes are used to carefully balance the relative importance of various attribute in the metadata. However, sometimes music metadata is unavailable, not precise enough, inaccurate or even ill-defined, which make the metadata similarity not so powerful. For this reason, we don't consider use metadata feature in this thesis.

Audio content similarity analysis relies on comparing features such as pitch, amplitude, rhythm and timbre. Previous audio content investigation mainly has two directions. In the first direction, many studies have been carried out to analyze symbolic representations of audio such as MIDI music data. The MIDI [2] technique provides standardized and efficient means of conveying musical performance information as electronic data. MIDI file does not contain the sampled audio data but only the instructions needed by a synthesizer to play the sounds. Further, some related techniques using pitch tracking are applied to find a melodic contour for each piece of music. Finally, string-matching techniques are used to compare the transcriptions for each song [3] [4] [5]. However, techniques based on MIDI are limited to music for which data exist in electronic form. Not all music has good-quality machine-readable score descriptions available, which makes this approach limited.

The second direction uses extracted features of audio to represent the music. Acoustic features ideally contain the information of the original music signal hence classifiers or similarity metrics can be applied on them. Features may exist either on a short, medium or long time scale. [6] analyzes different timescale features and combines short-time features on a larger timescale. Many of the investigations use short time feature data derived from MFCCs (Mel Frequency Cepstral Coefficients) [7]. In [8], audio pieces are chopped into short-time frames, and each frame is described by its MFCCs. A Gaussian mixture model (GMM) is constructed from these feature data, thus distance method is used as a measure of similarity. [9] discoveries the timbre similarity by modeling MFCC feature data of individual songs with GMMs and use Monte Carlo methods to estimate the Kullback Leibler divergence between them. [10] forms a signature for each song based on K-means clustering of spectral features instead of GMMs. The signatures are compared using the Earth Mover's Distance.

Audio content similarity and metadata similarity are both feature-based similarities. This kind of similarity requires specific knowledge about music. Besides

feature-based similarities, there is another similarity approach named compression-based similarity. It is non-featured and proved to work well in many domains[11]. An application of this compression-based similarity on music is proposed in [12]. The results obtained from experiment with music in MIDI format showed promising results. It makes us believed this compression-based similarity is worth exploring further.

This thesis works on audio objects. Our audio similarity approaches cover both feature-based and non-feature based similarities. The measures of likeness include tradition distance method and non-distance method. The Figure 1.1 shows which groups our similarity approaches belong to. Our similarity approaches are represented as grey circles.

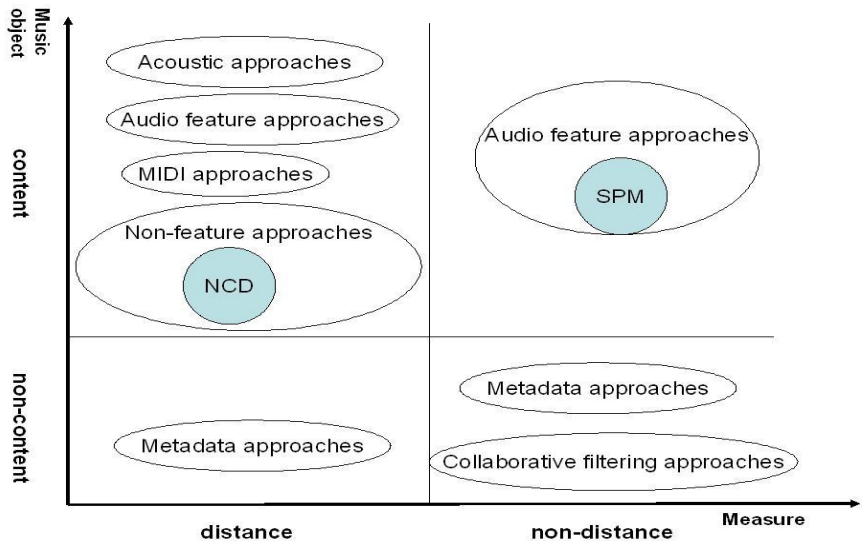

Figure 1.1: Similarity approaches

In [13], we proposed to utilize compression-based similarity on audio objects instead of MIDI. The similarity distance measure NCD in [12] calculates the distance between encoding files of audio objects. In this thesis, we investigate and verify NCD on audio objects.

The other approach we propose is feature-based similarity. Instead of deriving MFCC feature data, we extract amplitude feature from audio objects. As in [14], this approach also focuses on energy profile and multi-frames of audio objects. In this work, PrefixSpan algorithm is applied on audio objects, while a variant of dynamic programming algorithm is used in [14]. This variant is called Discrete

Time Warping, which aligns one object string to the other via a lattice. Results returned contain the best alignment path that takes one string into the other, and the matching cost of that path. The cost is the measure of object similarity. Sequential Pattern Mining applications have been applied in many fields, such as text analysis, medical treatment, natural disaster analysis and so on. In content-based music retrieval field, some works also have been done. In [15], melody feature is represented as a sequence of chord-sets, a modified sequential pattern mining algorithm finds the ordered patterns. Similar work is in [16] which focuses on music features represented as rhythmic and melodic sequences; patterns are generated and used for music classification. In this thesis, audio objects are represented as amplitude value sequence. PrefixSpan algorithm mines sequential patterns shared by audio objects in Database. Similarity between audio objects is measured based on the frequencies of occurrence in same patterns.

1.1.2 Playlist Generation

In [17], the problem of playlist generation is treated as a network flow problem. Given a collection of songs, where each song is labeled with a number of attributes. One start sink and one stop sink are set initially. What the algorithm does is to find a path (of user-defined length) between those two sinks through the network that satisfies some user-defined constraints. In [18], another approach for handling metadata is presented. According to the user-defined constraints, the metadata of each song (each node in the graph) is transformed into a cost function. The playlist is constructed by iteratively optimizing an initial randomly chosen playlist with regard to the cost function.

In this thesis, the playlist generation algorithms for NCD similarity approach are based on a distance matrix. What the algorithms do is to find a list of songs satisfies some user-defined constraints. One algorithm gives ordered playlists considered to be similar to the query song, these songs are ordered by the distances between them and query song. Another algorithm returns playlists in which songs are similar and transit smoothly from one to the next. For the case with more than one query songs from users, a scoring model is built to return a playlist of songs similar to all the query songs.

Based on Amplitude Pattern Mining approach, another playlist generation algorithm is proposed. It returns songs similar to a query song to certain degree regarding users constraints. If there is more than one query songs, algorithm uses the scoring model from NCD playlist generation.

1.2 Contribution

There are mainly three contributions of this work. Firstly, playlist generation models are proposed based on audio distance, and NCD is investigated in audio domain by doing several compressor experiments. A modified NCD is implemented on audio content. In NCD formula, an element denotes compressed size of the mixture of two objects. Two ways are used to get this mixture instead of a simple concatenation. The NCD performance on music collection does not yield satisfying results. So we conclude that NCD can not work well in audio contents.

Secondly, the amplitude feature is chosen to distinguish audio objects. Energy profile is computed for every audio object in the music collection, which is determined by computing the amplitude feature values. This energy profile tells the softness and hardness of songs, but experiment results show that energy profile over long time-scale does not yield satisfactory performance.

We improve this approach by using feature information than pure amplitude. Combination of amplitude and the compressed file size of audio piece is proposed and tested. Experiment shows these two features still do not give satisfactory results on audio objects.

Finally, based on all these observations and prior experiments on amplitude, audio object is chopped into a sequence of one second segments and represented as a sequence of amplitude values. Similarity between audio objects could be got by comparing the corresponding amplitude value sequences. Sequential pattern mining is introduced to extract amplitude patterns from audio objects, and a model is proposed to measure the similarity. The main idea is if two sequences share certain number of same amplitude variation patterns, two corresponding audio objects are similar to certain degree. This approach shows good results.

1.3 Structure of Thesis

In Chapter 2, some basic concepts are explained, including background knowledge used for our research. In Chapter 3, after a review of NCD and model construction based on NCD, all the NCD experiments on audio objects and the findings are shown. Then, in Chapter 4, firstly investigation of amplitude feature, feature combination are presented. Then a novel approach called amplitude pattern mining is introduced and evaluated, and an algorithm is also proposed for playlist generation. Finally, conclusions and future work are discussed in Chapter 5.

Chapter 2

Basic Concepts

In this chapter, some basic concepts and background knowledge are introduced that are related with our explorations in audio similarity. Audio combination is explained in section 2.2, and compression technique is discussed in 2.3. In the end, some definitions are given that serve as the basis of the audio similarity approaches we propose.

2.1 Audio file

Audio file is made of electronical audio signals transformed from analog signals for easy storage, reproduction and transfer. Analog audio signal in figure 2.1a is series of increase and decrease in pressure changes in air, resulting in sound that can be heard. These pressure changes occur very rapidly and the speed at which they alternate between a positive pressure and a negative pressure determines the frequency. Digital audio signal in 2.1b is a stream of discrete and quantized electrical impulses which represent acoustic audio signal. It is discrete because it is sampled from continuous signals and represented with a series of numbers. Also, it is quantized because it tries to approximate the continuous signal. For more details about continuous and digital signals, [19] can be consulted.

There are many factors that determine the quality of an audio file, two of them are:

- Sample rate - Sample rate refers to rate at which the the electric voltage is captured. It is measured in hertz (Hz) and should be at least twice the highest frequency represented [19]. Since humans can't hear frequency above 20,000Hz, 44,100Hz is chosen as the sample rate for audio CDs. Anything lower than this rate result in distortion and higher rate does not produce significant improvement in the reconstruction of the analog signal

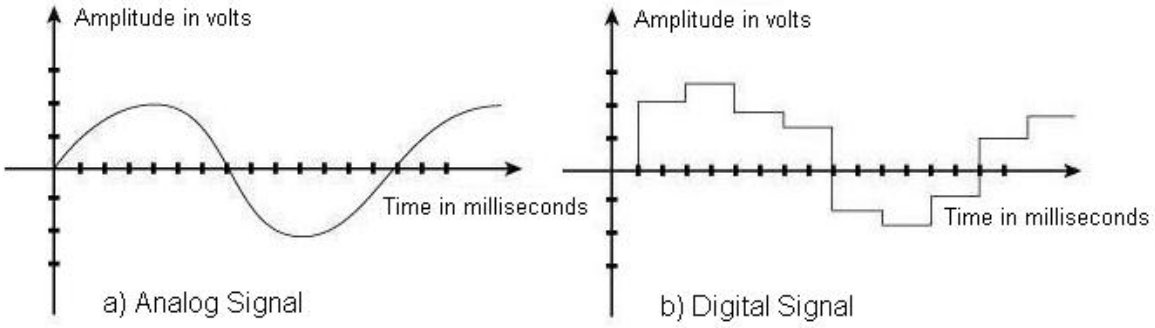

Figure 2.1: Example of acoustic and electric audio signal

- Sample size - Sample size refers to the quantization of analog audio signal. Small sample size result in coarsely quantized sample. For example "if a sound wave has a voltage of 1 volt, an 8 bit sample can differentiate voltage as low as 1/256v is 4mv. A sound that generates a wave of 2mv will be sampled as zero" [20]. In audio CD the sample size is 16 bits, this makes it possible to distinguish sound as low as 15mv.

2.2 Audio Combination

Combination of two different audio files is a common phenomenon in audio domain. There are many ways of combining audio files into a single clip, one way is to concatenate songs together. Another is by playing them simultaneously which is called mixing.

There are two common ways of concatenating audio files. One way is to append one audio file after the other. The other is to chop audio files into short time scale segments. The segments from the two audio files are selected in an interleaving manner and appended one after the other. For example, audio file A and B are chopped into segments respectively, which are represented as sequences of small segments, $A = (A1, A2, A3, A4)$ and $B = (B1, B2, B3, B4)$. After the concatenation, one result can be $(A1, B1, A2, B2, B3, A3, B4, A4)$.

Mixing is the simultaneous recording of two or more audio files into a single file. The audio files could be of equal or different length. Audacity [21] is an open source software which is used in this work for mixing.

2.3 Compression

Since compression technique is used for audio similarity in this thesis, we introduce some compression methods in this section. Firstly, an overview of data compression is given, including some common audio compression techniques. Finally three compressors used in this project are explained, namely *bzip*2, *flac* and *ogg*.

2.3.1 Data Compression

Data compression is the process of converting an input data stream into an output data stream that has a smaller size [20]. There are two reasons why data compression is popular. One is that old data can then be saved, occupying less space. For instance, when there is large amount of history records, such as in the library and bank, these information can still be saved after compression with less requirements for hard disk space. The other reason is that a smaller size data is more efficient to transfer. For example, when large files are transferred over the internet, such as multimedia, they are generally compressed before the transmission.

There are many different methods for compression, based on different ideas and for different types of data, such as image, audio, text, etc. They are all based on the same principle, which is to remove redundancy from the original data [20]. Generally, compression is done by assigning short codes to common events and long codes to rare ones. For example, in a text file where letter "E" appears very often while "Z" is rarely used, then a shorter code is assigned to "E" while a longer one to "Z". In this way, the so called "alphabetic redundancy" is removed.

A compression method can be either lossy or lossless. The lossy methods will have some information loss during compression. When decompressed, the result is not identical to the original input data stream. Lossy methods are commonly used to compress images and sounds, since human may not notice the difference when a small amount of data is removed. For example, if sounds of frequency out of the range (20Hz, 20,000Hz) are removed, human ears will not notice. In contrast, some data can't afford the loss, such as computer programs, where the lossless methods are necessary.

Audio compression

In this section, common compression methods designed for audio are introduced. A general overview is given for compressors used for audio compression nowadays.

Some methods are based on human auditory limits. The frequency range of the human ear is from about 20 Hz to 20,000 Hz, and the ear sensitivity differs de-

pending on the frequency. One lossy compression method delete audio samples2.1 that are out of the range of human hearing.

Two other properties of human hearing system are also used for audio compression, namely "frequency masking" and "temporal masking". Frequency masking, also known as simultaneous masking, is masking between two concurrent sounds that share a frequency band. It happens when a sound that human can normally hear is masked by another louder sound with a nearby frequency. A lossy compression can detect such cases and delete the signals corresponding to the sound being masked, since it can't be heard anyway. Temporal masking occurs when a loud sound A is preceded or followed in time by a lower sound B. If the time interval between A and B is short, then B will be masked by A. A lossy compressor will find such events and delete sounds like B.

Generally, lossy methods are most often used for compressing sound. This is because human ears can't notice the difference even though some information is lost during compression. Among the lossy audio codecs, MP3 codec which is a digital audio encoding and lossy compression format is commonly used nowadays. The Advanced Audio Coding (AAC) was promoted as the successor to MP3 for audio coding at medium to high bit rates and popularized by Apple Computer. The other example is an increasingly popular codec named OGG which is also a good choice.

There are also a lot of lossless music audio compressors such as Free Lossless Audio Codec(FLAC), RealAudio Lossless(RealPlayer), True Audio Lossless(TTA), WavPack lossless(WavPack), Windows Media Lossless(WMA Lossless) etc.

2.3.2 Compressors

In this thesis, we consider three different compressors, since they can be suitable compressors for similarity purpose. A "suitable" compressor can detect similarities between audio files. A compressor might be "suitable" even though the compressed file is of poor quality or with large size, as long as the similarity between files can be elicited. The three compressors used are *bzip2*, *flac* and *ogg*.

bzip2

As stated in [22], "*bzip2* compresses files using the Burrows-Wheeler block-sorting text compression algorithm, and Huffman coding". Burrows-Wheeler block-sorting works in a block mode, where the input stream is read block by block and each

block is encoded separately as one string. For each block, the idea is to rearrange the order of the characters so that substrings that occur frequently are clustered together. The result string has several places where a single character is repeated multiple times in a row. Then Run Length Encoding(RLE) is used for compressing the result string. Huffman coding assigns variable-length code to symbols in the original file based on the estimated probability of its occurrence.

Flac

Flac stands for Free Lossless Audio Codec, which is one of the lossless audio compression methods. The *flac* encoder has the following stages according to the introduction in [23]. Firstly, the input is broken up into many contiguous blocks. Then the block is passed through a prediction stage where the encoder tries to find a mathematical description of the signal. If the predictor does not describe the signal exactly, the difference between the original signal and the predicted signal is called the error or residual signal, which must be coded losslessly.

In [24], details of the technique used in *flac* is given which is called AudioPak. In the prediction stage, a linear predictor is applied to the signal sample in each block, which looks at most 3 samples backward to predict.

If the audio to be compressed has a sampling rate 4,4100 Hz, then the sample length is around 2.27e-5 seconds. That is the predictor will look at most 6.8e-5 seconds back from the current sample point.

Ogg Vorbis

Ogg Vorbis is a popular lossy compression techniques for audio. Technically, it uses the modified discrete cosine transform (MDCT) to convert sound data from the time domain to the frequency domain. The resulting frequency-domain data is broken into noise floor and residue components, and quantized and entropy coded using a codebook-based vector quantization algorithm [25]. Generally, as a lossy compression method, Ogg Vorbis removes sound that can not be heard by human, and use less bits to represent audio information without affecting the overall quality.

2.4 Basic Definitions

In this section, we define some basic notations and terms. These definitions serve as the basis for the explorations we do in audio similarity. Some definitions are redefined or extended in later chapters for use in specific contexts.

Definition 1 (Song) *A **song** is a music audio file in certain digital format.*

Examples of digital format are wav, ogg, etc.

Definition 2 (Song Database, Song ID) *A **Song Database** DB is a collection of songs, collected beforehand and taken as the mini-world. Each song in DB is assigned a unique **Song ID** i, $i \in N$. $song_i$ represents the ith song in DB.*

Definition 3 (Similarity Detection Algorithm) ***Similarity detection algorithm** is a methodology that discovers likeness between two songs.*

Two Similarity detection algorithms are proposed in this thesis, one is Music Distance called Normalized Compression Distance (NCD), and the other is Amplitude Pattern Mining (APM).

In this thesis, there are two kinds of **similarity** for two songs, algorithmically similar and perceptionally similar.

Definition 4 (Algorithmically Similar) *Two songs are considered to be **algorithmically similar** based on the results of the similarity detection algorithm.*

For example, when music distance approach NCD is applied, two songs are regarded to be algorithmically similar if their distance is close to zero. In the case of APM, two songs are algorithmically similar if they contain the same patterns.

Definition 5 (Perceptionally Similar) *Two songs are **perceptionally similar** if human listeners consider them to be similar after listening.*

When a user requests for similar songs, it is a query for songs that are perceptionally similar.

Definition 6 (Seed Song, Seed Song Set) *A **seed song** is a song $s \in DB$, which is selected by the user during a query. A number of seed songs make up a **seed song set** $S = \{s_1, s_2, ..., s_k\}$, in which $s_i \in S$ is a seed song, $1 \leq i \leq k$.*

Definition 7 (Nominee Song) *A **nominee song** is a song selected from DB using a similarity detection algorithm, which is algorithmically similar to the seed song.*

Definition 8 (Nominee List) *For each seed song s, there can be a number of nominee songs selected from DB. A **nominee list** L for seed song s is a list of nominee songs that are algorithmically similar to s. $L[j]$ represents the jth nominee song in L, $1 \leq j \leq |L|$.*

Definition 9 (Nominee List Set) *For a seed song set S, a **nominee list set** is a set of nominee lists, depicted as $L_S = (L_1, L_2, ..., L_k)$, L_i is the nominee list for $s_i \in S$, $1 \leq i \leq k$.*

Definition 10 (Playlist) *Given a seed song set S, a **playlist** P is a sequence of songs, which results from nominee list set L_S. For a query with one seed song s, nominee list L is returned as playlist P. When there is a set of seed songs $S = (s_1, s_2, ..., s_k)$, a nominee list set $L_S = (L_1, L_2, ..., L_k)$ is returned by similarity detection algorithm. Taking into account all nominee lists in L, a playlist P is produced.*

Chapter 3

Audio Similarity Using Compression

In this chapter, an approach called Normalized Compression Distance(NCD) is explored for audio similarity detection. Firstly, an overview of NCD is introduced. Then three models are proposed for playlist generation based on results from music distance algorithms, e.g. NCD. Further, experiments are made on NCD by applying three different compressors, namely *bzip2*, *flac* and *ogg*. Afterwards, tests are done for NCD using *ogg* with more songs. It turns out that NCD is not promising in audio similarity detection, and analysis of the reasons is given at the end of the chapter.

3.1 Normalized Compression Distance (NCD)

In [13], a similarity metric called NCD is proposed for audio similarity. NCD stands for Normalized Compression Distance, and is defined by Formula 3.1 to calculate distance between two objects, which can also be audio files. In Formula 3.1, $C(x, y)$ denotes the compressed size of the concatenation of x and y, $C(x)$ denotes the compressed size of x, and $C(y)$ denotes the compressed size of y. NCD is a non-negative number, $0 \leq ncd \leq 1+\epsilon$, indicating the degree of similarity of the two files. The closer NCD is to zero, the more algorithmically similar the two files are. The ϵ in the upper bound is due to imperfections in compression techniques in real world, but for most standard compression algorithms ϵ is seldom above 0.1 [26].

$$NCD(x, y) = \frac{C(x, y) - min\{C(x), C(y)\}}{max\{C(x), C(y)\}} \tag{3.1}$$

The intuition of NCD is that two objects are considered to be close if we can significantly "compress" one given the information in the other. In other words, if two objects are similar, then one can be more succinctly described given the other.

NCD tries to detect similarity between two objects by means of compression. Hence, to use NCD, a good compressor is necessary and objects compared have to be compressed.

The NCD method has been released in the public domain as an open-source software at [27], namely Complearn. The CompLearn Toolkit is a suite of simple utilities that applies NCD. It has three built-in compressors, namely *bzip2*, *zlib* and *google*.

3.2 Definitions

In this section, some concepts and notations are defined that are only used in this chapter. Some notations in Section 2.4 are also used, they are Song Database, Seed Song, Seed Song Set, Nominee Song, Nominee List, Nominee List Set and Playlist.

Definition 11 (Song Distance) *Given song s and t, the* ***song distance*** *between them is obtained by applying a Distance algorithm, depicted as $d : s \times t \rightarrow \mathbb{R}$.*

In the case of NCD, it is $NCD(s,t)$.

Based on Definition 5 in Section 2.4, algorithmically similar is further specified for audio distance algorithm.

Definition 12 (Algorithmically Similar) *Two songs s and t are* ***algorithmically similar*** *when song distance $d(s,t) < \epsilon$, ϵ varies in different contexts. The smaller $d(s,t)$ is, the more similar they are considered by the audio distance algorithm.*

In the case of NCD, s and t are algorithmically similar when $NCD(s,t)$ is close to zero. Based on Definition 7, a nominee song is a song whose song distance is close to the seed song.

Definition 13 (Song Distance Matrix) *A* ***song distance matrix*** *is used to depict result of audio distance algorithm. In an $n \times n$ song distance matrix M, $M_{i,j} = d(song_i, song_j)$, representing song distance between the ith and jth songs in DB, $n = |DB|$, $1 \leq i, j \leq n$.*

Definition 14 (Preference Vector) *A* ***preference vector*** *for seed song set $S = (s_1, s_2, s_3, ...s_n)$ is $W = (w_1, w_2, w_3, ...w_n)$, $w_i \in \mathbb{N}$ is the preference weight for seed song s_i, $1 \leq i \leq n$. A seed song with a higher preference weight is more favored.*

Definition 15 (Nominee List Maximum Size) *N_l is defined to be the maximum size for nominee list L.*

Definition 16 (Playlist Size) *N_p is defined to be the maximum size for a playlist P.*

3.3 Playlist Generation Models

Three models are proposed in this section. *distance ordering* and *smooth transition* are based on song distance matrix. Given a seed song, the models generate different nominee lists that satisfy different queries. In situations where there are more than one seed songs, a *scoring model* is proposed to produce a playlist.

3.3.1 Model of Distance Ordering (DOM)

DOM takes ID of seed song s and a song distance matrix M as input, and output a nominee song list L. Songs in L are those with least song distance to s, and they are ordered according to their distance with s. In L, the song at the top is the most algorithmically similar with s, while the bottom song is the least algorithmically similar.

Algorithm 1 $DOM(M, id, N_l)$

1: *Input* M: $n \times n$ distance matrix, $n = |DB|$; id: song ID of seed song s; N_l: nominee list maximum size
2: *Output:*L: nominee list, represented as an array of size N_l
3: *Define* A: an array of size n, each entry in format of (SongID, SongDistance)
4: **for** $i = 1$ to n **do**
5: $A[i] \leftarrow (i, M[id][i])$ {assign the idth row in M to A}
6: **end for**
7: QUICKSORT(A in ascending order of SongDistance)
8: **for** $i = 1$ to N_l **do**
9: $L[i] \leftarrow A[i].SongID$
10: **end for**
11: **return** L

Generally, DOM works by ordering the songs in DB according to their song distance with seed song s and pick out the top N_p nominee songs as a nominee list L. A description of the algorithm using pseudo code is given in Algorithm 1. Here is an example, there are five songs in DB, and the NCD matrix M is given

	Song1	**Song2**	**Song3**	**Song4**	**Song5**
Song1	0	0.00038	0.00164	0.00207	0.00293
Song2	0.00038	0	0.00539	0.00050	0.01310
Song3	0.00164	0.00539	0	0.06802	0.01485
Song4	0.00207	0.00050	0.06802	0	0.00219
Song5	0.00293	0.01310	0.01485	0.00219	0

Table 3.1: NCD Matrix for five songs in DB

in Table 3.2. Given Song2 as the seed song, N_l as five, the returned nominee list by DOM is (Song2, Song1, Song4, Song3, Song5).

3.3.2 Model of Smooth Transition (STM)

STM takes ID of a seed song s and a song distance matrix M as input, and outputs a nominee list L. Songs in L have the shortest song distance between neighbor songs, which means $L[i+1]$ has the least song distance to $L[i]$ among all songs in DB. For example, in a smoothest transition nominee list $L = (L[1], L[2], ...L[n]$, $L[2]$ has the least distance to $L[1]$, $L[3]$ is closest to $L[2]$, and so on. When the songs in L is played, the next song is algorithmically closest to the current one. In Algorithm 2, pseudo code is given for STM. For example, taking the same input as in model DOM, NCD matrix M in Table 3.2, seed song Song2, N_l as five, the returned nominee list by STM is (Song2, Song1, Song3, Song5, Song4).

3.3.3 Scoring Model

Since the above two models only work with one seed song, a scoring model is proposed for situations when more than one seed songs are given in a query. This model calculates score for each nominee song and considers user's preference for different seed songs.

Suppose there are k seed songs $\{s_1, s_2, s_3, ..., s_k\}$. For each seed song s_i, there is a nominee list L_i, $1 \leq i \leq k$. L_i is obtained by applying either of the two models in Section 3.3.1 and 3.3.2. For each nominee song $L_i[j]$ in L_i, Formula 3.2 is used to calculate $Score_i$ for it. If a nominee song y only appears in one nominee list, then $Score_i$ is its final score. Otherwise, if y appears in more than one nominee lists, the final score is got by summing up all $Score_i$ that is assigned to y in every nominee list L_i it appears. In the end, pick out top N_p songs with higher final scores, and output as playlist P. A description of scoring model is given in Algorithm 3 with

Algorithm 2 $STM(M, id, N_l)$

Input M: $n \times n$ distance matrix, $n = |DB|$;id: song ID of seed song s;N_l: nominee list maximum size
Output nominee list L, represented as an array of size N_l
Define A: an array of size n, each entry in format of (SongID, SongDistance)
$L[1] \leftarrow id$ {put seed song in the head of nonimee list L}
$i, j \leftarrow 1$
while $i < N_l$ **do**
  for $j = 1$ to n **do**
    $A[j] \leftarrow (j, M[id][j])$ {assign the idth row in M to A}
  end for
  QUICKSORT(A in ascending order of SongDistance)
  $k \leftarrow 1$
  while $k \leq n$ **do**
    if $A[k]$ is not in L **then**
      $L[i+1] \leftarrow A[k].SongID$ {put song $A[k].SongID$ as next song in L}
      $id \leftarrow A[k].SongID$
      BREAK
    end if
    $k \leftarrow k + 1$
  end while
  $i \leftarrow i + 1$
end while
return L

pseudo code.

$$Score_i(L_i[j]) = (N_l + 1 - j) \times w_i \quad 1 \leq j \leq N_l, 1 \leq i \leq k \tag{3.2}$$

$$Score\_final(y) = \sum_{i=1}^{k} Score_i(y) \quad 1 \leq i \leq k \tag{3.3}$$

Here is an example for Scoring model. Using the same input as in examples in Section 3.3.1 and 3.3.2. The seed song set is (S1, S2). Given seed song S1, the nominee list returned by DOM is L_1 = (Song1, Song2, Song3, Song4, Song5). Given seed song S2, DOM returns L_2 = (Song2, Song1, Song4, Song3, Song5). The preference vector (w_1, w_2) is (5, 10) for (Song1, Song2). Applying Formula 3.2, we get scores for songs in L_1 and L_2:

Algorithm 3 $Score(L_S, W)$

Input: nominee list set L_S, preference vector W
Output: playlist P
Define A: an array of size N_p, each element in the format of (SongID, FinalScore)
for each nominee list L_i in L, $1 \leq i \leq |L|$ **do**
  for each nominee song $L_i[j]$ in L_i, $1 \leq j \leq |L_i|$ **do**
    $Score_i(L_i[j]) \leftarrow (N_l + 1 - j) \times w_i$ {calculate score for each nominee song}
  end for
end for
for each song y in L **do**
  $Score\_final(y) \leftarrow \sum_{i=1}^{|W|} Score_i(y)$ {sum up scores for nominee songs that appear in more than one nominee lists}
  $A[i] \leftarrow (y.SongID, Score\_final(y))$, $1 \leq i \leq \sum_{i=1}^{|W|} |L_i|$
end for
QUICKSORT(A in descending order of FinalScore)
for $i = 1$ to N_p **do**
  $P[i] \leftarrow A[i].SongID$ {take top N_p songs in A as output of P}
end for
return P

- $\text{Score}_1(\text{Song1}) = \text{Score}_1(\text{L}_1[1]) = (N_l + 1 - 1) \times w_1 = (5 + 1 - 1) \times 5 = 25.$
- $\text{Score}_1(\text{Song2}) = \text{Score}_1(\text{L}_1[2]) = (N_l + 1 - 2) \times w_1 = (5 + 1 - 2) \times 5 = 20$
- $\text{Score}_1(\text{Song3}) = \text{Score}_1(\text{L}_1[3]) = (N_l + 1 - 3) \times w_1 = (5 + 1 - 3) \times 5 = 15$
- $\text{Score}_1(\text{Song4}) = \text{Score}_1(\text{L}_1[4]) = (N_l + 1 - 4) \times w_1 = (5 + 1 - 4) \times 5 = 10$
- $\text{Score}_1(\text{Song5}) = \text{Score}_1(\text{L}_1[5]) = (N_l + 1 - 5) \times w_1 = (5 + 1 - 5) \times 5 = 5$
- $\text{Score}_2(\text{Song2}) = \text{Score}_2(\text{L}_2[1]) = (N_l + 1 - 1) \times w_2 = (5 + 1 - 1) \times 10 = 50$
- $\text{Score}_2(\text{Song1}) = \text{Score}_2(\text{L}_2[2]) = (N_l + 1 - 2) \times w_2 = (5 + 1 - 2) \times 10 = 40$
- $\text{Score}_2(\text{Song3}) = \text{Score}_2(\text{L}_2[3]) = (N_l + 1 - 3) \times w_2 = (5 + 1 - 3) \times 10 = 30$
- $\text{Score}_2(\text{Song4}) = \text{Score}_2(\text{L}_2[4]) = (N_l + 1 - 4) \times w_2 = (5 + 1 - 4) \times 10 = 20$
- $\text{Score}_2(\text{Song5}) = \text{Score}_2(\text{L}_2[5]) = (N_l + 1 - 5) \times w_2 = (5 + 1 - 5) \times 10 = 10$

In the end, to get a final playlist, a final score is calculated for all the songs. And the finally playlist according to the final score is (Song2, Song1, Song3, Song4, Song5).

- $\text{Score\_final(Song1)} = \text{Score}_1\text{(Song1)} + \text{Score}_2\text{(Song1)} = 25 + 40 = 65$
- $\text{Score\_final(Song2)} = \text{Score}_1\text{(Song2)} + \text{Score}_2\text{(Song2)} = 20 + 50 = 70$
- $\text{Score\_final(Song3)} = \text{Score}_1\text{(Song3)} + \text{Score}_2\text{(Song3)} = 15 + 30 = 45$
- $\text{Score\_final(Song4)} = \text{Score}_1\text{(Song4)} + \text{Score}_2\text{(Song4)} = 10 + 20 = 30$
- $\text{Score\_final(Song5)} = \text{Score}_1\text{(Song5)} + \text{Score}_2\text{(Song5)} = 5 + 10 = 15$

3.4 NCD Tests Using Different Compressors

In this section, three different compressors are applied for NCD calculation. The goal is to find a suitable compressor for further tests. Additionally, we have a better understanding of both NCD and compressors in audio similarity.

3.4.1 Testing Source

We collect some song segments in *wav* format that are perceptionally similar. These segments are excerpted from songs over time, detailed description can be found in Appendix A.1.1. Some of them are different segments from the same song, some are of the same analog but different digital copies. Some have the same instrumental performance but different vocal components, and others are different tempo versions of the same excerpt. We listen to all of them, and they are perceptionally similar.

3.4.2 Text compressor: bzip2

We start the tests with a text compressor *bzip2*, which is the default compressor in Complearn Toolkit[27]. NCDs are calculated for songs in the testing source. It turns out most of the NCDs are near one, some are even greater than one, see A.1.3. According to the definition of NCD in section 3.1, it means these song segments are not algorithmically similar in terms of NCD using *bzip2*. On the other hand, *bzip2* is not designed for audio, so we decide to look for other audio compressors to apply NCD.

3.4.3 Audio compressor: flac, ogg

Two audio compressors are tested, namely *flac* and *ogg*, explained in Section 2.3.2. *flac* is a lossless compressor, while *ogg* is a lossy one with satisfying compression quality for music. NCD is calculated by applying Formula 3.1. In the formula,

$C(x, y)$ denotes compressed size of the concatenation of x and y. Since audio file is different from text file, we consider two ways to obtain $C(x, y)$, as described in Section 2.2. One is to concatenate x and y and get the file size, which is done by appending one audio file to the end of the other. The other way is to mix audio files x and y with an audio editing software, here we choose "Audacity"[21], and take the size of the mixture as $C(x, y)$. The mixing is done by playing the two files simultaneously and clipping the mixture as a single audio file.

Generally, NCD between file A.wav and B.wav is obtained in three steps, which is shown in Figure 3.1. Firstly, use a compressor C to compress A.wav and B.wav separately and we get audio files A1 and B1. Secondly, combine A.wav and B.wav to get a new file M.wav. Then compress M.wav with C and we get M1. In the last step, with the sizes of A1, B1, and M1, Formula 3.4 is used to calculate NCD, in which *sizeof()* is a function to get the length of input file.

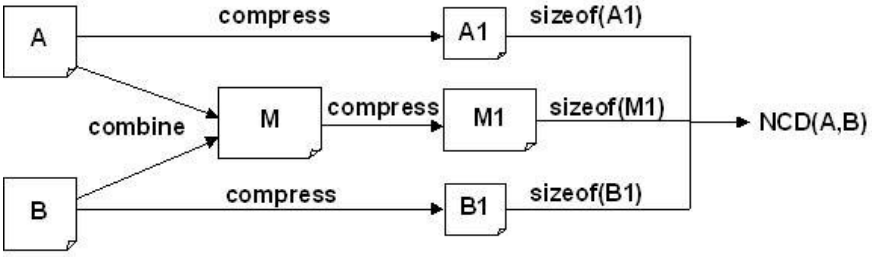

Figure 3.1: Steps of calculating NCD for A.wav and B.wav

$$NCD(A, B) = \frac{sizeof(M1) - min\{sizeof(A1), sizeof(B1)\}}{max\{sizeof(A1), sizeof(B1)\}} \tag{3.4}$$

Here is an example of how NCD is calculated for song segment "S1.wav" and "S2.wav". Firstly, compress them with *ogg* and obtain "S1.ogg" and "S2.ogg" respectively. Then with Audacity, mix these two segments, and we get "S1&S2.wav". Using *ogg* to compress this mixed file, we get "S1&S2.ogg". In the last step, Formula 3.1 is applied. Taking the length of the three *ogg* files, that is 306.226 bytes for "S1.ogg", 305.538 bytes for "S2.ogg", and 332.881 bytes for "S1&S2.ogg". NCD(S1, S2) is got by applying the formula as follows:

$$\begin{aligned} NCD(S1,S2) &= \frac{C(S1,S2)-min\{C(S1),C(S2)\}}{max\{C(S1),C(S2)\}} \\ &= \frac{sizeof(S1\&S2.ogg)-min\{sizeof(S1.ogg),sizeof(S2.ogg)\}}{max\{sizeof(S1.ogg),sizeof(S2.ogg)\}} \\ &= \frac{332.881-min\{306.226,305.538\}}{max\{306.226,305.538\}} \\ &= 0.0021 \end{aligned} \tag{3.5}$$

Firstly, we made experiments for *flac*, using both ways of combination for two songs, testing results are shown in Appendix A.1.2. It is found that by concatenation, NCD between two similar songs is very large. While by mixing songs with Audacity, the resultant NCD is closer to zero for similar songs, which indicates they are algorithmically similar. An example is shown in Figure 3.2, where there are three pairs of similar songs and NCD are calculated for them using both ways of combination. It can be seen that for similar songs, NCDs using concatenation are all nearly one, while by mixing NCDs are much smaller. In terms of audio similarity detection, NCD by concatenation fails, and we decide to get $C(x,y)$ by mixing in the rest of the tests.

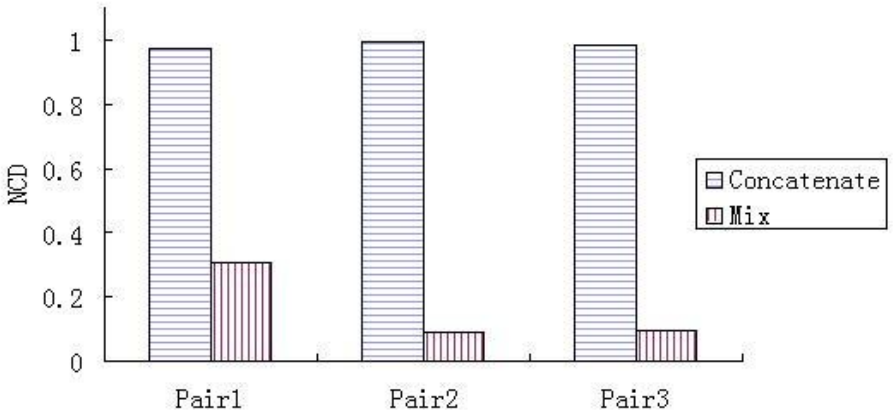

Figure 3.2: NCD by Concatenation and Mixing for 3 similar pairs of songs

In addition to lossless compressor *flac*, we choose a lossy compressor namely *ogg* for experiments, which is popular for the good quality of compressing music. Comparing the results of *flac* and *ogg*, we found that they give equivalent results except that NCD from *ogg* is smaller than that from *flac*. For example, in Figure

3.3, for similar song pairs, both *flac* and *ogg* give small NCDs, but the ones by *ogg* is relatively smaller than that from *flac*. Detailed results can be found in Appendix A.1.3. Additionally, *ogg* can maintain a good quality and give a smaller size, which is space efficient. So in the later tests, we choose *ogg* as the compressor.

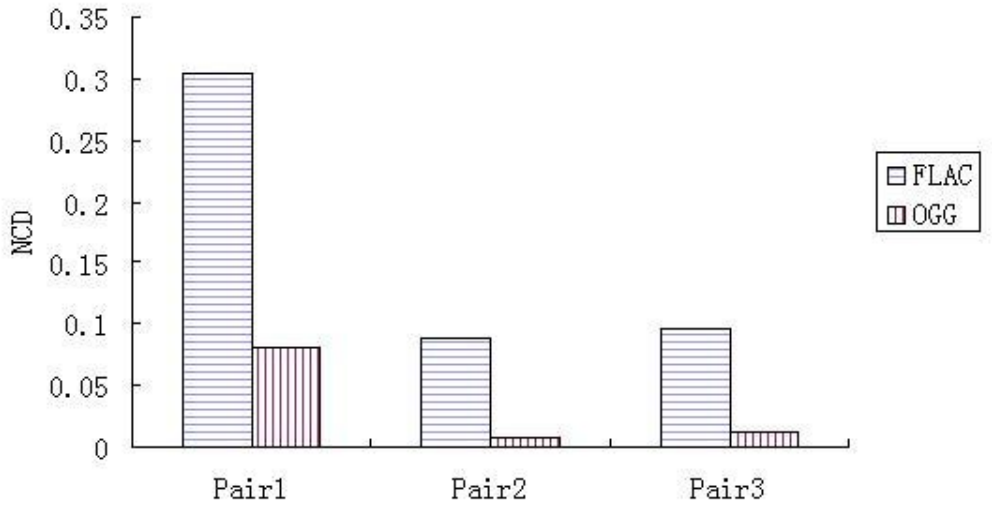

Figure 3.3: NCD using compressor *flac* and *ogg*

3.5 NCD Using *Ogg*

In this section, NCD tests are done for all songs in *DB*, using mixing to combine two songs and *ogg* as the compressor. The goal is to see how NCD perform on a larger amount of audio data.

3.5.1 Song Database

During the tests in Section 3.4, we discover that two different results are got when mixing two songs, if their sampling rates are different. For example, there are two song segments of 30 seconds. One is of CD quality with sampling rate 44,100Hz, and the other with sampling rate 22,050Hz. If two songs are mixed with sampling rate 44,100Hz, the result file is of size 6,145KB; Else if mixed with 22,050Hz, the mixed file is 3,073KB. In order to simplify the problem and get rid of other factors

that might affect the testing results, we decide to focus on audio with the same sampling rate and same length in time. In the following tests, we use songs of CD quality as our testing source.

We collect 100 songs to build the song database DB, defined in Definition 2. The songs are in different languages and are of different styles in order to simulate the universe of music songs as much as possible. Copyright issues are addressed by extracting 30 seconds of each song. 30 seconds is taken from 2:00 to 2:30 in time scale.

3.5.2 Experiments

Upon calculating NCD, an NCD matrix is obtained as in Figure 3.2. Given a song known as the seed song, NCD between it and the remaining songs in the DB is calculated. For example, NCD between $Song1$ and other songs in the database are in the first row, 0.0000, 0.03590, 0.9324,..., 0.0237. The distance of one song to itself is zero, therefore it is assumed to be the closest song to itself. To evaluate the result, songs are ordered in terms of NCD to a given seed song in ascending order. Then for one seed song, we listen to top ten songs in the order.

	Song1	**Song2**	**Song3**	**.**	**.**	**.**	**SongM**
Song1	0.0000	0.3590	0.9324	.	.	.	0.0237
Song2	0.3590	0.0000	0.6432	.	.	.	0.5327
Song3	0.9324	0.6432	0.0000	.	.	.	0.8743
.	.	.	.	0.0000	.	.	.
.	.	.	.	.	0.0000	.	.
.	.	.	.	.	.	0.0000	.
SongM.	0.0237	0.5327	0.8743	.	.	.	0.0000

Table 3.2: Example of a NCD matrix

It is observed that among the 10 songs that are the most algorithmically similar to the seed song, only 4 are perceptionally similar on average. In addition, these similar songs are not necessarily the top 4 songs in the ordering of their distance to the seed song. It means there are many misclassifications using NCD for song similarity detection.

3.6 Conclusion of NCD

We investigate the reasons that could have caused unexpected NCD results. There are mainly two reasons. The first reason is NCD can only work with the help of well-performing compressors. In Section 3.1, Formula 3.1 describes how NCD is calculated. According to this formula, for two similar songs a and b, the numerator in the formula $C(a,b) - min\{C(a), C(b)\}$ is supposed to be small. It means, after compressing the concatenation of a and b, the length of the compressed file should be close to the smaller one in $C(a)$ and $C(b)$. Only if a compressor can detect the similar information between a and b, a smaller NCD can be got for them. Unfortunately, to make this work, a suitable compressor is necessary. NCD works with the ideal compressor but not the audio compressors in real world.

Taking the two audio compressors used in this thesis for example. *flac* has a prediction stage to find similarities which is explained in Section 2.3.2. However, this prediction only looks backwards at most three frames, totally not more than 6.8e-5 seconds in time. So it is hard for *flac* to find similarities even for a song of one second, because one second could be more than ten thousand frames. That's why *flac* fails when most songs in our testing source is of 30 second. Regarding *ogg*, a lossy compressor for audio, it focuses more on discarding informations that can't be noticed by human ears, instead of discoving similarities in songs.

The following example shows how these two audio compressors fail the idea of NCD. Taking one song from our testing source, namely "1.wav", which is of 30 second. Concatenate the song to itself, we have "1+1.wav". Using both *flac* and *ogg*, we get Table 3.3. It can be seen, C(1+1.wav) by both *flac* and *ogg* almost doubles the size of C(1.wav), when it is just a song repeating itself twice.

Compressor	**C(1.wav)**	**C(1+1.wav)**
flac	**4087KB**	**8168KB**
ogg	**581KB**	**1158KB**

Table 3.3: Performance in similarity detection of *flac ogg*

The other reason for unexpected NCD results is the ways we use to get (x, y). When we don't get reasonable results by concatenating the songs, we choose mixing by playing songs together. Firstly, the mixing still does not help the compressors to find similarity, which can be seen from the testing results in Section 3.5. And this method actually causes some information loss, which can be noticed when we play the mixed songs.

Chapter 4

Audio Similarity Using Amplitude

In this chapter, firstly an introduction for amplitude and an amplitude measure *RMS* is introduced. Tests are then performed on amplitude to identify similar songs. Further, compressed file size is used together with amplitude for audio similarity. Finally a novel approach called Amplitude Pattern Mining (APM) is investigated.

4.1 Amplitude RMS

There is a certain correlation between amplitude [28, 14] and energy of a song. Sometimes, energy is interpreted as loudness, the sensation of loudness is affected by frequency which is the number of periods per second. The smaller the amplitude the higher the chance that the audio has rhythm of few beats per second and vice versa [29].

$$RMS = \sqrt{\frac{\sum ampPeak^2}{peakCount}} \tag{4.1}$$

To represent the amplitude of a song, it is not sufficient to compute the average peak amplitude due to the symmetric positive and negative values of the audio signal. Root mean square (RMS) is used to represent amplitude over a given time. One of the formulas that can be used to calculate RMS is Equation 4.1. In the rest of the thesis, amplitude RMS is referred as *RMS*. *ampPeak* in the equation refers to the peak amplitude. *peakCount* refers to the number of peaks over a given time period.

4.2 Amplitudal Feature Test

In this section, experiment is made for amplitude feature. The goal of the experiment is to investigate the possibility of using RMS described in 4.1 to identify similar songs. The song dataset in Section 3.5.1 is increased from 100 to 188 songs, which is used as DB for all experiments in this chapter. RMS is extracted for all the songs in the dataset. From the dataset, the least RMS of a music data is 0.059886 and largest is 0.408262. Before performing the test, the songs were perceptionally evaluated. It is observed that songs with small RMS value are soft songs, such as jazz, classical. Songs with large RMS value are identified to be hard songs, such as heavy metal, hard rock etc. Mild songs are songs that are neither hard nor soft. One example is pop music. Songs are grouped into equal RMS intervals of 0.024. The findings are presented in Table 4.1. The first column is RMS interval, the second column is the number of songs in each interval. The third column presents the distribution of different songs in that interval and the last column are song types.

4.2.1 Test Results

From Table 4.1, songs with RMS value less than 0.124 are soft songs, while songs with RMS greater than 0.325 are hard songs. When RMS value is greater than 0.124 but less than 0.325, similarity detection becomes a difficult task for amplitudal feature. This interval is called *critical*. RMS identifies soft and hard songs, but the coverage is not satisfactory.
It can be seen in Table 4.1 that the coverage of soft songs by interval (0.025, 0.124) is 35% while for hard songs by (0.325, 0.424) is 30%. Further,there are many music genre agglomerated within the *critical* interval. RMS could be suitable for nominee song searching for soft and hard songs.

4.3 Amplitude and Compressed File Size

As stated in Section 4.2, for songs whose RMS fall into interval (0.124, 0.325), amplitudal feature alone is hard to detect audio similarity. To solve the problem, tests are performed to see if any other feature can be used to help. After tests for NCD in section 3.4, each song in the dataset has a compressed file size. And it is observed that the size of the *ogg* file can be another feature to look into. In the rest of this section, fs is used to represent compressed file size. When the difference between two songs' RMS is less than 0.0009, and the difference between their fs is less than 10KB, then these two songs are perceptionally similar in most cases. So we make the hypothesis as follows.

Amplitude Interval	Song Count	Distribution(%)	Energy
0.025 - 0.049	1	100	soft
0.050 - 0.074	5	100	soft
0.075 - 0.099	4	75, 25	soft, mild
0.100 - 0.124	7	71.4, 28.6	soft, mild
0.125 - 0.149	10	50, 50	soft, mild
0.150 - 0.174	15	26.67, 73.33	soft, mild,
0.175 - 0.199	23	26.08, 34.78, 39.14	soft, mild, hard
0.200 - 0.224	15	46.67, 46.67, 6.56	soft, mild, hard
0.225 - 0.249	22	27.27, 81.83	mild, hard
0.250 - 0.274	25	4, 64, 32	soft, mild, hard
0.275 - 0.299	31	61.29, 38.71	mild, hard
0.300 - 0.324	17	58.22, 41.18	mild, hard
0.325 - 0.349	3	100	hard
0.350 - 0.374	11	9.09, 91.91	mild, hard
0.375 - 0.399	4	100	hard
0.400 - 0.424	1	100	Hard

Table 4.1: Support and confidence for soft and hard songs

Given two songs:

1. If their RMS is less than 0.0009 and their fs is less than 10KB, then they are similar songs.

2. Else they are different

Firstly, two examples are given to show if this hypothesis works for similar and different songs. In Figure 4.1, there are three groups of songs, which has been confirmed to be perceptionally similar. They are taken from DB, detailed information is in Table 4.2, and they are ordered in accordance with the columns in the figure from left to right. Songs in each group are close in RMS, that is the difference is less than 0.0009. Looking at their fs in Figure 4.1, it can be seen that the similar songs in each group also have close file sizes, the difference is less than 10KB. So for these three groups of songs, both conditions in the hypothesis hold.
The other example is shown in Figure 4.2, in which there are three groups of songs. Except the last song in each group, represented with a white column, the songs are the same as in Figure 4.1. In Table 4.2, they are marked with a star. It can be seen that in each group, the last song is close in RMS to all the other songs, but has more than 10KB difference in fs. Perception similarity test confirmed that the

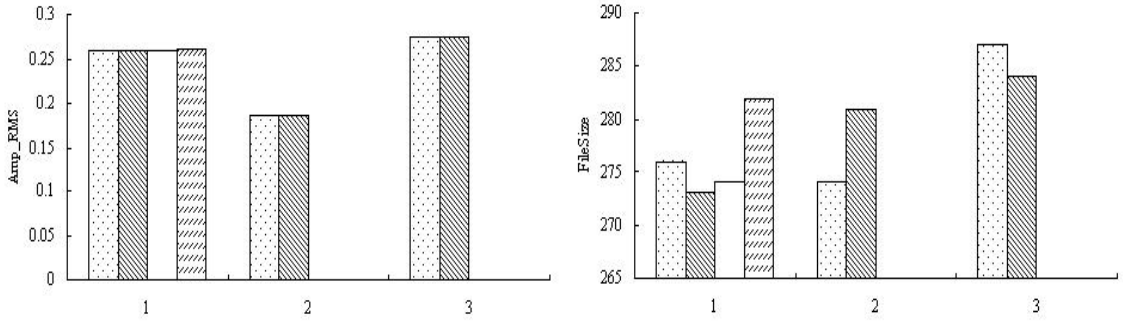

Figure 4.1: Three groups of similar songs

last song is different to other songs in the same group. So when the requirement for fs is not satisfied, the songs are not similar even though their RMS is close.

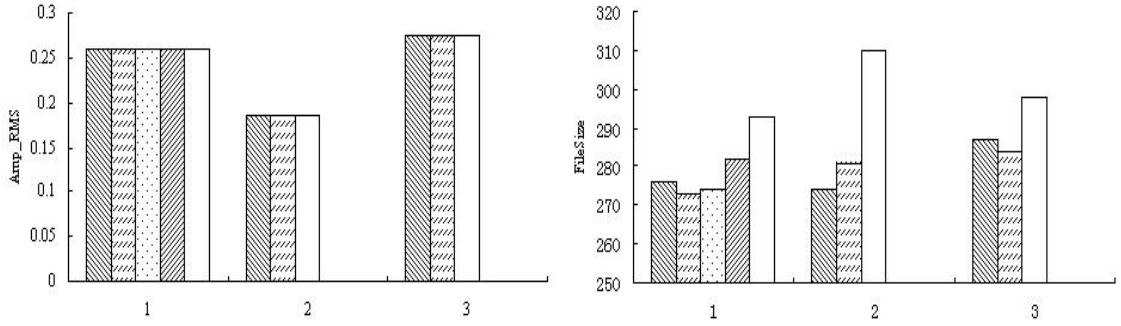

Figure 4.2: Different song examples

We therefore propose the following algorithm as a new approach to get similar songs.
On getting a query song Q:

1. Take Q's RMS and look for songs in the database whose RMS has less difference than ϵ. These songs form a set T_{rms}.

2. Compress the songs in set T_{rms} with a chosen compressor. And these compressed songs make up set T_{fs}

3. Checking the fs of files in T_{fs}. Select songs, whose fs has less than δ difference with Q. These songs are returned as a nominee list.

In the algorithm, ϵ represents the maximum difference between two songs' RMS, and δ symbolizes the upbound of their difference in fs. These two parameters can be set according to different requirements in accuracy.

Test Results

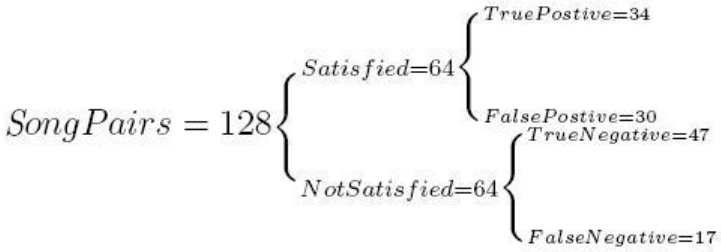

Figure 4.3: RMS and Files size result

Among the 188 songs in DB, there are 128 pairs of songs that meet the requirement for closeness in RMS. That is the two songs in each pair have less than 0.0009 difference in RMS. We look at the fs for these songs and also listen to them. It turns out that among these 128 pairs, only 47 pairs are not exactly following our hypothesis. So in 63.3% of the cases in DB, our hypothesis for audio similarity works by looking at both RMS and fs.

Among the 128 pairs, 64 pairs satisfy both requirements in RMS and fs in our hypothesis, and 34 pairs of them turn out to be perceptionally similar. So the confidence for getting correct similar songs by applying our hypothesis is $34/64 = 53.1\%$. And in the other 64 pairs of different songs who only statisfy the closeness in amplitude feature, 47 pairs turn out to be really different. That is our supposition has $47/64 = 73.4\%$ confidence to tell the difference between songs.

4.4 Sequential Pattern Mining - PrefixSpan

Sequence Pattern Mining (SPM) was first introduced by Agrawal and Srikant in [30] as follows:" *Given a sequence dataset made of sequence set, where each sequence consists of a list of elements and each element consists of a set of items, and given a user-specified support threshold, sequence pattern mining is to find all of the frequent subsequences, i.e., the subsequences whose occurrence frequency in the set of sequences is no less than support threshold.*"

Group Nr	SongName	RMS	fs
1	138.wav	0.259209	276
1	59.wav	0.25972	273
1	151.wav	0.259754	274
1	152.wav	0.260332	282
1	133.wav*	0.259486	293
2	19.wav	0.186287	274
2	100.wav	0.186301	281
2	58.wav*	0.186071	310
3	165.wav	0.275083	287
3	113.wav	0.275326	284
3	146.wav*	0.275183	298

Table 4.2: Descriptions for songs in Figure4.1 and 4.2

4.4.1 Definitions

In this subsection, terms are defined to help understanding of Prefixspan algorithm, which is further explained later with an example. Some definitions are also used when PrefixSpan is applied to audio similarity in Section 4.5.

Definition 17 (Element, Element Set, Sequence) *Let* $E = \{e_1, e_2, e_3, ..., e_m\}$ *be a set of* ***elements.*** *An* ***element-set*** *is a subset of elements. A* ***sequence*** *is a list of element-sets. A sequence is denoted by* $<q_1, q_2, q_3, ..., q_n>$*, where* q_j *is an element-set, i.e.* $q_j \subseteq E$ *for* $1 \leq j \leq n$*. The number of instance of elements in a sequence is called the* ***length*** *of the sequence.*

Definition 18 (Subsequence, Supersequence) *A sequence* $\alpha =< a_1, a_2, a_3, ..., a_k >$ *is called a* ***subsequence*** *of another sequence* $\beta =< b_1, b_2, b_3, ..., b_s >$ *and* β *is a* ***supersequence*** *of* α*, denoted as* $\alpha \sqsubseteq \beta$*, if there exist integers* $1 \leq j_1 < j_2 < ... < j_k \leq s$ *such that* $a_1 \subseteq b_{j_1}$*,* $a_2 \subseteq b_{j_2}$*,* $a_3 \subseteq b_{j_3}$*, ...,* $a_k \subseteq b_{j_s}$*.*

Definition 19 (Sequence Dataset, Support) *A* ***sequence dataset*** D *is a set of tuples* $<d_{id}, d>$*, where* d_{id} *is a* ***sequence_id*** *and* d *is a sequence. A tuple* $<d_{id}, d>$ *is said to contain a sequence* α*, if* α *is a subsequence of* d*. The* ***support*** *of a sequence* α *in* D *is the number of tuples in the dataset containing* α*, i.e.* $support_D(\alpha) = |\{< d_{id}, d > | (< d_{id}, d > \in D) \wedge (\alpha \sqsubseteq d)\}|$

Definition 20 (Sequential Pattern) *Given a positive integer* ϵ *as the* ***support threshold*** *, a sequence* α *is called a (frequent)* ***sequential pattern*** *in* D *if* α *is contained by at least* ϵ *tuples in* D*, i.e* $support_D(\alpha) \geq \epsilon$*.*

Definition 21 (Prefix) *Given a sequence* $\alpha =< a_1, a_2, a_3, ..., a_k >$, *a sequence* $\beta =< b_1, b_2, b_3, ..., b_s >$ $(s \leq k)$ *is a* ***prefix*** *of* α *if and only if (1)*$b_i = a_i$ *for* $(i \leq s-1)$*; (2)* $b_s \subseteq a_s$ *and (3)all the elements in* $(a_m - b_m)$ *are alphabetically after those in* b_m.

Definition 22 (Projection) *Given sequences* α *and* β *such that* β *is a subsequence of* α. *A subsequence* α' *of* α *is called a* ***projection*** *of* α *w.r.t* β *if and only if(1)*α' *has prefix* β *and (2)there exists no super-sequence* α'' *of* α *such that* α'' *is a subsequence of* α *and also has a prefix* β.

Definition 23 (Projected Dataset) *Let* α *be a sequential pattern in a sequence dataset* D, *the* α-***projected dataset*** *denoted as* $D|_\alpha$ *is the collection of postfixes of sequences in* D *w.r.t prefix* α.

4.4.2 PrefixSpan Example

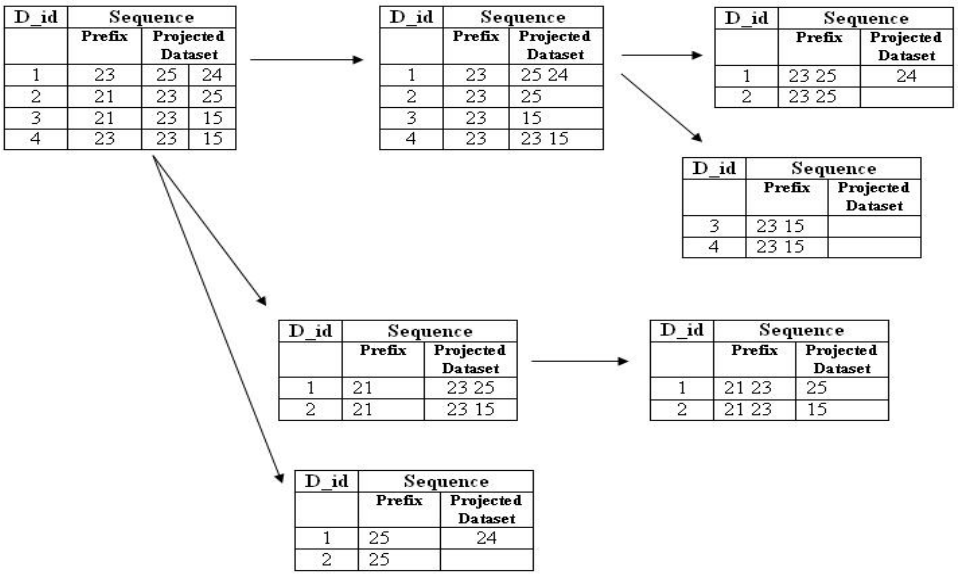

Figure 4.4: Example of sequence mining

In Figure 4.4, an example shows how prefixSpan algorithm works. Considering the sequence dataset located at the top left corner of the figure, it is made up of cells that contains the elements. Elements in the horizontal cells make a sequence and all the sequences make up the dataset. Assuming the requirement is a minimum support of 2. It is observed that elements 21, 23, 25, 15 are frequent, therefore, we call them our initial sequence patterns. For each α we create α-projected datasets, These datasets are represented by the tables projecting from the original dataset.

The first column of the projected dataset is the prefix and the rest of the dataset is the projection. Pattern 23 can be extended further by looking for frequent elements in its α-projected database. If there are no frequent elements in its projected database, we stop the grow of that pattern. However, after performing a search through pattern 23's projected dataset, it turns out that element 15 and 25 are frequent, hence new patterns, 23 15 and 23 25 are obtained with 23 as their prefix. For pattern 23 15 we stop extending the pattern because its projected dataset is empty. For pattern 23 25 we recursively make a search for frequent elements since its projected dataset is not empty. Nevertheless, we do not extend the pattern because it does not fulfill the support constraint of 2. By doing the same for other frequent elements in the dataset we have 23 15, 23 25, 21 23 and 25 as the result.

4.4.3 Gapless and Gappy Pattern

Prefixspan can extract two types of sequential patterns, *gapless* and *gappy*. Elements in a gappy pattern do not have to be continuous in the sequences from which the pattern is extracted. However, the elements need to be in the same order as they are in the sequence. For instance, sequence $\alpha =< a_1, a_2, a_3, ..., a_k >$ contains a gappy pattern $\alpha' =< a_{p_1}, a_{p_2}, a_{p_3}, ..., a_{p_m} >$, $1 \leq p_i \leq k$, $p_i < p_{i+1}$, $1 \leq m \leq k$.

In a gapless pattern, elements have to be continuous as they are in the original sequence and also have to follow the same order. For example, sequence $\beta =< b_1, b_2, b_3, ..., b_k >$ contains a gapless pattern $\beta' =< b_{p_1}, b_{p_2}, b_{p_3}, ..., b_{p_m} >$, $1 \leq p_i \leq k$, $p_{i+1} = p_i + 1$, $1 \leq m \leq k$.

Both gappy and gapless implementations of prefixSpan are applied in this thesis. Test are performed for both patterns in the same format.

4.5 PrefixSpan in Amplitude Pattern Mining

It is discovered that RMS changes during 30 seconds, so in this section we propose Amplitude Variation Mining, using PrefixSpan[31], for audio similarity detection. The idea is that songs with similar amplitude variations are algorithmically similar. We begin this section by giving some definitions that are used through the rest of the section. Then an overview of APM is given in Figure 4.5, and each step is further explained. Finally, experiments for APM approach are performed and results are evaluated.

4.5.1 Definitions

Definitions and notations are given that are used in the rest of this section. Some of them are based on definitions in Section 2.4.

Definition 24 (Sequence, Sequence Dataset (SDB)) *Given a 30-second song segment $t = (t_1, t_2, ...t_{30})$, t_i is one second segment, t_i and t_{i+1} are continuous over time, $0 \leq i \leq 30$. For each t_i, RMS is extracted, and t can be represented as a sequence $r = (r_1, r_2, ...r_{30})$, r_i is RMS for t_i, $1 \leq i \leq 30$. For each song in DB, a corresponding sequence is obtained, and Sequence Dataset (SDB) is made up of these sequences.*

Definition 25 (Pattern) *Given a sequence $r = (r_1, r_2, ...r_{30})$, a pattern is a sequence $pt = r_{j_1}, r_{j_2}, ...r_{j_m}$, $1 \leq j_m \leq 30$ and $j_m < j_{m+1}$, $1 \leq m \leq 30$.*

Definition 26 (RDB) *A set of patterns $P = \{p_1, p_2, ...p_n\}$ are mined from SDB, p_i is a pattern. For each pattern p_i, there is a set of songs who contain p_i, whose song IDs are returned as a set T_i. RDB is a dataset of T_i, $1 \leq, n \in \mathbb{N}$.*

Based on Definition 5 in Section 2.4, algorithmically similar is further specified for Amplitude Pattern Mining approach.

Definition 27 (Algorithmically Similar) *Given two songs s and t, and set of sequential patterns $\beta = \{\beta_1, \beta_2, \beta_3, \ldots, \beta_n\}$ extracted from DB. s and t are **algorithmically similar** if they contain the same set of patterns β', $\beta' \subseteq \beta$.*

Definition 28 (Similarity Degree) ***Similarity degree** sd is a number, $0 < sd \leq 1$, representing the degree of similarity between two songs. The larger sd is, the higher the degree of similarity. When $sd = 1$, two songs are exactly the same.*

Definition 29 (Candidate Song) *A candidate song has at least one similar pattern with the seed song.*

Definition 30 (Nominee Song) *A **nominee song** is a candidate song which is algorithmically similar to the seed song above the similarity degree threshold.*

4.5.2 Amplitude Pattern Mining

From experiment results in Section 4.1, RMS is promising in telling softness and hardness of songs. So we use RMS to investigate similar amplitude variation patterns. RMS values are extracted from sequential small frames of a song for all the songs and PrefixSpan algorithm is used to find songs with similar RMS variation

patterns. Amplitude pattern mining returns a set of nominee songs whose amplitude over time have a similar pattern to given a seed song.

Figure 4.5 illustrates how APM works to detect similarity in songs, using PrefixSpan. In the rest of this subsection, each step in the figure is further explained.

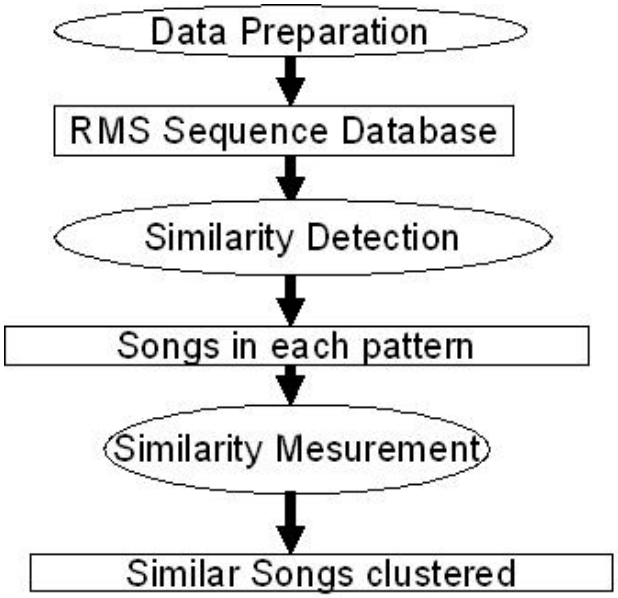

Figure 4.5: Our Approach

Data Preparation

Before using PrefixSpan, a sequential dataset SDB is build from DB. Due to the relatively small size of DB, we round up RMS values to 2 digits so that more patterns can be obtained.

Similarity Detection

In this step, songs that are algorithmically similar are found in DB. Two parameters need to be set to run PrefixSpan. One is minimum support threshold, as defined in Definition 19. We use 2 as the minimum support so that two songs are returned in the result if they contain at least one same pattern.

The second parameter is the minimum pattern length. The pattern needs to cover the sequence to some degree so that it can represent the sequence. At the same time, this length should be such that it covers the dataset as much as possible.

PrefixSpan is then run on SDB. The results is a RDB, containing sets of song IDs that participate in a same pattern.

Similarity measurement

To measure similarity, Formula 4.2 is used to calculate similarity degree. $CandCount$ is the number of times a candidate song having a pattern same to the seed song, $SeedCount$ is the number of patterns the seed song have.

$$SimilarityDegree = \frac{CandCount}{SeedCount} \tag{4.2}$$

We propose Algorithm 4 to measure similarity and return nominee song list w.r.t a seed song. The input parameters are, RDB which is the result from step in Section 4.5.2, ID of a seed song id, a similarity degree threshold ST. The output is a nominee song list that satisfies the defined threshold. This model only works for one seed song, if more than one seed song is given, Scoring model in Algorithm 3 is used to return a playlist.
An example is given in Table 4.3 to show how Algorithm 4 works. In the table, the first column is the pattern ID that are mined after running PrefixSpan on DB. For each pattern, there are sets of songs returned that contain that pattern, which is in the second column. For instance, given Table 4.3 as RDB, song23 as the seed song, 70% as the similarity degree threshold. Find all songs with similarity degree greater or equal to 70% .

Pattern	Sequence of Songs				
1	Song23	Song25	Song24	Song21	Song26
2	Song21	Song23	Song25	Song30	Song26
3	Song21	Song23	Song15	Song25	Song50
4	Song23	Song95	Song15	Song32	Song26
5	Song20	Song25	Song28	Song24	Song25
6	Song35	Song24	Song28	Song31	Song33
7	Song20	Song35	Song38	Song41	Song40

Table 4.3: Example of Song RDB

The first step is to count the number of patterns where Song23 occurs. In the table, Song23 contains pattern 1, 2, 3, 4, so $SeedCount = 4$. The songs that also contain these patterns are Song15, Song21, Song23, Song24, Song25, Song26, Song30, Song32, Song50 and Song95, they are the candidate songs. Further, for each candidate song, we get $CandCount$ by counting the number of times they appear in the same pattern sets with song23. For each candidate song, the degree of similarity is measured and only songs with similarity greater than or equal to

$ST = 70\%$ are add to L. For example, Song21 occurs in the same set with Song23 in pattern 1, 2, 3. So *CandCount* for Song21 is 3. Its degree of similarity to Song23 is $\frac{CandCount}{SeedCount} = \frac{3}{4}$. Since it is greater than threshold 70%, it is returns as a nominee song L. The same is done for all other candidate songs in the patterns 1, 2, 3 and 4. In the end, the returned L contains Song25, Song21, and Song23. Song25 and Song23.

4.5.3 Experiment

The first task in this section is to determine the best minimum pattern length. An ideal pattern length should be long enough to represent *RMS* variation of song, and short enough to cover the dataset. The test is performed by extracting sequential patterns and finding the songs that contain the same pattern. The test is initialized from the longest possible pattern in *RDB*. The pattern length is reduced by length 1 at each iteration. The songs are verified to be perceptionally similar. The test stops when the songs become different even though they contain the same pattern of certain length, no need to mention sequential all the time.

Tests are done for both gappy and gapless patterns with PrefixSpan, after performing the minimum pattern length test. For gappy pattern test, when the length is less than 10, the number of false positive is increased, and the coverage of *DB* is less than than 41%. In gapless pattern test, pattern length of 2 is used and the resulted coverage is 55.3%.

Experiment Result

Seed Song (SS)	**0.9**	**0.8**	**0.7**	**0.7**	**0.5**	**0.4**
22.wav	SS	SS	SS	SS	SS	No
12.wav	SS	SS	SS	SS	SS	SS
51.wav	SS	SS	SS	SS	SS	SS
32.wav	SS	SS	SS	SS	SS	No
49.wav	SS	SS	SS	SS	SS	No
88.wav	SS	SS	SS	SS	No	No
101.wav	SS	SS	SS	Yes	Yes	Yes
18.wav	SS	SS	SS	No	No	No
20.wav	SS	SS	SS	SS	SS	No
26.wav	SS	SS	SS	SS	SS	No

Table 4.4: Result for Similarity degree measurement with Gapless prefixSpan

Seed Song (SS)	0.9	0.8	0.7	0.7	0.5	0.4
22.wav	SS	SS	SS	SS	SS	Yes
12.wav	Yes	Yes	Yes	Yes	Yes	Yes
51.wav	SS	SS	Yes	Yes	Yes	Yes
32.wav	Yes	Yes	Yes	Yes	Yes	Yes
49.wav	SS	SS	SS	SS	SS	SS
88.wav	SS	SS	SS	SS	SS	No
101.wav	SS	SS	SS	SS	SS	SS
18.wav	SS	SS	SS	SS	SS	Yes
20.wav	SS	SS	SS	SS	SS	SS
26.wav	SS	Yes	Yes	Yes	Yes	Yes

Table 4.5: Result for Similarity degree measurement with Gappy prefixSpan

Table 4.4 and Table 4.5 presents the result of the similarity measure experiments, using gappy and gapless patterns. The tables are in the same format. In the first column are random seed songs chosen from RDB. In other columns are results for different similarity degree. SS means only the seed song is returned as nominee song. "Yes" signifies that there are nominee songs other than the seed song and all are similar to the seed song. "No" means that there are nominee songs other than the seed songs but not all are similar to the seed song.
Testing result shows that nominee songs are very few, less than 3 for each seed song on average. In most cases, the returned nominee song is only the seed song itself. A possible reason is due to the size of the DB and pattern variation of songs in DB. There are more misclassifications in gapless patterns detected in RDB in the column of degree 40% in Table 4.4. One possible reason for the misclassification is that the minimum pattern of length 2 is too short.

Conclusion

Gappy prefixSpan algorithm is better than its Gapless counterpart when appropriate pattern length is defined. When the database is not large enough, the APM does not return songs that do not satisfy the similarity degree. One drawback is the low coverage in spite of small size of DB. This is because PrefixSpan searches only for exact RMS value in the sequence. For example, close RMS value 0.24 and 0.23 are considered to be different, so that two songs can not participate in the same pattern even though the variation trend is similar. Another reason for low coverage is the pattern length, longer pattern results in less coverage. A solution could be to modify prefixSpan algorithm. A neighbourhood factor can be included so that neighbouring RMS values can be classified as similar.

Algorithm 4 PatternMining (RDB, id, ST)

1: *Input:* RDB represented as an array, ID of seed song id, similarity degree threshold ST.
2: *Output:* L, an array of nominee songs
3: *Define* A as an array of size |RDB|, each entry of format (SongID, CandCount)

4: $CandCount, Count \leftarrow 1$
5: $SeedCount \leftarrow 0$
6: **for** $i = 1$ to |RDB| **do**
7: **if** $id \in RDB[i]$ **then**
8: $SeedCount \leftarrow SeedCount + 1$
9: **for** each candidate song t in $RDB[i]$ **do**
10: **if** $t \notin A$ **then**
11: $A[Count] \leftarrow (t, 1)$
12: $Count \leftarrow Count + 1$
13: **else**
 Increment CandCount of t by 1
14: **end if**
15: **end for**
16: **end if**
17: **end for**
18: **for** $i = 1$ to |A| **do**
19: $d = SimilarityDegree(A[i].SongID) = \frac{A[i].CandCount}{SeedCount}$
20: **if** $d \geq ST$ **then**
21: $L[NomineeCount] \leftarrow A[i].SongID$
22: $NomineeCount \leftarrow$ NomineeCount + 1
23: **end if**
24: **end for**
25: **return** L

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Although NCD can detect similarity for music in MIDI format[12], based on our testing results, NCD fails in audio similarity detection. For NCD calculation, the ideal way to combine audio remains unknown and a suitable compressor is yet to be found. Amplitude RMS is a feature that can be used to classify soft and hard songs. Further, the combination of compressed file size and RMS improves performance in audio similarity detection. The variation of RMS can be taken advantage of to detect similarity. When two songs have similar amplitude variation patterns, they are similar to some degree. The idea of Sequential pattern mining is suitable for playlist generation. It guarantees minimal misclassifications and in the worst case only the seed song is returned.

5.2 Future Work

The following are some interesting directions that can be looked into for further investigation:

- As described in Section 4.5.3, one drawback of our approach is that only a limited number of patterns are obtained. One solution is to modify PrefixSpan, so that songs with close RMS values can also participate in the same pattern. For example, given two RMS sequence $\alpha = (a_1, a_2, ..., a_m)$ and $\beta = (b_1, b_2, ..., b_n)$, by introducing a δ into PrefixSpan algorithm, a pattern can be extracted from α and β if $0 \leq |a_i - b_i| \leq \delta$.

- Due to the small size of our song dataset, we are not able to simulate the real-life scenario. A possible direction is to perform more tests for both gapless and gappy patterns in a larger scale.

- Another direction is to mine patterns with other audio features, such as frequency. An idea could be to mine sequential patterns on various features of audio. Then weights are given to different feature patterns w.r.t their importance. Models can be built based on the results.

Appendix A

NCD Test

A.1 NCD Test

These source are audio pieces from Web, in each group, pieces are similar in some way. They are used to test the NCD formular, compressors and audio combination.

A.1.1 Source from Web

Segment	Length(seconds)	Start(sec.)	End(sec.)
myst10sec.wav	10	49.7	59.7
myst20sec.wav	20	44.9	64.9
myst30sec.wav	30	91.1	121.1

Table A.1: Summary times for The Magical Mystery Tour, by the Beatles

Segment	Length(seconds)	Start(sec.)	End(sec.)
spring10sec.wav	10	4.3	14.3
spring20sec.wav	20	8.4	28.4
spring30sec.wav	30	2.5	32.5

Table A.2: Summary times for Spring - Allegro, from The Four Seasons, by Antonio

Segment	Length(seconds)	Start(mm:ss)	Song Title
seg0902.wav	10	09:02	**Toto Para Me**
seg0912.wav	10	09:12	**Toto Para Me**
seg0005.wav	10	00:05	**Musica Si Theme**
seg0015.wav	10	00:15	**Musica Si Theme**

Table A.3: Different 10-second segments from same identical musical excerpt

Song	Tempo(bpm)	Song Title)
t110bpm.wav	110	**Tangerine by Apostrophe Ess**
t112bpm.wav	112	**Tangerine by Apostrophe Ess**

Table A.4: Different-tempo versions of the identical musical excerpt

Song1	Song2	Similarity	Source
e043.wav	**e108.wav**	same source,different digital copies	Mendels-SpringSong
e017.wav	**e107.wav**	same score,different performances	Tchaiko-PianoConcerto
e116.wav	**e117.wav**	same instrument,different vocalist	Kirka-SurunPyyhit
e106.wav	**e114.wav**	same melody,different otherwise	Beethoven-Symphony

Table A.5: Different types of similar music pairs.

A.1.2 Concatenation and Mixing using Flac compressor

Since NCD calculation needs the compressed file size of the combination of two audio pieces. This test is to find out which combination way is suitable for audio objects.

Song1	Song2	Concatenation(NCD)	Mixing(NCD)
myst10sec.wav	**myst20sec.wav**	0.981818181	0.523636363
myst20sec.wav	**myst30sec.wav**	0.9875	0.340
myst10sec.wav	**myst30sec.wav**	0.9875	0.62750
spring10sec.wav	**spring20sec.wav**	0.982300884	0.535398230
spring20sec.wav	**spring30sec.wav**	0.985507246	0.391304347
spring10sec.wav	**spring30sec.wav**	0.985507246	0.684057971
t110bpm.wav	**t112bpm.wav**	0.995762711	0.088983050
seg0015.wav	**seg0005.wav**	0.985549132	0.095375722

Table A.6: NCD examples for similar song segments returned by using *Flac* and differnt mixtures

A.1.3 Compressors Test: *bzip2*, *flac* and *ogg*

Three compressor are compared based on NCDs from the same test source. Since these pieces are smilar, the compressor giving smallest NCD values tells similarity best.

Song1	Song2	NCD(bzip2)	NCD(Flac)	NCD(Ogg)
myst10sec.wav	**myst20sec.wav**	0.648427609	0.523636363	0.490566037
myst20sec.wav	**myst30sec.wav**	0.995757267	0.340	0.640
myst10sec.wav	**myst30sec.wav**	0.996184255	0.62750	0.293333333
spring10sec.wav	**spring20sec.wav**	1.023187020	0.535398230	0.476190476
spring20sec.wav	**spring30sec.wav**	1.031593069	0.391304347	0.349206349
spring10sec.wav	**spring30sec.wav**	1.019983659	0.684057971	0.650793650
t110bpm.wav	**t112bpm.wav**	0.953075662	0.088983050	0.007407407
seg0015.wav	**seg0005.wav**	0.985603815	0.095375722	0.012195121
seg0912.wav	**seg0902.wav**	0.970450775	0.089364270	0.020408163
e043.wav	**e108.wav**	1.025265128	0.304347826	0.080808080
e116.wav	**e117.wav**	0.994940921	0.170316301	0.028735632
e071.wav	**e107.wav**	1.077054695	0.436781609	0.103448275
e106.wav	**e114.wav**	1.061302121	0.394531250	0.240

Table A.7: NCD examples for similar song segments returned by using *bzip2*,*flac* and *ogg*

A.2 Song Database

Two hunderds songs are used as our music data collection. Following table shows songs' IDs and their categories.

Segments	Category	Number
1-3	**Metal**	3
4-9	**Death Metal**	6
10-15	**Black Metal**	6
21-24	**New Age**	4
25-28	**Rock**	4
29-32	**Pop**	4
33-36	**Rock**	4
37-40	**Blue**	4
41-45	**Rock**	5
46-51	**Pop**	2
52,56,59	**R&B**	3
53,55,57,60,65,67	**Pop**	6
54,58,61-64,66,68	**Axe**	8
69-77	**Pop**	9
78-82	**R&B**	4
83-90	**Rock**	8
91-100	**Pop**	10
101-108	**Progressive Rock**	8
109-134	**Rock**	26
135-144	**R&B**	10
145-154	**Folk Rock**	10
155-168	**Pop**	14
169-171	**R&B**	3
172-188	**Christian**	17
189-193	**Rock**	5
194-196	**Classic**	3
197-200	**Jazz**	4

Table A.8: Two hundreds testing songs

A.3 NCD Results

This table gives an expression of what the NCD matrix contains: song IDs and NCDs. This example contains four songs and the their ten most similar songs with corresponding distance values.

SongID	SongID	NCD-value	SegmentID	SegmentID	NCD-value
1	81	-0.024514438	2	57	-0.01862017
1	9	-0.024121511	2	64	-0.015638157
1	4	-0.023587049	2	69	-0.014843587
1	77	-0.023215634	2	71	-0.014843587
1	61	-0.022766399	2	88	-0.013789436
1	29	-0.017516167	2	68	-0.013308262
1	3	-0.017216479	2	4	-0.012573838
1	65	-0.017166441	2	56	-0.01212432
1	64	-0.01602092	2	49	-0.011684299
1	58	-0.014233966	2	44	-0.011516521
3	64	-0.019513534	4	81	-0.036284573
3	1	-0.017216479	4	9	-0.032927753
3	4	-0.013949815	4	7	-0.029447634
3	49	-0.010265436	4	58	-0.024470052
3	9	-0.010136717	4	1	-0.023587049
3	11	-0.008057665	4	49	-0.023045559
3	68	-0.007904092	4	5	-0.020711995
3	61	-0.006450053	4	10	-0.020563136
3	87	-0.005914182	4	13	-0.0194255
3	2	-0.003969686	4	46	-0.01627664

Table A.9: NCDs example: smallest ten NCD values for first four songs by using Ogg compressor

Bibliography

[1] S. Pauws and B. Eggen. Realization and evaluation of an automatic playlist generator. *In Proceedings 3rd International Symposium on Music Information Retrieval (ISMIR)*, 2002.

[2] http://www.midi.org/about-midi/tutorial/tutor.shtml, October 2005.

[3] S. Blackburn and D.D. Roure. A tool for content based navigation of music. *In Proceeding ACM Conference on Multimedia*, 1998.

[4] R.J. McNab, L.A. Smith, and I.H. Witten. Towards the digital music library: Tune retrieval from acoustic input. *In Proceeding ACM Digital Libraries Conference*, 1996.

[5] A. Ghias, J. Logan, D. Chamberlin, and B. Smith. Query by humming - musical information retrieval in an audio database. *In Proceeding ACM Conference on Multimedia*, 1995.

[6] A. Meng, P. Ahrendt, and J. Larsen. Improving music genre classification by short-time feature integration. *(IEEE) International Conference on Acoustics, Speech, and Signal Processing*, pages 497–500, 2005.

[7] J.J. Aucouturier and F. Pachet. Music similarity measures: What's the use? *In Proceedings of the 3rd International Symposium on Music Information Retrieval (ISMIR02)*, 2002.

[8] E. Pampalk, A. Flexer, and G. Widmer. Improvements of audio based music similarity and genre classification. *In Proceedings of the 6th International Symposium on Music Information Retrieval (ISMIR)*, 2005.

[9] J.J. Aucouturier and F. Pachet. Improving timbre similarity : How high's the sky? *journal of Negative Results in Speech and Audio Sciences*, 2004.

[10] B. Logan and A. Salomon. A music similarity function based on signal analysis. *In Proceedings of IEEE Intl Conf on Multimedia and Expo (ICME)*, 2001.

[11] R. Cilibrasi and P.M.B. Vitanyi. Clustering by compression. *Information Theory, IEEE Transactions*, 2005.

[12] R. Cilibrasi, P.M.B. Vitanyi, and R.D. Wolf. Algorithmic clustering of music. *In Web Delivering of Music,2004. Proceedings of the Fourth International Conference*, 2004.

[13] Ruobing Li and Ruoran Zhou. Generation of music play-list using compression-based similarity. Aalborg University, 2005.

[14] J. Foote. Arthur: Retrieving orchestral music by long-term structure. *In Proceedings of the International Symposium on Music Information Retrieval (ISMIR)*, 2000.

[15] M.K. SHAN and F.F. KUO. Music style mining and classification by melody. *IEICE TRANSACTIONS on Information and Systems,Vol.E86-D, No.3*, 2003.

[16] C.R. Lin, N.H. Liu, Y.H. Wu, and A.L.P. Chen. Music classification using significant repeating patterns. *DASFAA International Conference on Database Systems for Advanced Applications*, 2004.

[17] M. Alghoniemy and A. Tewfik. A network flow model for playlist generation. *In Proceedings of IEEE Intl Conf on Multimedia and Expo (ICME)*, 2001.

[18] J.J. Aucouturier and F. Pachet. Scaling up music playlist generation. *In Proceedings of IEEE Intl Conf on Multimedia and Expo (ICME)*, 2002.

[19] B. Leslie. Signals and systems: an introduction. *Prentice Hall international (UK) Ltd*, (0-13809351-2), 1991.

[20] D. Salomon. Data compression: the complete reference. *Springer-Verlag New York, Inc.*, (0-387-95045-1), 2000.

[21] http://audacity.sourceforge.net, February 2006.

[22] http://www.bzip.org/1.0.3/bzip2-manual-1.0.3.htmlintro, March 2006.

[23] http://flac.sourceforge.net/format.html, March 2006.

[24] M. Hans and R.W. Schafer. Lossless compression of digital audio. *IEEE Signal Processing Magazine*, month 2001.

[25] http://xiph.org/vorbis/doc/vorbis_i_spec.html, March 2006.

[26] M. Li, X. Chen, X. Li, B. Ma, and P. Vit'anyi. The similarity metric. *In Proceeding of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[27] http://complearn.sourceforge.net, September 2005.

[28] J. Paulus and A. Klapuri. Measuring the similarity of rhythmic patterns. *In Proceedings of the International Symposium on Music Information Retrieval (ISMIR)*, 2002.

[29] G. Tzanetakis and P. Cook. Musical genre classification of audio signals. *IEEE Trans. Speech and Audio Process*, 10(5), July 2002.

[30] R. Agrawal and R. Srikant. Mining sequential patterns. *Proc. 1995 Int. Conf. Data Engineering (ICDE)*, pages 3–14, March 1995. Taipei Taiwan.

[31] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, and Helen Pinto *et. al. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth.*