Repeating Segment Detection in Songs using Audio Fingerprint Matching

Regunathan Radhakrishnan* and Wenyu Jiang†
* Dolby Laboratories Inc, San Francisco, USA
E-mail: regu.r@dolby.com
† Institute for Infocomm Research, Singapore
E-mail: wjiang@i2r.a-star.edu.sg

***Abstract*—[1] We propose an efficient repeating segment detection approach that doesn't require computation of the distance matrix for the whole song. The proposed framework first extracts audio fingerprints for the whole song. Then,for each time step in the song we perform a query to match a sequence of M fingerprint codewords against the fingerprints of the rest of the song. In order to find a match for the first fingerprint query, a search tree data structure is built with the fingerprints of the rest of the song. For subsequent fingerprint queries for the rest of the song, the matching process dynamically updates the search tree data structure to exclude the M fingerprint codewords corresponding to each time step. For each matching segment, we record the time offset from the query segment. Following the matching process for the whole song, we compute the histogram of the number of matching segments for each offset. The peaks in this histogram correspond to offsets at which matches were found more often than others and can be used to pick out a set of repeating segments.**

I. Introduction

Popular music often consists of repeating segments that can be thought of as audio thumbnails. These segments are potentially good representative segments for the whole song and often referred to as chorus segments. Automatically detecting these segments can help in the browsing of music collections by allowing an end-user to listen only to the chorus segments of one song to the next. Chorus playback facilitates instant recognition and identification of the songs for browsing of known songs and assessment of liking or disliking for unknown songs.

Past work on repeating pattern detection and music structure analysis in general can be divided into the following two different approaches.[1],[2],[3],[4],[5]

- In the "clustering approach" the songs are segmented into different sections by using clustering techniques. The underlying assumption is that the different sections (such as verse, chorus) of a song share certain properties that discriminates these parts from the other parts.
- The "pattern matching approach" (also denoted as sequence approach) relies on the assumption that the chorus is a repetitive section in a song. Repetitive sections are identified by matching the different sections of the song with each other.

[1]† This work was completed when Dr. Wenyu Jiang was with Dolby Laboratories Inc

The pattern matching approaches described in the literature first represent the whole song as a sequence of features extracted from N frames of the audio signal. Then, a full distance matrix is computed, which contains the distance of all combinations of all N frames of the audio signal. The computation of this matrix is computationally expensive, since the distance needs to be computed for all $\frac{N^2}{2}$ combinations.

The repeating pattern detection framework that we propose in this paper follows the pattern matching approach but doesn't require computation of the the distance matrix for the whole song. The proposed framework first extracts audio fingerprints for the whole song. Then,for each time step in the song we perform a query to match a sequence of M fingerprint codewords against the fingerprints of the rest of the song. In order to find a match for the first fingerprint query, a search tree data structure is built with the fingerprints of the rest of the song. For subsequent fingerprint queries for the rest of the song, the matching process dynamically updates the search tree data structure to exclude the M fingerprint codewords corresponding to each time step. For each matching segment, we record its time offset from the query segment. Following the matching process for the whole song, we compute the histogram of the number of matching segments for each offset. The peaks in this histogram correspond to offsets at which matches were found more often than others and can be used to pick out a set of repeating segments.

The rest of the paper is organized as follows. In the next section, we describe the proposed repeating pattern detection framework based on audio fingerprint matching. In section 3, we present experimental results illustrating the efficiency of the proposed framework.

II. Proposed Framework

Figure 1 below illustrates the three main stages of the proposed framework to detect repeating segments in a song. Input audio is first represented as a sequence of audio fingerprints. Then, for each time step short sequences of fingerprints are queried against the rest of the fingerprints of the song. The best matching sequence is found using a 256-ary tree based search algorithm. Finally, the matching results are analyzed to detect repeating segments of the song. In the following subsections, we describe each of the three processing blocks in detail.

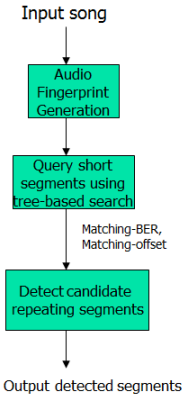


Fig. 1. Proposed Framework for Repeating Segments Detection

A. Audio Fingerprint Extraction

The goal of audio fingerprint extraction is to create a compact bitstream representation of the underlying content that can serve as an identifier. In general, audio fingerprints are designed to be robust against a lot of signal processing operations including audio coding, Dynamic Range Compression (DRC), equalization etc. However, for finding repeating segments in the same song, the robustness requirements of generated audio fingerprints can be relaxed as the matching is done within the same song. The usual attacks on an audio fingerprinting system are absent in the context of this application. In the rest of this section, we describe the audio fingerprint extraction method designed for this application.

The audio fingerprint extraction is based on a coarse spectrogram representation. First, we downmix the audio to mono signal and downsample it to 16khz . Then, we divide it into overlapping chunks and create a spectrogram from each of the chunks. Finally, we create a coarse spectrogram by averaging along both time and frequency. This operation provides robustness against small changes in the spectrogram along time and frequency. Note that the coarse spectrogram created could choose to emphasize certain parts of the spectrum more than others.

The input audio is first divided into chunks of duration T_{ch} = 2 seconds with a step size of $T_o = 16$ ms. For each chunk of audio data (X_{ch}) we compute a spectrogram with certain time resolution (128 samples or 8 ms) and frequency resolution (256 sample FFT). Then, we tile the computed spectrogram X_{ch} with time-frequency blocks. Finally, we sum up the magnitude of the spectrum within each of the time-frequency blocks to obtain a coarse representation of the spectrogram. Let us represent the spectrogram by S. We obtain a coarse representation (Q) of S by averaging the magnitude of frequency coefficients in time-frequency blocks of size $W_f \times W_t$. Here, W_f is the size of block along frequency and W_t is the size of block along time. Let F be the number of blocks along frequency axis and T be the number of blocks along time axis and hence Q is of size ($F \times T$).

Finally, we create a low-dimensional representation of a coarse spectrogram (Q) of the input audio frame by projecting the spectrogram onto pseudo-random vectors, which can be thought of as basis vectors. We generate K pseudo-random vectors each with the same dimensions as the matrix, Q ($F \times T$). The matrix entries are uniformly distributed random variables in [0, 1]. The state of the random number generator is set based on a key. Let us denote these random vectors by $P_1, P_2, ...P_K$ each of dimension $F \times T$. We compute the mean of matrix P_i and subtract it from each matrix element in P_i ($i = 1, 2...K$). Then, the matrix Q is projected onto these K random vectors to generate H_k projected values. Using the median of these projections (H_k) as a threshold, we generate K hash bits for the matrix Q. We generate a hash bit 1 for k^{th} hash bit if the projection H_k is greater than the threshold. Otherwise, we generate a hash bit of 0. In our implementation, $K = 24$. Therefore, we generate 24 fingerprint bits every 16 ms of the audio. A sequence of these 24 bit fingerprint codewords is then used as an identifier for that particular chunk of audio that it represents.

B. Audio Fingerprint Matching

The goal of the fingerprint matching block is to quickly identify offsets/lags at which repeating segments appear in a song. For every 0.64s of the input song, a sequence of 488 24-bit fingerprint codewords corresponding to ˜8s of audio is used as a query. The matching algorithm finds the best match for this sequence of bits from the rest of the song. For instance, at the current time step t = 0 a sequence of fingerprint codewords corresponding to 8s of audio is used as a query. The best matching sequence of bits is found from the database of fingerprint bits. Since we expect the repeating chorus duration to be at least 10s and do not expect the chorus to repeat one after the other consecutively, the database doesn't include the sequence of the fingerprint bits corresponding to (˜20s) 19.2 seconds starting from the current time step. For the next time step, t = 0.64, the fingerprints corresponding to 0.64s to 8.64s is used as a query and the section of the song corresponding to (19.2s to 19.84s) is removed from the database and the section corresponding to the previous time step (0 to 0.64s) is added to the database. At each time step, the database is updated and a search is performed to find the best matching sequence of bits for a sequence of query fingerprints, as illustrated in Figure 2. For each search, the following two results are recorded:

- the offset at which the best matching segment is found
- the hamming distance between the query and the best matching sequence from the database

The search itself is performed efficiently using a 256-ary tree data structure that finds approximate nearest neighbors in high-dimensional binary spaces [6].

1) Tree Data Structure based Matching: A fingerprint is a sequence of codewords with a specified length, or in essence, a binary sequence (of 488*24 = 11712 bit long in this paper). Therefore, the best match to a query fingerprint is defined as the reference fingerprint with the smallest Hamming distance to the query fingerprint. Since Hamming distance is

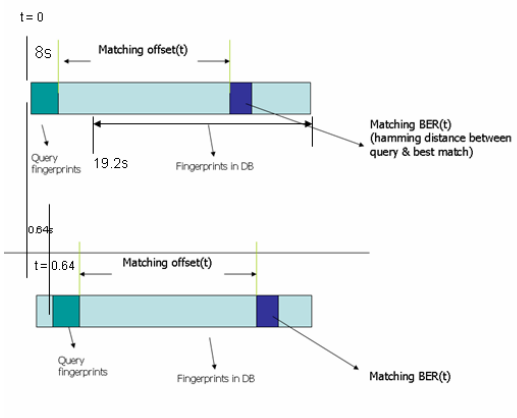


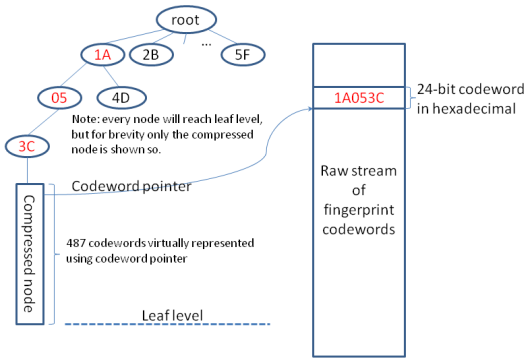
Fig. 2. Fingerprint Matching

the number of differing bits between two binary sequences, finding the best match becomes nearest neighbor (NN) search where each dimension is binary. The number of bits, i.e., the number of dimensions of a fingerprint is usually high (11712 bits in our case), and this makes NN search challenging. We implement a tree-based search structure proposed in [6], where the tree is 256-ary, i.e., each level = 1 byte, and its depth is the same as a fingerprint (488*24/8 = 1464 levels). Each reference fingerprint is inserted into the tree at 16ms stepping.

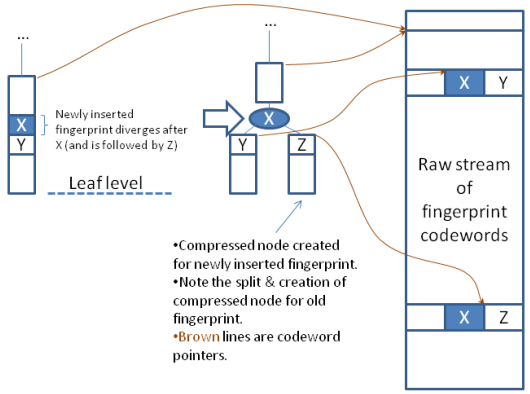
The search is implemented as a form of depth-first tree traversal starting from the root, and to reduce time complexity, at each (byte) level the branches are sorted by their Hamming distance (only on this level) to the query fingerprint's corresponding byte, and lowest Hamming distance branches are traversed first. In addition, a heuristic function is used to estimate how likely a branch will lead to leaf node whose Hamming distance will be lower than the best match so far, and traversal will continue only if the estimate is favorable. These time complexity saving measures are from [6].

In addition, we significantly reduced space complexity of such a search tree, by using codeword pointers whenever there is only one reference fingerprint underneath some intermediate node, or whenever more than one reference fingerprint have more than one byte of data in common, both of which are used in tree nodes referred to as *compressed nodes*. This is illustrated in Figure 3(a). Although [6] describes the former case as a "compressed node", there was no suggestion of use of codeword pointers and would presumably be less memory efficient. We estimate the use of codeword pointers provided about 90% savings in memory usage.

The search tree is updated with two main operations: *insertion* and *deletion* of a reference fingerprint. To facilitate the use of codeword pointers, both operations are passed with the codeword pointer of the beginning of the reference fingerprint, i.e., using the pass-by-reference instead of pass-by-value calling convention. There are two kinds of tree nodes: a sparse node (a normal node with a list of branches) and a compressed node (which has only one branch underneath for



(a) Compressed node with codeword pointer in a 256-ary search tree



(b) A compressed node being split during insertion

Fig. 3. Search tree structure and update process

multiple levels and has a corresponding codeword pointer). A leaf node is always defined to be a compressed node.

During insertion, the to-be-inserted fingerprint is traversed one level (byte) at a time, to follow the tree structure. If it has a byte value at some level that is not present in the tree, the tree is added with a new branch consisting of this byte value, and the new branch would point to a newly created compressed node whose codeword pointer points to the to-be-inserted fingerprint plus its current traversal level. Although the definition of sparse and compressed nodes are simple, insertion and deletion are not so trivial. For example, a new branch may be added in the middle of a compressed node, instead of in a sparse node, and this would require splitting the compressed node into a shorter compressed node, a single branch to a subsequent sparse node with two branches (one branch to the latter half of the old compressed node, the other to the new compressed node), and the new compressed node. This is illustrated in Figure 3(b), and note how the original codeword pointer is still kept at the new upper compressed node after the split.

During deletion, the to-be-deleted fingerprint is also traversed one level (byte) at a time, to follow the tree structure. If it exists in the tree, it can eventually be found at a compressed node (whether leaf level or not), by comparing the compressed node's codeword pointer to the passed-in codeword pointer.

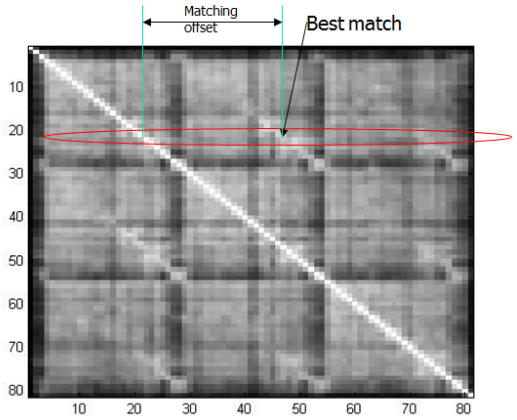


Fig. 4. Fingerprint Matching Vs Similarity Matrix based Approaches

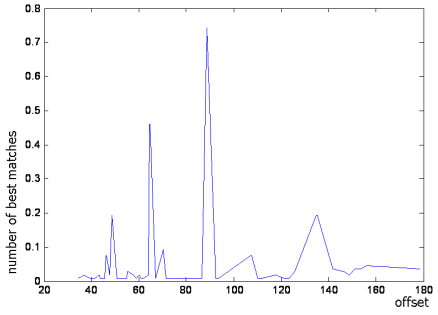


Fig. 5. Histogram of offsets

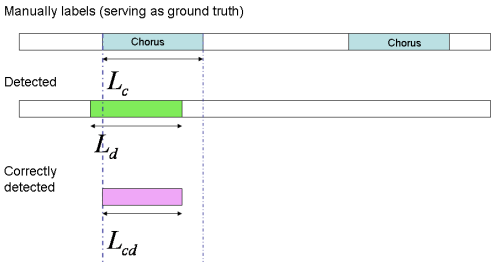


Fig. 6. F Measure Computation

If they match, and if this codeword pointer has been used in several levels of compressed nodes, then an alternative codeword pointer (from underneath the affected compressed node) must be found to replace it at each of its occurrences. In the right part of Figure 3, if fingerprint corresponding to XY is to be deleted, then the upper compressed node's codeword pointer must be updated with e.g. the codeword pointer of fingerprint XZ (and adjusted to its corresponding offset).

In short, the codeword pointer based tree structure provides significant memory savings, but requires fairly complex code to properly implement insertion and deletion.

C. Detection of significant offsets

The fingerprint matching block returns the offset of the best-matching segment in a song for every 0.64s in the song. In order to relate the proposed fingerprint matching based approach to similarity (distance) matrix based approaches, refer to Figure 4. Note that repeating patterns in the song appear as parallel lines off the diagonal in the similarity matrix. The proposed fingerprint matching based approach doesn't compute the full similarity matrix but uses a efficient tree-based search data structure to find the best match and time offset of the match for each query point. Furthermore, the fingerprint query can be efficiently executed for a song on the order of several minutes. However, it becomes computationally expensive to compute similarity matrix based on audio frame features from a song of similar length.

In this block, we determine a number of significant offsets by computing a histogram based on all offsets obtained in the previous matching step. As can be seen from the off-diagonal lines in the similarity matrix, neighboring time points within a repeating segment tend to have the same matching offset. Therefore, we first detect a set of offsets where there are significant number of matches by computing a histogram of number of matches for each offset value. Figure 5 shows an example of this histogram. The significant offsets are offsets at which there are a significant number of matches and manifest as peaks in this histogram.

Finally, the start times and end times of the repeating segments correspond to sections of the song where the best matching offsets are equal to one of the significant offsets.

III. Experimental Results

A. Objective Performance

We collected a total of 95 popular songs and manually labelled the start and end times of the repeating segments that are at least 12 seconds long. Then, we analyze each of the 95 songs using the proposed framework and record the start and end times of the detected repeating segments. We use F-measure as an objective measure to quantify how good the detected repeating segments are.

The F-measure is a common performance measure for detection tasks. It is used in the context of chorus extraction work in [3][4]. The F-measure corresponds to the amount of overlap of two different segments. For our task it specifies the overlap between the detected repeating segment and the manually labeled repeating segment (serving as ground truth). The F-measure is defined as the harmonic mean of the recall rate R and the precision rate P and is given by $F = \frac{2RP}{R+P}$ (see Figure 6). Here R denotes the recall rate and is given by the ratio of the correctly detected length to the correct part (i.e $R = \frac{L_{cd}}{L_c}$) , thus it reaches its maximum value if the detected segment fully covers the correct segment (and even exceeds it). The precision rate ($P = \frac{L_{cd}}{L_d}$) denotes the ratio of the correctly detected segment and the detected segment, thus it reaches its maximum if the detected segment does not exceed the range of the correct segment.

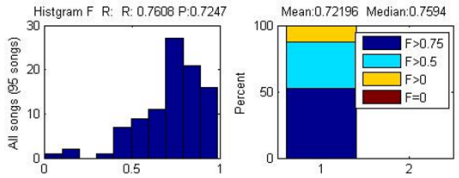


Fig. 7. Distribution of F-Measure for the set of Detected Repeating Segments from the 95 popular songs

Figure 7 shows the distribution of the F-measure values for the detected repeating segments from the 95 songs. The mean F-measure for this database of songs is 0.72196.

IV. Conclusions & Future Work

We proposed an efficient repeating segment detection approach that doesn't require computation of the distance matrix for the whole song. The proposed framework first extracts audio fingerprints for the whole song. Then, for each time step in the song we perform a query to match a sequence of M fingerprint codewords against the fingerprints of the rest of the song. In order to find a match for the first fingerprint query, a search tree data structure is built with the fingerprints of the rest of the song. For subsequent fingerprint queries for the rest of the song, the matching process dynamically updates the search tree data structure to exclude the M fingerprint codewords corresponding to each time step. For each matching segment, we record the time offset from the query segment. Following the matching process for the whole song, we compute the histogram of the number of matching segments for each offset. The peaks in this histogram correspond to offsets at which matches were found more often than others and can be used to pick out a set of repeating segments.

References

[1] Bartsch M. A, Wakefield G. H., "To catch a chorus: Using chroma-based representations for audio thumbnailing,"in Proceedings of the IEEE Workshop on Applications of Signal Processing to Audio and Acoustics. New Paltz, NY, 2001.

[2] Bartsch, M.A.; Wakefield, G.H.; "Audio thumbnailing of popular music using chroma-based representation," IEEE Transactions on Multimedia, Volume: 7 , Issue: 1 , Publication Year: 2005 , Page(s): 96 104

[3] Goto, M.; "A chorus section detection method for musical audio signals and its application to a music listening station," IEEE Transactions on Audio, Speech, and Language Processing Volume: 14 , Issue: 5 , Publication Year: 2006 , Page(s): 1783 1794

[4] Eronen A; "Signal Processing Methods for Audio Classification and Music content analysis," Phd Thesis, Tampere university of technology, 2009

[5] Paulus, J.; Klapuri, A.; " Music Structure Analysis Using a Probabilistic Fitness Measure and a Greedy Search Algorithm," IEEE Transactions on Audio, Speech, and Language Processing Volume: 17 , Issue: 6 ; Publication Year: 2009 , Page(s): 1159 1170

[6] Miller, Matthew L.; Rodriguez, Manuel Acevedo; Cox, Ingemar J.; "Audio Fingerprinting: Nearest Neighbor Search in High Dimensional Binary Spaces," Journal of VLSI Signal Processing Volume: 41, Number: 3, Publication Year: 2005, Page(s): 285-291