Query by Rhythm
An Approach for Song Retrieval in Music Databases*

James C. C. Chen and **Arbee L.P. Chen**
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 300, R.O.C.
Email : alpchen@cs.nthu.edu.tw

Abstract

*In this paper, we propose techniques for retrieving songs by rhythm from music databases. The rhythm of songs is modeled by rhythm strings. The song retrieval problem is then transformed to the string matching problem. In order to allow approximate string matching, we define similarity measures on rhythm strings. An index structure, called L-tree, is proposed to support efficient sub-string matching. Retrieval algorithms based on L-tree are then designed to provide approximate and sub- song retrieval. Experimental results s*how that this approach is effective and efficient.*

1 Introduction

As the speed and capacity of computers and networks increase, more and more new types of data are stored in databases for access, such as image, video and audio. Traditional information retrieval techniques can be applied to these data[3]. However, the applications of these data will be limited if the database lacks appropriate ways to manage the data. Recently, many software tools have been used to help humans deal with music works in composing, storing and playing. Songs are searched by some descriptions attached to them, such as file names or keywords excerpted from *lyrics*, the words of a song. However, they do not allow searching the database by melody or rhythm.

* This work was partially supported by the Republic of China National Science Council under Contract No. NSC 87-2213-E-007-001.

Wold, Blum, Keislar, and Wheaton[9] used acoustic features, such as *pitch*, *loudness*, *brightness*, and *bandwidth* to represent sounds. These acoustic features are stored as feature vectors. A model is defined to classify the sounds in the database into classes. The sound retrieval is performed by retrieving all sounds whose feature vectors fall in a set of ranges in the feature space.

Chou, Chen, and Liu[1] proposed a chord decision algorithm which transforms songs and queries into *chord* strings. The similarity of songs is implied by their chord representations. A PAT-tree like structure is used to search the chord strings in the database. Ghias, Logan, Chamberlin, and Smith[5] used a pitch tracking method to convert the songs and queries into strings of relative *pitch* changes. An approximate string searching algorithm is employed, which tolerates errors in the matching process.

In this paper, we treat music information as strings of *notes*. The typical query over music databases is *similar excerpt search*, in which the query is an excerpt of the target song. The database should return all songs containing patterns similar to the query, together with the places in the songs where the patterns appear. We propose an approach for song retrieval by rhythm as follows.

1. Represent the rhythm of a song using *rhythm string*.
2. Define the similarity relation between two rhythm strings.
3. Design an easy-to-maintain data structure for storing rhythm strings of the songs in the music database. The data structure can support efficient song retrieval with exact or similar matches.

We carry experiments on a set of folk songs to verify the effectiveness of song retrieval by rhythm and the performance of our index structure and retrieval

algorithms.

This paper is organized as follows. Section 2 describes how to represent the rhythm of songs. Sections 3 and 4 present our index structure and retrieval techniques, respectively. Experimental results for a set of folk songs are included in Section 5. Section 6 gives the conclusion.

2 Representing the Rhythm of Songs

In this section, we describe how to represent the rhythm of songs. A good representation will make the search efficient and allow flexible user interfaces and tools to be constructed[2][7].

2.1 Mubols and Rhythm Strings

We propose representing the rhythm of songs by using rhythm strings.

Definition 1. A *mubol* is the rhythm pattern of a *measure* in the song.

Definition 2. A rhythm string of a song is the string of mubols formed from the measures of the song.

Mubols are the basic units of rhythm strings. They are constructed by ignoring the *pitch* values of the notes in the measure, and considering only the *duration* of each note in the measure.

Definition 3. The *occurrence time* of each note in the mubol is the duration of all notes including *rests* before the note in the mubol.

Figure 1 shows the rhythm string of an excerpt of a song.

Figure 1(a) is an excerpt of a song and Figure 1(b) is its corresponding rhythm string which consists of 4 mubols. The pitch value of each note in the measure is ignored, only the duration remains.

2.2 Similarity Relations of Mubols

In order to define similarity relations between rhythm strings, we first define similarity relations of mubols.

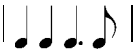
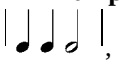
Definition 4. One mubol is the *contraction* of the other mubol if the occurrence time of each note in the two mubols coincides, and the duration of some note in the first mubol is terminated by a rest. In contrast, the latter mubol is the *extension* of the former one.

Example 1. By definition 3,  is the contraction of  , and  is the extension of  .

Definition 5. For two mubols one is the *shifting* of the other if they have the same number of notes, and exactly one note in a mubol occurs at a different time from the corresponding note in the other mubol.

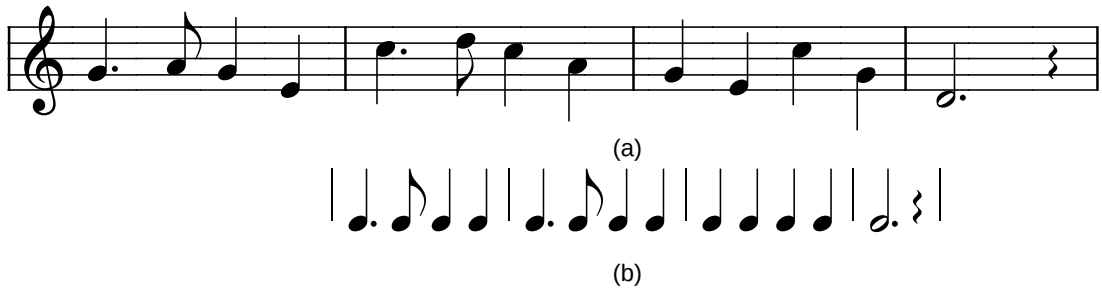
Example 2.  is the shifting of  , and vice versa.

Definition 6. A mubol is the *splitting* of the other mubol if the duration of some note in the other mubol is divided by two notes in this mubol. Meanwhile, the latter mubol is the *merging* of the former one.

Example 3.  is the splitting of  , and  is the merging of  .

2.3 Similarity Measures of Rhythm Strings

Usually two rhythm strings are similar when most notes in these rhythm strings are synchronized. Based on the similarity relations defined on mubols, we define the



(a)

(b)

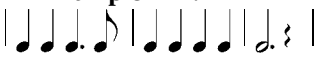
Figure 1. An excerpt of a song and its corresponding rhythm string.

similarity measure of rhythm strings of the same length in the following.

Definition 7. Two rhythm strings are the same if all mubols in the corresponding positions of these two strings are the same.

Definition 8. Two rhythm strings are k-similar to each other if one rhythm string differs from the other by k *changes* of mubols. A change of a mubol can be a replacement of its contraction, extension, shifting, splitting or merging.

Two 0-similar rhythm strings are also the same.

Example 4.  and  are 1-similar to each other, while  and  are 3-similar to each other.

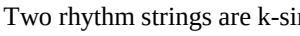
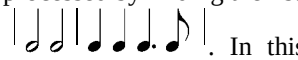
The rhythm strings are invariant to *transposition* (a change of *key*) since the pitch values of the notes are ignored in the rhythm strings. Different transpositions of the same song will have the same rhythm string.

3 Index Structure : L-Tree

In our approach of song retrieval by rhythm, the capability of sub-string matching is needed since we assume queries over music databases are excerpts of the target songs. The index structures should fit for sub-string matching.

3.1 Basic Idea

Since the lengths of queries can be mostly around some value, we want our index structure to have the best performance for queries which are shorter (in terms of the number of mubols) than a predefined value n. Our basic idea is as follows. For all possible sub-strings of length i, or *i-grams* where $1 \le i \le n$, of the rhythm strings, we maintain a list of occurrences of the i-grams within the rhythm strings. To process a query of length $\le n$, we can search the list of occurrences of the corresponding i-gram. Queries of lengths $> n$ are discussed in Section 4.2.

Example 5. A rhythm string and its lists of occurrences of all 1-grams, 2-grams, and 3-grams are shown in Figure 2. The number before the colon in the lists indicates the song ID of the i-gram, and the numbers after the colon are the positions of the i-gram within the rhythm string. A query, say  , over the rhythm string can be processed by finding the list of occurrences of the 2-gram  . In this case, we know  appears in the 3rd and 6th measures of the first song from the list of occurrences of .

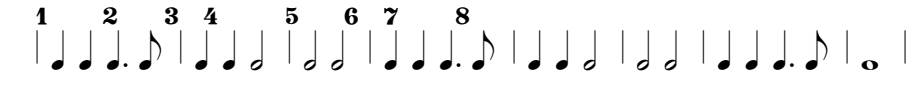
3.2 L-Tree

The lists of occurrences of the i-grams of the rhythm strings can be organized as a *trie* structure, named *L-tree*. A trie is a digital search tree constructed from a set of strings[6]. The ith symbols of the strings are stored in the internal nodes in level i of the tree. The symbols of a string are stored in the nodes in the path from the root node to a leaf node. The symbols in the internal nodes are used to direct the trie construction and string searching.

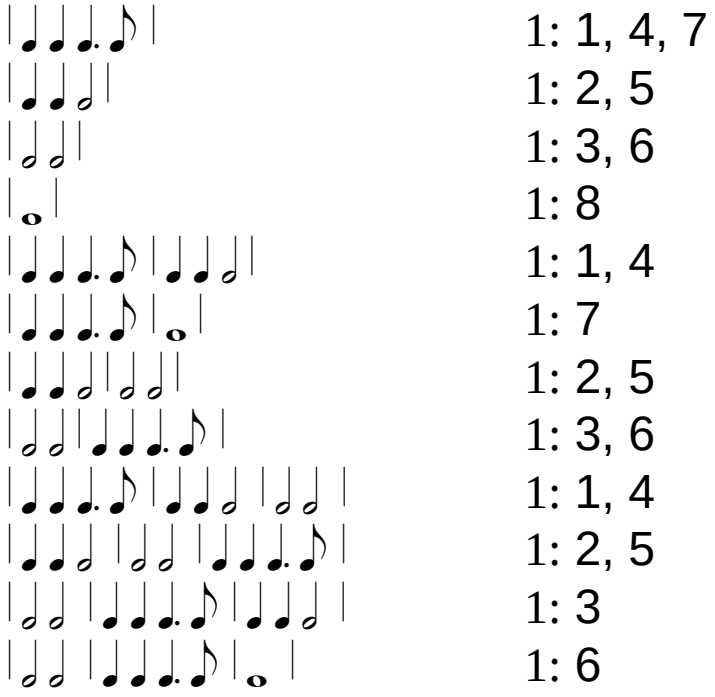
There is a parameter n of the L-tree which indicates that all sub-strings of the strings of length $\le n$ will be inserted to the L-tree.

Example 6. The lists of occurrences shown in Figure 2(b) can be organized as an L-tree as shown in Figure 3. The nodes connected by the solid arrows form the trie structure. Level 1 of the L-tree stores the first symbol of the i-grams, and level 2 store the second symbol, and so on, $1 \le i \le n$. The dotted arrows point to the lists of the occurrences of the i-grams stored in the path from the root to the nodes in level i.

L-tree can achieve index space reduction since the common prefix of the i-grams is represented as a single path of the L-tree. That is, the common prefix of the i-grams is stored only once in the L-tree. This space reduction is especially important when indexing very large databases.



(a)



(b)

Figure 2. A rhythm string and the lists of occurrences of the *i*-grams.

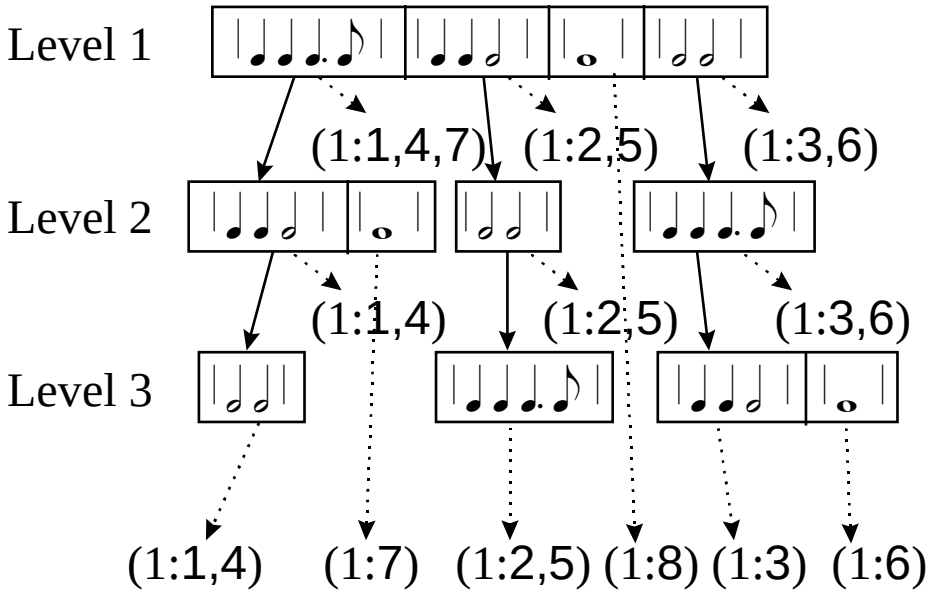


Figure 3. An L-tree built on the lists of the occurrences shown in Figure 2(b).

3.3 The Size of L-Tree

In this subsection, we examine the space required by the L-tree. The number of internal nodes in the L-tree is bounded by *nl*, *l* is the total length of the rhythm strings in mubols. The number of internal nodes in the L-tree would be less than *nl*, if some *n*-grams share the same prefix. Moreover, the lists in level *i* of the L-tree are the merger of the lists in level *i+1*. If the lists to be merged in level *i+1* contain occurrences of the same rhythm string, the duplicated song IDs can be removed. The size of the total lists in level *i* would be smaller than the size of the total lists in level *i+1*. Thus the size of the total lists in the L-tree is bounded by *2nl*. Note that these lists can be compressed via Huffman or other run length codings[8].

Thus we have an index structure of linear space complexity **O**(*nl*), with respect to the database size. Also, the size of the L-tree can be controlled by *n*.

3.4 Similarity Table

In order to process queries with some degree of similarity, another data structure called *similarity table* is needed, which records the possible changes of each mubol that exists in the index.

An example similarity table is shown in Figure 4.

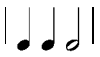
Mubol	Possible Changes
	
	

Figure 4. An example similarity table.

4 Retrieval Techniques

Using the similarity measure of the rhythm strings defined in Section 2.3, we consider how to retrieve songs by rhythm, and show how to process queries using L-tree. The queries can be classified by their matching criteria as follows.

1. Exact match. Given a rhythm string as a query, we want to find all songs containing the rhythm string.
2. *K*-Similar match. Given a rhythm string as a query, we want to find all songs containing rhythm strings that are *m*-similar to the query, where $0 \le m \le k$.

The database should report the song IDs along with the occurrences of the query within these songs.

Depending on the length of the query, two different kinds of retrieval strategies are used.

4.1 Query Length ≤ *n*

In this subsection, we describe how to process the query when the length of the query is ≤ *n*. We start from the algorithm that retrieves songs exactly containing the query.

4.1.1 Exact Match If the length of a query is ≤ *n*, the search ends down to some node of the L-tree where we exhaust the query. The list of the node contains all occurrences of the query. This terminates the search algorithm.

Example 7. Search  on the L-tree in Figure 3. We test the first mubol  in level 1; following the 2nd branch, we test the second mubol  in level 2. The mubol  appears in level 2, so we get its list, (1: 2, 5) as the answer.

As a result the search time is proportional to the length of the query.

4.1.2 K-Similar Match The basic idea to retrieve songs with *k*-similar match is by finding all rhythm strings containing variants of the query within *k* changes using the L-tree.

The system first generates a *template* of the query by referencing all the possible changes of the mubols in the query in the similarity table. Level *i* in the template stores the *i*th mubol of the query and all its possible changes. The template is used to search strings that are *k*-similar to the query on the L-tree. The search process is like the one in exact match except that when finding the *i*th mubol of the query in level *i* of the L-tree, it can be replaced by its possible changes.

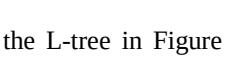
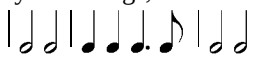
Example 8. Assume we want to retrieve all songs which contain rhythm strings 1-similar to  using the L-tree in Figure 3 and the similarity table in Figure 4. The template, shown in

Figure 5(a), is represented by linking the three levels with solid lines. The paths formed by dotted lines indicate that the candidate strings are 1-similar to the query, which exist in the L-tree. In this example, two rhythm strings, shown in Figure 5(b), are 1-similar to  and their lists contain the answers.

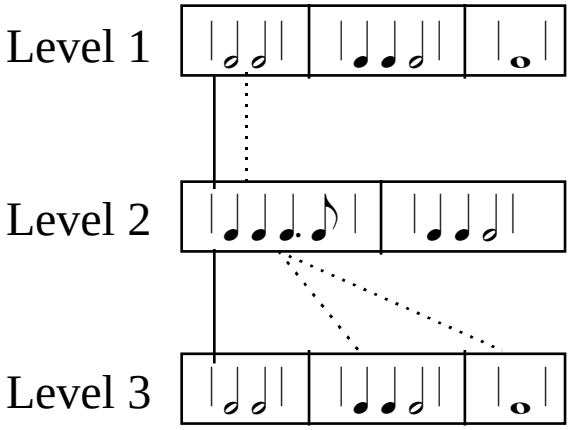
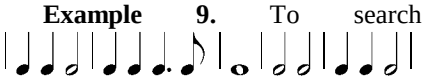
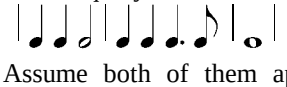
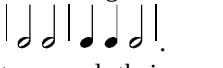


Figure 5. 1-similar match for a query shorter than *n*.

4.2 Query Length > *n*

4.2.1 Exact Match In the case where the length of a query is > *n*, the query is divided into sub-queries of lengths ≤ *n*. For each sub-query, we search its corresponding lists on L-tree. The lists of these sub-queries are then aggregated to get the lists of the query. The lists are aggregated by a difference *d*, where *d* is determined by the length of the first sub-query to be aggregated. An occurrence in the first list is preserved only if there is a corresponding occurrence in the second list which has the same song ID and is larger than the first occurrence by *d*.

Example 9. To search the query  with exact match, the query is divided into a 3-gram,  and a 2-gram  . Assume both of them appear in the L-tree, and their corresponding lists are as shown in Figure 6. These two lists are aggregated by a difference 3 to get the list of the query. In this case, the list is (2: 5; 4: 1).

4.2.2 K-Similar Match In the case of *k*-similar match, the query is divided into sub-queries of lengths ≤ *n*. For each sub-query, we invoke *k*-similar match as discussed in 4.1.2. The rhythm strings that are *k*-similar to the query are *concatenated* from the strings that are *k*-similar to the sub-queries. The lists of occurrences of the rhythm strings that are *k*-similar to the query are the aggregation of the lists of occurrences of the strings that are *k*-similar to the sub-queries. Note that the number of changes in the concatenated strings equals to the sum of the changes in the strings that are concatenated. The list aggregations need not be performed if the number of changes of the concatenated strings has already exceeded *k*.

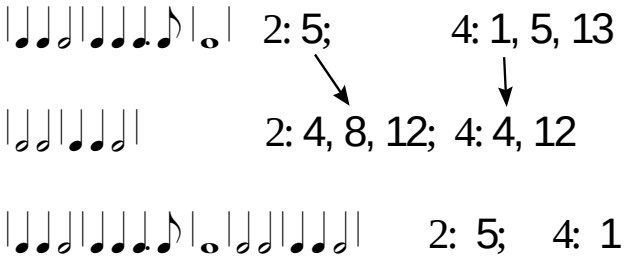


Figure 6. The list aggregation of a 3-gram and a 2-gram.

4.2.3 Ways of Query Division If the length of a query is > *n*, there are many ways we can divide the query into sub-queries. Different ways of query division will lead to different performances. We use a simple model to illustrate the performance of different ways of query division. This model computes the total length of the lists to be aggregated using the following L-tree property: the list of occurrences of some non-leaf node is the merger of all its children's lists.

Assume each non-leaf node in the L-tree has *b* children nodes, and the length of the lists at each leaf node is *f*. Then the length of the lists at the node in level *i* is $f * b^{(n-i)}$ where n is the number of levels of the L-tree.

Example 10. If the length of the query is 5, we can either divide the query into two sub-queries of lengths 4 and 1 or divide it into two sub-queries of lengths 3 and 2. Assume $n = 4$, $b = 3$ and $f = 1$. The total length of lists to be aggregated when dividing the query into sub-queries of

lengths 4 and 1 is 28 ($3^0 + 3^3$), while the total length of lists to be aggregated when dividing it with lengths 3 and 2 is 12 ($3^1 + 3^2$). In this case, the latter way of query division will lead to better performance.

In general, it is better to divide the query string into sub-strings of similar lengths.

5 Experiments

We carried out experiments to examine the power of querying songs by rhythm and analyze the performance of the L-tree.

5.1 Experiment Setup

The L-tree is built on rhythm strings of 102 folk songs. The average notes per measure is 3.72. We examine the power of retrieving songs by rhythm and investigate the following questions:

1. The discriminability of songs using different lengths of rhythm strings. What is the suitable length of rhythm strings for the song retrieval by rhythm to be effective?
2. The size of L-tree? How does it grow as a function of *n*?
3. The relationship between the lengths of queries and *n*? We compare the response times of query processing using L-tree and PAT-tree[4] for the case of exact match. PAT-tree is also a trie structure constructed over all *sistring* of a string. A sistring is a suffix of a string starting at some position.
4. How does the parameter k of k-similar match affect the response time?

5.2 Song Discriminability

To see the discriminability of songs using different lengths of rhythm strings, for all sub-strings of the rhythm strings in the database whose lengths are between 1 and 8, we average the numbers of songs containing each of these sub-strings. As shown in Figure 7, the horizontal axis indicates the lengths of the sub-strings of the rhythm strings. Sub-strings with four mubols are sufficient to distinguish a song from others.

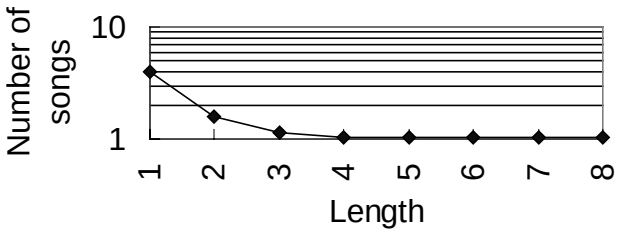


Figure 7. The average numbers of songs that contain the sub-strings of the rhythm strings.

5.3 Size of L-Tree

We built L-trees for four different *n's*, 1, 2, 3, 4 and compared the size of L-trees and PAT-tree. Figure 8 shows that the size of L-tree grows linearly as a function of *n*. For small *n*'s, the size of L-tree is much smaller than that of PAT-tree.

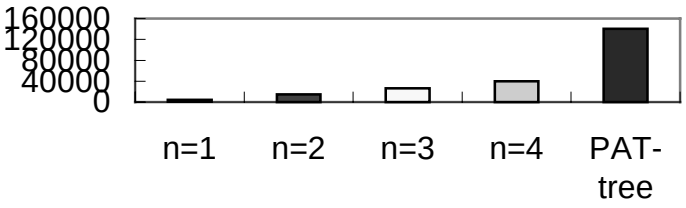


Figure 8. The size of L-tree.

5.4 Response Time for Exact Match

To verify the performance of the retrieval algorithms using L-tree, we examined the response time of exact match for n = 4. We varied the lengths of queries and compared the response time of query processing using L-tree and PAT-tree.

Figure 9 shows the result. The horizontal axis indicates the length of queries . In general, a larger *n* will lead to better performance. Note that if the length of queries is ≤ 4, the result can be obtained directly on L-tree, no list aggregation is needed. We have very short response time. However, the PAT-tree needs to invoke a prefix search and merge all lists of occurrences. For a query of length > 4, we divide it into sub-queries of length = 4 and ≤ 4. The system needs more time to aggregate the lists of occurrences of the sub-queries. On the other hand, the PAT-tree approach prefers longer queries. The best case for PAT-tree is that the query happens to be a string in the

PAT-tree. In this case, no prefix search and list merger are needed. The response time when the length of query is > 4 can be shortened by either applying different ways of query division or using a larger *n*.

5.5 Response Time for *K*-Similar Match

To see how different *k*'s affect the response time for *k*-similar match, the response times for exact match, 1-similar match and 2-similar match are shown in Figure 10. The way of query division employed here is the same as that in Section 5.4. A larger *k* will induce a longer response time since there are more sub-queries to be connected and more lists to be aggregated.

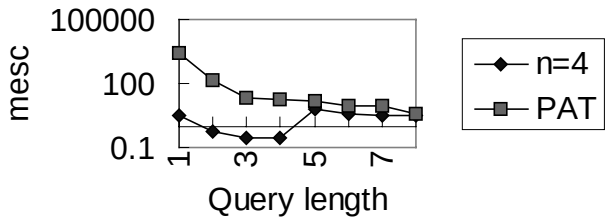


Figure 9. Response time for exact match.

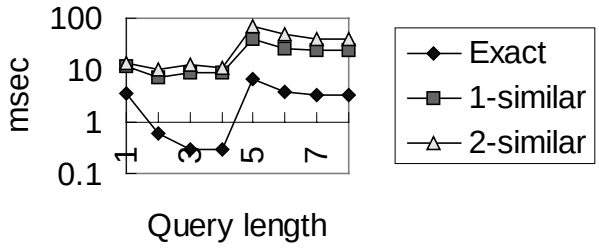


Figure 10. Response time for *k*-similar match.

6 Conclusion

In this paper, we presented the techniques for retrieving songs by rhythm. The rhythm strings are chosen as the features of songs. The definitions of similarity relations on rhythm strings capture the similarity among songs. The *n*-grams excerpted from the rhythm strings areorganized as an L-tree to reduce the space overhead. By adjusting the *n* of the L-tree, we can control the response time and index size as we want. We also propose methods for exact match and *k*-similar match of rhythm strings.

The future work include:

1. Extend the rhythm string model to represent the *melody* of songs. This can be done either by considering the pitch values of the notes in the measures, or extending the similarity relations of rhythm strings.
2. Use the approach to analyze songs and discover *main themes* of songs.
3. Apply L-tree to other time series data which can be modeled using strings over another alphabet.

References

[1] Ta-Chun Chou, Arbee L. P. Chen, and C. C. Liu. Music database: indexing technique and implementation. In *Proc. of IEEE Intl. Workshop on Multimedia Data Base Management System*, 1996.

[2] Roger B. Dannenberg. Music representation: issues, techniques, and systems. In *Computer Music Journal*, Vol. 17, No. 3, pp. 20-30, 1993.

[3] Christos Faloutsos. A survey of information retrieval and filtering methods. Tech. Report CS-TR-3514, Dept. of Computer Science, Univ. of Maryland, August 1995.

[4] William B. Frakes, and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, New Jersey. 1992.

[5] Asif Ghias, Jonathan Logan, David Chamberlin, and Brian C. Smith. Query by humming: musical information retrieval in an audio database. In *Proc. of ACM Multimedia Conference*, pp. 231-236, 1995.

[6] Ellis Horowits and Sartaj Sahni. *Data Structures in Pascal*. W.H. Freeman and Company, New York, 1990.

[7] Hagit Shatkey and Stanley B. Zdonik. Approximate queries and representation for large data sequences. In *Proc. of IEEE 12th International Conference on Data Engineering*, pp. 536-545, 1996.

[8] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York. 1994.

[9] Erling Wold, Thom Blum, Douglas Keislar, and James Wheaton. Content-based classification, search, and retrieval of audio. In *IEEE MULTIMEDIA*, Vol. 3, No. 3, pp. 27-36, Fall 1996.