A Qualitative Comparison (Comparative Study) of Various Techniques for Content-based Music Information Retrieval

Jia-Lien Hsu and Arbee L.P. Chen[1]

Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 300, R.O.C.
Email: alpchen@cs.nthu.edu.tw

ABSTRACT

In this article, we discuss the techniques used in content-based music information retrieval. The techniques include the methods to represent music objects, the similarity measures of music objects, and indexing and query processing for music object retrieval. To represent music objects, we introduce three coding schemes, i.e., chord, mubol, and music segment. Various similarity measures are then presented, followed by various index structures and the associated query processing algorithms. The index structures include suffix tree, n-gram, and augmented suffix tree. A qualitative comparison of these techniques is finally presented to show the intrinsic difficulty of the problem of content-based music information retrieval. In addition, the platform for evaluating MIR approaches is introduced. Some preliminary results of efficiency study are illustrated.

1. INTRODUCTION

Rapid progress of hardware and software technologies makes it possible to manage and access large volumes of multimedia data. While most researchers focus on the content-based retrieval of image and video data, it is getting more attention on the content-based retrieval of audio and music data. In the area of multimedia databases, the content representation, indexing and searching techniques for multimedia data are key issues.

A friendly interface for users to pose queries is important in content-based music data retrieval. The queries are usually a piece of music, which should not be restricted to be the *incipits*, the beginning of music objects, but can begin at any position of the music objects. Meanwhile, it is required to provide the functionality of *approximation* in music databases when taking into account the errors caused by frequent inexact query specifications. It is also

[1] To whom all correspondence should be sent.

reasonable to provide functionalities of *transposition*[2] invariant and scale invariant for query processing, which will be discussed in Section 2.

To meet the requirements of content-based retrieval, the techniques in music databases are developed, as shown in Figure 1. By using proper coding schemes and similarity measures, it fulfills the requirement of approximation, transposition invariant and scale invariant. With the benefit of index structures, the functionality of partial matching can be supported, and user queries can be efficiently processed even when a large number of music objects are stored in the databases.

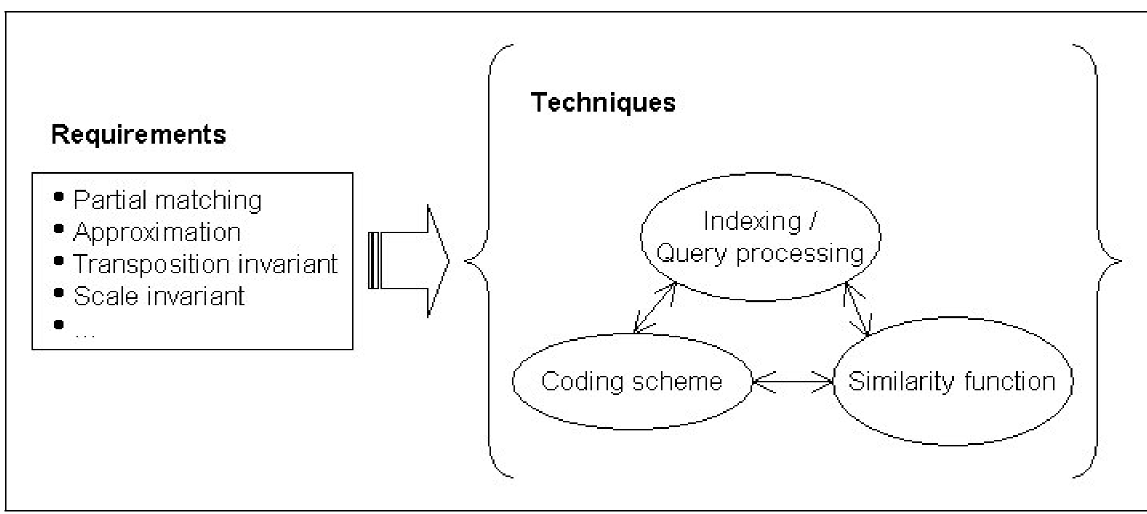


Figure 1: Techniques used in music databases to meet the requirements.

In this article, we introduce various techniques for content-based music retrieval. The issues of representing music objects, similarity measure, index construction, and query processing are discussed. The rest of this article is organized as follows. In Section 2, we focus on the representation issues and present three coding schemes of music objects. The similarity functions based on these schemes are presented as well. The index structures and query processing are described in Section 3 and Section 4, respectively. In Section 5, the discussion and comparison of various techniques are presented. We describe the platform for evaluating the performance of each approach in Section 6, as well as the experiment results to show the efficiency. Finally, Section 7 concludes this article.

1.1. Related Work

The contributions in the field of music databases are presented as follows. Selfridge-Field (1998) provides a survey of clarifying and resolving conceptual and representational issues in melodic comparison. Ghias, *et al.* (1995) propose an approach for modeling the content of

[2] The shifting of composition into another key without any alteration of melodic intervals so that the result sounds exactly the same within itself apart from being at a different basic pitch (Burke, 1993).

music objects. A music object is transformed into a string which consists of three kinds of symbols, ‘U’, ‘D’, and ‘S’ which represent a *note* is higher than, lower than, or the same as its previous note, respectively. The problem of music data retrieval is then transformed into that of approximate string matching. Prather (1996) proposes a linked list data structure to store the *scores* of music data. By applying the *harmonic analysis method* to traverse the scores, the *chord* information can be derived for further processing. In the paper (Blackburn and DeRoure, 1998; DeRoure and Blackburn, 2000), a system supporting the content-based navigation of music data is presented. A sliding window is applied to cut a *music contour* into sub-contours. All sub-contours are organized as an index structure for the navigation. Tseng (1999) proposes a content-based retrieval model for music collections. The system uses a pitch profile encoding for music objects and an *n*-gram indexing for approximate matching. Similar techniques have also been employed in the papers (Yanase and Takasu, 1999; Uitdenborgerd and Zobel, 1999; Downie and Nelson, 2000; Yip and Kao, 2000). The work (McNab, *et al.*, 2000) focuses on music retrieval from a digital library. The issues of *melody* transcription and the requirements of string matching are analyzed to show the trade-off between the matching criteria and retrieval effectiveness.

When considering polyphonic music objects, Uitdenbogerd and Zobel (1998) propose a method for extracting the monophonic melody from a polyphonic music object in MIDI format. A framework is also proposed in which the music objects are organized as an *n*-gram structure for efficient searching (Uitdenborgerd and Zobel, 1999). Lemstrom and Perttu (2000) also present a prototype handling polyphonic music objects. Using dynamic programming, a bit-parallel algorithm is presented for efficiently searching melodic excerpts. In the bit-parallel processing, the whole table for dynamic programming need not be created, and thus it leads to a better performance. Clausen, *et al.* (2000) design a web-based tool for searching polyphonic music objects. The applied algorithm is a variant of the classic inverted file index for text retrieval. A prototype is implemented and its performance is investigated.

To develop a content-based music database system, we have implemented a system called *Muse* (Chen and Chen, 1998; Chou, Chen and Liu, 1996; Liu, Hsu, and Chen, 1999a). In this system, we use various methods for content-based music data retrieval. The *rhythm*, melody, and chords of a music object are treated as music feature strings and a data structure called *1D-List* is developed to efficiently perform approximate string matching (Liu, Hsu, and Chen, 1999a). Similarity measures in the approximate string matching algorithm are based on music theory. Moreover, we consider music objects and music queries as sequences of chords (Chou, Chen and Liu, 1996) and mubol strings (Chen and Chen, 1998). An index structure is developed for each approach to provide efficient matching capability. In the paper (Chen and

Chen, 1998), we propose an approach for retrieving music objects by rhythm. Instead of using only melody (Blackburn and DeRoure, 1998; DeRoure and Blackburn, 200; Chou, Chen and Liu, 1996; Ghias, *et. al.*, 1995; Liu, Hsu, and Chen, 1999a) or rhythm of music data, we consider both of the information plus the music contour and define *music segments* to represent music objects (Chen, *et. al.*, 2000). Two index structures, called *one-dimensional (1-D) augmented suffix tree* and *two-dimensional (2-D) augmented suffix tree*, are proposed to speed up the query processing. By specifying the similarity thresholds, we provide the capability of approximate song retrieval. When considering more than one feature of music objects for query processing, we propose multi-feature index structures (Lee and Chen, 2000). With the multi-feature index, both exact and approximate search functions on various music features are provided.

In addition to the issues of music data retrieval, we also address the feature extraction problem (Hsu, Liu, and Chen, 2001; Hsu, Liu, and Chen, 1998; Liu, Hsu, and Chen, 1999b). We define *repeating patterns* as sequences of notes which appear more than once in music objects. Repeating patterns are considered one of the most expressive features of music objects which meet both efficiency and semantic-richness requirements for content-based music data retrieval. We propose two efficient algorithms to extract repeating patterns from music data in (Hsu, Liu, and Chen, 2001; Hsu, Liu, and Chen, 1998; Liu, Hsu, and Chen, 1999b).

2. THE REPRESENTATION OF MUSIC OBJECTS

Instead of handling raw data of wave format in audio retrieval system, music objects are often handled in score-like or symbolic format, such as MIDI format (MIDI) in music information retrieval. Music objects can be *monophony*[3] or *polyphony*[4]. In this article, we only consider monophonic music objects.

In the following, we introduce three approaches of representing music objects and the corresponding similarity measures.

2.1. Chord

It is not expected that queries can be exactly specified, especially for non-expert users. Four types of approximation, *i.e.*, duplication, elimination, disorder, and irrelevance, are defined in the paper (Chou, Chen, and Liu, 1996), which are handled by the chord

[3] Music for a single voice or part, e.g., plainchant and unaccompanied solo song. The term is contrasted with 'polyphony' (music in two or more independent voices) (Sadie, 1988).
[4] Term, derived from the Greek for 'many-sounding', used for music in which two or more notes sound simultaneously. It is used in distinction to monophony ('one-sounding', for music consisting of a single line) (Sadie, 1988).

representation. A chord is the simultaneous sounding of two or more notes (Stanley, 1980). The use of chords is the basic foundation of harmony. A chord set is given in Figure 2.

C1: do	C2: do, mi	C: do, mi, sol	C7: do, mi, sol, si
D1: re	D2: re, fa	Dm: re, fa, la	Dm7: re, fa, la, do
E1: mi	E2: mi, sol	Em: mi, sol, si	Em7: mi, sol, si, re
F1: fa	F2: fa, la	F: fa, la, do	F7: fa, la, do, mi
G1: sol	G2: sol, si	G: sol, si, re	G7: sol, si, re, fa
A1: la	A2: la, do	Am: la, do, mi	Am7: la, do, mi, sol

Figure 2: A chord set.

Given a music object, the coding process based on the chord set is described as follows. For each measure, the five principles are applied one by one to determine its chord representation. As a result, a music object is represented as a *chord string*.

THE PRINCIPLES FOR THE CHORD DECISION:

1. Find the candidates of chords which contain the most notes in the measure.
2. Preserve the minimal-length chords.
3. Preserve the chords whose *roots*[5] have the maximal occurrence frequency.
4. Preserve the chords whose *fifths* have the maximal occurrence frequency.
5. Preserve the chords whose *thirds* have the maximal occurrence frequency.

Example 1

Given a music object of two measures, say |sol, mi, mi| do, mi, la, sol, do|. For each measure, we apply the five principles to determine the chord for the measure.

MEASURE 1:

By applying Principle 1, the set of candidate chords is {‘C’, ‘C7’, ‘E2’, ‘Em’, ‘Em7’, ’Am7’}. Next, according to Principle 2, only ’E2’ is left, which represent the measure.

MEASURE 2:

By applying Principle 1, the set of candidate chords is {’C’, ’C7’, ’Am’, ’Am7’}. Then, according to Principle 2, the candidate set is {’C’, ‘Am’}. By applying Principle 3, chord representation of the measure is determined as ’C’.

Therefore, the chord string of the music object is ‘E2’-‘C’.

5 The “root” of a chord is the note on which it seems to be built (Stanley, 1980). The “third” is the note which is three degree to the root and the “fifth” is the note which is five degree to the root.

2.2. *Mubol*

Considering the rhythm information of music objects, the representation of music objects by mubol strings is discussed as follows.

DEFINITION 1:

A *mubol* is the rhythmic pattern of a measure in a music object. A *mubol string* of a music object is the string of mubols which are determined by the measures of the music object.

Mubols are the alphabet of mubol strings. For example, as shown in Figure 3, the mubol string R consists of four mubols to represent the four measures of the original music object.

Figure 3: An example of music object and its corresponding mubol string R.

Five operators over mubols are defined in the paper (Chen and Chen, 1998) and illustrated in Figure 4. By means of these operators, the similarity measure of two mubol strings is defined as follows.

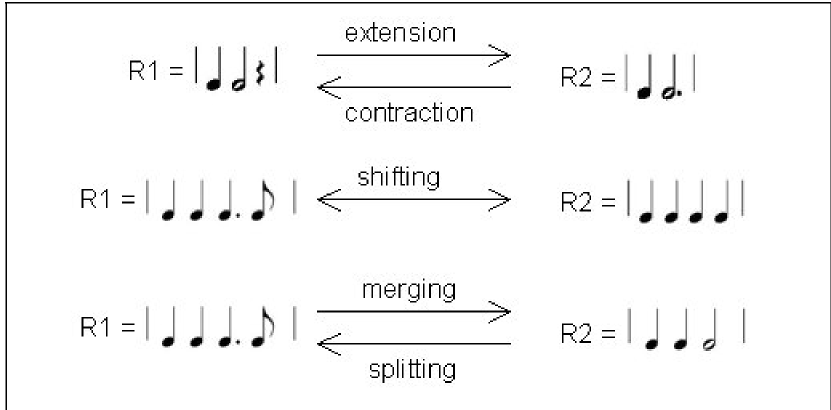


Figure 4: The illustration of five operators over mubols.

DEFINITION 2:

Two mubol strings A and B of the same length are *k-similar* to each other if A can be changed to B by applying *k* times of the operators on A, or vice versa.

Example 2

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

2.3. *Music Segment*

The concept of music contour has been employed in (Ghias, *et. al.*, 1995; Blackburn and DeRoure, 1998; Liu, Hsu, and Chen, 1999a; Selfridge-Field, 1998) for retrieval of music data. From the study (Selfridge-Field, 1998), "recently studies in musical perception suggest that durational values may outweight pitch values in facilitating melodic recognition", in this subsection, we describe an approach which takes into account of music contour and rhythm to represent music objects.

A *music segment* is a triplet which consists of the *segment type* and the associated beat and pitch information. There are four segment types defined to model the music contour, *i.e.*, (type A), (type B), (type C), and (type D). Define the *segment base* as the horizontal part of a music segment. The beat information of a music segment is represented by the *segment duration* which is the number of beats in the corresponding segment base. The pitch information of a music segment is represented by the *segment pitch* which is the *note number* in the MIDI standard of the corresponding segment base minus the note number of the segment base of the previous segment base. For example, for the piece of music shown in Figure 5, the corresponding representation as a sequence of music segments is shown in Figure 6. The music segment (A, 1, +1) indicates that it is a type A segment with the segment duration and segment pitch being 1 and +1, respectively. When coding by music segments, the first music segment and the last music segment are ignored due to lack of information to assign the segment type.

Figure 5: A piece of music.

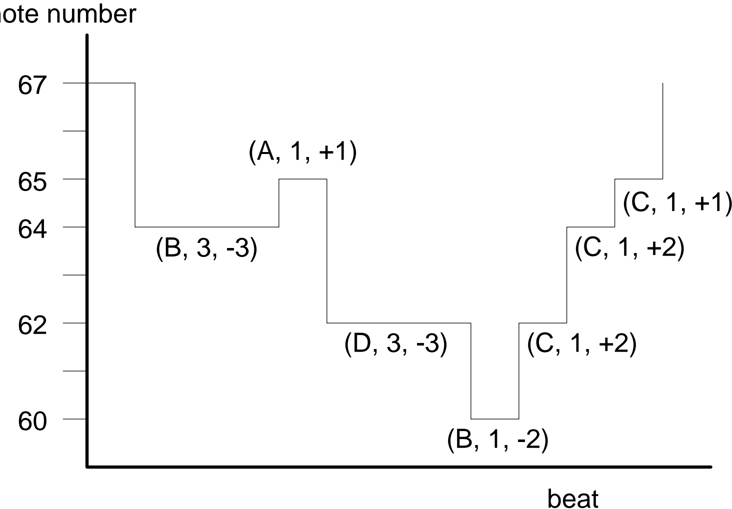


Figure 6: The corresponding sequence of music segments of the score in Figure 5.

There exist transposition and inaccurate pitch or rhythm in the queries. The music contour, however, is usually correct. Therefore, the music segment approach (Chen, *et. al.*, 2000) requires that the music contour of the query (*i.e.*, the sequence of the segment types of the query) have to exactly match with the results from the database. A dissimilarity measure for approximate retrieval of music data based on the beat and pitch information is defined as follows.

DEFINITION 3: *DIS_SIM* (*Q, C*)

Given two sequences of music segments of the same length, a query sequence $Q = (i_1, j_1, k_1)$ (i_2, j_2, k_2) ... (i_n, j_n, k_n), and a candidate sequence from the database $C = (i_1, x_1, y_1)$ (i_2, x_2, y_2) ... (i_n, x_n, y_n).

$$DIS_SIM(Q,C) = w_{duration} \times dis_sim_{duration} + w_{pitch} \times dis_sim_{pitch}$$

where (1) $w_{duration} + w_{pitch} = 1$ and $0 \le w_{duration}, w_{beat} \le 1$

(2) $dis_sim_{duration} = \frac{1}{n}\sum_{i=1}^{n}\frac{f_d(i)}{MaxDuration}$, and $f_d(i) = \text{MIN}(MaxDuration, |j_i - x_i|)$

(3) $dis_sim_{pitch} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{f_p(i)}{MaxPitch}\right)^2}$, and $f_p(i) = \text{MIN}(MaxPitch, |k_i - y_i|)$

The $dis_sim_{duration}$ and dis_sim_{pitch} stand for the dissimilarity degrees of the beat and pitch, respectively. As in (2), the $dis_sim_{duration}$ is a value between zero and one, where the value of one means the two sequences are the most dissimilar with respect to beat. The system-defined constant *MaxDuration* specifies the maximal value when calculating the difference of note

duration. Also in (3), the dis_sim_{pitch} is a value between zero and one with respect to pitch. The constant *MaxPitch* specifies the maximal value when calculating the difference of note pitch. The $w_{duration}$ and w_{pitch} are the coefficients for the two terms, and the weighted dissimilarity degree, *DIS_SIM* (*Q*, *C*), is within zero and one, where the value of one means the two sequences are the most dissimilar.

Example 3

Given a sequence of music segments (B,3,-3) (A,1,+1) (D,3,-3) (B,1,-2) (C,1,+2) (C,1,+2) (C,1,+1) (A,5,+2) (B,3,-3) (A,1,+1) (D,3,-3) (B,1,-2) (C,1,+4) (A,2,+3) (D,4,-3) (B,5,-2) (C,1,+2) (A,2,+1) (B,5,-1) (C,1,+1) (A,3,+2) (B,3,-3) (A,1,+1) (D,3,-3) (B,1,-2) (C,1,+4) (A,2,+3). The *Code_*(*a~b*) denotes the subsequence ranging from the *a*-th position to the *b*-th position in the sequence of music segments.

Assume the query sequence *Q* is (B,1,-2) (C,1,+5) (A,2,+3), the *MaxDuration* is 3 and the *MaxPitch* is 4, and both weights $w_{duration}$ and w_{pitch} are 0.5. Only the four subsequences, *i.e.*, *Code_*(12~14), *Code_*(16~18), *Code_*(19~21), and *Code_*(25~27) have the same sequences of segment types. For *Code_*(12~14), the dissimilarity *DIS_SIM*(*Q*, *Code_*(12~14)) is 0.072, the $dis_sim_{duration}$ is 0 and the dis_sim_{pitch} is 0.144. For *Code_*(16~18), the *DIS_SIM*(*Q*, *Code_*(16~18)) is 0.425, the $dis_sim_{duration}$ is 0.333 and the dis_sim_{pitch} is 0.520.

2.3.1) An Extension

The coding scheme of music segments preserves the property of transposition invariant and is able to handle the cases of inaccurate pitch and speed of singing when posing queries.

However, for two music object A and B, if all the notes of MA and MB all the same, but the beats of each note in A is directly proportional to the corresponding one in B, the coding of MA and MB will not be the same. Moreover, the dissimilarity measure would be also substantial to distort the ranking of retrieval results. For example, MA = (B, **3**, -3) (A, **1**, +1) (D, **3**, -3) and MB = (B, **6**, -3) (A, **2**, +1) (D, **6**, -3). As considering the scale invariant on duration, MA will be exactly same as MB.

If the scale invariant on duration is one of main concerns, we provide an extension of the coding scheme by music segment to accommodate the situation. Instead of absolute duration of segment, we use the relative duration of segment to represent the beat information of music objects. The *n*-th relative duration of segment is the ratio of the *n*-th segment duration to the (*n*–1)-th segment duration, such that the refined coding scheme has the property of scale invariant on duration. For the same music object in Figure 5, the refined sequence of music segment is shown as in Figure 7.

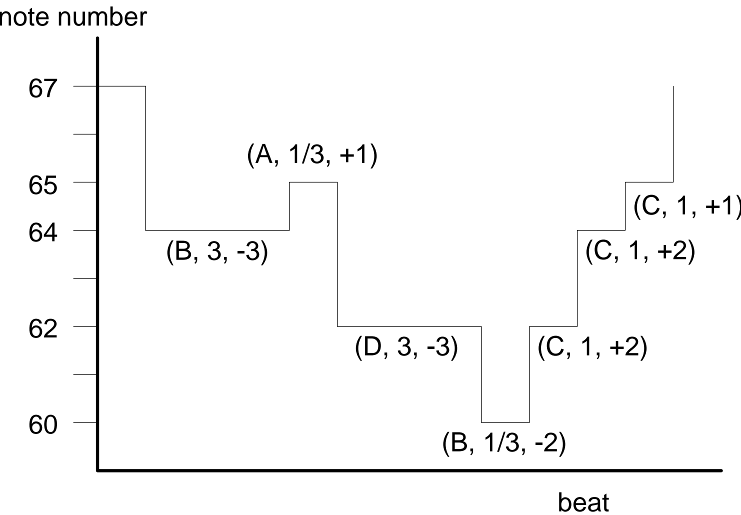


Figure 7: The sequence of music segments of the score in Figure 5 by using the relative duration.

3. INDEXING

In this section, we introduce three indexing methods. For ease of reading, we illustrate ideas through examples.

3.1. Suffix Tree-based Indexing

We first introduce the data structure of suffix tree, and then take the chord strings as an example to use the suffix tree-based indexing.

3.1.1) Suffix Tree

A suffix tree is originally developed for substring matching (Chen and Seiferas, 1984; McCreight, 1976, Gusfield, 1997). For example, Figure 8 shows the suffix tree of the string S="ABCAB". Each leaf node (denoted by a box) corresponds to a substring starting at the position indicated in the node in S, and each link is labeled with a symbol α, where $\alpha \in \Sigma \cup \{\$\}$, Σ is the alphabet of S and '$' is a special symbol denoting end-of-string. As a result, all the suffixes, *i.e.*, "ABCAB", "BCAB", "CAB", "AB", and "B", are organized in the tree.

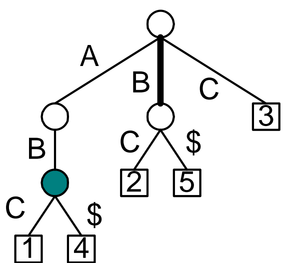


Figure 8: The suffix tree of the string S = "ABCAB".

3.1.2) Indexing Chord Strings

We make use of suffix tree to construct the index of music objects in chord strings. In Figure 9, the leaf node (denoted by a box) in the suffix tree indicates the string id with the associated substring organized in the suffix tree.

Example 4

Given four music objects coded as chord strings, the suffix tree of chord strings is constructed as shown in Figure 9.

S1: ’Am’-’F2’-‘Dm’-‘Am’

S2: ‘C’-‘C’-‘F’-‘C’

S3: ‘G’-‘Em’-‘C’-‘D’

S4: ‘E1’-‘G’-‘Am’-‘Bm’

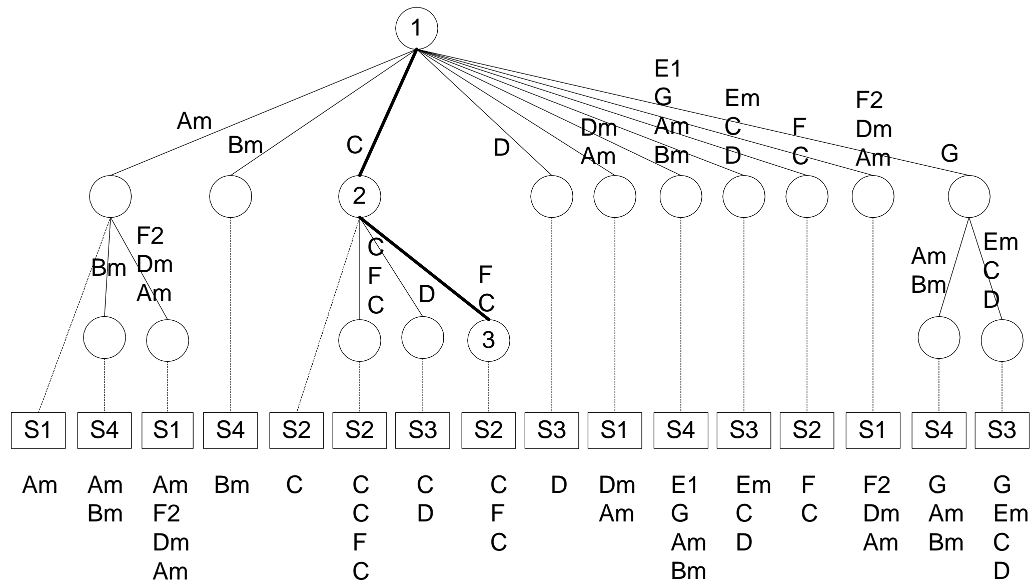


Figure 9: The suffix tree of chord strings in Example 4.

3.2. (N-gram+Tree)-based Indexing

Similarly, we first introduce the *n*-gram indexing, and then take the mubol strings as an example to use the (*n*-gram+tree)-based indexing.

3.2.1) N-gram indexing

The *n*-gram indexing has been widely used in the area of information retrieval (Frakes and Baeza-Yates, 1992; Witten, Moffat, and Bell, 1994). For a given string, each *n*-gram is determined by associating continuously *n*-characters fragment with the position where the *n*-characters fragment is located. Given a query string of length l, where $l \geq n$, the *n*-gram method decomposes the query string into $(l-n+1)$ *n*-grams which are treated as subqueries. The

results to the (l–n+1) n-grams are collected for further checking to avoid the situation in which all the (l–n+1) n-grams match, but not in right order.

3.2.2) Indexing Mubol Strings

The n-grams organized as a tree, we call it (n-gram+tree)-based index structure, are used to index mubol strings of music objects in the paper (Chen and Chen, 1998). The index structure, called L-tree, is a tree with two kinds of links, *i.e.*, solid and dotted lines. The internal nodes are connected with solid lines, while the leaf nodes with the associated positions are indicated by dotted lines.

In priori to constructing the L-tree index, the system-defined parameter h, which is the height of L-tree, has to be determined. The height of L-tree is a trade-off between the elapsed time of query processing and the index size. It is suggested to be the reasonable length of query strings.

Example 5

The music object of eight measures is coded in the mubol string R1, as shown in Figure 10. The n-grams of R1, where n = 1, 2, …, h (h is set to 3), with the associated positions are also listed in the table of Figure 10. All the prefixes of an n-gram can be found in the (n–1)-grams, (n–2)-grams, …, and 1-grams. With the table, it is able to construct the L-tree in a level-wise manner. As in Figure 11, the nodes in Level 1 of the tree indicate the first mubols of 1-grams. The nodes in Level 2 indicate the second mubols of 2-grams, and so on.

n-grams of mubol string, n=1,2,3	position
	R1:1,4,7
	R1: 2,5
	R1: 3,6
	R1: 8
	R1: 1,4
	R1: 7
	R1: 2,5
	R1: 3,6
	R1: 1,4
	R1: 2,5
	R1: 3
	R1: 6

Figure 10: The mubol string R1 and its *n*-grams associated with the corresponding positions.

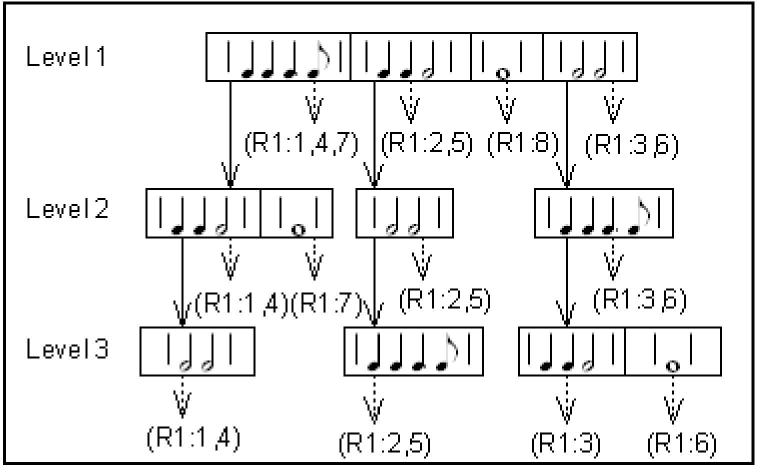


Figure 11: The L-tree of the mubol string R1.

3.3. *Augmented Suffix Tree-based Indexing*

The previous index structures are suitable for strings, such as chord strings and mubol strings, but not for sequences of pairs or triplets, such as sequences of music segments. For handling such kind of cases, the augmented suffix tree is proposed in the paper (Chen, *et. al.*, 2000). In the following, we take music segments as an example to present the one-dimensional and two-dimensional augmented suffix tree index structures.

3.3.1) The One-dimensional Augmented Suffix Tree

A 1-D augmented suffix tree is a suffix tree with the segment duration information being added to the edge labels. First, a suffix tree based on the sequences of the segment types is constructed. Each edge of the suffix tree refers to a symbol appearing in one or more positions in the sequence. For example, let the sequence of music segments be (A,2,+1) (B,5,-1) (C,1,+1) (A,3,+1) (B,3,-2). Using only the segment types, the suffix tree can be constructed as shown in Figure 8. The bold-faced edge in Figure 8 refers to the 'B' in the second and fifth position. Since the corresponding segment durations are 5 and 3, we attach the range of segment duration <*min*, *max*> = <3, 5> to the edge label. This range can be used to filter out some results which cannot be answers during query processing. Figure 12 shows an example of a 1-D augmented suffix tree.

To exploit the filtering effect, the range <*min, max*> should be as compact as possible. For a given population of segment durations, such as {1, 2, 2, 3, 7, 8, 8}, two ranges <1, 3> and <7, 8> are better than one range <1, 8>. Thus, the edge should be split into two edges labeled with <1, 3> and <7, 8>, respectively. Better ranges can be found by applying a clustering algorithm for a given population of segment durations. This method is called *dynamic splitting*. Some well-developed clustering algorithms are available (Jain and Dubes, 1988). We will not address the details of the clustering algorithms in this article. In some cases, however, if it is hard to find compact ranges from a given population, we may apply *static splitting* method by splitting a range into some predefined smaller ranges which can be obtained from the statistics of data set.

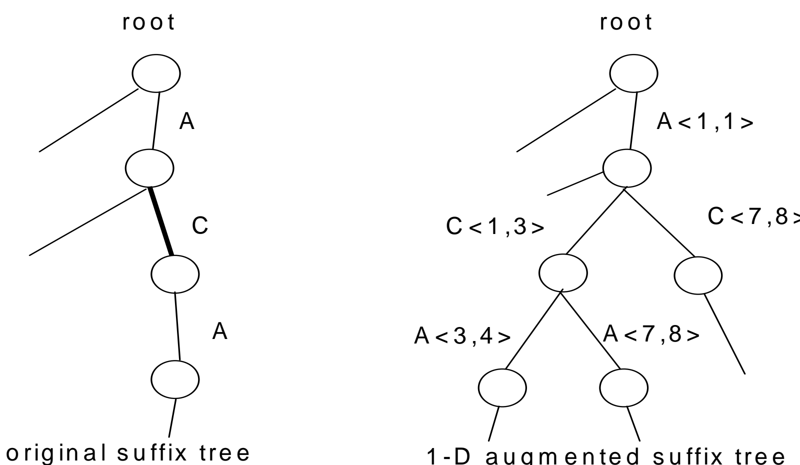


Figure 12: An example of the 1-D augmented suffix tree.

The algorithms of 1-D augmented suffix tree construction are described as follows. Three steps are included in **`Algorithm C1`**. Given a set of music objects in MIDI format, the first step is to code these music objects as the sequences of music segments. Based on the segment

types in the sequences of music segments, we construct a suffix tree in the second step. In the third step, for each edge of the suffix tree, we collect the corresponding music segments as `MuSeg`. The `MuSeg` will be evaluated by the **Algorithm C2** to decide whether to split the edge or not. In the **Algorithm C2**, we apply a clustering algorithm on the set of one-dimensional points which are the segment durations in `MuSeg`. If two or more clusters are determined, the edge will be split accordingly. Finally, we attach the range information to the edge or all the split edges. For example, as in Figure 12, when processing the bold-faced edge C of the original suffix tree, the set of segment durations in `MuSeg` is {1, 2, 2, 3, 7, 8, 8}. Two clusters are found by the clustering algorithm, and the corresponding range, <1, 3> and <7, 8>, will be attached to the splitted edges, respectively.

```
Algorithm C1 Construct_1-D_Augmented_Suffix_Tree(S, T)
//  parameters:
//      S, the set of songs in MIDI format
//      T, the 1-D augmented suffix tree to be constructed
Begin
//  Step 1: coding
//  for each song s in S,
//      parse the midi file to derive its melody and rhythm information,
//      and then code s by the music segments
CodedS = Coding(S);

//  Step 2: construct an original suffix tree
T = Construct_Original_Suffix_Tree(CodedS);

//  Step 3: edge-split
for each edge e in T
    // retrieve the corresponding music segments as the set MuSeg
    DescS = retrieve the songs descending from the edge E;
    MuSeg = sample the music segments, which are specified by the
                edge E, of all the songs in DescS;
    // evaluate the set MuSeg to decide whether split the edge or not
    if (Evaluate_1D(MuSeg) == true)
        split the edge e with the range being added to the edge labels;
    else
        add the range, <min, max>, to the edge label;
End.
```

```
Algorithm C2 Evaluate_1D(MuSeg)
//  parameters:
//      MuSeg, a set of music segments
//  return:  true or false
Begin
    // PS = {d1, d2, …}, a set of one-dimensional points
    PS = retrieve all the values of the segment duration in MuSeg;
    // apply a clustering algorithm on PS to find out clusters if any.
    no_clusters = clutering_algm(PS);
    if (no_clusters ≥ 2)
        return true;
```

```
    else
        return false;
End.
```

3.3.2) The Two-dimensional Augmented Suffix Tree

The 2-D augmented suffix tree is an extension of the 1-D augmented suffix tree by attaching both the segment duration and the segment pitch information to the edge label. The construction of the 2-D augmented suffix tree are similar to the 1-D augmented suffix tree.

4. QUERY PROCESSING

With proper indexing structures, the query processing methods, especially for the partial matching and approximate matching, are described in this section.

4.1. Exact Matching

With all three tree-based index structures, the functionality of exact partial matching can be easily supported. We go through some examples to show the query processing of exact matching.

Example 6

Given a suffix tree as shown in Figure 8, suppose that the query string is “AB”. We follow the leftmost path to the black node, and all the leaf nodes rooted from the black node are the results, *i.e.*, the substrings “ABCAB” and “AB”.

Example 7

Given the same music objects as in Example 4, suppose that the query has been transformed into the chord string ‘C’-‘F’ by the chord decision algorithm. When processing the first chord ‘C’, from the root node, the branch associated with the label ‘C’ will be followed and we reach node 2. For the next chord ‘F’, we select the right branch of node 2 with the label ‘F’-‘C’, and stop at node 3. All leaf nodes descending from node 3 are the answers to the query, *i.e.*, S2 in this example.

Example 8

Given the L-tree, shown in Figure 11, suppose that the query string is |♩ ♩ 𝅗𝅥|𝅗𝅥 𝅗𝅥| for exact searching. The query string will be processed by traversing the L-tree in level-wise manner. When processing the first mubol |♩ ♩ 𝅗𝅥| against Level 1 of the L-tree, node A is matched and its children will be reached for processing the next mubol. When processing the

second mubol |𝅗𝅥 𝅗𝅥| against Level 2, those children (only node B in this example) will be compared to the second mubol. Since node B is matched to the second mubol, the two children of node B will be reached for further processing. Moreover, all the mubols of the query have been processed, and the node (R1:2,5) will be the answer.

The L-tree is a (n-gram+tree)-based index structure. In the approach of n-gram indexing and query processing, if the length of the query string is larger than n, the cases of false match might happen. For the L-tree of tree height h, if the query length is larger than h, the query will be divided into subqueries and the intermediate answers with respective to each subqueries will be merged and confirmed by the join processing. Different ways of query division will lead to different performance. A simple model to estimate the cost of each division is presented as follows, which the cost is defined as the total number of intermediate answers to be joined.

Assume each non-leaf node in the L-tree has b children and the length of the list at each leaf node is f. Therefore, the length of the list at the leaf node in Level i is $f \times b^{(h-i)}$ where h is the tree height of L-tree.

For example, assume $h = 4$, $b = 3$, and $f = 1$. If the length of query is 5, the query can be either divided into two subqueries of lengths 4 an 1 or the ones of length 3 and 2. In the former case, the total length of lists, *i.e.*, the number of intermediate answers to be joined, is $28\times(3^0+3^3)$. In the latter case, it is $12\times(3^1+3^2)$. The estimated cost of the latter case is smaller than the former case.

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

4.2. Template-based Matching

Given a query in mubol string *QR*, to find all the music objects containing the mubol strings which are m-similar to *QR*, where $0 \le m \le K$, is called a K-similar searching.

The template-based matching is designed for the K-similar searching. The basic idea of template-based matching is to construct a *template* containing the variants of the query within K operators. The system first generates a template of the query by referencing all the possible mubol operators in the L-tree. The nodes in Level i of the template indicate the i-th mubol of the query and all the mubols by possible operators. Therefore, all the candidates can be represented in the template. For each candidate, we apply the method of exact searching to confirm if the candidate is matched.

Example 9

Given the L-tree, shown in Figure 11, suppose that the query string is for 1-similar searching. According to the query string, we first generate the template, shown in Figure 13(a). In the template, the query string is represented by the nodes in three levels with solid lines. The paths formed by dotted lines indicate the candidate mubol strings which are 1-similar to the query. Accordingly, the L-tree will be traversed by the method of exact searching for each candidate. In this example, two candidates are matched, shown in Figure 13(b). The answers are (R1:3) and (R1:6).

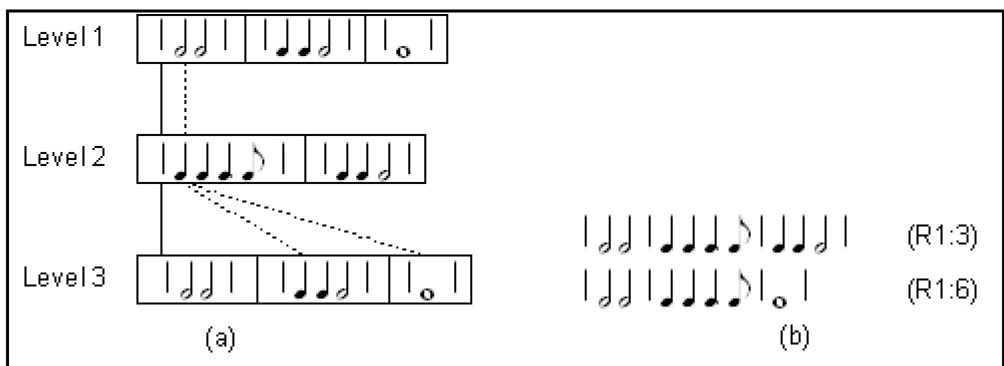


Figure 13: The template for 1-simliar searching and query results in Example 7.

4.3. Thresholding-based Matching

For the case of approximate queries in which the approximation is specified by means of thresholds, it is called thresholding-based matching.

We take the music segments as an example. When posing an approximate query, the user specifies two thresholds for segment duration and segment pitch, denoted $\delta_{duration}$ and δ_{pitch}, respectively. The sequences whose $dis_sim_{duration}$ and dis_sim_{pitch} are less than $\delta_{duration}$ and δ_{pitch}, respectively, will be the answers, provided an exact match for the sequence of segment types exists. All the results will be sorted by the dissimilarity, *DIS_SIM*.

Example 10

As in Example 3, assume the query sequence *Q* is (B,1,-2) (C,1,+5) (A,2,+3), the $\delta_{duration}$ and δ_{pitch} are 0.2 and 0.3, respectively, the *MaxDuration* is 3 and the *MaxPitch* is 4, and both weights $w_{duration}$ and w_{pitch} are 0.5. By comparing the sequences of the segment types, *Code*_(12~14), *Code*_(16~18), *Code*_(19~21), and *Code*_(25~27) are the candidates. For the candidate *Code*_(12~14), the dissimilarity *DIS_SIM*(*Q*, *Code*_(12~14)) is 0.072, the $dis_sim_{duration}$ is 0 and the dis_sim_{pitch} is 0.144. Since the $dis_sim_{duration}$ and the dis_sim_{pitch} are both less than the thresholds, *Code*_(12~14) is one of the answers. For the candidate

*Code*_(16~18), the *DIS_SIM*(*Q*, *Code*_(16~18)) is 0.425 , the $dis_sim_{duration}$ is 0.333 and the dis_sim_{pitch} is 0.520. Since the $dis_sim_{duration}$ is greater than $\delta_{duration}$ or the dis_sim_{pitch} is greater than δ_{pitch}, *Code*_(16~18) is not an answer. Further checking shows that *Code*_(19~21) is not an answer while *Code*_(25~27) is.

In the following, we develop query processing methods for thresholding-based matching in augmented suffix trees.

4.3.1) Query Processing for the One-dimensional Augmented Suffix Tree

Given a query in the form of the sequence of music segments and a threshold $\delta_{duration}$, we traverse the 1-D augmented suffix tree to find the answers. Based on the query specification in which n is the number of query segments and *MaxDuration* is the predefined constant, we first define the threshold $\Delta_{duration}$ as follows.

$$\Delta_{duration} = MaxDuration \times n \times \delta_{duration}$$

The threshold $\Delta_{duration}$, which denotes the maximal accumulated difference of note duration, is used for internal processes when traversing the 1-D augmented suffix tree. When processing the first music segment, it is compared with the edges descending from the root node. Only the edge whose label matches the segment type of the first music segment and its range satisfies the following conditions (*i.e.*, Case 2 and Case 3) will be followed for processing the second music segment. Specifically, when processing the *i*-th music segment of the query, we compare the query range $<d_i-\Delta_{duration}', d_i+\Delta_{duration}'>$ with the range of the corresponding edge label $<min, max>$, where d_i denotes the segment duration of the *i*-th music segment in the query, and $\Delta_{duration}'$ denotes a reduced threshold to be detailed in the following. There are three cases to consider, as shown in Figure 14:

Case 1: $(max < d_i-\Delta_{duration}')$ or $(d_i+\Delta_{duration}' < min)$

The range $<d_i-\Delta_{duration}', d_i+\Delta_{duration}'>$ and the range $<min, max>$ have no intersection. The edge need not be followed.

Case 2: $(d_i-\Delta_{duration}' < max < d_i)$ or $(d_i < min < d_i+\Delta_{duration}')$

The range $<d_i-\Delta_{duration}', d_i+\Delta_{duration}'>$ intersects with the range $<min, max>$. The edge has to be followed. However, the threshold $\Delta_{duration}'$ can be reduced by the *threshold propagation function* in which $\Delta_{duration}''$ represents a further reduced threshold. If $\Delta_{duration}''$ is less than zero, the searching process will terminate immediately.

DEFINITION 4: Threshold Propagation Function for the One-dimensional Augmented Suffix Tree

$$\Delta_{duration}'' = \begin{cases} \Delta_{duration}' - \mathrm{MIN}(MaxDuration, (d_i - max)) & \text{if } d_i - \Delta_{duration}' < max < d_i \\ \Delta_{duration}' - \mathrm{MIN}(MaxDuration, (min - d_i)) & \text{if } d_i < min < d_i + \Delta_{durationt}' \end{cases}$$

Case 3: $min < d_i < max$

When d_i is covered by the range <*min*, *max*>, the edge has to be followed for processing the next music segment without any reduction of the threshold.

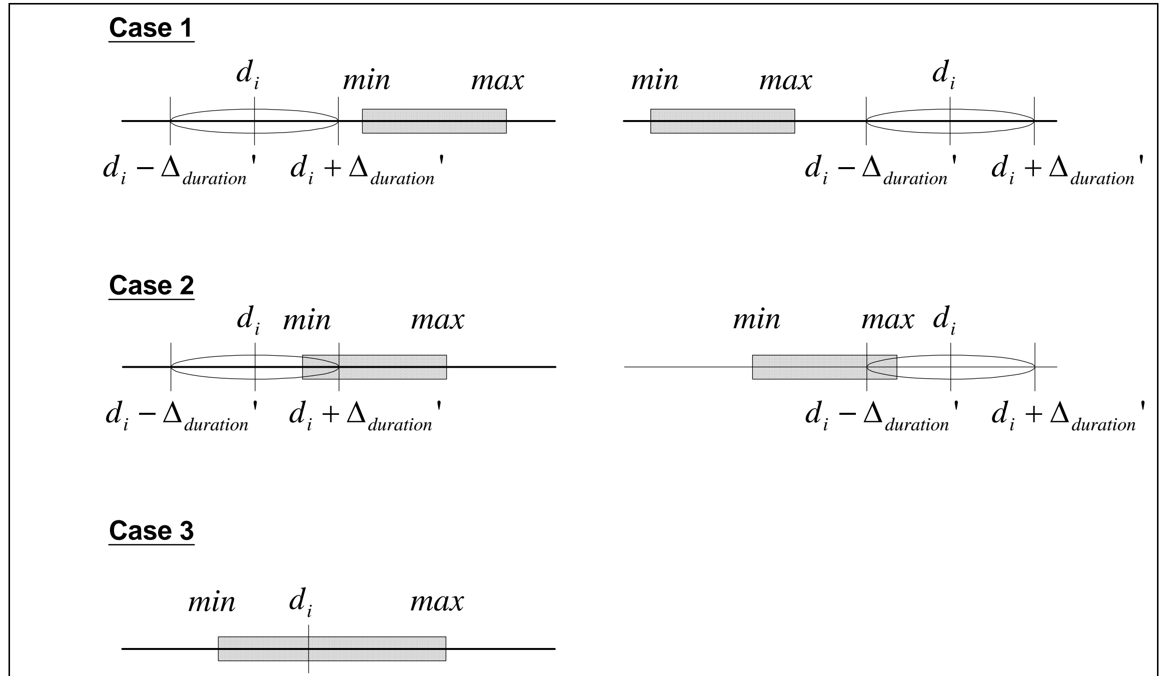


Figure 14: The illustration of three cases when processing a music segment.

Example 11

Based on the 1-D augmented suffix tree in Figure 12, we give an example to illustrate the traversal process. Given the threshold $\delta_{duration}$ = 0.22, and the query (A,2,–) (C,5,–) (A,4,–). When posing and processing 1-D queries, the segment pitch (*i.e.*, the third element of the triplet in the query sequence of music segments) is not needed and denoted by '–'.

First, the threshold $\Delta_{duration}$ is calculated as 3 × 0.22 × 3 = 1.98. When processing the first music segment (A,2,–), Case 2 holds. We apply the threshold propagation function and the threshold is reduced to 0.98. When processing the second music segment (C,5,–), there are two edges to consider. However, in both situations, Case 1 holds. The searching process stops and there is no answer for the query.

The algorithms of query processing for the 1-D augmented suffix tree are described in **Algorithm Q1** and **Algorithm Q2**. There are two procedures in **Algorithm Q1**. The procedure `ProcNode` is to filter out impossible songs and derive a candidate set, CR. For each candidate in the set CR, we make a further check if it satisfied the given thresholds. All the answers will be ranked by the dissimilarity *DIS_SIM*, in the procedure `Check_and_Sort`.

The **Algorithm Q2** is a recursive procedure. The first part is the boundary condition of recursion. Once the reduced threshold is less than zero or there are no more edges of the tree to be matched against the query, the procedure `ProcNode` returns empty set. If all the music segments of the query have been matched, all the descending leafs will be returned as the candidate results. Followed by the boundary condition, the edges of the node N will be retrieved for comparison. For each edge, we consider three cases as described above. If Case 2 or Case 3 hold, we invoke a recursive call of the procedure `ProcNode` to process the next music segment of the query.

```
Algorithm Q1 Query_Process_1D(Q, th, T, R)
// parameters:
//     Q, the user query Q = (t1, d1)(t2, d2) … (tn, dn)
//         where t is segment type and d is segment duration
//     th, the threshold of segment duration
//     T, the 1-D augmented suffix tree
//     R, the result set of songs which satisfied Q and th
Begin
    CR = ∅;    // CR, the candidate result set
    th = n * MaxDuration * th;
    CR = ProcNode(root of T, 1, th)
    R = Check_and_Sort(CR);
    return R;
End
```

```
Algorithm Q2 ProcNode(node N, i, th)
// parameters:
//     N, the current node when traversing the tree T
//     i, the i-th music segment to be matched when calling the ProcNode
//     th, the reduced threshold

Begin
    // the boundary condition of recursion
    if ((th < 0) or (node == NULL))
        return ∅;
    else if (m == n)
        TR = retrieve all leafs descending from the node N;
        return TR;

    // body
    E = retrieve all the edges of the node N;
    TR = ∅;
```

```
    for each edge e = (e_type, e_duration) of E do
        if (e_type ≠ ti)
            skip the edge e;
        else
            // e_duration = (e_min, e_max)
            // case 1:
            if ((e_max < (di-th)) or ((di+th) < e_min))
                skip the edge e;
            // case 2:
            else if (((di-th) < e_max < di) or (di < e_min < (di+th)))
                a_th = apply_1-D_threshold_propagation(th);
                TR = TR + ProcNode(child node of edge e, (m+1), a_th);
            // case 3:
            else
                TR = TR + ProcNode(child node of edge e, (m+1), th)
    return TR;
End.
```

4.3.2) Query Processing for the Two-dimensional Augmented Suffix Tree

The 2-D augmented suffix tree is an extension of the 1-D augmented suffix tree. The query processing for the 2-D augmented suffix tree is similar to the one for the 1-D augmented suffix tree, but considering both information of duration and pitch attached to the edges. The threshold propagate function has to be revised for considering both thresholds. Define the threshold Δ_{pitch} as the maximal accumulated difference of note pitch.

$$\Delta_{pitch} = MaxPitch^2 \times n \times \delta_{pitch}^{\ 2}$$

Let d_i and p_i denote the segment duration and segment pitch of the i-th music segment in the query, respectively, and $<B_{min}, B_{max}>$ and $<N_{min}, N_{max}>$ be the ranges of the corresponding edge for the segment duration and segment pitch, respectively. The threshold propagation function for the 2-D augmented suffix tree is defined as follows.

DEFINITION 5: Threshold Propagation Function for the Two-dimensional Augmented Suffix Tree

$$\Delta_{duration}'' = \begin{cases} \Delta_{duration}' - \mathrm{MIN}\left(MaxPitch, \left(d_i - B_{max}\right)\right) & \text{if } d_i - \Delta_{duration}' < B_{max} < d_i \\ \Delta_{duration}' - \mathrm{MIN}\left(MaxPitch, \left(B_{min} - d_i\right)\right) & \text{if } d_i < B_{min} < d_i + \Delta_{duration}' \end{cases}$$

$$\Delta_{pitch}'' = \begin{cases} \Delta_{pitch}' - \left(\mathrm{MIN}\left(MaxPitch, \left(p_i - N_{max}\right)\right)\right)^2 & \text{if } p_i - \Delta_{pitch}' < N_{max} < p_i \\ \Delta_{pitch}' - \left(\mathrm{MIN}\left(MaxPitch, \left(N_{min} - p_i\right)\right)\right)^2 & \text{if } p_i < N_{min} < p_i + \Delta_{pitch}' \end{cases}$$

5. DISCUSSION

We have discussed the issues and methods for the representation and indexing and query processing. In this section, we provide a qualitative comparison of various methods to show the intrinsic difficulties of the problem of content-based music information retrieval.

5.1. Comparison of Representation Methods

The comparison of three representation methods described in Section 2 is summarized in the table of Figure 15.

Representation	Melody	Rhythm	Transposition invariant	Scale invariant	Approxi-mation	Similarity measure	Restriction of measure boundary
Chord	✓			✓	✓		✓
Mubol		✓	✓		✓	# of mubol operators	✓
Music Segment	✓	✓[1]	✓	✓[2]	✓	Distance function, DIS_SIM	

Note:
[1] only considering the duration information.
[2] when applying the extension of music segment in Section 2.3.1.

Figure 15: The comparison of representation methods.

First of all, the chord decision algorithm can be applied, provided that the measure boundaries of music objects are determined first. However, in some cases of coding music objects, especially user queries, such kind of restriction makes the representation of chord unfeasible. Also, the restriction of measure boundary is applied to the representation of rhythm when transforming music objects into mubol strings.

Some questions remain unsolved and deserve further study. It is believed that there is no a universal representation method suitable for any kinds of applications. For the three representation models, the target domain and applications, such as music genre of the collections, have not been identified. Also, it is not convinced whether the semantics captured by each representation method coincides with the user perception. For example, in the chord representation model, it is not been demonstrated that similar music objects from user point of view will be coded as the same chord strings. Similarly, the issue remains in the chord and music segment representation model.

In the rhythm representation model, however, the mapping processing from measures to mubols is not clear enough. The set of mubols should be defined explicitly, as well as the mapping processing. For example, suppose two measures are almost the same, but in little difference. The two measures map into either one mubol or two distinct mubols. The former case introduces the issues of similarity measurements, while the latter case results in a large

mubol set (too many kinds of mubols). Between the two cases, there is a trade-off to be explored.

In the music segment representation model, if the consecutive notes are the same in pitch, the adjacent notes with same pitch will be combined and coded as one music segment. It might be distinguishable for users that the case of four consecutive one-beat notes in same pitch is different from the case of only one note in four beats. However, the coded segments for the two cases are the same. Besides, the way of dealing with *rests* is not discussed. If considering *grace notes* and unstable pitch from user queries by singing, the smoothing techniques can be involved when coding music contours into music segments.

5.2. Comparison of Indexing and Query Processing

The comparison of indexing and query processing methods described in Section 3 and Section 4 is summarized in the table of Figure 16.

Indexing	Exact matching	Partial matching	Approximate matching	Space	Efficiency (filtering effect)
Suffix tree-based	✓	✓		2	As only the exact matching is required
(*n*-gram+tree)-based	✓	✓	template-based	1	When query length ≤ tree height
Augmented suffix tree-based	✓	✓	thresholding-based	3	1. When handling sequences of pairs or triplets 2. In the cases of specifying thresholds

Figure 16: The comparison of indexing and query processing methods.

The exact matching and partial matching are supported for all three methods by using suffix tree and *n*-gram techniques. Considering the approximate matching, the template-based and the thresholding-based query processing methods are applied to achieve the approximation with the (*n*-gram+tree)-based and the augmented suffix tree-based indexing, respectively. Compared to template-based matching, the way of handling approximation by the thresholding-based matching is more accurate. In the thresholding-based matching, the approximation degree can be described by the one or two user-specified thresholds rather than number of operators.

Considering the space consumption, the rank of storage required in the three index structures is L-tree, suffix tree, augmented suffix tree, in an increasing order. Suffix trees, including augmented suffix trees, suffer from the space consumption. Moreover, the height of suffix tree is the maximal length of all feature strings, such as chord string and mubol string, in databases. On the contrary, the height of L-tree is predefined, not related to the data set. Meanwhile, typical length queries would not be as long as the length of feature strings in

databases. Therefore, it is not needed to spend such space to maintain full suffix tree just for queries of unreasonably long length. Such that, in real application, the tree height of suffix tree might be set to a certain value, which will be discussed in Section 6.

Regarding efficiency of indexing and query processing, although the complexity analysis doe not include in the table, the efficiency in terms of filtering effect is discussed as follows.

When the query length is less than the tree height of L-tree, the (*n*-gram+tree)-based indexing is suggested. As user queries are longer than the tree height, the queries have to be divided into subqueries of shorter lengths which are determined by the cost model. However, the cost model seems too rough such that it can not handle the unbalanced L-tree from real data set.

When dealing with queries of general length in exact matching, the suffix tree-based indexing is suggested. If considering the data as sequences of pairs or triplet and/or approximate matching, the augmented suffix tree-based indexing is suggested rather than the suffix tree-based indexing.

For example, with the coding scheme by music segments, the music objects are transformed into sequences of music segments. Intuitive, we can construct a suffix tree-based index structure for the sequences of the segment types of the songs in the database. For a query Q and the thresholds $\delta_{duration}$ and δ_{pitch}, we traverse the suffix tree according to the sequence of the segment types of Q and collect the leaf nodes as the candidates. For each candidate, if the $dis_sim_{duration}$ and dis_sim_{pitch} satisfy the thresholds, the candidate will be considered as an answer and the *DIS_SIM* is then computed for ranking all the answers. Since the suffix tree is constructed based on the sequences of segment types, there might still be lots of candidates. Moreover, the computation of $dis_sim_{duration}$ and dis_sim_{pitch} can only be done when traversing down to the leaf nodes. On the contrary, the augmented suffix tree-based indexing makes use of pitch and duration information attached to edges for applying threshold propagation function to prune out the impossible music objects and reduce the search space. As a result, it leads a better filtering effect.

5.3. Intrinsic Difficulties

With having various techniques of coding schemes, indexing, and query processing, it is crucial but not easy to formulate the conditions, even the strategies and heuristics, of choosing proper methods to build a content-based music information retrieval system.

The overall and comparative study with real-scale experiments among these techniques is currently in process. Without the support by experimental results, some critical statements,

such as the effectiveness of indexing and the discriminability of representation, could not be easily justified.

Although the intrinsic difficulties of content-based music information retrieval, we have presented three approaches based on the representations of chord, mubol, and music segment (denoted by APC, APM, APS) in the paper (Chou, Chen, and Liu, 1996; Chen and Chen, 1998; Chen, *et. al.*, 2000), respectively. The approaches are summarized in the table of Figure 17.

Approach	Techniques employed	Coding scheme	Similarity functions / operators	Index and query processing
APC	Chord + Suffix tree + Exact	A		P
APM	Mubol + L-tree + Exact, Template-based		A	P+A
APS	Segment + AST + Exact, Thresholding-based	A+T+S	A	P+A

Note:
P: partial matching
A: approximation
T: transposition invariant
S: scale invariant

Figure 17: The comparison of three approaches on ways of handling PATS.

Concerning the applicability of three approaches, APS outperforms APC and APM, because of not restricted to determining measure boundary. Also in the table of Figure 17, all the four properties, including P, A, T, and S, can be preserved when apply APS approach.

Compared to the space requirement of the chord suffix tree, the space requirement of L-tree will not explode, because of predefined tree height. For the benefit of space consumption, APM has the advantage of fixed tree height, rather than APC and APS.

Although, only part of efficiency study in elapsed time of query are concluded in the preliminary experiment results. We still can share some observations when comparing the three approaches. By adding extra information to the edges in augmented suffix trees, the fan-outs of node would be APS, APM, and APC, in descending order. In general, higher fan-out of each node in balanced trees suggests better filtering effects. Note that too many fan-outs also introduce more computation.

The partial matching, one of most essential functionalities, is provided for all three methods by using suffix tree and *n*-gram techniques. As for the approximate matching, different techniques are applied to achieve the approximation in the three approaches. In APC, the approximation can be just done by chord decision algorithm. Only exact matching is provided by the index structure and its query processing algorithms. In APM, the approximation is measured by the number of applying mubol operators. The dedicated query processing algorithm is developed for approximation matching. User can specify the approximation degree by the parameter K. Compared to APC, the way of handling

approximation by APM is more flexible. In APS, the approximation is done by three ways. First, the coding scheme by music segment provides an approximation. We also devise a similarity function to measure the difference between two segment strings. Similar to query processing algorithms of APM, the approximation degree can be described by the user-specified thresholds. Moreover, the functionality of scale invariant is provided by the refined coding scheme, detailed in Section 2.3.1.

Among the three approaches, the comparative study of effectiveness, by means of precision and recall, is not easily concluded without the experimental support. We cannot formulate the discriminability between the representation of melody (the chord in APC, the music segment in APS) and rhythm (mubol in APM). However, it is clear that the discriminability of coding by music segment is better than the one by chord. Note that too much discriminability does not always suggest better retrieval effectiveness, especially for non-expert users.

6. EXPERIMENTS: THE PRELIMINARY RESULTS

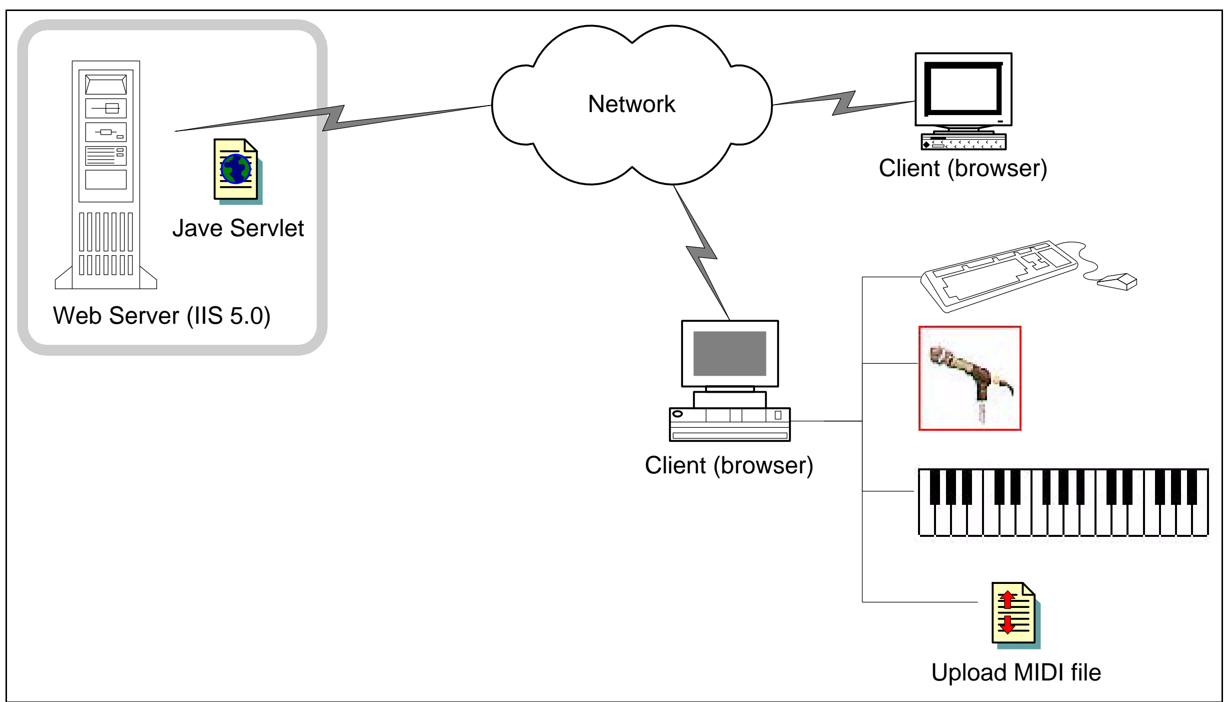
We initiate the Ultima project to construct a platform with the goal to make a comprehensive and comparative assessment of various MIR approaches. Therefore, a quantitative study for performance is achieved. In the first subsection, we illustrate the platform. In the following, the efficiency issues of approaches are demonstrated, as well as the experiment results to support the comparison study described in the previous section. The Ulrtima project and performance study are detailed in the report (Hsu and Chen, 2001)

6.1. Implementation

The system is implemented as a web server, which is running on the machine of Intel Pentium III/800 with 1GB RAM, on MS Windows 2000 by JDK 1.3. In addition to provide the realistic performance of our approaches, the system is designed to serve as a real-scale and unified platform. Therefore, under the same environment with the real data set, we will have a comparative results to support the quantitative study in Section 5, and even to provide guidelines of choosing appropriate representation schemes, indexing structures, and query processing methods when building a MIR system.

In designing the platform, the issues of architecture, interface, efficiency and effectiveness study, relevance judgment, generation of query set, and measurement are considered. The system architecture, shown in Figure 18, is described as following. For posing queries at the client end, we provide the ways of humming songs, playing the piano keyword, uploading MIDI files, and the set of computer keyboard and mouse.

The server end consists of a mediator, four modules, and a data store, as shown in Figure 1. The mediator receives user queries and coordinates with other modules. The music objects and the corresponding information, such as title, composer, and genre, are organized as standard MIDI files and relational tables, respectively. The summarization module aims to resemble and visualize query results. The query generation module aims to generate parameterized user queries for performance evaluation, as discussed in Section 2.3. The implementations of the two modules are not finished yet. The report module aims to monitor and assess the performance of the system, such as the elapsed time of query processing, space of indices, and precision and recall of the retrieved results. The query processing module aims to resolve queries from the client end or the query generation module. The query processing module is designed as a "container" to which each query processing methods can be "plugged-in". Whenever a new method is proposed, it can be easily plugged into the module for performing experiments under the same environment. Both APC and APS are suffix tree-like approaches. Moreover, the suffix tree-based approach can be considered as a special case of the augmented suffix tree-based approach. In the current stage, we have implemented three approaches, *i.e.*, 1D_List (Liu, Hsu, and Chen, 1999a), APS, and APM.



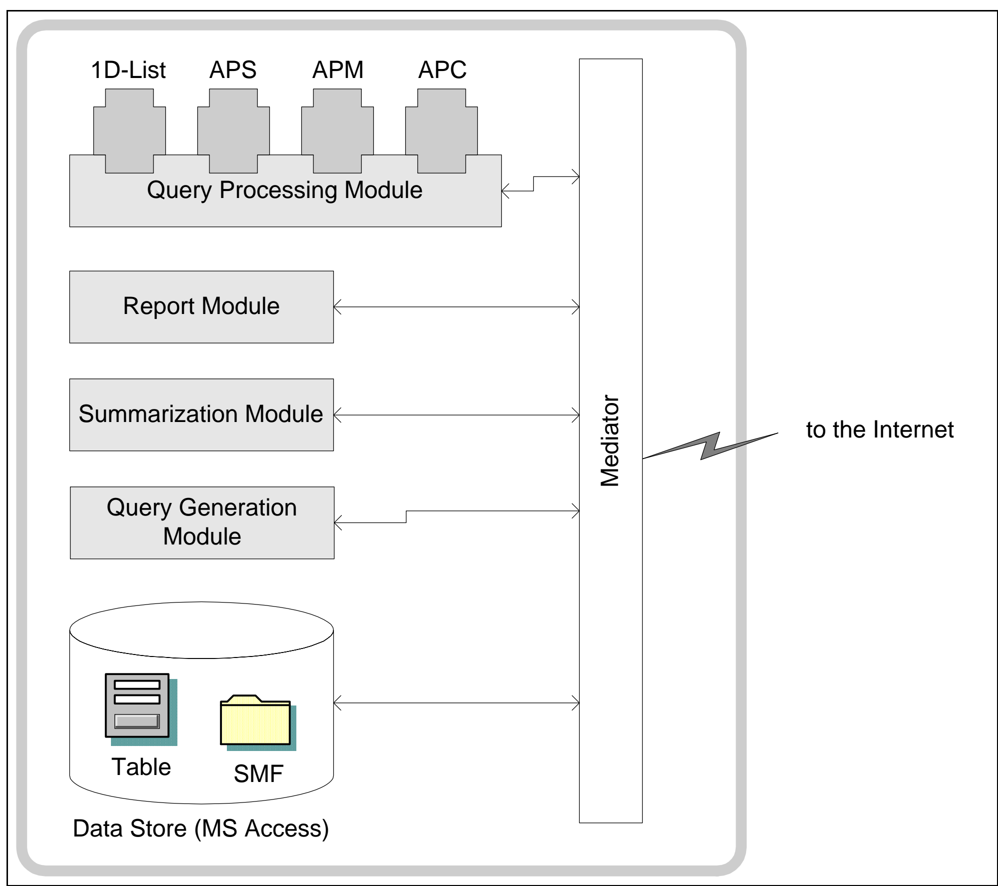


Figure 18: The system architecture: (a) the client-sever system, (b) function blocks of the server.

6.2. *Efficiency Study*

To show the efficiency of the approaches, a series of experiments are performed. We describe the testing data, and then illustrate some experiment results regarding performance issues, *i.e.*, index construction and exact query processing, of APM and APS.

The testing data of music objects, from CWEB Technology, Inc., is a collection of 3500 single track and monophonic MIDI files. Most of them are pop music of Chinese and English songs in various genre.

The average object size is 328.05 notes. When coding these objects in the mubol and music segment representation, the average object size is 78.34 (mubol) and 272 (segment), respectively. The *note count* is defined as the number of distinct notes appearing in a music object. According to the MIDI Standard, the alphabet size is 128. Therefore, the note count of every melody string is between 1 and 128. According to the experiments, the average note count of the real music objects is 13.46. The distributions of the object size and note count of the testing data are shown in Figure 19 and Figure 20.

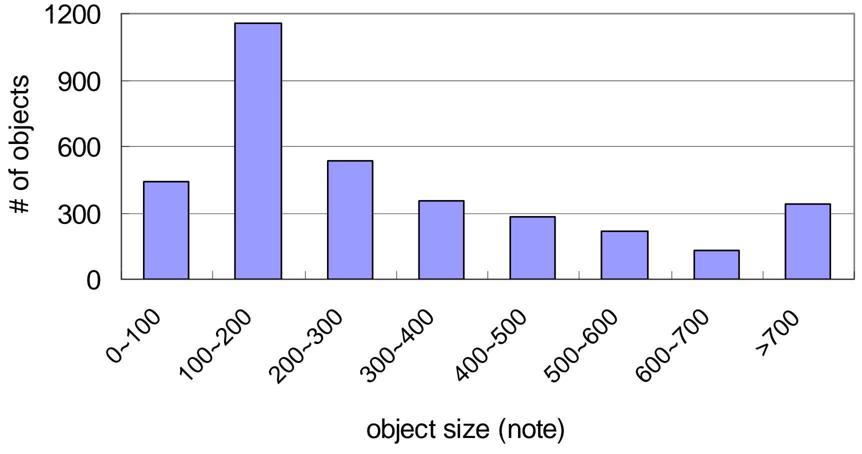


Figure 19: The distribution of object sizes of CWEB data set.

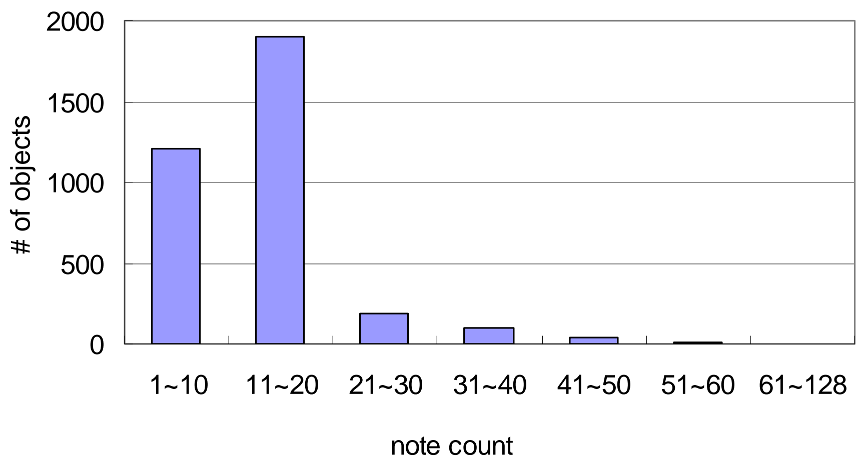


Figure 20: The distribution of note counts of CWEB data set.

The elapsed time and space requirement of constructing index of APR and APS are illustrated as follows.

For APM, the tree height of L-tree is set to 6 in our experiments. As shown in Figure 21 and Figure 22, the L-tree scales well as increasing the number of music objects by means of elapsed time and memory usage.

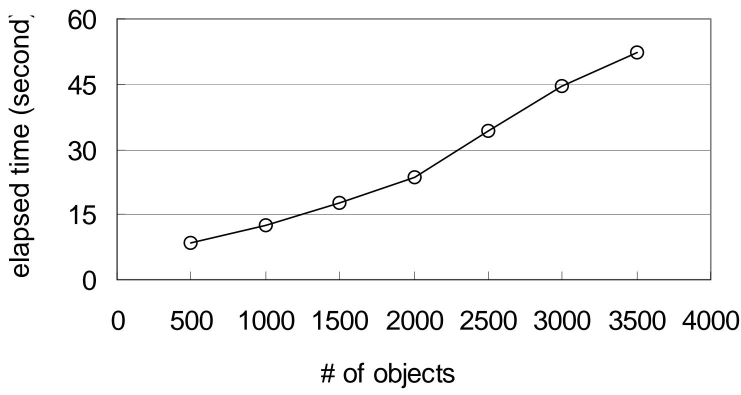


Figure 21: The index construction of APM (L-tree, elapsed time vs. # of objects).

Figure 22: The index construction of APM (L-tree, memory usage vs. # of objects).

For APS, the augmented suffix tree also suffers from the space consumption. Meanwhile, it is not reasonable to construct a full and complete augmented suffix tree just for handling rare cases of extremely long-length queries. On the contrary, an augmented suffix tree of longer tree height is benefit to the efficiency of query processing. In our experiments, the tree height of augmented suffix tree is set to be 4, 8, 16, 32, and 64. The index construction of APS is shown in Figure 23 and Figure 24, where 'h_4' indicates the augmented suffix tree of tree height four, 'h_8' indicates the one of tree height eight, and so on. The augmented suffix tree of a certain tree height scales well as increasing the number of music objects. However, as increasing the tree height, it introduce large amount of space consumption. In our platform, we can construct the augmented suffix tree whose tree height is at most 32.

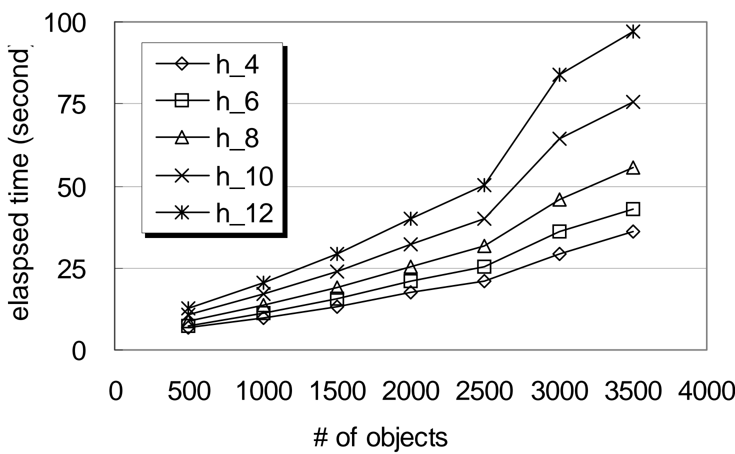


Figure 23: The index construction of APS (2-D AST, elapsed time vs. # of objects).

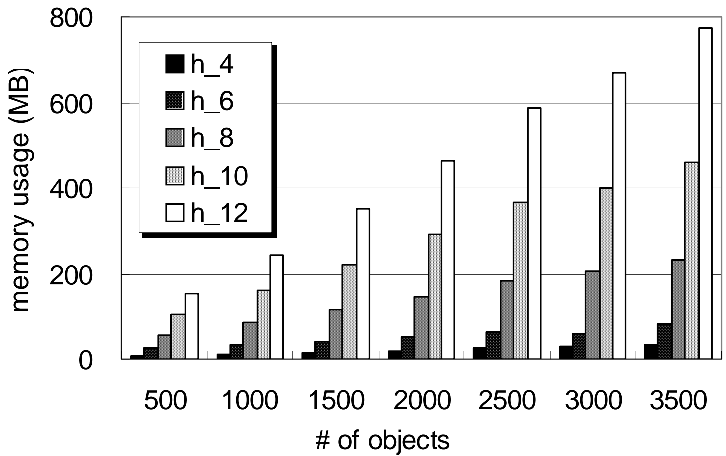


Figure 24: The index construction of APS (2-D AST, memory usage vs. # of objects).

In the following, we discuss the efficiency of processing exact queries for APM and APS approaches. The factor of query length is explored, as shown in Figure 25 and Figure 26. For APM, Figure 25 shows the elapsed time versus query length, where ‘obj_0.5K’ indicates five hundred music objects in the collection, ‘obj_1.0K’ indicates one thousand objects, and so on. When processing queries of length from one to six, the elapsed time decreases rapidly. As processing queries of length seven, the elapsed time rises up substantially. In the experiment setting, the tree height of L-tree is six. As in Section 4.1, if the query is of length seven, it will be divided into two subqueries, *i.e.*, two times of L-tree traversal. In addition, the join processing also contributes an extra computation. For queries of length from seven to thirteen, similar behavior can be obtained. When processing queries of length from seven to twelve, the elapsed time decreases. As processing queries of length thirteen, the elapsed time rises up again, and so on.

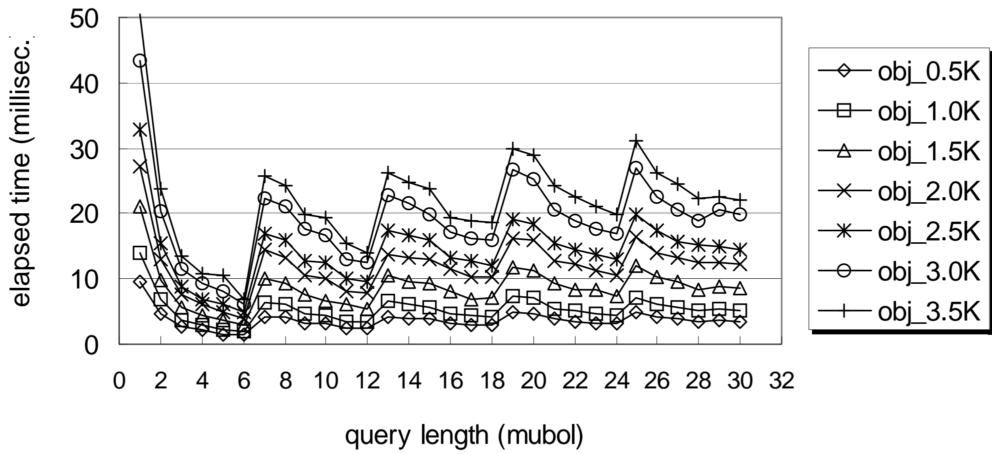


Figure 25: The query processing of APM (L-tree, elapsed time vs. query length).

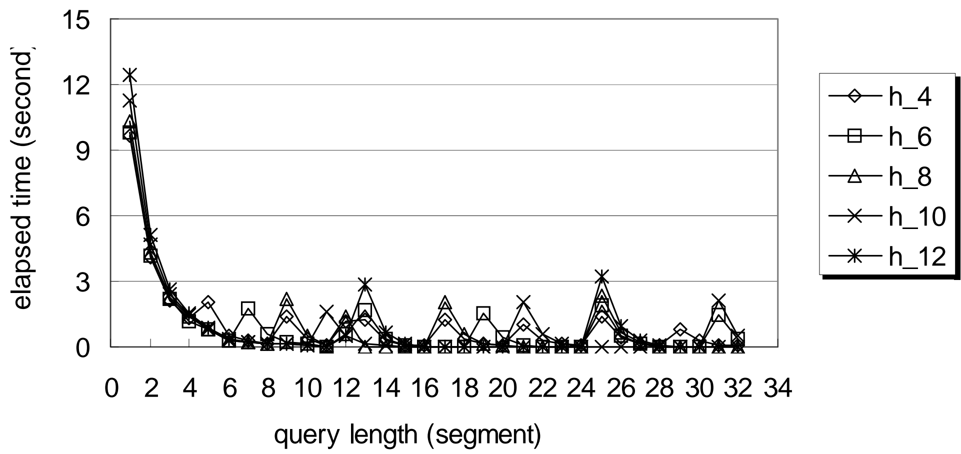


Figure 26: The query processing of APM (2D-AST, elapsed time vs. query length).

For APS, Figure 26 shows the elapsed time versus query length. The curves in Figure 26 have a similar trend to the curves in Figure 25. However, for shorter queries ranging from one to eight music segments, such kind of trend is not obvious in APS. Two reasons are given as follows. In APM, leaf nodes are regarded as results, while leaf nodes of APS are just candidates for further confirmation. In addition, the number of leaf nodes retrieved in APS is much more than the one in APM. Post processing of a large number of candidates results in extra computation which smoothes the curves.

The total elapsed time of query processing in APS consists of three parts, *i.e.*, tree traversal, joining processing (if the query length is longer than the tree height), and post processing (for similarity computation). Among the three parts, the post processing consumes most of the elapsed time. For example, with the 2-D AST of tree height ten, the total elapsed time of processing a ten-segment query is 811 milliseconds, where 10 milliseconds for tree traversal and 801 milliseconds for computing similarity. When processing queries whose

length is longer than the tree height, the query will be divided into subqueries. The number of candidates will be reduced after the joining processing. However, our database of 3500 music objects is only of moderate size. No matter what the tree height is, the number of candidates does not change much.

7. CONCLUSION

In this article, we discuss the issues of representation, similarity measure, indexing and query processing in content-based music information retrieval. Various methods for each issue are presented. For the representation of music objects, we introduce the chord, rhythm, and music segment model, as well as the similarity measurements. All music objects are coded into feature strings, accordingly. To support efficiently matching, the feature strings are organized as an index structure, such as suffix tree-based, (n-gram+tree)-based, and augmented suffix tree-based indexing. Regarding the methods of query processing, the exact matching and approximate matching, including template-based and thresholding-based, are presented in cooperation with appropriate index structures. Finally, a discussion and comparison of these techniques is presented, which shows the intrinsic difficulty of the problem of content-based music information retrieval. To show the performance of our approaches, we are currently building a platform and the comparative study of efficiency is also illustrated.

7.1. Future Work

Some of further improvements to polish various methods of representation and indexing and query processing have been described in Section 5. In addition, two issues regarding the augmented suffix tree-based indexing will be investigated in the future. First, when splitting an edge in the tree construction, the clustering algorithm will be applied to find clusters, and then the edge will be split based on the clustering results. Thus, the filtering effect when resolving queries will be enhanced. On the other hand, too many clusters will also introduce processing overhead when resolving queries. The impact of clustering algorithms and the trade-off between the number of clusters and filtering effect remains unclear. Second, the suffix tree-like data structures suffer from space consumption. One solution to the problem is to cut the augmented suffix tree in a certain height which is similar to L-tree. Another possible solution is to partition the augmented suffix tree into some subtrees. When applying our approach on moderate memory space, the partitioned subtrees still can be fitted in such environment.

In our work of modeling music objects, some components of music objects are ignored for simplicity, such as grace note and rest. We are considering an improvement of coding schemes for handling the two cases. Meanwhile, we are also considering various features, such as

density and tempo, to capture semantics of music objects for better filtering, classification, and recommendation. In addition to continuing our research on monophonic objects, the modeling, representation, indexing and query processing for polyphonic music objects are also one the future research directions.

Furthermore, we are working on the project to provide a unified platform in real scale for assessing the efficiency and retrieval effectiveness of our approaches. Without any benchmarks (standard testing data), the relevance judgment and query set generation are the key issues in this project, as well as the measurement of effectiveness. In the first stage, we will implement the approaches of representing music objects by mubol and music segment, and the 1D-List approach to make a comparative study regarding the key issues. Some preliminary results are reported in Section 6, and the comprehensive and comparative study is still in developing. In the long term, the platform will be continuously maintained and served as the testbed whenever new approaches of content-based music information retrieval are proposed.

ACKNOWLEDGMENT

We would like to thank the CWEB Technology, Inc., for sharing us the data set used in our experiments.

REFERENCES:

Burke, J. (1993). The illustrated dictionary of music. Warner Books and BLP Publishing Limited.

Blackburn, S. & DeRoure, D. (1998). A tool for content-based navigation of music. In Proceedings of the 6th ACM International Multimedia Conference, (pp. 361-368).

Chen, A. L. P., Chang, M., Chen, J., Hsu, J. L., Hsu, C. H. & Hua, S. Y. S. (2000). Query by music segments: An efficient approach for song retrieval. In Proceedings of IEEE International Conference on Multimedia and Expo. New York.

Chen, J. C. C. & Chen, A. L. P. (1998). Query by rhythm: An approach for song retrieval in music databases. In Proceedings of the 8th International Workshop on Research Issues in Data Engineering, (pp. 139-146).

Chou, T. C., Chen, A. L. P., & Liu, C. C. (1996). Music databases: Indexing techniques and implementation. In Proceedings of IEEE International Workshop on Multimedia Data Base Management System.

Chen, M. T. & Seiferas, J. (1984). Efficient and elegant subword-tree construction. In Apostolico A. & Galil, Z. (Ed.), Combinatorial algorithms on words, volume 12 of NATO ASI Series F: Computer and System Sciences, (pp. 97-107), Berlin, Germany: Springer-Verlag.

Clausen, M., Engelbrecht, R., Mayer, D. & Smith, J. (2000). PROMS: A web-based tool for searching in polyphonic music.

DeRoure, D. & Blackburn, S. (2000). Content-based navigation of music using melodic pitch contours. Multimedia Systems, 8(3), Springer. (pp. 190-200).

Downie, S. & Nelson, M. (2000). Evaluation of a simple and effective music information retrieval method. In Proceedings of ACM SIGIR, (pp. 73-80).

Frakes, W. B. & Baeza-Yates, R. (1992). Information retrieval: Data structures and algorithms, Prentice-Hall.

Ghias, A., Logan, H., Chamberlin, D., & Smith, B. C. (1995). Query by humming: Musical information retrieval in an audio database. In Proceedings of the Third ACM International Conference on Multimedia, (pp. 231-236).

Gusfield, D. (1997). Algorithms on strings, trees, and sequences. Cambridge University Press.

Hsu, J. L. & Chen, A. L. P. (2001). Building a platform for performance study of various music information retrieval approaches. technical report.

Hsu, J. L., Liu, C. C., & Chen, A. L. P. (2001). Discovering non-trivial repeating patterns in music data. IEEE Transactions on Multimedia (accepted).

Hsu, J. L., Liu, C. C., & Chen, A. L. P. (1998). Efficient repeating pattern finding in music databases. In Proceedings of Seventh International Conference on Information and Knowledge Management (CIKM'98).

Jain, A. K. & Dubes, R. C. (1998). Algorithms for clustering data, Prentice-Hall, Inc.

Lee, W. & Chen, A. L. P. (2000). Efficient multi-feature index structures for music data retrieval. In Proceedings of SPIE Conference on Storage and Retrieval for Image and Video Databases.

Liu, C. C., Hsu, J. L., & Chen, A. L. P. (1999a). An approximate string matching algorithm for content-based music data retrieval. In Proceedings of International Conference on Multimedia Computing and Systems (ICMCS'99).

Liu, C. C., Hsu, J. L., & Chen, A. L. P. (1999b). Efficient theme and non-trivial repeating pattern discovering in music databases. In Proceedings of the 15th International Conference on Data Engineering.

Lemstrom, K. & Perttu, S. (2000). SEMEX: An efficient music retrieval prototype. In Proceedings of International Symposium on Music Information Retrieval.

McCreight, E. M. (1976). A space economical suffix tree construction algorithm. Journal of Assoc. Comput. Mach., 23, 262-272.

MIDI Manufactures Association (MMA), MIDI 1.0 Specification, http://www.midi.org/.

McNab, R. J., Smith, L. S., Witten, I. H., & Henderson, C. L. (2000). Tune retrieval in the multimedia library. Multimedia Tools and Applications, 10(2/3), Kluwer Academic Publishers.

Prather, R. E. (1996). Harmonic analysis from the computer representation of a musical score. Communication of the ACM, 39(12), pp. 119, (pp. 239-255 of Virtual Extension Edition of CACM).

Sadie, S. (1988). The norton/grove concise encyclopedia of music, W. W. Norton & Company.

Selfridge-Field, E. (1998). Conceptual and representational issues in melodic comparison. In Hewlett, W. B. & Selfridge-Field E. (Ed.), Melodic similarity: Concepts, procedures, and applications (Computing in Musicology: 11), The MIT Press.

Stanley, S. (1980). The new grove dictionary of music and musicians, Macmillan Publishers Limited.

Tseng, Y. H. (1999). Content-based retrieval for music collections. In Proceedings of ACM SIGIR.

Uitdenbogerd, A. & Zobel, J. (1998). Manipulation of music for melody matching. In Proceedings of the 6th ACM International Multimedia Conference, (pp. 235-240).

Uitdenbogerd, A. & Zobel, J. (1999). Melodic matching techniques for large music databases. In Proceedings of the 7th ACM International Multimedia Conference, (pp. 57-66).

Witten, I. H., Moffat, A., & Bell, T. C. (1994). Managing gigabytes: compressing and indexing documents and images, Van Nostrand Reinhold, International Thomson Publishing company.

Yanase, T. & Takasu, A. (1999). Phrase based feature extraction for musical information retrieval. In Proceedings of IEEE Pacific Rim Conference on communications, Computers, and Signal Processing.

Yip, C. L. & Kao, B. (2000). A study on *n*-gram indexing of musical features. In Proceedings of IEEE International Conference on Multimedia and Expo. New York.

Abstract 1
1. Introduction 1
1.1. Related Work 2
2. The Representation of Music Objects 4
2.1. Chord 4
2.2. Mubol 6
2.3. Music Segment 7
2.3.1) An Extension 9
3. Indexing 10
3.1. Suffix Tree-based Indexing 10
3.1.1) Suffix Tree 10
3.1.2) Indexing Chord Strings 11
3.2. (N-gram+Tree)-based Indexing 11
3.2.1) N-gram indexing 11
3.2.2) Indexing Mubol Strings 12
3.3. Augmented Suffix Tree-based Indexing 13
3.3.1) The One-dimensional Augmented Suffix Tree 14
3.3.2) The Two-dimensional Augmented Suffix Tree 16
4. Query Processing 16
4.1. Exact Matching 16
4.2. Template-based Matching 17
4.3. Thresholding-based Matching 18
4.3.1) Query Processing for the One-dimensional Augmented Suffix Tree 19
4.3.2) Query Processing for the Two-dimensional Augmented Suffix Tree 22
5. Discussion 23
5.1. Comparison of Representation Methods 23
5.2. Comparison of Indexing and Query Processing 24
5.3. Intrinsic Difficulties 25
6. Experiments: The Preliminary Results 27
6.1. Implementation 27
6.2. Efficiency Study 29
7. Conclusion 34
7.1. Future Work 34
Acknowledgment 35
References: 35