# Efficient Multi-Feature Index Structures for Music Data Retrieval

Wegin Lee and Arbee L.P. Chen[1]

Department of Computer Science, National Tsing Hua University,
Hsinchu, Taiwan 300, R.O.C.

## ABSTRACT

In this paper, we propose four index structures for music data retrieval. Based on suffix trees, we develop two index structures called Combined Suffix Tree and Independent Suffix Trees. These methods still show shortcomings for some search functions. Hence we develop another index, called Twin Suffix Trees, to overcome these problems. However, the Twin Suffix Trees lack of scalability when the amount of music data becomes large. Therefore we propose the fourth index, called Grid-Twin Suffix Trees, to provide scalability and flexibility for a large amount of music data. For each index, we can use different search functions, like exact search and approximate search, on different music features, like melody, rhythm or both. We compare the performance of the different search functions applied on each index structure by a series of experiments.

**Keywords:** Content-based music data retrieval, index structure, suffix tree, query processing

## 1. INTRODUCTION

In the area of multimedia databases, content-based retrieval of images is an active research area, where many systems, like QBIC [7], Virage [13], Photobook [12], and MARS [11], have been developed in addition to some image search engines, like VisualSeek [19] and NetView [20]. To meet the retrieval purposes of different users, they all combine different features extracted from image data to support effective and efficient queries.

For content-based retrieval of music data, there are some research works [2][4][5][6][9][17]. Most approaches extract a feature from the music data and develop an index to manage it. Users pose queries based on the feature and the system searches the answers by using theses indexes. Ghias et al. [2] use a sequence of 3 symbols ('U', 'D', and 'S', representing that a note is higher than, lower than, or the same as its previous note) to represent the content of music data. The music search is transformed into processing approximate string matching without using any index. The representation method is very rough and other important music features are not included. For efficient string matching, a suffix-tree-like index is developed in [4] to manage the feature. Additionally an interface is designed to navigate the answers. In [5], we use chord sequences to represent the content of music data and develop an index structure to provide efficient approximate matching. In [6], we propose an approach for retrieving music data by rhythm and define five kinds of similarity relationships between different rhythm strings. An index structure is proposed for managing and searching rhythm strings. In the works mentioned above, a specific index is developed (or not) to manage a music feature and to provide query functions for a single feature.

In [9], we propose an efficient algorithm to enable exact and approximate string matching for any music feature, like melody, rhythm or chord, in a link-list-like index. Because the index structure is quite simple, it just can provide exact search and approximate search with a difference of one. In addition to exact and approximate search, some other search functions, such as range search, are also needed. Moreover, in [8] we show that the property of music repetition is useful for searching and clustering music data. Hence it should also be provided.

In this paper, we propose four indexes to manage different features extracted from music data and to support different search functions. We first develop two indexes, called Combined Suffix Tree and Independent Suffix Trees, which are similar to suffix trees, to manage different feature-strings. We found that some search functions still show bad performance for a specific index. Hence we propose a third index structure, called Twin Suffix Trees, to improve these problems. However, this method lacks scalability when the amount of music data is large. Therefore, we invent a fourth index structure, named Grid-Twin Suffix Trees, that provides the scalability and flexibility to overcome these shortcomings.

[1] Correspondence: Email: alpchen@cs.nthu.edu.tw

The paper is organized as follows: In Section 2 we describe the features of music data used in this paper and introduce the suffix tree. Later we explain the four search functions often used for suffix trees. Section 3 describes the Combined Suffix Tree and Independent Suffix Trees. We also explain the shortcomings of these two indexes. In Section 4, we develop an enhanced index, called Twin-Suffix Trees, to improve these problems. Section 5 proposes the fourth index, called Grid-Twin Suffix Trees, to manage large music data. In Section 6, we make experiments to evaluate the performance of the search functions applied on different indexes. Section 7 contains the conclusion and the future work.

## 2. RELATED WORK

In this section, we explain the music features used in this paper, introduce the suffix trees, and explain four important search functions for suffix trees.

### 2.1. The music features

We assume the music feature strings contain the melody and rhythm features. For example, Figure 1 shows a piece of "You Are My Sunshine". The corresponding melody feature string is "sol-do-re-mi-mi-mi-mi-re-mi-do-do" and the corresponding rhythm string is "1-1-1-2-2-1-1-1-1-2-2". The music feature string is represented as "sol-1, do-1, re-1, mi-2, mi-2, mi-1, mi-1, re-1, mi-1, do-2, do-2". For simplicity we use letters (i.e., 'a', 'b', 'c', …) to represent the melody symbols (i.e., 'do', 're', 'mi', …). For our example, the music feature string is represented as "$e_1a_1b_1c_2c_2c_1c_1b_1c_1a_2a_2$".

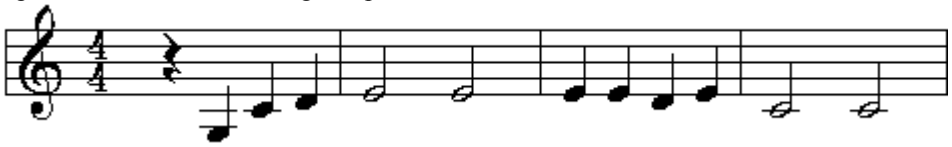

Figure 1: A piece of "You Are My Sunshine"

### 2.2. Suffix trees

There are several research works about suffix trees [10][15][18], because they provide efficient solutions for many string matching problems. The suffix tree is improved from suffix trie that is originally evolved from patricia tree [3]. Moreover, suffix tree is similar to position tree [1], compact suffix tree [18] and pat tree [3].
The suffix tree used in this paper is similar to a suffix trie. For example, given a string "ababc", the suffixes are "ababc", "babc", "abc", "bc", and "c". Figure 2 shows the suffix tree for our example. The information in the leaf nodes stands for the starting position of the suffixes in the string "ababc".

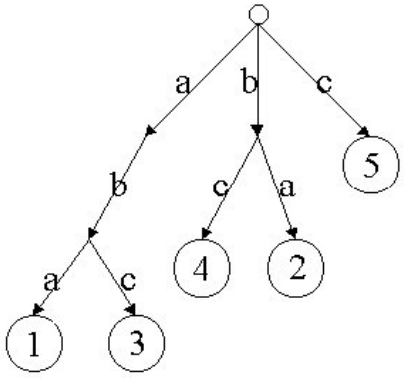


Figure 2: Construction of a suffix tree

### 2.3. Search functions for suffix trees

There are four general search functions for suffix trees [3], which we explain in the following.
*Exact search*: Given a query, like "aba", we find all strings containing "aba". In Figure 3 we traverse the path "a-b-a" to an internal node in the suffix tree. The answer consists of all leaf nodes under the internal node.
*Approximate search*: Given a query and an allowable difference *k*, we find all strings within the difference *k* to the query string. Here we only use the *transition distance*. For example, given a string "aba" and a difference of one, "aaa", "aba",

"abb", and "bba" are all answers. We do not consider "ab" or "abaa" as answers, because they have one delete or add difference, respectively. There are two approaches for the approximate search. One is to create all possible query strings within the difference and then do exact search for each string. The other approach is to use the depth-first recursive algorithm to traverse the suffix tree and to find all approximate paths within the difference. Moreover, there are other research works [14][16] providing more efficient algorithms on suffix trees for approximate search.

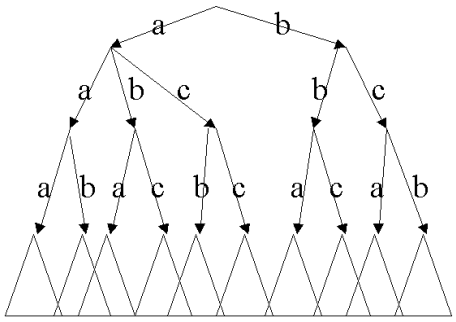


Figure 3: Search functions for suffix trees

*Repetition search*: When we do exact search, there can be two or more answers in the same music object. We say that there is repetition in the music object for the query string. We can do repetition search by examining all the answers to see whether some of them are from the same music object.

*Range search*: If there is an order for the symbols of the feature strings, we can provide range search in suffix trees. For example, assume the order of symbols in Figure 3 is "a⇨b⇨c". Given two strings, "aab" and "acc", the range search finds all paths between the two strings in Figure 3. The leaf nodes below the paths "aba", "abc", and "acb" are all answers, because they are lexicographically between "aab" and "acc".

## 3. TWO INDEXES BASED ON SUFFIX TREES

In this section, we introduce two index structures, Combined Suffix Tree and Independent Suffix Trees, which are similar to suffix trees. We also show some shortcomings of these two indexes.

### 3.1. Combined Suffix Tree

The feature strings are directly used to construct the index in the index structure Combined Suffix Tree. If there are three melody symbols ('a', 'b', and 'c') and two rhythm symbols ('1' and '2'), each internal node in the Combined Suffix Tree can have an out-degree up to 6 (i.e. "$a_1$", "$a_2$," "$b_1$," "$b_2$","$c_1$", and "$c_2$"), pointing to child nodes. Figure 4 shows the index that is constructed from the feature string "$a_1b_1a_1b_1c_1$".

There are 3 general cases for string matching in a suffix tree. We use examples to explain these cases.

*Case 1*: We search the string "$a_1b_1$" in Figure 4. We can directly get the answer {①,③} from the leftmost path of the Combined Suffix Tree.

*Case 2*: We search the string "$b_1a_1b_1$" in Figure 4. We match the first two symbols "$b_1a_1$" with the path "$b_1$-$a_1$" and reach a leaf node. We read the strings from the music database to continue string matching, starting at the position saved in the leaf node. In the example, we match the string and get the answer {②}. In other cases there may be no match.

*Case 3*: We search the string "$a_1c_1$" in Figure 4. There is no such path in the tree. Therefore, there is no result.

This index structure is efficient for multi-feature search functions. From the above Case 1, we can see that the multi-feature exact search is straightforward. Moreover, we can directly perform the multi-feature approximate search by using a depth-first recursive traverse. However, the index is not suitable for single music features. When we perform single feature exact search for the rhythm string "121", the search algorithm must find all possible paths in the index, such as "$a_1b_2b_1$", "$a_1b_2b_1$", "$b_1a_2c_1$", "$b_1b_2a_1$", "$c_1a_2c_1$", and "$c_1c_2c_1$" (represented by bold lines in Figure 5).

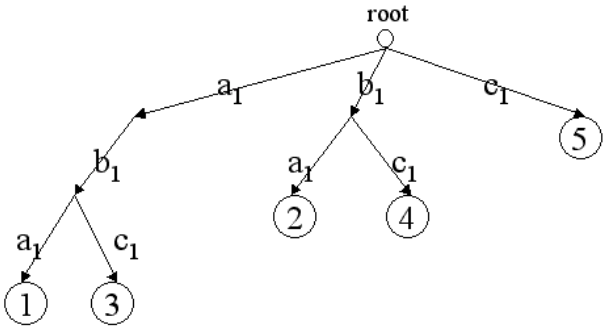

Figure 4: An example of a Combined Suffix Tree

We limit the height of the Combined Suffix Tree to keep its size small (e.g. 3 in Figure 5). Therefore, a linked list of leaf nodes is maintained to record the suffixes with the same prefix. We also use this limit for the following indexes.

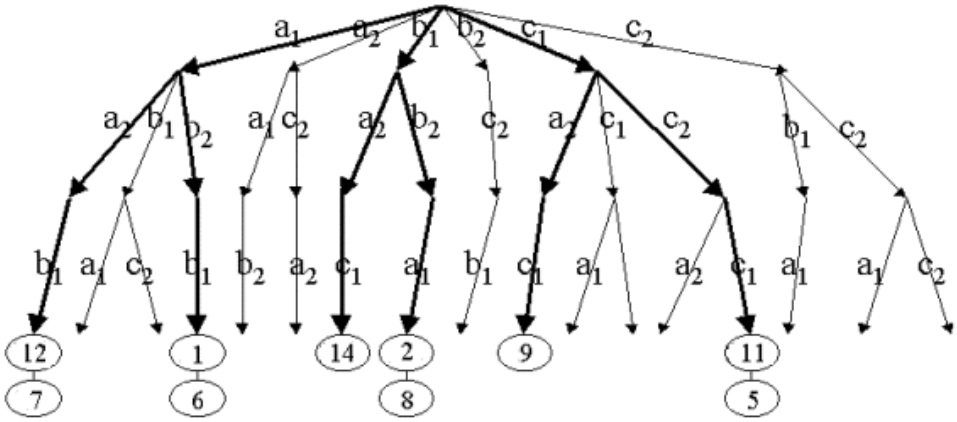

Figure 5: Search in the Combined Suffix Tree

### 3.2. Independent Suffix Trees

The index structure Independent Suffix Trees separates the feature strings into a melody and a rhythm string and stores them in two independent suffix trees. Figure 6 shows the Independent Suffix Trees for the feature string "$a_1b_2a_1b_2c_2$", where $ marks the end of the string.

Obviously the index outperforms the Combined Suffix Tree for single feature search functions. For example, when we perform exact search for the rhythm string "121" in the rhythm suffix tree of Figure 6, we traverse only the link "1-2-1" and get the answer {①}. We use the depth-first recursive procedure to traverse the suffix trees for the approximate search.

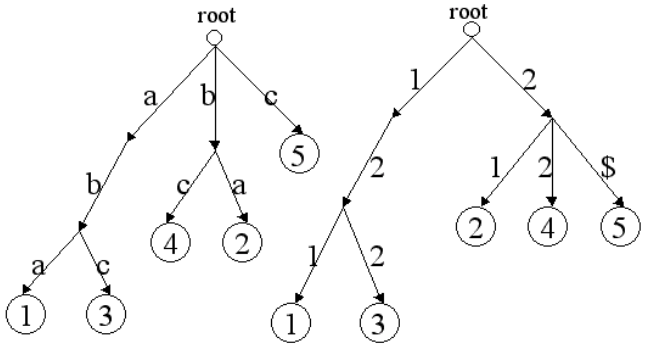

Figure 6: An example of the Independent Suffix Trees

The disadvantage of this index structure is, that it is time-consuming for multi-feature search functions. For example, if we perform an exact search for "$a_1b_2$", we first separate the string "$a_1b_2$" into two feature strings, i.e., "ab" and "12". We get the

melody answer set {①,③} and rhythm answer set {①,③} (see Figure 6). Finally we intersect the two answer sets to get the final result {①,③}.

## 4. TWIN SUFFIX TREES

### 4.1. Construction of Twin Suffix Trees

The index structure Twin Suffix Trees is constructed by adding additional information to the Independent Suffix Trees. This index structure consists of a melody and a rhythm suffix tree with links pointing from each *melody node* in the melody suffix tree to the corresponding *rhythm nodes* in the rhythm suffix tree.

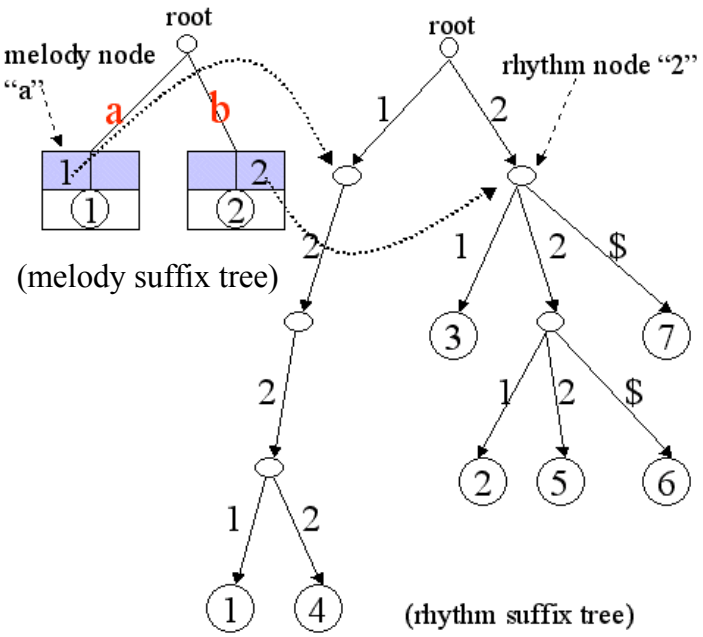


Figure 7: Construction of the Twin Suffix Trees

We describe the construction of the Twin Suffix Trees by using the example "$a_1b_2a_2b_1a_2b_2c_2$". First, we construct the rhythm suffix tree using the rhythm string "1221222" as shown in Figure 7. Then we construct the melody suffix tree and link each melody node to the corresponding rhythm nodes in the rhythm suffix tree. For "$a_1b_2a_2b_1a_2b_2c_2$", we first create the melody node 'a' under the root of the melody suffix tree and find the corresponding rhythm node '1' under the root of the rhythm suffix tree. We link melody node 'a' with the corresponding rhythm node '1'. Then, for the suffix "$b_2a_2b_1a_2b_2c_2$", we create melody node 'b' under the root of the melody suffix tree and find the corresponding rhythm node '2' under the root of the rhythm suffix tree. We also make a link between these two nodes (see Figure 7). Each melody node consists of two parts, i.e. a link and a data part. The link part consists of n out-links to the corresponding rhythm nodes in the rhythm suffix tree, where n denotes the number of distinct rhythm symbols. The data part of the leaf nodes contains the starting position of the corresponding suffix. For internal nodes it is empty and therefore it does not appear in the Figures. If we have a link from a melody node m to a rhythm node n, but there exists already a link to another rhythm node with the same rhythm symbol n, a new melody node is created. This node is placed under the corresponding link part n of melody node m. The final index is shown in Figure 8.

### 4.2. The multi-features exact search in Twin Suffix Trees

We use the three cases mentioned in Section 3.1 to explain different situations for multi-feature exact search in Twin Suffix Trees. We use the index shown in Figure 8.

*Case 1*: When we search "$a_2b_2$", we first go to the melody node 'a' under the root in the melody suffix tree and to the rhythm node '2' under the root in the rhythm suffix tree. There is a link pointing from the melody node 'a' to the corresponding rhythm node '2'. Then we go to the next melody node 'b' in the melody suffix tree and the next rhythm node '2' in the rhythm suffix tree. We find two links in the melody node 'b' pointing to different rhythm nodes. Only one link points to the

corresponding rhythm node '2' in the path "root-2-2". Therefore we get the answer sets {①,③,⑤} and {②,⑤,⑥}. The intersection of the two answer sets lead to the result {⑤}.

*Case 2*: Given the search string "$a_2b_1a_2$", we directly traverse the melody nodes 'a' and 'b' in the melody suffix tree and the rhythm nodes '2' and '1' in the rhythm suffix tree. We reach a leaf node in the rhythm suffix tree. Therefore, we read the rhythm string from the music database to continue string matching, starting at the third position. If the string matching fails, we stop the search and return an empty result. If it succeeds, we go to the next node 'a' in the melody suffix tree and get the answer set {①,③}. We intersect {①,③} and {③}, and get the final result {③}.

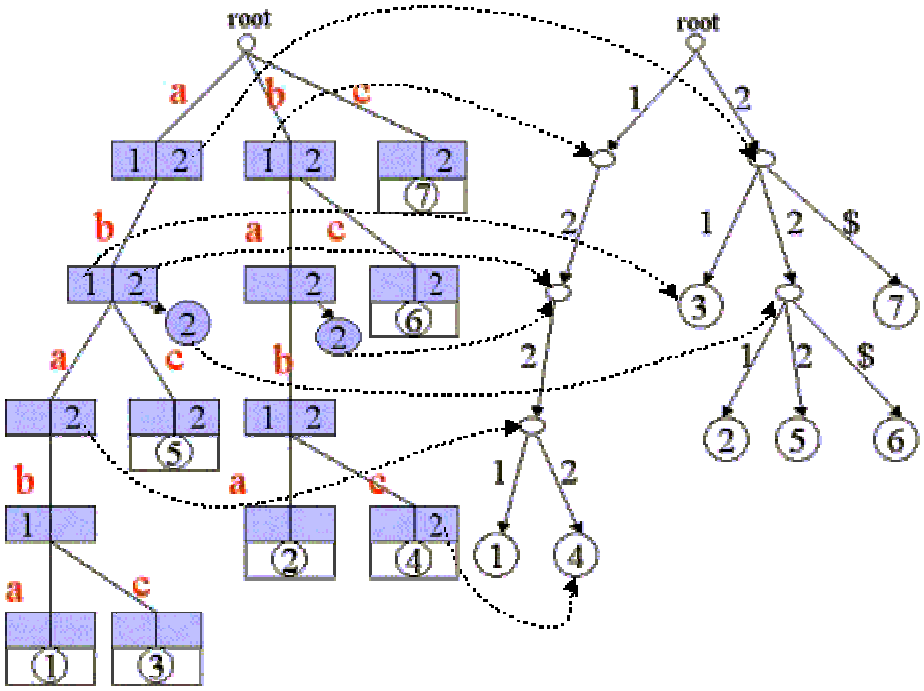


Figure 8: The Twin Suffix Trees constructed from "$a_1b_2a_2b_1a_2b_2c_2$" (not all links are shown)

*Case 3*: Given the search string "$a_2b_2a_1$", we first traverse the melody nodes 'a' and 'b' and rhythm nodes '2' and '2'. We check the links between the melody and the rhythm nodes. From melody node 'b' we go to node 'a', and from node '2' in the rhythm tree we go to node '1'. Because there is no link pointing from melody node 'a' to rhythm node '1', we stop the search and return an empty result. A main advantage of the Twin Suffix Trees is that an empty result can be detected earlier (than in the Independent Suffix Trees).

### 4.3. The multi-feature approximate search in Twin Suffix Trees

Because of additional links in the melody suffix tree, we can perform the multi-feature approximate search using a depth-first traverse algorithm synchronously for the two suffix trees. As an example, we perform the approximate search for the string "$a_1b_2$" within a difference of one in Figure 8. First we go to melody node 'a' and rhythm node '1' in the melody and rhythm suffix trees, and check if melody node 'a' links to the rhythm node '1'. This is true and the difference is zero. We go to the next melody node 'b' and the next rhythm node '2'. There are three different links pointing from melody node 'b' to three different rhythm nodes. Among them, one link points to the rhythm node '2'. The difference is still zero. We continue traversing the two suffix trees to get the melody and rhythm answers {①,③,⑤} and {①,④}. After the intersection, the final result is {①}.

We go back to melody node 'a' and find that there are no other child nodes. We check the other link in melody node 'a'. It points to rhythm node '2' in the rhythm suffix tree. The difference is now one. We go to melody node 'b' and find that there are two links pointing to the rhythm nodes '1' and '2', which are child nodes of the rhythm node '2'. Therefore we get two approximations, "$a_2b_1$" and "$a_2b_2$". The difference between "$a_2b_1$" and "$a_1b_2$" is two and the difference between "$a_2b_2$" and "$a_1b_2$" is one. Only "$a_2b_2$" is a valid approximation. Hence we continue to traverse the two suffix trees to get the answers {①,③,⑤} and {②,⑤,⑥}. After the intersection, the final result is {⑤}. We go back to the melody node 'a' and there is no

other link to a rhythm node. We just go back to the root of the melody suffix tree to search the rest of the tree for other approximate paths.

## 5. GRID-TWIN SUFFIX TREES

The previous index structures lack scalability when the amount of music data increases. Therefore we propose a flexible index that provides scalability. It also can be adopted in different computing environments.

### 5.1. Construction of Grid-Twin Suffix Trees

Figure 9 shows an overview of the Grid-Twin Suffix Trees. We first use a hash function to map each suffix of the feature string into a specific bucket of a 2D grid. The hash function uses the first n symbols of the suffix to map it into a specific bucket. After hashing each suffix, several suffixes may be mapped into the same bucket.

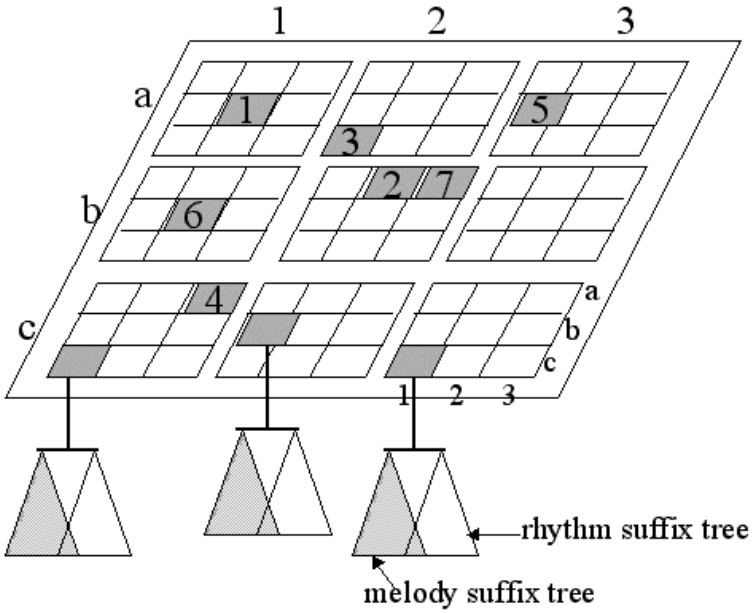


Figure 9: An example of the Grid-Twin Suffix Trees

The following is the hash function:

$$P(x, y) = \sum_{i=1}^{n} \left(\frac{(Num_m)^n}{(Num_m)^i} M_i, \frac{(Num_r)^n}{(Num_r)^i} R_i\right),$$

where $x$ and $y$ are the row and column coordinates respectively, and P(x,y) denotes the position of the bucket. $Num_m$, $Num_r$, $M_i$ ,and $R_i$ are the sizes of the melody and rhythm alphabets and the value of the $i_{th}$ melody and rhythm symbols, respectively. The length of the suffix is denoted by n. $Num_m$ and $Num_r$ are assumed to be three in Figure 9. The values that represent melody symbols 'a', 'b', and 'c', and rhythm symbols '1', '2', and '3' are 0, 1, and 2, respectively. To insert the first suffix (with length two) of the feature string "$a_1b_2a_2c_1a_3b_2b_2a_3a_1$" for example, we hash the first symbol "$a_1$" to $P(x,y)_1 = (3^2/3^1*0, 3^2/3^1*0) = (0,0)$. Here $i$ is 1 and $M_i$ and $R_i$ are both 0. $P(x,y)_2$ equals to $(3^2/3^2*1, 3^2/3^2*1) = (1,1)$ for the second symbol "$b_2$". Then we sum the two position values (0,0) and (1,1) to (1,1) and map the suffix into the (1,1) bucket in Figure 9. The second suffix "$b_2a_2c_1a_3b_2b_2a_3a_1$" is mapped into bucket (3,4). The mapped buckets for the other suffixes are shown in Figure 9. They are labeled by numbers. After hashing all suffixes, we construct the Twin Suffix Trees "under" the buckets. The construction of the Twin Suffix Trees is mentioned in Section 4.1.

### 5.2. Search functions for Grid-Twin Suffix Trees

The search functions for Grid-Twin Suffix Trees consist of two phases. The first phase is to hash the first n symbols of the query string into the corresponding buckets of the grid. The second phase is to apply the search functions on the Twin Suffix Trees for the rest of the query string. In the following, we explain the search functions for the Grid-Twin Suffix Trees.

*Multi-feature exact search*: When we search "$b_2b_1b_1b_2$" for example, we first hash the first two symbols "$b_2b_1$" into the (4,3) bucket (see Figure 10). We continue the multi-feature exact search in the Twin Suffix Trees under the (4,3) bucket for the rest of the query string, i.e., "$b_1b_2$". The search process for the Twin Suffix Trees is described in Section 4.2.

*Single feature exact search*: When we search "21112" for example, we hash the first two symbols "21" into the fourth column. (summation of (3) and (0) ). For all buckets of the column, we perform single feature exact search in the Twin Suffix Trees for the rest of the query string, i.e., "112". In the example, we have to perform the single feature exact search on 9 buckets, (0,3), (1,3), …, and (8,3).

*Multi-feature approximate search*: We explain the approximate search for the feature string "$a_2b_1b_1b_2$" within a difference of one. For the first two symbols "$a_2b_1$", we generate all possible strings within a difference of one, i.e., "$a_1b_1$", "$a_2b_1$", "$a_2b_2$", "$a_2b_3$", "$a_2c_1$", …, and "$c_3b_1$". We hash them into the corresponding buckets. Then we perform multi-feature approximate search or exact search for the Twin Suffix Trees under these buckets for the rest of query string, i.e., "$b_1b_2$". In the example, only the Twin Suffix Trees under the bucket (1,3) need to be checked by approximate search, because the difference of the prefix string "$a_2b_1$" is zero. We perform multi-feature exact search for the other Twin Suffix Trees.

*Single feature approximate search*: We make the approximate search for the string, "bcba" within a difference of one. For the first two symbols "bc", we generate all possible strings within one difference, i.e., "bc", "ba", "bb", "cc", and "ac". They are hashed into rows (5), (3), (4), (8), and (2). For each bucket of these rows, we continue performing single feature approximate search or exact search in the Twin Suffix Trees for the rest of query string, i.e., "ba". In the example, we perform single feature approximate search only for the Twin Suffix Trees under the buckets of row (5), because the difference of the prefix string "bc" is zero.

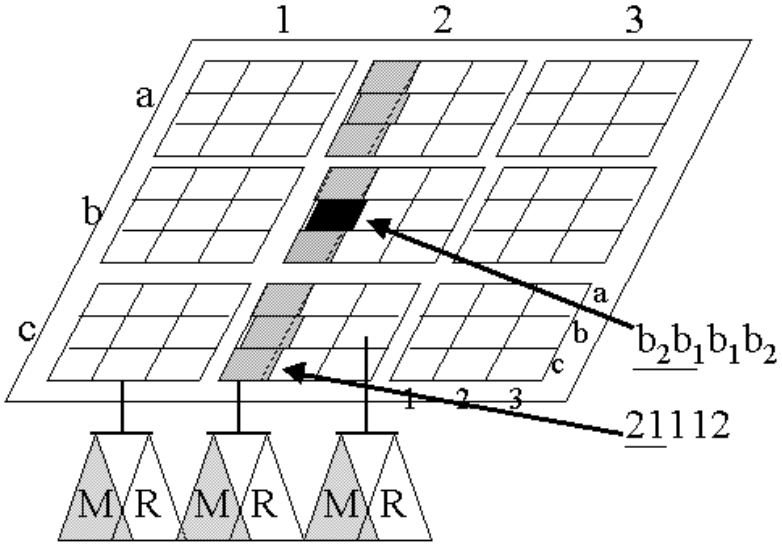


Figure 10: Search on Grid-Twin Suffix Trees

### 5.3 Evaluation of Grid-Twin Suffix Trees

The structure of the Grid-Twin Suffix Trees is a tradeoff between the size of the grid and the size of the Twin Suffix Trees. If we choose a small grid, the Twin Suffix Trees below the buckets will be large. On the other hand, the Twin Suffix Trees become small if we choose a large grid. Due to its structure, it can be partitioned into a set of small grids with the associated Twin Suffix Trees and be executed in a parallel or distributed environment.

The index structure is of scalability. In previous index structures the time spent on queries increases rapidly with the amount of music data, especially for short queries. We take Figure 11 as an example. We construct the tree shown in Figure 11(a) with a small, and the tree in Figure 11(b) with a larger set of music data. When we request the string "aba", we have to traverse the dashed triangle in both indexes to get the answers. In Figure 11(b), it needs more time to traverse the triangle, because there are more answers in the tree. Therefore there is a problem of scalability in previous indexes, especially in Twin

and Independent Suffix Trees. The Grid-Twin Suffix Trees is of scalability to improve the above situation. The results of the experiments in Section 6.3 show clearly this advantage.

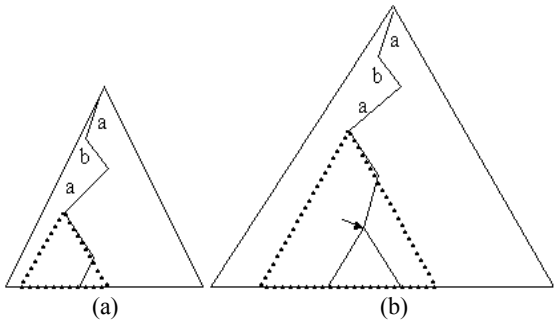


Figure 11: An example for the scalability

## 6. EVALUATION OF EXPERIMENTS

In this section, we make several experiments to compare the performance of search functions for different index structures. We extract music feature strings from 100 songs in MIDI format. The length of each feature string is about 400. Additionally we generate 900 music feature strings randomly. The length of the random feature strings is 500. For simplicity, we just use 22 melody and 10 rhythm symbols. All experiments are run on a Desktop PC with Intel CPU Pentium 200 MHz, 64 MB RAM and 2GB hard disk.

### 6.1. Comparison of multi-feature search functions

Figure 12 shows the performance of the multi-feature exact search function for the four index structures. We test 100 query strings of length 3 constructed from different numbers of songs, (i.e., 200, 400, 600, 800 and 1000). The Figure shows that only the performance of the Independent Suffix Trees is not good. However we do not clearly know which index structure is the best in Figure 12.

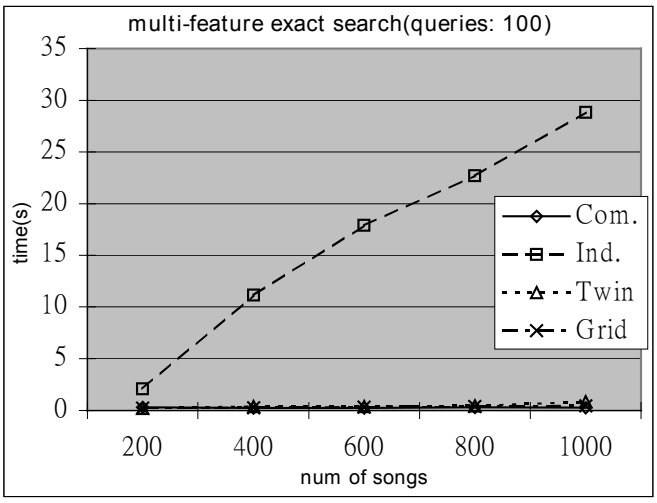


Figure 12: Multi-feature exact queries with length 3

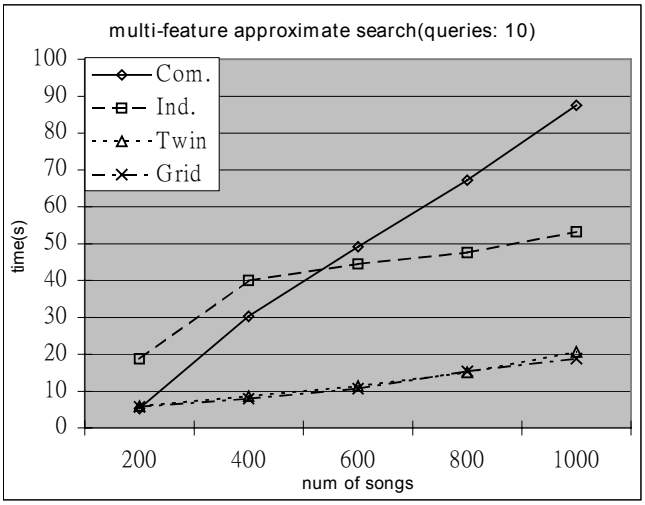


Figure 13: Multi-feature approximate queries with length 3

Figure 13 shows the performance of multi-feature approximate search for the four index structures. Ten queries of length 3 are tested on these index structures and the allowable difference is one. The performance of the Twin Suffix Trees almost equals that of the Grid-Twin Suffix Trees. The processing time for these indexes is better than that for the other two index structures. For small music databases, the Independent Suffix Trees perform worser than the Combined Suffix Tree. However, when the amount of music data increases, the performance of the Independent Suffix Trees is better than that for the Combined Suffix Tree.

### 6.2. Comparison of single feature search functions

Figure 14 shows the processing time of the four index structures for melody exact search. The performance of the Grid-Twin Suffix Trees is the best. The difference between the performance of the Independent Suffix Trees and Twin Suffix Trees is small. The Combined Suffix Tree shows the highest processing time.

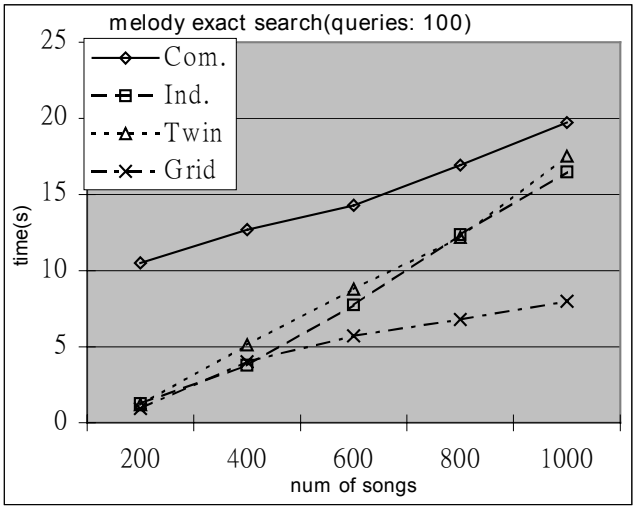


Figure 14: Melody exact queries with length 3

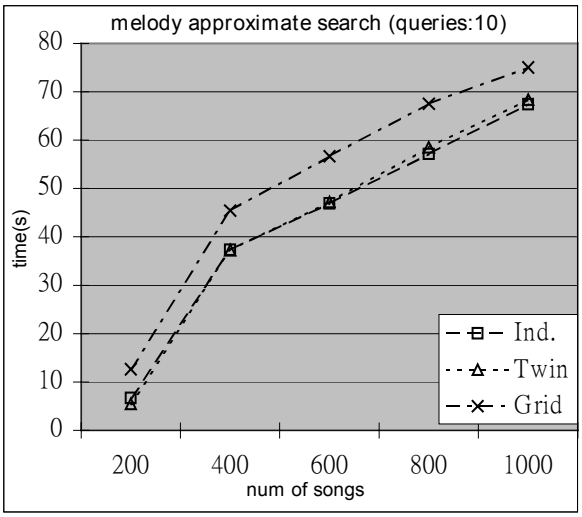


Figure 15: Melody approximate queries with length 3

Figure 15 compares the performance of melody approximate search for the four index structures. For this experiment, we do not show the performance of the Combined Suffix Tree, because its processing time is high compared to the other three index structures. Ten query strings of length 3 are tested on different index structures and the allowable difference is one. The performance of the Grid-Twin Suffix Trees is worser than that of the Twin Suffix Trees and Independent Suffix Trees, which show almost the same performance.

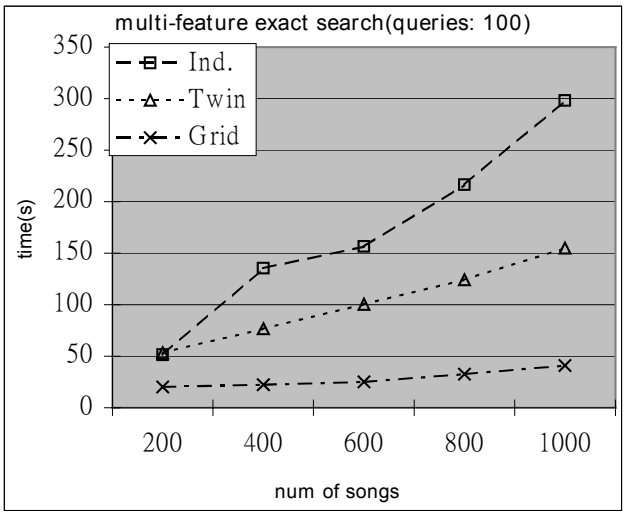


Figure 16: Multi-feature exact queries with length 2

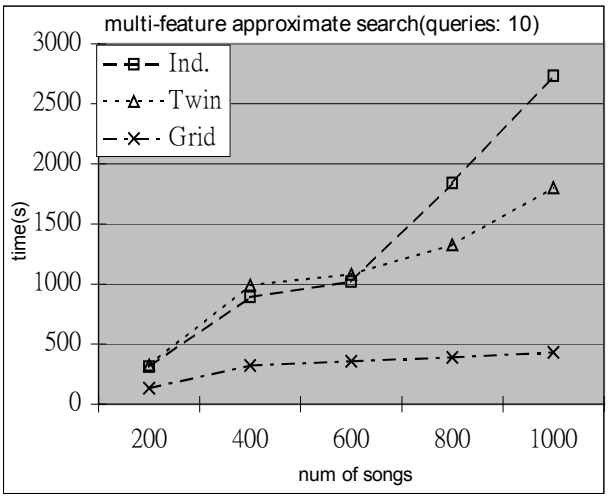


Figure 17: Multi-feature approximate queries with length 2

### 6.3. The scalability of Grid-Twin Suffix Trees

In this subsection we examine the scalability of the Grid-Twin Suffix Trees. In the previous experiments, we can not see the effect of scalability clearly, except for Figure 14. In Figure 16, the multi-feature exact search for the different index structures is tested on 100 query strings of length 2. Only the processing time of the Grid-Twin Suffix Trees increases smoothly.

In Figure 17, we use 10 multi-feature query strings to perform approximate search on different index structures with an allowable difference of one. The length of the query strings is also 2. We can see that the processing time for the search functions for the Grid-Twin Suffix Trees increases smoothly when the amount of music data grows. In Table 1, we use 100 multi-feature query strings of length 4 or 5 to test the different index structures. The performance difference between these index structures is not very conspicuous. It means that the query strings are only matched to a few answers in all index structures. Hence the search time is small for all index structures.

| | multi-feature exact search( number of queries: 100 ) | | | | | | |
|---|---|---|---|---|---|---|---|
| | query length: 5 | | | | query length: 4 | | |
| number of songs | Com. | Ind. | Twin | G.-T. | Ind. | Twin | G.-T. |
| 200 | 0.17(s) | 0.22 | 0.05 | 0.11 | 0.06 | 0.06 | 0.06 |
| 400 | 0.11 | 0.22 | 0.11 | 0.11 | 0.83 | 0.05 | 0.06 |
| 600 | 0.11 | 0.22 | 0.11 | 0.16 | 1.37 | 0.06 | 0.06 |
| 800 | 0.11 | 0.28 | 0.17 | 0.16 | 2.47 | 0.11 | 0.11 |
| 1000 | 0.11 | 0.28 | 0.27 | 0.2 | 3.41 | 0.11 | 0.16(s) |

Table 1: 100 multi-feature queries with query length 4 or 5

## 7. CONCLUSION

This paper provides four index structures using multiple music features to manage and search music data. Based on suffix trees, we develop the Combined Suffix Tree and the Independent Suffix Trees index structures. There are some shortcomings for the search functions for the two index structures. Therefore we propose the third index structure, Twin Suffix Trees, to overcome these problems. However, this index structure lacks scalability when the amount of music data increases. Finally, we propose the fourth index, Grid-Twin Suffix Trees. Its processing time only increases smoothly if the amount of music data grows. Moreover, it can be adopted on different computing environments. Our experiments compare the performance of the different search functions for the four index structures and they show clearly the effect of scalability for the Grid-Twin Suffix Trees. In the future, we will develop an algorithm for the Twin Suffix Trees to manage larger music databases. We will also implement a parallel version of the Grid-Twin Suffix Trees to improve its performance. Additionally we will explore new similarity measurements which are more suitable for music data.

## 8. REFERENCES


[1] A. A., Hopcroft J., and J. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, Section 9.5, 1974.

[2] A. Ghias, J. Logan, D. Chamberlin, and B. C. Smith, "Query by Humming: Musical Information Retrieval in an Audio Database", *Proc. ACM Multimedia*, pp. 231-236, 1995.

[3] W. B. Frakes and R. A. Baeza-Yates, "Information Retrieval: Data Structures and Algorithms", Prentice Hall, Chapter 5, 1992.

[4] S. Blackburn and D. DeRoure, "A Tool for Content-based Navigation of Music", *Proc. ACM Multimedia*, pp. 361-368, 1998.

[5] T.C. Chou, A.L.P. Chen, and C.C. Liu. "Music Databases: Indexing Techniques and Implementation", *Proc. IEEE International Workshop on Multimedia Data Base Management Systems,* 1996.

[6] J.C. Chen and A.L.P. Chen, "Query by Rhythm: An Approach for Song Retrieval in Music Databases", *Proc. IEEE International. Workshop on Research Issues in Data Engineering*, pp. 139-146, 1998.

[7] M. Flickner et al., "Query by Image and Video Content: The QBIC System", *IEEE Computer*, Sep. 1995.

[8] J.L. Hsu, C.C. Liu, and A.L.P. Chen, "Efficient Repeating Pattern Finding in Music Databases", *Proc. ACM International Conference on Information and Knowledge Management*, 1998.

[9] C.C. Liu, A.J.L. Hsu, and A.L.P. Chen, "An Approximate String Matching Algorithm for Content-Based Music Data Retrieval", *Proc. IEEE International Conference on Multimedia Computing and Systems*, 1999.

[10] E. McCreight, “A Space-Economical Suffix Tree Construction Algorithm”, *Journal of Association for Computing Machinery*, pp. 262-272, 1976.
[11] M. Ortega, S. Mehrotra, Y. Rui, and K. Chakrabar, "Supporting Similarity Queries in MARS", *Proc. ACM Multimedia*, pp. 403-412, 1998.
[12] A. Pentland, R. W. Picard, and S. Sclaroff, “Photobook: Tool for Content-Based Manipulation of Image Databases ”, *Proc. Storage and Retrieval for Image and Video Databases II, SPIE,* pp. 34-37, 1994.
[13] J. R. Bach, C. Fuller, A. Gupta, and A. Hampapur, “The Virage Image Search Engine: An Open Framework for Image Management”, *Proc. SPIE Storage and Retrieval for Still Image and Video Databases,* Vol. 2670, pp. 76-87, 1996.
[14] U. Manber and G. Myers, “An Algorithm for String Matching with a Sequence of Don’t Cares”, *Information Processing Letters*, Vol. 37, pp. 133-36, 1991.
[15] E. Ukkonen, “Approximate String-Matching over Suffix Trees”, *Combinatorial Pattern Matching*, pp. 228-242, 1993.
[16] E. Ukkonen, “On-Line Construction of Suffix Trees”, *Algorithmica*, Vol. 10, pp. 353-364, 1993.
[17] L. Uitdenbogerd and J. Zobel, “Manipulation of Music for Melody Matching”, *Proc. ACM Multimedia*, pp. 235-240, 1998.
[18] P. Weiner, “Linear Pattern matching algorithms”, *Proc. IEEE Ann. Symp. on Switching and Automata Theory*, pp. 1-11, 1973.
[19] J.R Smith and S.-F. Chang, “VisualSeek: a fully automated content-based image query systems”, *Proc. ACM Multimedia*, pp. 87-98, 1996.
[20] A. Zhang, W. Chang, and G. Sheikholeslami, “NetView: Integrating Large-Scale Distributed Visual Databases”, *IEEE Multimedia*, Jul.-Sep., 1998.