Efficient K-NN Search in Polyphonic Music Databases Using a Lower Bounding Mechanism

Ning-Han Liu
Department of Computer Science
National Tsing Hua University
Hsinchu,Taiwan 300, R.O.C
886-3-5715679
nhliou@yahoo.com.tw

Yi-Hung Wu
Department of Computer Science
National Tsing Hua University
Hsinchu,Taiwan 300, R.O.C
886-3-5715131-2847
yihwu@mx.nthu.edu.tw

Arbee L.P. Chen
Department of Computer Science
National Tsing Hua University
Hsinchu,Taiwan 300, R.O.C
886-3-5715131-1065
alpchen@cs.nthu.edu.tw

ABSTRACT
Querying polyphonic music from a large data collection is an interesting and challenging topic. Recently, researchers attempt to provide efficient techniques for content-based retrieval in polyphonic music databases where queries can also be polyphonic. However, most of the techniques do not perform the approximate matching well. In this paper, we present a novel method to efficiently retrieve k music works that contain segments most similar to the user query based on the edit distance. A list-based index structure is first constructed using the feature of the polyphony. A set of candidate approximate answers is then generated for the user query. A lower bounding mechanism is proposed to prune these candidates such that the k answers can be obtained efficiently. The efficiency of the proposed method is evaluated by real data set and synthetic data set, reporting significant improvement over existing approaches in the response time yielded.

Categories and Subject Descriptors
H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing – *indexing methods.* H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval – *search process.*

General Terms
Algorithms, Management, Performance

Keywords
Polyphonic music information retrieval, indexing methods, search process, lower bounded edit distance

MIR'03, November 7, 2003, Berkeley, California, USA.

1. INTRODUCTION
The amount of music data in digital formats has been increasing with the progress of computer and network technologies. To provide content-based retrieval on these music data, a variety of approaches [13] have been proposed, which address the issues including data representation and index structure. Regarding the representation of music data, several approaches [13] have been introduced to model music data based on various features, such as pitch, rhythm, interval, chord, and contour. To deal with the efficiency issue, different techniques [2] have been presented in the literature, including string matching, dynamic programming, n-gram indexing, list-based indexing, and tree-based indexing.

Two types of approaches are commonly used for content-based retrieval on polyphonic data. The first type of approaches is to transform the polyphony into a monophony and then apply the techniques on monophonic data for content-based retrieval. For example, Themefinder [6] represents each polyphonic datum by monophonic themes. Given a query, similar themes are retrieved by using the string matching algorithms. Meldex [9] transforms each polyphonic datum into a monophony based on the highest-pitch approach proposed by Uitdenbogerd and Zobel [16]. The common drawback in this type of approaches lies in its assumption that the polyphonic data can be represented as monophonic data, which is not always true [13].

The second type of approaches is to design specialized algorithms for content-based retrieval on polyphonic data. SEMEX [7] extends the well-known shift-or algorithm [18] for string matching to find all the segments in the polyphonic data, which contain the monophonic query. This system also enables its searching with transposition invariant. To enable polyphonic query processing, PROMS [1] adopts the inverted-file based method to record the information about every note in the polyphonic data. The recorded information includes pitch, bar, and note onset time. It only returns the music segments with exactly the same pitches and onset times of notes as the ones of the query. This system is good for exact matching but ineffective for approximate matching because the onset times of notes still have to match with those of the query. On the other hand, Dovey [3] proposes a dynamic programming based algorithm for processing a polyphonic query. In addition to exact matching, this algorithm further takes approximate matching into consideration. However, due to the limitation of dynamic programming, its performance suffers from both the long length of a music datum and the large number of data in the database. Pickens [14] uses

HMM (Hidden Markov Model) to describe the probability distributions of chords in polyphonic data and polyphonic queries respectively. These probability distributions are represented as the probability matrices. In this way, the distance between a query and a datum can be estimated according to the difference between their probability matrices. Such approach is appropriate for the music with variation. However, the probability distribution of a string can be very different from the probability distribution of its substring. Therefore, the HMM-based approach can only be used for the case where the query and data have almost the same lengths.

Given any two strings, a variety of scoring functions can be used to estimate the similarity degree between them. Given a query string and a scoring function, two approaches are commonly used to select the proper answers, i.e., to prune the improper ones. One is to prune the data whose scores are below a predefined threshold. However, it is difficult to find a threshold that is suitable for different queries. The other is to select only a fixed number of data whose scores are the highest (named the k nearest neighbors search and abbreviated as K-NN search). This approach is more suitable for music retrieval since only a pre-determined number of music works are returned to the users.

The set of notes that begin at the same time is defined as an event [3]. Therefore, a piece of music is regarded as a string of events. Owing to the variety of events, the conventional indexing method such as suffix tree [10, 12] or n-gram [4] may generate a complex and costly index and the K-NN search can be inefficient. Some methods for music reduction [13] have been proposed to reduce the complexity of the index structure by grouping similar events into one. However, a change of the similarity measure for events may result in the reconstruction of the index.

In this paper, we design a query processing method that is efficient to K-NN search given a query and a similarity measure of events. An index structure is constructed which keeps the position of every event in a music datum. Given a query as a string of events, all candidate approximate answers under the user provided similarity measure will be identified from the index structure. A method to prune impossible candidates is then designed such that the K-NN results can be obtained efficiently. From the experimental results, our method performs much better than the previous works [3, 12]. The remainder of this paper is organized as follows. In Section 2, we define a similarity measure for polyphonic data. Section 3 presents the indexing structure and the methods for query processing are introduced in Section 4. Section 5 shows the experiment results and performance analysis. Finally, Section 6 summarizes the contributions of this work with some future research directions.

2. SIMILARITY MEASURE FOR THE POLYPHONY

There are several symbolic representations in digital music, such as MIDI [11] and CHARM [17]. We adopt the MIDI representation because of its popularity. Moreover, we focus on the classical music because most of them are polyphonic music. We determine the pitch scales following the MIDI standard, where the pitch values are non-negative integers smaller than 128. In this way, an event is represented as a set of pitch values and a piece of music is represented by a string of events, where only the ordering of the note onsets is preserved. The representation is often named homophonic reduction [13] that assumes independence between the notes with the overlapping duration. Figure 1 is an example of the above representation for the polyphony. The signs '<' and '>' imply an event and the values between them are the pitch values of the notes. For example, <60> means the event only contains the middle-C note and <64,67> means the event contains both the E note and the G note.

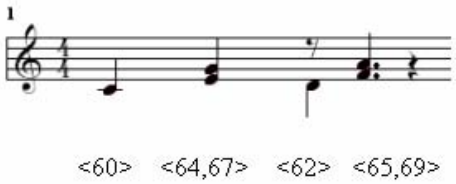


Figure 1: An example of a string of events

To provide the ability of approximate matching, we adopt the *edit distance* based approach to measure the similarity degree between two strings. Based on the definition in [2], there are three types of *local transformations* from string A (denoted as $a_1 \ldots a_m$) to string B (denoted as $b_1 \ldots b_n$) as follows, where a_i and b_i denote a single symbol (or value) and λ means a null character.

- Insertion: $\lambda \rightarrow b_j$
- Deletion: $a_i \rightarrow \lambda$
- Replacement: $a_i \rightarrow b_j$

The edit distance between strings A and B is the minimum number of local transformations required to transform A into B. The cost of the local transformations is defined as follow:

Both the costs of insertion and deletion are set to 1. For the cost of replacement $cost(a,b)$, we define the similarity degree $sim(a,b)$ first, where $0 \leq sim(a,b) \leq 1$. Moreover, we define $g(a,\mu_a)$ as the set of events whose similarity degrees with a are larger than a predefined threshold μ_a. The cost function $cost(a,b)$ for a replacement between a and b is then computed as $1\text{-}sim(a,b)$ if $b \in g(a,\mu_a)$, and 1 otherwise. Notice that since events a and b may have different thresholds, $b \in g(a, \mu_a)$ does not imply $a \in g(b, \mu_b)$ and vice versa. From the cost of replacement as defined above, it implies that an event b matches with an event a when b is similar enough with a. Otherwise, it is considered as an unmatched symbol in A and B, which spends the same cost as insertion and deletion. The unmatched symbols include the symbols of deletion, insertion and replacement with cost 1. The *gap* is defined as the number of consecutive unmatched symbols between two strings. In our method, we also adopt the *gap constraint* δ [3] to limit the maximum length of gaps between two strings such that none of the gaps in the returned answers has a length larger than the gap constraint. In this way, both the edit distance and the gap constraint are used to determine the final answers.

To perform the experiments for our approach, a similarity measure of the events is proposed in the following. This measure is based on the number of pitches in common between the events, which follows the proposed methods in [3, 14]. Moreover, the user can specify the degree of significance to each note in the event based on the user's need. An event is regarded as a vector with 128 dimensions following the MIDI standard. For example,

an event <60, 63> corresponds to the vector with 128 dimensions in which only the dimensions 60 and 63 have values of 1 and the others are 0. Furthermore, for each event *a*, a *weighting function* f_a^i is used to adjust the value of dimension *i* whose original value is 1 to keep the degree of significance for the corresponding pitch with respect to this event. In this way, we can use the parametric cosine function to measure the similarity degree between two events as follows.

$$sim(U,V) = \frac{\sum_{i=1}^{128} f_U^i(u_i) * f_V^i(v_i)}{\sqrt{\sum_{i=1}^{128} f_U^i(u_i)^2} * \sqrt{\sum_{i=1}^{128} f_V^i(v_i)^2}}, \quad \sum_{i=1}^{128} f_U^i(u_i) = 1, \quad \sum_{i=1}^{128} f_V^i(v_i) = 1 \qquad (1)$$

U, *V*: the vectors of events with 128 dimensions
u_i, v_i: the values of dimension *i* in *U*, *V* vectors respectively
$f_U^i(u_i)$, $f_V^i(v_i)$: the weighted functions of *U*, *V* for dimension *i* respectively

From this formula, when two events are the same, the cosine value is 1. When all of the notes in these two events are different, the cosine value is 0. For example, assume there are three events <65,67>, <60,63>, and <60,67>. If the user feels that the highest note in an event plays the most important role in the polyphony, by respectively assigning the higher and lower notes the weights of 70% and 30%, the above formula will compute the similarity degree between <65,67> and <60,67> as 0.845. Similarly, the similarity degree between <60,63> and <60,67> is computed as 0.5. Therefore, it implies that <65,67> is closer to <60,67>. By contrast, if the user assigns the same weights to both notes (i.e., 50%), the similarities will become the same.

3. INDEXING STRUCTURE

Many data structures such as *suffix trees* and *suffix arrays* [2] have been proposed to store the substrings of a string for fast string searching. However, these index structures are not adequate to solve the problem mentioned in this paper, because they do not deal with the k approximate strings searching efficiently. On the other hand, the n-gram based indexing techniques [2] are also not adequate to this problem, because too many types of events in the polyphony will make their performance poor. In our previous work [8], we designed an index structure called the *1D-list* [8] to index music strings. An example is shown in Figure 2(a). Each node in the linked list keeps a pair of information (i : j), which indicates the j-th event of the i-th music in the database. All the events in the database of the same type are linked together in order.

For instance, in Figure 2(a), the 1D-list of event <60> has three nodes, indicating the 7th event of the 1st music, the 1st event of the 2nd and the 9th events of the 3rd music piece, respectively.

Based on the 1D-list, the query processing for exact answer is illustrated by an example as follows. Given a query Q=(<60>, <64,67>, <62>), the three 1D-lists involved in Q are retrieved and two dummy nodes are added as shown in Figure 2(b). We check every pair of nodes in the adjacent two linked lists. For two nodes x (i_x:j_x) and y (i_y:j_y) where x is to the left of y, we build a link from x to y if $i_x=i_y$ and $j_x=j_y-1$. Such a link indicates a segment in a music work that matches a part of Q. In Figure 2(b), four such links are found and drawn as solid bold lines. We then start from the dummy node "start" to traverse these links. If there exists a path to the dummy node "end", it represents an exact answer to Q.

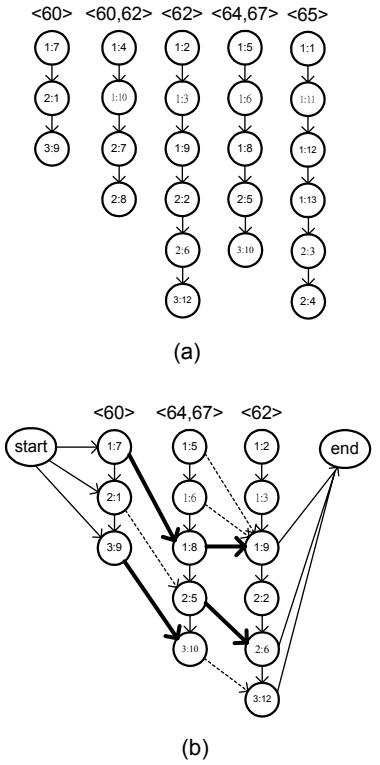


Figure 2: (a) A 1D-list (b) Traversal of the 1D-list

For approximate matching, three kinds of errors, i.e., deletion, insertion and replacement, should be considered. Among them, the processing of deletion and replacement will be discussed in Section 4. In this section, we consider the matched segments with insertion errors. The query processing is similar to the case of exact matching except that extra links from x to y where $i_x=i_y$, $j_x<j_y-1$ and $j_y-j_x-1 \le \delta$ are allowed. The value of j_y-j_x-1 implies the number of insertion errors between x and y. For instance, in Figure 2(b), the dotted lines indicate the extra links. Two paths from "start" to "end," which contain an extra link are constructed to form possible approximate answers, as shown in Figure 2(b). Further comparison with other approximate answers generated from considering deletion and replacement is needed to determine the final K-NN answers.

4. QUERY PROCESSING FOR K-NN ANSWERS

In this section, we introduce our strategy of query processing in detail. The user submits a segment of music as a query that is transformed into a string of events. In addition, the user also specifies the following parameters to control the quality of query results.

1. The number of query results k that the system has to return.

2. The weighting function f_a^i to adjust the value of the i-th dimension in event a.
3. The threshold μ_a for event a to determine the set of events $g(a,\mu_a)$.
4. The gap constraint δ to restrict the number of continuous differences between two strings.

Our approach of query processing is summarized as follows. It will be detailed in Sections 4.1, 4.2, and 4.3.

All the candidate approximate answers (abbreviated as *candidates*) with deletion and replacement errors are first generated from a query based on the above parameters. After the candidates are generated, the system evaluates the distance of each candidate and the query by summing up the costs of deletions and replacements between them. This distance is called the lower-bound distance (abbreviated as *LB distance*) of the candidate. The system then selects the candidate with the minimal *LB* distance to search for possible answers from the 1D-list index structure. The edit distances of the possible answers are computed considering the insertion errors. In the next iteration, the *LB* distances of the remaining candidates are refined and the above process is repeated until none of the *LB* distances of the remaining candidates are less than the maximum of the edit distances of the k possible answers with the minimal edit distances. At this moment, all the answers are returned to the user.

4.1 Generation of Candidates

The generation of all candidates consists of three steps.

Step 1: For each event a in the query, the similarity degree between a and each event in the database, i.e., all the different types of events in the 1D-list, is computed by formula (1) and the events with similarity degrees larger than μ_a are collected as $g(a, \mu_a)$. Note that one event may be involved in more than one group. For example, Figure 3 shows the four groups generated from a query, where each alphabet stands for a distinct event.

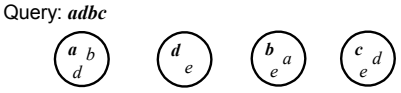


Figure 3: A query and its associated groups

Step 2: A candidate is formed from concatenating the events, each from a distinct group. Considering the groups in Figure 3, the candidates generated includes *adbc*, *adbd*, *adb*, etc. Define a segment in the query, which does not match with the events in a candidate, as a *cut* of the candidate, and the length of the cut the *cut length*. For example, the candidate *adb* has a cut *c*. Notice that any cut length must be smaller or equal to δ since a candidate cannot have a gap larger than δ.

Step 3: For each candidate, we evaluate the initial value of the *LB* distance as follows. Without loss of generality, we represent the query S_q with respect to the candidate $S_1=a_{11}a_{12}...a_{1n}$ as $S_q=C_0a_{q1}C_1a_{q2}C_2...a_{qn}C_n$.

Note that a_{qi} and a_{1i} are events where $a_{1i}\in g(a_{qi},\mu)$, and C_i is a cut of S_1 with respect to S_q. Let $L(C_i)$ be the cut length of C_i. The function $cost(a_{qi},a_{1i})$ as defined in Section 2 stands for the cost for a local transformation from event a_{qi} to a_{1i}. We initialize the *LB* distance (denoted by $LB(S_1,S_q)$) by the following formula:

$$LB\ (S_1,S_q) = \sum_{i=0}^{n} L(C_i) + \sum_{i=1}^{n} cost(a_{qi},a_{1i}) \qquad (2)$$

For the refinement of the *LB* distance, we use an array (named *candidate array*) to store the costs of the unmatched symbols, i.e., deletions, insertions and replacements with cost 1 for the candidate. The value in each cell of the array is denoted as $Cell(a_xa_y)$ where a_xa_y is the *identifying string* of the cell. In the beginning, only the costs of deletions are stored. Figure 4 shows an example where x is an unmatched symbol (deletions):

$S_q=a_{q1}a_{q2}\,x\,a_{q3}\,x\,a_{q4}\,x\,x\,a_{q5}\,x\,x\,a_{q6}\,x\,a_{q7}\,x\,a_{q8}\,x\,a_{q9}$ (query)

$S_1=a_{11}a_{12}\ a_{13}\ a_{14}\ a_{15}\ a_{16}\ a_{17}\ a_{18}\ a_{19}$ (candidate)

0	1	1	2	2	1	1	1

$a_{11}\ a_{12}\ a_{13}\ a_{14}\ a_{15}\ a_{16}\ a_{17}\ a_{18}\ a_{19}$

$$LB(S_1,S_q) = 9 + \sum_{i=1}^{9} cost(a_{qi},a_{1i})$$

Figure 4: An example candidate array

4.2 Computing Edit Distances of Possible Answers

After candidates are generated, the candidate with the minimal *LB* distance is selected to process. Denote this candidate as $S_1=a_{11}a_{12}...a_{1n}$. We use the method of approximate matching described in Section 3 to find the possible answers and compute their edit distances. The possible answers include all the events in S_1 and keep the order of the events, which can be represented as $S_1^{ans}=a_{11}I_{11}a_{12}I_{12}...I_{1n-1}a_{1n}$ where I_{1i} denotes insertion errors. Let $L(I_{1i})$ be the number of the insertions between a_{1i} and a_{1i+1}, which can be obtained by traversing the 1D-list. The edit distance between the possible answer S_1^{ans} and the query S_q is evaluated by the following formula:

$$dist(S_1^{ans},S_q) = L(C_0) + \sum_{i=1}^{n-1} max(L(C_i),L(I_{1i})) + L(C_n) + \sum_{i=1}^{n} cost(a_{qi},a_{1i}) \qquad (3)$$

Because each event in the candidate aligns with an event in the query, the events of a possible answer that appear in the candidate aligns with the same events in the query. Under this alignment of events, the cost of unmatched symbols between any two consecutive events in the candidate is the maximum of the cost on deletions (i.e., $L(C_i)$) and the one on insertions (i.e., $L(I_{1i})$). From formula (2) and (3), we see that $LB(S_1,S_q)\le dist(S_1^{ans},S_q)$.

4.3 Refinement of the *LB* Distances

The query processing terminates when none of the *LB* distances of the remaining candidates is less than the maximum edit distance of the k possible answers with the minimal edit distances. This is because none of the remaining candidates can produce a possible answer better than the current k answers. In this way, the *LB* distance can be used to reduce the generations of possible answers.

Because the initial *LB* distance only considers the costs of deletions and replacements, it is a loose bound to the real minimum edit distance of the possible answers. To reduce the generations of possible answers, the minimum number of the insertions, which can be computed after the traversal of the 1D-list, is used to tighten the bounds of the remaining candidates. Based on the *max* function in formula (3) if we consider the minimal number of the insertions to increase the *LB* distance, the refined *LB* distance is guaranteed to be lower than the edit distance of the associated possible answers.

Data1: *acb* Data2: *addbec*

Event-Pattern	Interval
ab	*1*
bc	*1*
cd	*infinite*
abc	*3*
abcd	*infinite*

Candidate: *abcd*

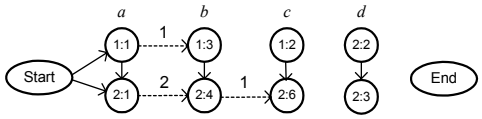


Figure 5: Evaluation of MNIs

During a traversal of the 1D-list, two sets of substrings of the candidate are collected, which are called *event-patterns*. One set consists of all the substrings containing only two events. The other set contains all the substrings that are prefixes of the candidate. For each event-pattern, the minimum number of the insertions (denoted MNI) can be computed. Consider the example shown in Figure 5. For the candidate *abcd*, the 1D-lists of *a*, *b*, *c*, *d* are traversed. After the links are constructed, it can be seen that there are two links connecting list *a* to list *b*. We compute the MNI of *ab* as follows. Since the insertion from node (1:1) to node (1:3) is 1 and from node (2:1) to node (2:4) is 2, the MNI of *ab* is 1. The MNI of an event-pattern implies the lowest cost of the insertions between the query and any string in the database, which contains the event-pattern. Moreover, for the event-patterns with more than two events, which is a prefix of the candidate, we can compute their MNIs in the same way. For instance, the MNI of *abc* is 3 because there is only one such path and the total number of the insertions from event *a* to event *c* through event *b* is 3. In addition, the MNI of an event-pattern is set to infinite if there is no path connecting all the evens in the event-pattern, which implies the candidates containing this pattern are not needed to be further considered.

After the MNIs of the event-patterns are computed, we refine the *LB* distance of the remaining candidates as follows.

We use a heuristic method to refine the *LB* distance using the MNIs. First, for each remaining candidate, an *extra candidate array* that initially is a duplicate of the candidate array is created for recording the refined costs of the unmatched symbols. Moreover, referring to Figure 6(a), we associate with an event-pattern $a_x a_{x+1} \ldots a_{y-1} a_y$, $y \geq x+1$, a single cell named *event-pattern cell* with the value of the cell being the MNI of the event-pattern. Second, the values of the affected cells in the extra candidate array are recomputed, which can be separated into two cases. The first case is that the ends of the event-pattern cell align with the ends of the affected cells as shown in Figure 6(a). If the sum of the values in the affected cells is less than the MNI of the event-pattern, then the event-pattern cell will replace the affected cells. The other case is that only one or none of the ends of the event-pattern cell aligns with the ends of the affected cells as shown in Figure 6(b). The *uncovered part* is defined as the identifying string of the non-overlapping part of the event-pattern cell and the affected cells. For an uncovered part $a_i a_{i+1} \ldots a_{x-1} a_x$, the estimated cost of the unmatched symbols is the sum of the values in $Cell(a_k a_{k+1})$ of the candidate array, where k is from i to x-1. When the sum of the MNI and the estimated costs of the uncovered parts is larger than the sum of the values in the affected cells, the affected cells of the extra candidate array are replaced by the event-pattern cell with the associated MNI, and for the uncovered parts the corresponding cells from the candidate array with their associated values are filled. Moreover, since the values of the cells in the candidate array is reused to compute the estimated costs of the uncovered parts in the extra candidate array, these values have to be kept up to date by the event-patterns with two events. For the event-pattern $a_x a_{x+1}$, if the MNI is larger than the value in $Cell(a_x a_{x+1})$ of the candidate array, it will be replaced with the MNI as shown in Figure 6(c).

After the adjustment of the extra candidate array, the refined *LB* distance can be computed by the values inside the cells.

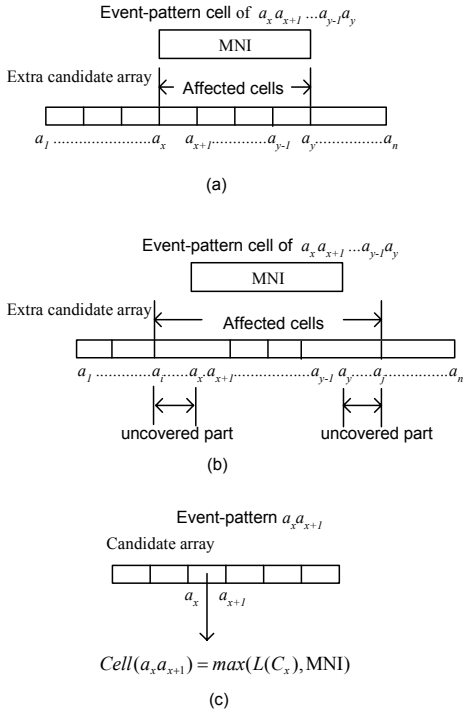


Figure 6 Adjustment of the affected cells for an event-pattern

Consider the query S_q in Figure 4 as an example. The refinement of the *LB* distance is shown in Figure 7(a). Assume there is an event-pattern $a_{16}\ a_{17}\ a_{18}\ a_{19}$ with the MNI=5. The three affected cells are $a_{16}\ a_{17}$, $a_{17}\ a_{18}$ and $a_{18}\ a_{19}$. Because the MNI is larger than 3, which is the sum of the values in the affected cells, we replace these three cells to one and store the MNI, i.e., 5 in this extended cell. As another example shown in Figure 7(b), there is an event-pattern $a_{15}\ a_{16}\ a_{17}$ with the MNI=9. The two affected cells are $a_{15}a_{16}$ and $a_{16}a_{17}a_{18}a_{19}$. Since 11, the sum of the MNI, *Cell*($a_{17}a_{18}$) and *Cell*($a_{18}a_{19}$) is larger than 7, the sum of the values in the affected cells $a_{15}\ a_{16}$ and $a_{16}\ a_{17}\ a_{18}\ a_{19}$, the cells are adjusted and the *LB* distance is refined.

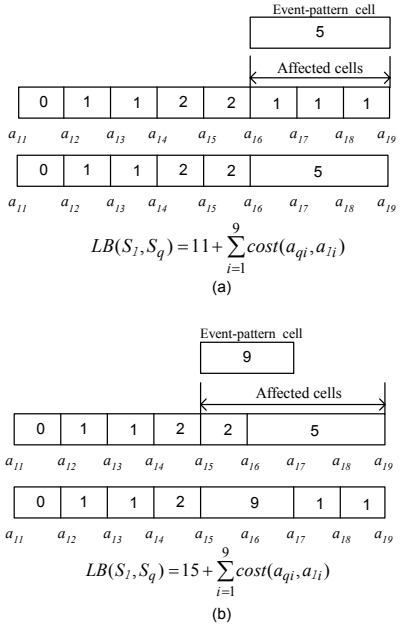


Figure 7 Examples of the *LB* distance refinement

4.4 A Query Processing Example

Assume there are two music works in the database and the 1D-list is constructed as shown in Figure 8(a). A query *aba* is posed with the request of the music work containing the most similar segment, that is, k is set to 1. Suppose $g(a, \mu_a)=\{b\}$ and $g(b, \mu_b)=\{a\}$, the distance between *a* and *b* is 0.4, and the gap constraint is 3. After the generation of candidates, 26 candidates are produced with their *LB* distances shown in Figure 8(b). The candidate *aba* with the minimum *LB* distance is selected and its associated 1D-list traversed for searching the answers, refer to Figure 8(c). At this round of query processing, no answer is found but two event-patterns *ab* and *ba* with their MNIs are recorded. The MNI of the event-pattern *ba* is set to infinite because none of the links constructed satisfy the gap constraint 3. The next step is to refine the *LB* distances of all the candidates by the MNIs of the event-patterns and some of the candidates are dropped out since their *LB* distances are infinite, which is shown in Figure 8(e). We repeat these steps, shown in Figure 8(f), 8(g) and 8(h), until an answer is found which is from position 1 through position 2 to position 3 in data 2 with the edit distance 0.8 as shown in Figure 8(i). The query processing terminates at this point since the *LB* distances of all the remaining candidates are larger than the edit distance of the answer.

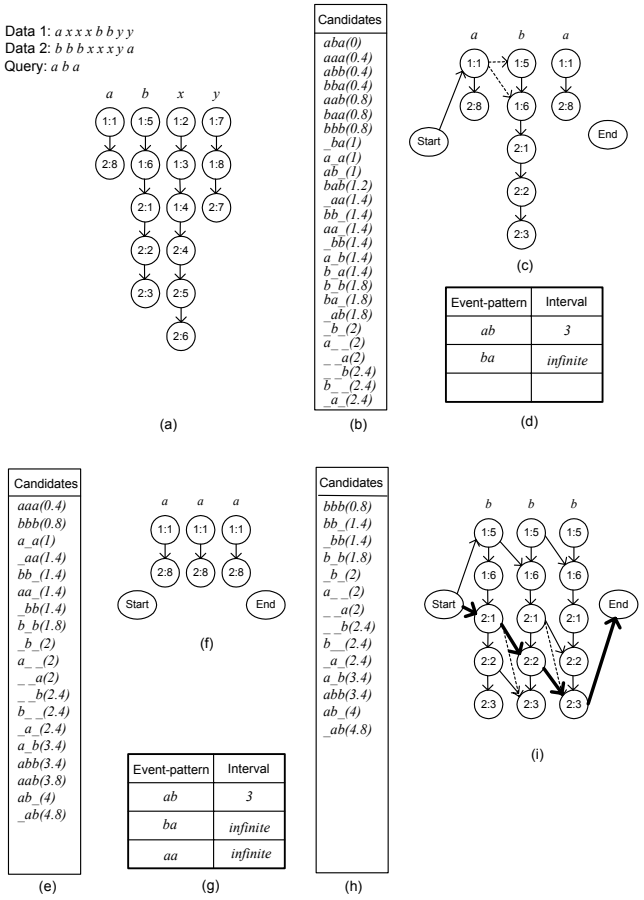


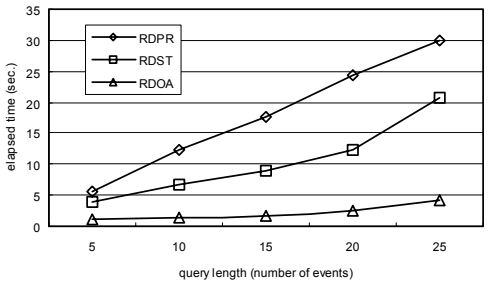
Figure 8: An example of query processing

5. PERFORMANCE ANALYSIS

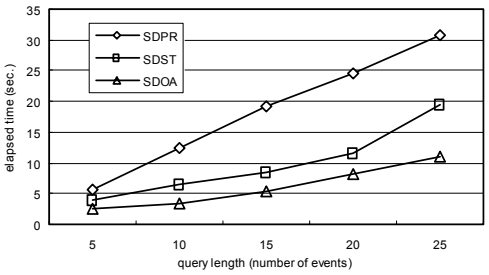
In this section, we show the experiment results about the efficiency of our method. For comparison, we modify a dynamic programming algorithm named *piano roll* [3] by adding the similarity measure as defined in this paper. We also compare our method with the suffix tree method proposed in [12]. In that method, the suffix tree is traversed by using the depth first search and its branches are pruned by the gap constraint to speed up the processing time.

There are two data sets used in the experiments. One is a real data set, which contains 2800 pieces of classical music with average 2954 events in each MIDI file. The other is a synthetic data set, which contains 2800 randomly generated text files with average 3000 events for each file. The parameters k, μ and δ are set to 10, 0.8 and 5, respectively. Moreover, we set f_a^i such that each note in *a* has an equal weight. Figure 9(a) and Figure 9(b) illustrate the

elapsed time versus the query length for the two data sets respectively. In both types of data, the elapsed time of the piano roll algorithm grows linearly with the query length, which follows the property of the dynamic programming algorithm. Moreover, the elapsed time of the suffix tree method grows rapidly. The reason is that because the common prefixes between any two suffixes in our data sets are rare, there are a large number of branches in the suffix tree. When the query length increases, the number of branches to be traversed also increases. Figure 9(a) and Figure 9(b) indicate that our method performs much better than the other methods. Figure 10(a) illustrates the elapsed time versus the threshold μ for the real data set. The parameters k, δ and query length are set to 10, 5 and 15, respectively. Figure 10(b) shows the elapsed time versus different k values for the real data set. Figure 10(a) and Figure 10(b) indicate that the elapsed time of our approach is influenced by the threshold μ and the k value. The reason is that when the threshold μ is looser or the k value is larger, our method will generate more candidates and spend more time to produce the answers. Moreover, the elapsed time of the suffix tree method is also influenced by the two parameters which have impacts on the number of traversed branches.



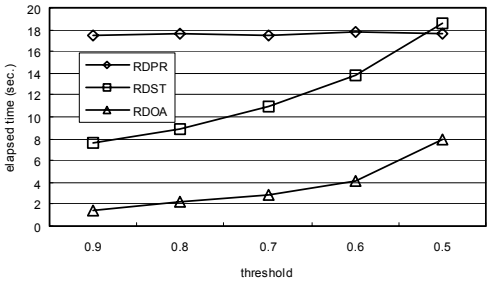
(a) Elapsed time vs. query length (real music data set)



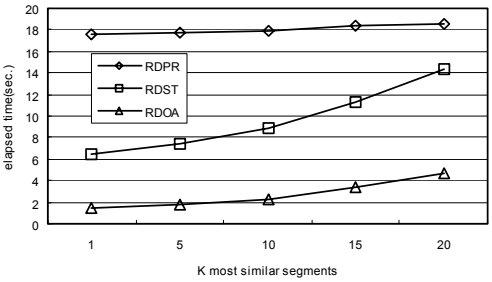
(b) Elapsed time vs. query length (synthetic data set)

RDPR: Real Data with Piano-Roll
RDST: Real Data with Suffix Tree
RDOA: Real Data with Our Approach
SDPR: Synthetic Data with Pinao-Roll
SDST: Synthetic Data with Suffix Tree
SDOA: Synthetic Data with Our Approach

Figure 9: Experiment results of different query lengths



(a) Elapsed time vs. similarity degree threshold



(b) Elapsed time vs. K value

RDPR: Real Data with Piano-Roll
RDST: Real Data with Suffix Tree
RDOA: Real Data with Our Approach
SDPR: Synthetic Data with Pinao-Roll
SDST: Synthetic Data with Suffix Tree
SDOA: Synthetic Data with Our Approach

Figure 10: Experiment results of threshold and K value

6. CONCLUSION

This paper provides a novel approach to support approximate search in the polyphonic music databases. The similarity measure for the polyphony and the efficient methods for indexing and query processing are proposed. In this approach, the candidate approximate answers for a query are generated, followed by the computation of the *LB* distances for efficient K-NN search. According to the experiment results, this approach performs better than the previous works using real data and synthetic data.

The main contribution of this paper is to provide an efficient K-NN search method for the polyphonic music. Unlike the traditional methods which use a predefined distance measure when taking account of the efficiency, our approach provides a lower bounding mechanism to efficiently derive the answers, given a flexibly defined distance measure.

Some important issues of our approach need to be further investigated. First, when the number of the candidates is large, this approach costs a lot to refine the *LB* distances of candidates. To deal with this problem, we will design a new approach to reduce the number of the candidates in the future. A straightforward way is to use an inverted file [15] to associate

each event with a set of music works that contain this event. By this way, we can prune a candidate when no music works contain all the events associated with this candidate. Second, the adjustment of the extra candidate array uses a heuristic method. Better strategies should be considered to tighten the lower bound of the edit distance under a feasible load of the computation. Third, the weighted function in the similarity measure can be enhanced based on musicology. Finally, the effectiveness of our method should be further investigated using the music evaluation platform [5].

7. ACKNOWLEDGEMENTS

This work was partially supported by the MOE Program for Promoting Academic Excellent of Universities under the grant number 89-E-FA04-1-4, and NSC under the contract number 91-2213-E-259-010.

8. REFERENCES

[1] M. Clausen, R. Engelbrecht, D. Meyer and J. Schmitz. Proms: A web-based tool for searching in polyphonic music. In Proceedings of the 1st International Symposium on Music Information Retrieval (ISMIR), 2000.

[2] M. Crochemore and W. Rytter, Text Algorithms, Oxford University Press, 1994.

[3] M. J. Dovey. A technique for "regular expression" style searching in polyphonic music. In Proceedings of the 2nd International Symposium on Music Information Retrieval (ISMIR), 2001.

[4] S. Downie and M. Nelson. Evaluation of a simple and effective music information retrieval method. In Proceedings of ACM Special Interest Group on Information Retrieval (SIGIR), 2000.

[5] J. L. Hsu, A. L. P. Chen, H. C. Chen and N. H. Liu. The Effectiveness Study of Various Music Information Retrieval Approaches. In Proceedings of ACM International Conference on Information and knowledge Management (CIKM), 2002.

[6] A. Kornstadt. Themefinder: A Web-Based Melodic Search Tool. Computing in Musicology 11, MIT Press, 1998.

[7] K. Lemström and S. Perttu. SEMEX – An Efficient Music Retrieval Prototype. In Proceedings of the 1st International Symposium on Music Information Retrieval (ISMIR), 2000.

[8] C. C. Liu, J. L. Hsu and A. L. P. Chen. An Approximate String Matching Algorithm for Content-based Music Data Retrieval. In Proceedings of International Conferences on Multimedia Computing and Systems (ICMCS), 1999.

[9] R. J. MacNab, L. A. Smith, D. Bainbridge and I. H. Witten. The New Zealand Digital Library MELodyinDEX. Digital Library Magazine, May 1997.

[10] E. M. McCreight. A space-economical suffix tree construction algorithm. Journal of the ACM, 23(2): 262—272, 1976.

[11] MIDI Manufaturers Association, Los Angeles, California. The Complete Detailed MIDI 1.0 Specification, 1996.

[12] S. Park, W. W. Chu, J. Yoon and C. Hsu. Efficient Searches for Similar Subsequences of Different Lengths in Sequence Databases. In Proceedings of the International Conference on Data Engineering (ICDE), 2000.

[13] J. Pickens. A Survey of Feature Selection Techniques for Music Information Retrieval. In Proceedings of the 2nd International Symposium on Music Information Retrieval (ISMIR), 2001.

[14] J. Pickens and T. Crawford. Harmonic Models for Polyphonic Music Retrieval. In Proceedings of ACM International Conference on Information and knowledge Management (CIKM), 2002.

[15] G. Salton and M. J. McGill. Introduction to Modern Information Retrieval. Mc Graw Hill. 1983.

[16] A. Uitdenbogerd and J. Zobel. Melodic Matching Techniques for Large Music Databases, In Proceedings of ACM Multimedia, 1999.

[17] G. Wiggins, E. Miranda, A. Smaill and M. Harris. A framework for the evaluation of music representation systems, Computer Music Journal, 17(3): 31--42, 1997.

[18] S. Wu and U. Manber. Fast text searching allowing errors. Communications of the ACM, 35(10): 83--91, October 1992.