Using Multiple Indexes for Efficient Subsequence Matching in Time-Series Databases

Seung-Hwan Lim, Hee-Jin Park, and Sang-Wook Kim

College of Information and Communications, Hanyang University, Korea
{firemoon, hjpark, wook}@hanyang.ac.kr

Abstract. *Time-series subsequence matching* is an operation that searches for such data subsequences whose changing patterns are similar to a query sequence from a time-series database. This paper addresses a performance issue of time-series subsequence matching. First, we quantitatively examine the performance degradation caused by the *window size effect*, and then show that the performance of subsequence matching with a single index is not satisfactory in real applications. We claim that *index interpolation* is a fairly effective tool to resolve this problem. Index interpolation performs subsequence matching by selecting the most appropriate one from multiple indexes built on windows of their distinct sizes. For index interpolation, we need to decide the sizes of windows for multiple indexes to be built. In this paper, we solve the problem of selecting optimal window sizes in the perspective of *physical database design*. For this, given a set of pairs $\langle length, frequency \rangle$ of query sequences to be performed in a target application and a set of window sizes for building multiple indexes, we devise a formula that estimates the overall cost of all the subsequence matchings. By using this formula, we propose an algorithm that determines the optimal window sizes for maximizing the performance of entire subsequence matchings. We formally prove the optimality as well as the effectiveness of the algorithm. Finally, we perform a series of experiments with a real-life stock data set and a large volume of a synthetic data set to show the superiority of our approach.

1 Introduction

Around us, there are a variety of objects such as stock prices, temperature values, and money exchange rates whose values change as time goes by. The list of such changing values sampled at a time interval is called a *data sequence* for the object[1, 2, 7]. For example, a list of temperature values in New York, which were measured at every 1:00 AM during a year, could be a data sequence. Also, a set of data sequences stored in a database is called a *time-series database*[1, 2, 6, 7, 9, 10, 12].

In a time-series database, it is possible to predict future values of an object by analyzing its past values. Let us assume that we have a time-series database consisting of stock price sequences of several companies for past 10 years. We can predict how the stock price of our company will fluctuate next week by referencing to sequences whose changing patterns are similar to that of our

M.L. Lee, K.L. Tan, and V. Wuwongse (Eds.): DASFAA 2006, LNCS 3882, pp. 65–79, 2006.

company's in the last week. *Similar sequence matching* is an operation that finds such sequences whose changing patterns are similar to that of a given *query sequence* from a time-series database[1, 2, 7, 12]. Similar sequence matching is classified into two categories as follows[7, 12]:

(1) *Whole matching*: Given N data sequences $S_1, \ldots, S_N$, a query sequence Q, and a tolerance ϵ, we find such data sequences S_i that are similar to Q. Here, we note that the data and query sequences should be of the same length.
(2) *Subsequence matching*: Given N data sequences $S_1, \ldots, S_N$, a query sequence Q, and a tolerance ϵ, we find all the sequences S_i, one or more subsequences of which are similar to Q, and the offsets in S_i of those subsequences.

Since subsequence matching is a generalization of whole matching, it is applicable to practical applications more than whole matching. In this paper, we focus our attention on subsequence matching.

As a measure for determining the similarity for arbitrary two sequences $X(=[x_0, x_1, \ldots, x_{n-1}])$ and $Y(=[y_0, y_1, \ldots, y_{n-1}])$ of the same length n, the *Euclidean distance* $D(X, Y)$ defined below is widely used as a *basic* similarity measure[1, 4, 5, 7, 8, 13, 14][1]. Two sequences X and Y whose $D(X, Y)$ is below a user-specified tolerance value ϵ are regarded *similar* and are also said to be in ϵ-*match*[12].

$$D(X, Y) = \sqrt{\sum_{i=0}^{n-1} (x_i - y_i)^2} \tag{1}$$

There have been two basic methods proposed in references [7] and [12] for subsequence matching. Following reference[12], we call them *FRM*[7] and *Dual-Match*[12], respectively. Both of them use an index for efficient processing of subsequence matching. Also, they employ the concept of a *window* as an indexing unit. The window is a subsequence of a fixed-size w extracted from query and data sequences. Their common idea for performing subsequence matching is summarized as follows.

For indexing, windows of size w are extracted from every data sequence. Then, each window is transformed into a point in $f(\ll w)$-dimensional space by using the *Discrete Fourier transform*(DFT) or *wavelet transform*. All these points are stored into an R*-tree[3], a multidimensional index structure.

For subsequence matching with a tolerance ϵ, windows of size w are extracted from a query sequence of length $l(\geq w)$, and are transformed into points in f-dimensional space. For each point, a range query of a range $\epsilon / \sqrt{p} (p = \lfloor l / w \rfloor)$ is performed on the R*-tree built in the indexing stage. This process is called an *index searching step*. As a result, candidate subsequences, each of which

[1] In addition to the Euclidean distance, the Manhattan distance, the maximum distance in any pair elements[2], and the time warping distance can be also used as a similarity measure.

has a high possibility to be in ϵ-match with a query sequence, are found. For resolving *false alarms*[1, 7], which are recommended as the candidates but not true answers, each candidate subsequence is accessed from disk, and its actual *Euclidean distance* to the query sequence is computed. This process is called a *post-processing step*.

In both methods, the performance of subsequence matching is highly dependent on the window size. That is, the performance tends to deteriorate as the difference between the size of a window and the length of a query sequence gets larger. This phenomenon is called a *window size effect*[12]. In *FRM* and *Dual-Match*, the size of a window is determined by the minimum among lengths of query sequences to be issued, and subsequence matching performs by using *only one* R*-tree. This approach, however, has a problem that the performance of subsequence matching degrades seriously as the length of a query sequence increases. We can consider building of indexes for all the lengths of query sequences to be issued in a target application. This requires a large cost of maintaining a large number of indexes, thereby being infeasible in real applications.

In this paper, we propose a novel subsequence matching method based on the concept of *index interpolation*[11]. *Index interpolation* constructs multiple indexes on different sizes of windows and processes subsequence matching by selecting the most appropriate one for a given query sequence. Index interpolation is applicable to both *FRM* and *Dual-Match*, and is expected to enhance the performance of subsequence matching significantly.

In index interpolation, more indexes provide better performance, but require a higher cost for their maintenance. This paper mainly focuses on the selection of the sizes for multiple windows that maximize the performance of subsequence matching when the number of indexes is given.

We summarize the contributions of the paper in the following. First, we quantitatively show the performance degradation of subsequence matching due to the window size effect, and then reveal that the overall performance of subsequence matchings using a single index is not satisfactory in real applications. We claim that the concept of index interpolation is quite useful for solving this problem. For subsequence matching by using index interpolation, we need to determine the sizes of windows on which multiple indexes are built. We employ the *physical database design methodology* to select optimal sizes of windows. That is, we devise a formula that estimates the entire cost of performing all the subsequence matchings when there are a set of pairs ⟨length, frequency⟩ of query sequences to be issued and a set of sizes of windows on which indexes are built. By using this formula, we propose an algorithm that decides the optimal sizes of windows that maximize the overall performance of all the subsequence matchings. We also formally verify the optimality and effectiveness of the proposed algorithm. Finally, we show the effect of performance enhancement by the proposed algorithm over previous ones via extensive experiments.

The paper is organized as follows. Section 2 briefly introduces previous methods for subsequence matching, and discusses their advantages and disadvantages. Section 3 presents a result of preliminary experiments that show how the gap

between the length of a query sequence and the size of a window affects the performance of subsequence matching. Section 4 proposes a new method based on index interpolation, and addresses the selection of multiple window sizes for optimizing the performance of subsequence matching. Section 5 verifies the superiority of the proposed method via a series of experiments. Finally, Section 6 summarizes and concludes the paper.

2 Related Work

2.1 FRM

Reference [7] proposed a subsequence matching method that allows data and query sequences of arbitrary lengths. Following reference [12], we call this method *FRM*. *FRM* uses the concept of a *window* of a fixed length for R*-tree indexing.

For indexing, *FRM* extracts *sliding windows* of size w from every possible position inside each data sequence S of length $len(S)(\geq w)$, and then it converts every sliding window into a point in $f(\ll w)$-dimensional space by using *DFT*. The number of points extracted from each data sequence S is $(len(S)-w+1)$. As a result, a large number of points appear in this way, and thus storage overhead for storing these points individually also gets large. For alleviating this problem, *FRM* forms the minimum bounding rectangles(MBR) enclosing multiple points and builds an R*-tree[3] on these *MBR*s instead of points.

For subsequence matching, *FRM* extracts p *disjoint windows* of size w from a query sequence of length $len(Q)(\geq w)$ where $p = \lfloor len(Q)/w \rfloor$, and then converts every disjoint window into a point in f-dimensional space by using *DFT*. For each point, *FRM* performs a range query on an R*-tree by using the point as a center and $\epsilon/\sqrt{p}$ as a range. This *index searching step* finds the points that correspond to the candidate subsequences that are highly likely to be true answers. To discard false alarms, it performs the *post-processing step*; i.e., it accesses all the sequences containing the candidate subsequences from the disk, and computes their *Euclidean* distance to the query sequence. Finally, it returns the final result set containing only the true answers after leaving out the false alarms.

2.2 Dual-Match

In order to reduce storage overhead, *FRM* stores the *MBR*s, each of which encloses multiple points, instead of storing individual points in an R*-tree. Inherently, these *MBR*s have *dead space*[3] inside. This dead space is the primary cause of false alarms, and thus degrades the overall performance of subsequence matching[12]. Moon et al.[12] proposed a method called *Dual-Match* to overcome this problem.

In contrast to *FRM* that locates sliding windows on data sequences and disjoint windows on a query sequence, *Dual-Match* extracts disjoint windows from data sequences and sliding windows from a query sequence. By this simple role exchange, *Dual-Match* reduces the number of points to be stored in the R*-tree

by the ratio of $1/w$. This makes it possible to store individual points themselves rather than *MBR*s in an R*-tree. As a result, *Dual-Match* does not suffer from the problem caused by the dead space inside *MBR*s any longer, and thus achieves considerable performance improvement.

3 Motivation

3.1 Window Size Effect

Given a query sequence, subsequence matching with *FRM* or *Dual-Match* tends to incur more false alarms as the window size gets smaller. For example, the window size for an R*-tree R1 is larger than that for an R*-tree R2. In this case, the candidate set returned by searching R2 would contain extra false alarms, which do not exist in the set returned by searching R1. More false alarms require more time in the post processing step, thereby degrading the overall performance of subsequence matching. This phenomenon is called *window size effect*[12]. Therefore, the choice of a large window in R*-tree construction is so beneficial to efficient processing of subsequence matching.

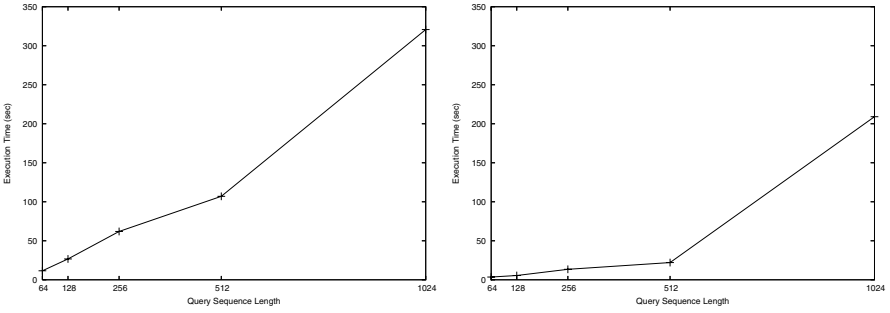
On the other hand, the R*-tree thus built is useless in subsequence matching when its window size is larger than the length of (in *FRM*) and half the length of (in *Dual-Match*) a query sequence[7, 12]. In this case, subsequence matching takes much time since the *sequential scan* should be employed for finding matched subsequences. Therefore, it is crucial to choose a proper size of windows in the indexing stage for efficient processing of subsequence matchings.

Let us denote *minQLen* as the minimum among the lengths of query sequences to be used in a target application. The previous methods determine the window size for indexing as *minQLen* (in *FRM*) or $\lfloor (minQLen + 1)/2 \rfloor$ (in *Dual-Match*). In real applications, however, query sequences of various lengths are issued regardless of the window size employed in indexing. Thus, the performance degradation of subsequence matching becomes fairly serious in case the difference between the length of a query sequence and the size of window is large.

3.2 Preliminary Experiments

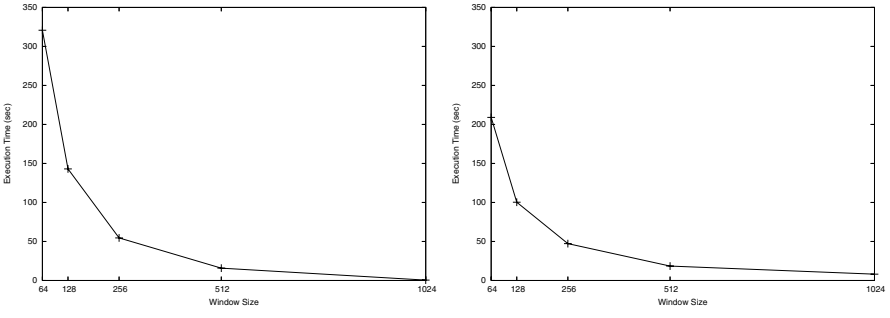
We used 620 Korean stock price sequences of length 1,024 in experiments. Other experiment environments such as hardware and software settings, extraction of windows from data sequences, the lower-dimensional transform, and construction of indexes are the same as those explained in detail in Section 5.

We performed two preliminary experiments. The first experiment used only a single index of the fixed window size and observed the performance tendency of subsequence matching while changing the length of query sequences. The window size was set to $w = 64$ and the lengths of query sequences were set to $Len(Q) =$ 64, 128, 256, 512, and 1,024. The second experiment used query sequences of the fixed length and observed the performance tendency of subsequence matching while changing the window size. The length of query sequences used was $Len(Q)$ = 1,024, and the window sizes were $w = 64$, 128, 256, 512, and 1,024.



(a) *FRM* (b) *Dual-Match*

Fig. 1. Variation of Total Execution Time According to Query Sequence Lengths



(a) *FRM* (b) *Dual-Match*

Fig. 2. Variation of Total Execution Time According to Window Sizes

We used the total execution time of subsequence matchings for all the query sequences as a performance factor. For every query, we adjusted a tolerance ϵ so that 20 subsequences should be returned as a final result set.

Figures 1(a) and 1(b) show the results of the first experiment for *FRM* and *Dual-Match*, respectively. In the figures, the horizontal axes represent the length of query sequences, and the vertical axes the total execution time in the unit of seconds. The results show that the total execution time increases as the query sequence gets longer for both *FRM* and *Dual-Match*. The rationale of the results is that, as the difference between the length of query sequences and the size of windows increases, the number of candidate subsequences obtained from the index searching step also increases due to the window size effect.

Figures 2(a) and 2(b) show the results of the second experiment. The horizontal axes represent the window size, and the vertical axes the total execution time. The results show that the total execution time rapidly decreases as the window size increases for both *FRM* and *Dual-Match*. The rationale of the results is the same as that of the first experiment.

In summary, the performance of subsequence matching dramatically deteriorates as the gap of the length of query sequences and the size of windows

increases. This implies that the performance of subsequence matchings is not satisfactory to users when their processing is done with only a single R*-tree built on windows whose size is determined by considering the minimum length of query sequences as in the prior work.

4 The Proposed Method

In this section, we propose a novel method based on the concept of index interpolation to overcome the performance degradation caused by the window size effect. In index interpolation, we build multiple indexes on windows of different window sizes, and then use an index whose window size is the most appropriate for a given query sequence in subsequence matching.

As the number of indexes increases, the cost for maintaining indexes also increases while subsequence matching performs better. The cost includes not only the storage space for storing indexes but also the time for updating indexes when data is inserted, deleted, or modified. Thus, it is necessary to build as few indexes as possible.

In this section, we consider determining a list of *optimal window sizes* for a given set of pairs ⟨lengths and its frequencies⟩ of query sequences. The lengths and frequencies of query sequences can be easily obtained by analyzing a target application, which is a widely-accepted assumption in physical database design.

4.1 Optimal Window Size

For further presentation, we first introduce some notations and definitions. We denote the length of a query sequence by l_i, $i \geq 1$, and do the list of n query sequence lengths by $\langle l_1, l_2, \ldots, l_n \rangle$ where $l_1 < l_2 < \cdots < l_n$. Similarly, we denote the frequency of a query sequence length l_i by f_i and do accordingly the frequency list of $\langle l_1, l_2, \ldots, l_n \rangle$ by $\langle f_1, f_2, \ldots, f_n \rangle$. We denote a window size by w_i, $i \geq 1$, and do the list of m window sizes by $\langle w_1, w_2, \ldots, w_m \rangle$ where $w_1 < w_2 < \cdots < w_m$. In index interpolation, we perform subsequence matching by selecting the most appropriate one from $\langle w_1, w_2, \ldots, w_m \rangle$ and by using its corresponding index. We call this window size *an optimal one for the length of the query sequence,* l_k, and denote it by $w_{opt}(l_k)$.

We show that the optimal window size $w_{opt}(l_k)$ for a query sequence length l_k in a window list $\langle w_1, w_2, \ldots, w_m \rangle$ is computed by

$$w_p = max\{w_i | w_i \leq l_k \ (1 \leq i \leq m)\} \text{ for } \textit{FRM} \text{ and} \tag{2}$$

$$w_q = max\{w_i | w_i \leq \lfloor (l_k + 1)/2 \rfloor \ (1 \leq i \leq m)\} \text{ for } \textit{Dual-Match} \tag{3}$$

We show that w_p is the optimal window size for *FRM*. (One can show w_q is the optimal window size for *Dual-Match* in a similar way.) We first show any window size in $\langle w_{p+1}, \ldots, w_m \rangle$ cannot be the optimal one for query sequence length l_k. By definition of w_p, windows sizes $w_{p+1}, \ldots, w_m$ are all larger than l_k and the indexes for those window sizes cannot be used in subsequence matching for a

query sequence of length l_k[7]. Thus, in this case, we have to perform sequential scan for a query sequence of length l_k, which shows as poor performance as no indexes are built.

Now, we show that w_p is the optimal window size for l_k among $\langle w_1, \ldots, w_p \rangle$. Every window in $\langle w_1, \ldots, w_p \rangle$ is not larger than l_k and thus an index built for any window size in $\langle w_1, \ldots, w_p \rangle$ can be used for processing a query sequence of length l_k without any concern of false dismissal[7]. According to the window size effect, as a window size is nearer to l_k, the performance becomes better. Therefor, it is hold that w_p is $w_{opt}(l_k)$ the optimal window size for l_k.

4.2 Cost Function

Given a query sequence length l_k and its frequency f_k, the cost of processing a query sequence of length l_k over the list of window sizes $W = \langle w_1, w_2, \ldots, w_m \rangle$, denoted by $C(l_k, f_k, W)$, is defined as follows.

$$C(l_k, f_k, W) = f_k l_k / w_{opt(k)}$$

This cost function is inferred from the observation from our preliminary experiments in Section 3: The cost of subsequence matching was found to be roughly proportional to the query sequence length and to be inversely proportional to the window size.

Now, we extend the cost function to a more general case. Given a list of query sequence lengths $L = \langle l_1, l_2, \ldots, l_n \rangle$ and a list of their frequencies $F = \langle f_1, f_2, \ldots, f_n \rangle$, the processing cost of subsequence matchings using W, denoted by $C(L, F, W)$, is the sum of $C(l_k, f_k, W)$'s for all $1 \leq k \leq n$, i.e.,

$$C(L, F, W) = \sum_{k=1}^{n} f_k l_k / w_{opt(k)}. \tag{4}$$

Also, the cost function for a sublist $L[i..j] = \langle l_i, \ldots, l_j \rangle$ and $F[i..j] = \langle f_i, \ldots, f_j \rangle$ over W is defined analogously.

$$C(L[i..j], F[i..j], W) = \sum_{k=i}^{j} f_k l_k / w_{opt(k)}. \tag{5}$$

4.3 Computing of Optimal Window Size List

In this section, we present an algorithm for determining a list of optimal window sizes W when L and F are given. First, we give a formal definition of W, the optimal window size list.

Definition 1. *For $L = \langle l_1, l_2, \ldots, l_n \rangle$, $F = \langle f_1, f_2, \ldots, f_n \rangle$, and m, a window size list $W = \langle w_1, w_2, \ldots, w_m \rangle$ is considered optimal if and only if $C(L, F, W) \leq C(L, F, W')$ for any window size list W' of length m. The cost $C(L, F, W)$ is called the optimal cost of L and F over the window size lists of length m and also denoted by $O_m(L, F)$.*

We show that every window size in W, the optimal window size list, corresponds to *some* query sequence length l_j in the next lemma. This implies that, for determining W, we just need to consider ${}_nC_m$, instead of ${}_lnC_m$, lists of windows sizes, where C represents *combination*.

Lemma 1. *If a window size list* $\langle w_1, w_2, \ldots, w_m \rangle$ *is an optimal window size list of length* $m(\leq n)$ *for* $L = \langle l_1, l_2, \ldots, l_n \rangle$ *and* $F = \langle f_1, f_2, \ldots, f_n \rangle$, $\langle w_1, w_2, \ldots, w_m \rangle = \langle l_{g(1)}, l_{g(2)}, \ldots, l_{g(m)} \rangle$ *for some* $1 \leq g(1) < g(2) < \cdots < g(m) \leq n$.

Proof. We only need to show that each window size w_i for $1 \leq i \leq m$ is the same as l_j for some $1 \leq j \leq n$. We prove it by contradiction. Assume that w'_i is a window size that is not same as any l_j for $1 \leq j \leq n$. Then, there exist two consecutive query sequence lengths l_{a-1} and l_a satisfying $l_{a-1} < w'_i < l_a$. One can show that the cost of subsequence matchings of L and F by using $\langle w_1, \ldots, w'_i, \ldots, w_m \rangle$ is larger than that by using $\langle w_1, \ldots, l_a, \ldots, w_m \rangle$, which contradicts that $\langle w_1, w_2, \ldots, w_m \rangle$ is optimal. Hence, every window size w_i is the same as some l_j.

Next, we show how to compute the optimal cost $O_m(L, F)$ for $L = \langle l_1, l_2, \ldots, l_n \rangle$ and $F = \langle f_1, f_2, \ldots, f_n \rangle$. We note that we can obtain the optimal window size list of length m as a result of computing the optimal cost $O_m(L, F)$. One can consider a naive approach that computes the costs over all possible window size lists of length m and then gets the minimum of them. However, this approach should compute $O(n^m)$ values, which are too much. In this paper, we present an algorithm that computes the optimal cost $O_m(L, F)$ in $O(mn^2)$ time using *dynamic programming*. The main idea is that we first compute the optimal costs for query sequence sublists and then extend them to get the optimal cost for the whole query sequence list.

The computation of the optimal cost $O_m(L, F)$ consists of two steps. In step 1, we compute an $n \times n$ array NC where each entry $NC(i, j)$ for $1 \leq i \leq j \leq n$ stores $C(L[i..j], F[i..j], \langle l_i \rangle)$, i.e., the cost of the sublists $\langle l_i, \ldots, l_j \rangle$ and $\langle f_i, \ldots, f_j \rangle$ over the window size list $\langle l_i \rangle$. In step 2, we compute the optimal cost $O_m(L, F)$ using the array NC.

Step 1. Compute the array NC: We compute $C(L[i..j], F[i..j], \langle l_i \rangle)$ for each $1 \leq i \leq j \leq n$ and store it into $NC(i, j)$. We show how to compute all $NC(i, j)$'s in $O(n^2)$ time. By equation (5), $NC(i, j) = \sum_{k=i}^{j} f_k l_k / l_i$. Thus, $NC(i, i) = f_i$ and $NC(i, j) = NC(i, j-1) + f_j l_j / l_i$ for all $1 \leq i < j \leq n$, which means that we can compute $NC(i, i)$ in $O(1)$ time, and we also can compute $NC(i, j)$ in $O(1)$ time from $NC(i, j-1)$. Hence, we get the following lemma.

Lemma 2. *We can compute all* $NC(i, j)$*'s for* $1 \leq i \leq j \leq n$ *in* $O(n^2)$ *time.*

Step 2. We compute the optimal cost $O_m(L, F)$: Let $C'(i, j)$ for $1 \leq i \leq n$ and $1 \leq j \leq m$ denote the optimal cost of the sublists $\langle l_i, \ldots, l_n \rangle$ and $\langle f_i, \ldots, f_n \rangle$ over the list of j window sizes whose smallest window size w_1 is l_i. Then, $O_m(L, F) = C'(1, m)$. We show that we can compute $C'(1, m)$ by dynamic programming. We first show the following recurrence is satisfied for $C'(i, j)$ for $1 \leq i \leq n$ and $1 \leq j \leq m$.

Lemma 3. $C'(i,j) = \min_{k=i+1}^{n-j+2}\{NC(i,k-1) + C'(k,j-1)\}$.

Proof. By definition of $C'(i,j)$, the smallest window size w_1 is l_i. Consider the second smallest window size w_2. The window size w_2 can be one of the query sequence lengths $l_{i+1}, \ldots, l_{n-i+1}$ by Lemma 1. If w_2 is l_k, $C'(i,j) = NC(i,k-1)$ $+ C'(k,j-1) \leq NC(i,k'-1) + C'(k',j-1)$ for any $i+1 \leq k' \leq n-i+1$. Hence, the recurrence is satisfied for $C'(i,j)$.

Finally, we show we can compute $C'(1,m)$ in $O(mn^2)$ time. Since we can compute $C'(i,j)$ in $O(n)$ time by Lemma 3, and we compute at most mn values of $C'(i,j)$'s, we can compute $C'(1,m)$ in $O(mn^2)$ time and we get the following lemma.

Lemma 4. *We can compute the optimal cost $O_m(L,F)$ in $O(mn^2)$ time.*

The pseudo-code for computing $C'(1,m)$ is depicted in Figure 3. In lines 1-4, we compute the array NC. In lines 5-10, we initialize some elements of the C' array. In lines 11-15, we compute all the elements of the C' array.

```
1: for i := 1 to n do
2:     NC[i][i] := f_i
3:     for j := i + 1 to n do
4:         NC[i][j] := NC[i][j − 1] + f_j l_j / l_i
5: for i := 1 to n do
6:     C[i][1] := NC[i][n]
7:     for j := 2 to n − i do
8:         C[i][j] := ∞
9:     for j := n − i + 1 to m do
10:        C[i][j] := 0
11: for i := n − 2 downto 1 do
12:    for j := 2 to min{m, n − i} do
13:        for k := i + 1 to n − j + 2 do
14:            temp := NC[i][k − 1] + C[k][j − 1]
15:            if (temp < C[i][j]) C[i][j] := temp
```

Fig. 3. Pseudo-code for computing $C'(1,m)$

5 Performance Evaluation

5.1 Experiment Environment

We used a real-life data set called *K_Stock_Data* and a synthetic data called *Syn_Data*. *K_Stock_Data*, the same one used for our preliminary experiments presented in Section 3, consists of 620 stock price sequences whose length is 1,024. *Syn_Data* is a synthetic data set comprising random walk data sequences s = $<s_1, s_2, \ldots, s_n>$ generated as follows[1].

$$s_{i+1} = s_i + z_i \tag{6}$$

Here, z_i is a random variable that takes an arbitrary value from an interval [-0.1, 0.1] and s_1, the first element of a sequence, is a special value obtained randomly from the interval [1, 10]. For performing our experiments extensively, we generated five sets of *Syn_Data* that comprise 2,000, 3,000, 4,000, and 5,000 data sequences of length 1,024, respectively and another five sets of *Syn_Data* that consist of 1,000 data sequences whose lengths are 2,000, 3,000, 4,000, and 5,000, respectively. On all these data sets, we built R*-trees in the same way as in our preliminary experiments.

Table 1. Number of Query Sequences in Each Group

Number of query sequence groups	Number of query sequences in each group	Sub-total number of query sequences
4	30	120
5	10	50
6	5	30
16	1	16
Total: 31		216

Also, query sequences have their lengths of multiples of 32 in the range [64, 1,024], and each query sequence belongs to a group by its length. The total number of groups is 31. We generated query sequences over groups as in Table 1, which follow the features in real applications. We see that query sequences in four groups frequently appear, and those in 16 groups do not. As a performance factor, we used the average execution time for subsequence matchings with the total of 216 query sequences. We also adjusted a tolerance ϵ so that 20 final answers are returned.

The hardware platform used in our experiments is a 2.8 GHz Pentium 4 PC equipped with 512MB RAM and 9GB hard disk. The software platform is MS Windows 2000 Server. The language used in development is Microsoft Visual C++. We set the size of a page for storing both data and R*-trees to 1KB. For dimensionality reduction, we used the *DFT*, and extracted six features for indexing. Since reference [12] already verified that *Dual-Match* performs much better than *FRM*, we only used *Dual-Match* in our experiments.

We compared the performance of the three methods: (A) *Dual-Match* with only one index (as in the original approach), (B) *Dual-Match* with multiple indexes whose window sizes are evenly chosen in the range of the minimum and maximum query sequence lengths, (C) *Dual-Match* with multiple indexes whose window sizes are chosen by our approach as shown in Section 4. Hereafter, we shortly call them methods (A), (B), and (C), respectively.

5.2 Results and Analyses

We ran three types of experiments for performance evaluation. In Experiment 1, we compared the performance of the three methods (A), (B), and (C) using

K_Stock_Data with different numbers of indexes. In Experiment 2 and Experiment 3, we compared the performance of the three methods using *Syn_Data* while changing the number and the length of data sequences, respectively.

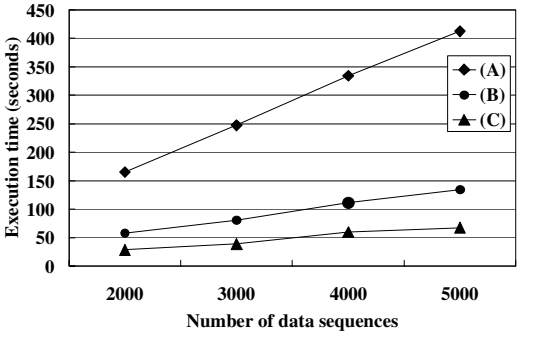


Fig. 4. Performance with Different Numbers of Data Sequences

Figure 4 shows the results of Experiment 1. The horizontal axis represents the number of R*-trees employed for subsequence matchings, and the vertical axis does the average execution time in the unit of seconds. As shown in the figure, method (A) showed the worst performance, and method (C) showed the best performance. When using five R*-trees, method (C) performs 7.8 times and 1.5 times better than methods (A) and (C), respectively. Also, it showed performance 5.6 times and 3.2 times better than methods (A) and (C), respectively. The performance gain tends to get larger with a smaller number of R*-trees employed in methods (B) and (C).

In Experiment 2, we examined the performance tendency of the three methods while changing the number of data sequences of length 1,024 to 2,000, 3,000, 4,000, and 5,000. We built four R*-trees for methods (B) and (C).

Figure 5 shows the results of Experiment 2. The horizontal axis represents the number of data sequences, and the vertical axis does the average execution time.

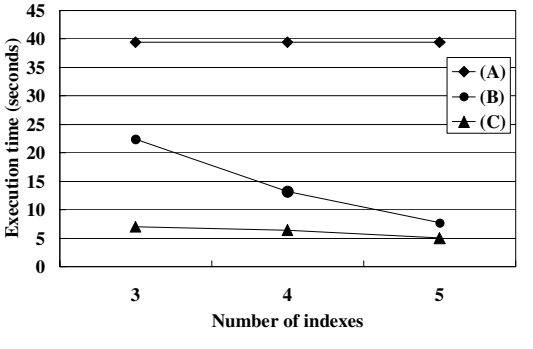


Fig. 5. Performance with Different Numbers of Indexes

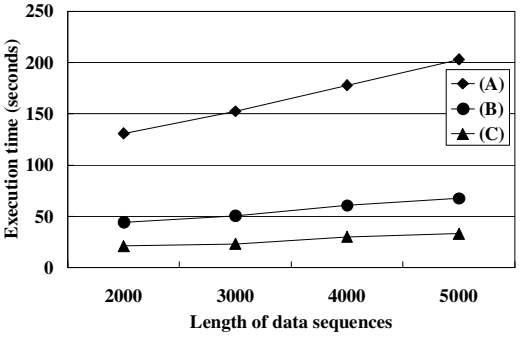


Fig. 6. Performance with Different Lengths of Data Sequences

In all cases, the performance of method (C) was shown to be better than that of method (B), which was shown to have better performance than method (A). Also, the performance gain of method (C) was shown to be about 5.7 times to 6.1 times when compared with method (A), and about 1.9 to 2.1 when compared with method (B).

In Experiment 3, we investigated the performance change of the three methods with 1,000 synthetic data sequences of length 2,000, 3,000, 4,000, and 5,000. As in Experiment 2, we employed four R*-trees for methods (B) and (C).

Figure 6 shows the results of Experiment 3. The horizontal axis represents the length of data sequences, and the vertical axis does the average execution time. The results appeared to be quite similar to that of Experiment 2. Regardless of the length of data sequences, method (C) showed the best performance, and achieved significant speedup about 6.2 times and 2.0 times over methods (A) and (B), respectively.

In summary, by employing the concept of index interpolation, we could improve the performance of subsequence matching significantly compared with the prior approach that uses only a single R*-tree. Also, our method for selecting the optimal window sizes for multiple R*-trees was shown to be fairly effective when compared with the simple one that chooses window sizes from even positions within a possible range.

6 Conclusions

In this paper, we have proposed a novel method for time-series subsequence matching based on index interpolation[11] that resolves the performance degradation caused by the window size effect.

The main contributions can be summarized as follows.

(1) Via preliminary experiments, we have first verified that the performance of subsequence matching by previous methods that employ only one R*-tree is not satisfactory to users. Then, we have claimed that index interpolation is a good choice to resolve this performance problem.

(2) We have derived a formula that estimates the cost for all the subsequence matchings when a set of pairs ⟨length, frequency⟩ of query sequences to be issued and a set of windows sizes for the R*-tree building are provided.
(3) Using the cost formula, we have proposed an efficient algorithm that determines an optimal set of window sizes that maximize the overall performance of all the subsequence matchings performed in a target application. We have formally shown the optimality and effectiveness of the proposed algorithm.
(4) We have quantitatively verified the effect of performance improvement obtained from the proposed method through a series of experiments.

The results reveal that the proposed approach outperforms the previous one up to 7.8 times. Currently, the proposed method provides the optimal list of window sizes, but not the optimal number of indexes. As a future study, we are considering tackling this issue by reflecting the update costs as well as subsequence matching costs.

Acknowledgement

This research was supported by the MIC (Ministry of Information and Communication), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Assessment) (IITA–2005–C1090–0502–0009).

References

1. R. Agrawal, C. Faloutsos, and A. Swami: Efficient Similarity Search in Sequence DataBases, In Proc. Int'l Conf. on Foundations of Data Organization and Algorithms (FODO), pp. 69-84, Chicago, Illinois, Oct. 1993.
2. R. Agrawal et al: Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Database, In Proc. Int'l Conf. on Very Large Data Bases (VLDB), pp. 490-501, Zurich, Switzerland, Sept. 1995.
3. N. Beckmann et al: The R*-tree: An efficient and Robust Access Method for Points and Rectangles, In Proc. Int'l Conf. on Management of Data, ACM SIGMOD, pp. 322-331, Atlantic City, New Jersey, May 1990.
4. K. P. Chan and A. W. C. Fu: Efficient Time Series Matching by Wavelets, In Proc. Int'l Conf. on Data Engineering (ICDE), IEEE, pp. 126-133, Sydney, Australia, Mar. 1999.
5. K. K. W. Chu and M. H. Wong: Fast Time-Series Searching with Scaling and Shifting, In Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), ACM, pp. 237-248, Philadelphia, Pennsylvania, May 1999.
6. T. Argyros, C. Ermopoulos: Efficient Subsequence Matching in Time Series Databases Under Time and Amplitude Transformations, ICDM, 2003
7. C. Faloutsos et al: Fast Subsequence Matching in Time-Series Databases, In Proc. Int'l Conf. on Management of Data, ACM SIGMOD, pp. 419-429, Minneapolis, Minnesota, May 1994.

8. D. Q. Goldin and P. C. Kanellakis: On Similarity Queries for Time-Series Data: Constraint Specification and Implementation, In Proc. Int'l Conf. on Principles and Practice of Constraint Programming, pp. 137-153, Cassis, France, Sept. 1995.
9. T. Kahveci, K. S. Ambuj: Variable Length Queries for Time Series Data, In Proc. Int'l Conf. on Data Engineering, 2001
10. T. Kahveci, K. S. Ambuj: Optimizing Similarity Search for Arbitrary Length Time Series Queries, IEEE Trans. Knowl. Data Eng. Vol. 16, No. 4, 418-433, 2004
11. W. K. Loh, S. W. Kim and K. Y. Whang: A Subsequence Matching Algorithm that Supports Normalization Transform in Time-Series Databases, Data Mining and Knowledge Discovery Journal, Vol. 9, No. 1, pp. 5-28, Jul. 2004.
12. Y. S. Moon et al: Duality-Based Subsequence Matching in Time-Series Databases, In Proc. Int'l Conf. on Data Engineering (ICDE), IEEE, pp. 263-272, Heidelberg, Germany, Apr. 2001.
13. D. Rafiei and A. Mendelzon: Similarity-based Queries for Time-Series Data, In Proc. Int'l Conf. on Management of Data, ACM SIGMOD, pp. 13-24, Tucson, Arizona, June 1997.
14. D. Rafiei: On Similarity-Based Queries for Time Series Data, In Proc. Int'l Conf. on Data Engineering (ICDE), IEEE, pp. 410-417, Sydney, Australia, Mar. 1999.
15. Weber, R. et al: A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces, In Proc. Int'l Conf. on Very Large Data Bases (VLDB), pp. 194-205, New York, Aug. 1998.