Available online at www.sciencedirect.com
ScienceDirect
Information Sciences 177 (2007) 5691–5706
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

Using multiple indexes for efficient subsequence matching in time-series databases

Seung-Hwan Lim, Heejin Park, Sang-Wook Kim *

College of Information and Communications, Hanyang University, Republic of Korea

Received 14 September 2005; received in revised form 21 June 2007; accepted 3 July 2007

Abstract

A *time-series database* is a set of data sequences, each of which is a list of changing values of an object in a given period of time. *Subsequence matching* is an operation that searches for such data subsequences whose changing patterns are similar to a query sequence in 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 solve 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 from the perspective of *physical database design*. 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 performed in a target application. 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 show the superiority of our approach by performing extensive experiments with a real-life stock data set and a large volume of synthetic data sets.

Keywords: Time-series databases; Subsequence matching; Index interpolation; Physical database design

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 [1,2,7,10,13]. For example, a list of temperature values in New York, which have

* Corresponding author.
E-mail addresses: firemoon@ihanyang.ac.kr (S.-H. Lim), hjpark@hanyang.ac.kr (H. Park), wook@hanyang.ac.kr (S.-W. Kim).

0020-0255/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2007.07.004

been 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* [1,2,13,22].

In a time-series database, it could be possible to predict future values of an object by analyzing its past values. 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 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 [1,2,13,22]. It is of growing importance in many new applications such as data mining and data warehousing [3,11,12,14,16,19,23,25,28,31].

Similar sequence matching is classified into two categories as follows [13,22]:

(1) *Whole matching*: Given N data sequences $S_1, \ldots, S_N$, and a query sequence Q. 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.[1]
(2) *Subsequence matching*: Given N data sequences $S_1, \ldots, S_N$, and a query sequence Q. We find all the sequences S_i, one or more subsequences of which are similar to Q, and the offsets within S_i of those subsequences. Here, the data and query sequences are allowed to be of arbitrary lengths.

Since subsequence matching is more applicable to practical applications than whole matching, in this paper, we focus our attention on subsequence matching.

As a *basic* measure for determining the similarity for two arbitrary 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 [1,8,9,13,15,18,25,26,21].[2] Two sequences X and Y whose $D(X, Y)$ is below a user-specified tolerance ϵ are regarded *similar* and are also said to be in ϵ-*match* [22]:

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

There have been two basic methods proposed in Refs. [13,22] for subsequence matching. Following Ref. [22], we call them FRM [13] and *Dual-Match* [22], 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 [4], a multidimensional index structure.

For subsequence matching with a tolerance ϵ, windows of size w are extracted from a query sequence of length $l(\geqslant w)$, 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 is likely to be in ϵ-match with a query sequence, are found. For resolving *false alarms* [1,13], which are recommended as candidate subsequences 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* [22], and will be explained in more detail in Section 3. 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, which

[1] This is because most approaches employ the Euclidean distance as a basic similarity measure.
[2] In addition to the Euclidean distance, the Manhattan distance [29], the maximum distance in any pair elements [2], and the time warping distance [5,17,24,30] can be also used as a similarity measure.

has been built on the windows of the size thus determined. This approach, however, has a problem that the performance of subsequence matching degrades seriously as the length of a query sequence get larger. We can consider building of indexes for all the lengths of query sequences to be issued in a target application. However, 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* [20,21]. *Index interpolation* constructs multiple indexes on windows of different sizes 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 by solving the performance problem caused by the window size effect.

In index interpolation, more indexes provide better performance, but require a higher cost for their maintenance. Here, the cost includes the space overhead for storing those indexes and the update overhead for making those indexes consistent when sequences in a database are inserted, deleted, and modified. This paper addresses how to choose 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 caused by 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 choose the 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 $\langle \text{length}, \text{frequency} \rangle$ 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 formally prove the optimality and the 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 shows 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

This section reviews previous work related to similar sequence matching. We first discuss the characteristics of prior methods proposed for efficient processing of similar sequence matching based on the Euclidean distance, and then briefly summarize other research efforts related to similar sequence matching.

2.1. Whole matching

The first solution to similar sequence matching was introduced in Ref. [1], which assumed that all the data and query sequences are of the same length. In the indexing stage, each data sequence of length l in a time domain is transformed into the one in a frequency domain by using DFT, and the first $f(\ll l)$ features are extracted. As a result, every sequence is mapped into a point in f-dimensional space. For indexing a large number of such points, an R^*-tree [4] is used. With this dimensionality reduction from l to f, the problem of the *dimensionality curse* [5,6,27] can be successfully avoided. The performance issue related to choosing f was also discussed in [1].

In the sequence matching stage, a query sequence of length l is transformed into a point over f-dimensional space in the same way, and a range query is formed with the point as a center and a tolerance ϵ as a range. Next, the R^*-tree built in the indexing stage is traversed to perform a range query. As a result of performing

this *index searching step*, candidate sequences, whose corresponding points in f-dimensional space are in ϵ-match with that of the query sequence, are found.

A sequence matching method is said to incur *false dismissals* when it misses some true answers in the final result set. By using the *Parseval theorem*, Ref. [1] proved that this method does guarantee no false dismissals. The index searching step, however, incurs *false alarms*, which are just candidates but not true answers, because it uses only $f(\ll l)$ features. Thus, for eliminating such false alarms, the *post-processing step* is performed. For this, every candidate sequence is accessed from disk, and its actual *Euclidean distance* to the query sequence is computed. If the distance is larger than ϵ, the candidate sequence is regarded as a false alarm, and thus is discarded from the final result set.

2.2. FRM

Ref. [13] proposed a subsequence matching method that allows data and query sequences of arbitrary lengths by extending the method proposed in [1]. Following Ref. [22], 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 $\text{len}(S)(\geqslant w)$, and then 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 the 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 [4] on these MBRs instead of points.

For subsequence matching, FRM extracts p *disjoint windows* of size w from a query sequence of length $len(Q)(\geqslant w)$ 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 contained in the final result set. 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 after leaving out the false alarms. Ref. [13] proved that FRM does not incur any false dismissals.

2.3. Dual-match

In order to reduce storage overhead, FRM stores the MBRs, each of which encloses multiple points, instead of storing individual points in an R^*-tree. Inherently, these MBRs have *dead space* [4] inside. This dead space is the primary cause of false alarms, and thus degrades the overall performance of subsequence matching [22]. Moon et al. proposed a method called *Dual-Match* to overcome this problem [22].

For indexing, *Dual-Match* divides every data sequence S of length $len(S)$ into *disjoint windows* of length w, and then converts each disjoint window into a *data window point* in $f(\ll w)$-dimensional space by using DFT. The number of data window points extracted from each data sequence S is approximately $len(S)/w$. Thus, *Dual-Match* reduces the number of data window points to be stored in the R^*-tree by approximately the ratio of $1/w$, compared with FRM. This makes it possible to store individual data window points themselves rather than MBRs in an R^*-tree. As a result, *Dual-Match* no longer suffers from the dead space inside MBRs, and thus achieves considerable performance improvement in subsequence matching.

For subsequence matching, *Dual-Match* extracts *sliding windows* of length w from a query sequence of length $len(Q)$, and then converts every sliding window into a *query window point* in f-dimensional space by using DFT. Next, *Dual-Match* forms a *query MBR* containing all those query window points. It also maintains those points themselves together with the query MBR. Next, it performs a range query on an R^*-tree by using the query MBR enlarged with $\epsilon/\sqrt{p}$ where $p = \lfloor (len(Q) + 1)/w \rfloor - 1$. This *index searching step* constructs a candidate set by comparing each query window point in the query MBR and each point obtained from the range query, thus discarding the false alarms caused by using the query MBR. Finally, it performs the post-processing step that discards all the remaining false alarms from the candidate set, and returns the final result set. Ref. [22] proved that *Dual-Match* guarantees no false dismissals in subsequence matching.

2.4. Other previous work

Most previous approaches [1,8,13,18,22,23] in early work employed the Euclidean distance between two sequences of the same length as a similarity measure. Kim et al. [18] proposed a method that optimizes the R^*-tree searches in time-series subsequence matching by using the concept of *window-join*. Chan and Fu [8] provided in-depth discussions on the application of the discrete wavelet transform instead of the discrete Fourier transform to subsequence matching. Yi and Faloutsos [29] suggested an indexing method for fast subsequence matching that allows arbitrary L_p norms. Moon et al. [23] proposed a subsequence matching method called *GeneralMatch* that generalizes *Dual-Match* in constructing windows for indexing and querying.

Time-series subsequence matching often fails to search for the data sequences that are actually similar to a query sequence in the users' perspective when employing only the Euclidean distance as a similarity measure. Therefore, later work on similar sequence matching tends to support various types of transformations such as scaling, shifting, normalization, moving average, time warping, and their combination.

Normalization is a transformation that enables finding sequences with similar fluctuation patterns even though they are not close to each other before the transformation. Das et al. [10] presented a similarity model based on normalization that takes into account outliers, shifting, and scaling. They also proposed statistical approximation techniques to speed-up the finding of similar sequences. Goldin and Kanellakis [15] suggested an efficient whole matching method that supports scaling and shifting by using on the normalization transformation. Agrawal et al. [2] introduced a similarity model that supports scaling, shifting, and noise, and also presented a method that finds all pairs of similar subsequences efficiently from a time-series database. Chu and Wong [9] defined a similarity model that considers scaling and shifting, and proposed an effective indexing mechanism for fast time-series subsequence matching. Loh et al. [21] presented an efficient subsequence matching method that supports normalization transform by using the concept of *index interpolation*.

Moving average is a transformation that is useful for finding the trend of time-series by reducing the effect of noise inside. Refs. [25,26] proposed whole sequence matching methods that support the moving average transformation of arbitrary order using only one index. Ref. [20] extended those methods, and suggested a subsequence matching method supporting a moving average transformation of arbitrary order by using multiple indexes.

Time warping is a transformation that allows any sequence element to replicate itself as many times as necessary and thus enables sequences with similar patterns to be found even when they are of different lengths. Berndt and Clifford [5] introduced a method for finding similar time warped sequences by using *dynamic programming*. Yi et al. [30] proposed a filtering mechanism that reduces the CPU processing time. They also proposed a method for approximate indexing of time warping that uses the *FastMap* technique. Refs. [24,18,16] dealt with approaches for exact indexing of time warping for efficient similar sequence matching.

Zhou and Wong [31] proposed a segment-wise time warping method for time scaling searching. Fu et al. [14] introduced a technique that handles both time warping and uniform scaling simultaneously. Wu et al. [28] introduced a solution for tumor respiratory motion analysis, clustering, and online prediction by subsequence matching over tumor motion data in cancer radiation treatment. Kim et al. [19] defined a *shape-based similarity model* that combines normalization, moving average, and time warping together, and proposed an efficient algorithm for subsequence matching based on this model.

3. Motivation

This section shows how the difference between the length of a query sequence and the size of a window affects the performance of subsequence matching. In Section 3.1, we first explain the concept of a *window size effect*. In Section 3.2, we examine the execution times of subsequence matching by previous methods while changing the length of query sequences and the size of windows.

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 candidate 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* [22]. 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) or half the length of (in *Dual-Match*) a query sequence [13,22]. In this case, subsequence matching takes much time since the *sequential scan* should be employed for finding matched subsequences. Therefore, it is important to choose a correct 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 $\min QLen$ (in FRM) or $\lfloor (\min QLen + 1)/2 \rfloor$ (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 a window is large.

In the next subsection, via a series of experiments, we quantitatively examine the window size effect on the performance of subsequence matching. Also, based on the performance result, we propose a method to accelerate subsequence matching by overcoming the performance degradation caused by the window size effect.

3.2. Preliminary experiments

We used 620 Korean stock price sequences of length 1024 in experiments. To generate query sequences, we extracted subsequences from the random positions of the arbitrarily chosen data sequences, and then added a random positive or negative value obtained from a small range to each element value of the subsequences. Other settings on experiment environment are the same as those explained in detail in Section 5.

We performed two preliminary experiments. The first experiment used only a single index on windows of the fixed size and observed the performance tendency of subsequence matching while changing the length of query sequences. The window size was set to 64 and the length of query sequences was set to 64, 128, 256, 384, 512, 640, 768, 896, and 1024. In order to exclude other factors that affect the performance, we extracted query sequences of different lengths from the same sequence. More specifically, we used query sequences of lengths 64, 128, 256, 384, 512, 640, 768, and 896 extracted from a query sequence of length 1024. 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 set to 1024, and the window size was set to 64, 128, 256, 384, 512, 640, 768, 896, and 1024.

As a performance factor, we used the execution time averaged over 50 subsequence matchings with different query sequences of the same length. We chose a tolerance that makes subsequence matching with a query sequence of length 1024 produce 3 true answers, and then used the tolerance for query sequences of all other lengths in the experiments.

Fig. 1(a) and (b) shows 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 average execution time in the unit of seconds. For both FRM and *Dual-Match*, the results show that the average execution time increases as the query sequence gets longer. 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.

Fig. 2(a) and (b) shows the results of the second experiment. The horizontal axes represent the window size, and the vertical axes the total execution time. For both FRM and *Dual-Match*, the results show that the total execution time rapidly decreases as the window size increases. The rationale of the results is the same as that of the first experiment: As the difference between the length of query sequences and the size of windows decreases, the number of candidate subsequences obtained from the index searching step also decreases due to the window size effect.

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

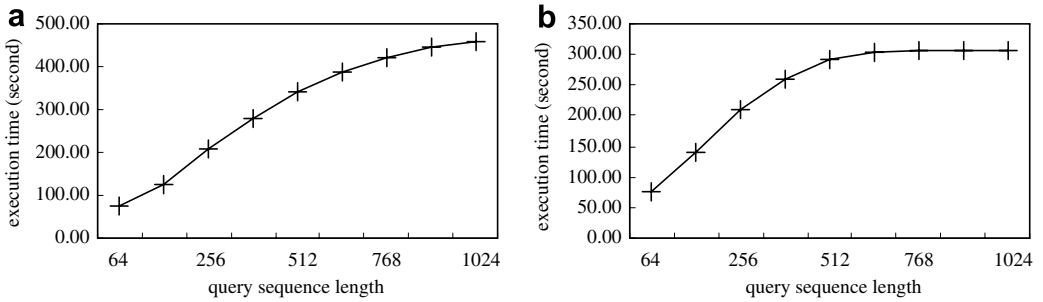

Fig. 1. Performance with different query sequence lengths: (a) FRM; (b) Dual-Match.

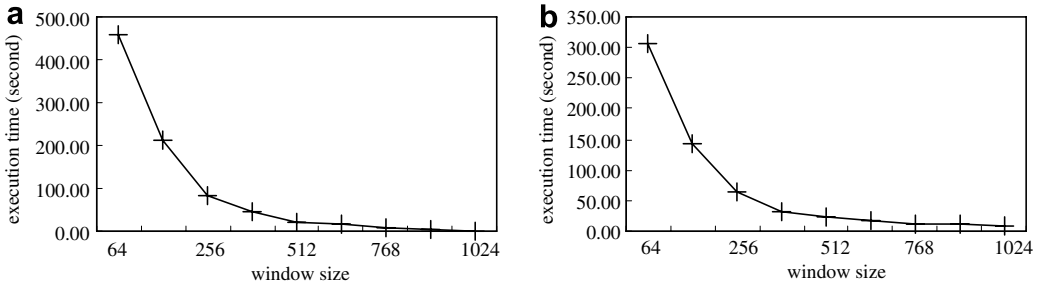

Fig. 2. Performance with different window sizes: (a) FRM; (b) *Dual-Match*.

matchings is not satisfactory to users when their processing is performed with only a single R^*-tree built on windows whose size is set to the minimum length of query sequences as in 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 faster. The cost includes not only the storage space for storing indexes but also the time for updating indexes when a data sequence is inserted, deleted, or modified. Thus, it is important to build as few indexes as possible for minimizing the cost for maintenance.

In this section, we consider determining a list of *optimal window sizes* for a given set of pairs ⟨length and its frequency⟩ of query sequences.[3] We first discuss how to select the optimal one from a given list of window sizes for subsequence matching of a query sequence in Section 4.1. In Section 4.2, we propose a cost function that estimates the overall cost of performing all the subsequence matchings when there are given lengths of query sequences, their frequencies, and a list of window sizes. In Section 4.3, we give an algorithm that computes a list of optimal window sizes that minimize the cost estimated by the proposed cost function for a given set of pairs ⟨length, its frequency⟩ of query sequences to appear in a target application.

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, $i \geqslant 1$, and do the list of n query sequence lengths by $\langle l_1, l_2, \ldots, l_n \rangle$ where $l_1 < l_2 < \cdots < l_n$.

[3] 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.

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, $i \geqslant 1$, 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 $w_{\text{opt}}(l_k)$.

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

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

and

$$w_q = \max\{w_i | w_i \leqslant \lfloor (l_k + 1)/2 \rfloor \ (1 \leqslant i \leqslant m)\} \quad \text{forDual-Match} \tag{3}$$

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 [13]. 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 dismissals [13]. According to the window size effect, as a window size is nearer to l_k, the performance becomes better. Therefore, it is hold that, w_p is $w_{\text{opt}}(l_k)$, 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:

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

where α is some positive constant. 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 (as shown in Fig. 1) and to be inversely proportional to the window size (as shown in Fig. 2). It should be noted that the purpose of the cost function is not to estimate the performance of our index interpolation method but to find the optimal window size list. For this purpose, the value of α in the cost function is not important because the optimal window size list optimizing the cost function for some α does also optimize the cost functions with other α's. Therefore, without loss of generality, we can assume that $\alpha = 1$.

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 $1 \leqslant k \leqslant n$, i.e.,

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

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:

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

4.3. Computing optimal window size list

In this section, we present an algorithm for determining an *optimal window size list* when L and F are given. We first give a formal definition of an 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 $C(L, F, W) \leqslant C(L, F, W')$ 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 is also denoted by $O_m(L, F)$.

The next lemma shows that, in an optimal window size list $W = \langle w_1, w_2, \ldots, w_m \rangle$, every w_i in W corresponds to some l_j in L. This implies that, in determining an optimal window size list, we only need to consider $\binom{n}{m}$ rather than $\binom{l_n}{m}$ window size lists.

Lemma 1. *If $W = \langle w_1, w_2, \ldots, w_m \rangle$ is an optimal window size list 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 $1 \leqslant g(1) < g(2) < \cdots < g(m) \leqslant n$.*

Proof. We prove this by showing that $W = \langle w_1, w_2, \ldots, w_m \rangle$ cannot be an optimal window size list for L and F if it has some window sizes that are not the same as any l_j for $1 \leqslant j \leqslant n$. Let w_a be a window size that is not the same as any l_j. Let l_b denote the smallest query length larger than w_a. If $w_a > l_n$, l_b is not defined.

If $w_a > l_n$ or $l_b = w_{a+1}$, consider a window size list W' obtained by removing w_a from W and inserting a new window w' such that $w' = l_j$ for some $1 \leqslant j \leqslant n$ and $w' \neq w_i$ for all $1 \leqslant i \leqslant m$. Since $m \leqslant n$, such w' always exists. Then, the cost using W' is always smaller than the cost using W because removing w_a does not change the cost and inserting w' surely decreases the cost. Thus, W cannot be an optimal window size list in this case. For the other case ($w_a \leqslant l_n$ and $l_b \neq w_{a+1}$), consider a window size list W' that is obtained by replacing w_a by l_b. The cost using W' is always smaller than the cost using W because, by the definition of the cost function, when w_a is replaced by l_b, the costs for queries of length l_k for $k < b$ remain the same, those of length l_b are reduced, and those of length l_k for $k > b$ either remain the same or are reduced. Hence, W cannot be an optimal window size list in this case and overall. □

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 $F = \langle f_1, f_2, \ldots, f_n \rangle$. Since we can obtain the optimal window size list of length m during computing the optimal cost $O_m(L, F)$, we only describe how to compute the optimal cost $O_m(L, F)$. A naive approach can be considered which 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 to solve this problem. In this paper, we show the problem can be defined recursively and also can be solved with $O(mn^2)$ computations by using *dynamic programming*.

We first describe the recursive structure of this problem. We define $C'(i,j)$ and $NC(i,j)$. The $C'(i,j)$ for $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant m$ is the optimal cost for $L[i..n]$ and $F[i..n]$ over the window size lists of length j whose smallest window size w_1 is l_i. For example, $C'(3,2)$ when $n = 5$ is the optimal cost for $L[3..5]$ and $F[3..5]$ over the window size lists of length 3 such that $w_1 = l_3$. It should be noted that $C'(1, m) = O_m(L, F)$. $NC(i,j)$ for $1 \leqslant i \leqslant j \leqslant n$ is the cost of sublist $L[i..j]$ over a window size list $\langle l_i \rangle$, i.e., $C(L[i..j], F[i..j], \langle l_i \rangle)$. By Eq. (5), $NC(i,j) = \sum_{k=i}^{j} f_k l_k / l_i$. The next two lemmas show recurrence relations for $C'(i,j)$ and $NC(i,j)$, respectively.

Lemma 2. $C'(i,j) = \min_{k=i+1}^{n-j+2} \{NC(i,k-1) + C'(k,j-1)\}$ *for* $1 \leqslant i \leqslant n$ *and* $2 \leqslant j \leqslant m$.

Proof. By the 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 $l_{i+1}, \ldots, l_{n-i+1}$ by Lemma 1. Assume that $w_2 = l_{k'}$ for some k'. Since $w_2 = l_{k'}$, $C'(i,j) = NC(i,k'-1) + C'(k',j-1)$ and thus $C'(i,j) = NC(i,k'-1) + C'(k',j-1) \leqslant NC(i,k-1) + C'(k,j-1)$ for any $i+1 \leqslant k \leqslant n-i+1$. Hence, the recurrence is satisfied for $C'(i,j)$. □

Lemma 3. $NC(i,j) = NC(i,j-1) + f_j l_j / l_i$ *for all* $1 \leqslant i < j \leqslant n$.

Proof. Since $NC(i,j) = \sum_{k=i}^{j} f_k l_k / l_i$ and $NC(i,j-1) = \sum_{k=i}^{j-1} f_k l_k / l_i$, $NC(i,j) = NC(i,j-1) + f_j l_j / l_i$. □

Fig. 3 shows the recursion tree of a problem instance when $n = 5$ and $m = 4$. The optimal solution $C'(1,4)$ is computed from either $NC(1,1) + C'(2,3)$ or $NC(1,2) + C'(3,3)$ by Lemma 2 where $C'(2,3)$ and $C'(3,3)$ are also computed recursively in the similar way and $NC(1,2)$ is computed recursively by Lemma 3. We note that the shaded values in the recursion tree are computed more than once. If the values in the recursion tree are

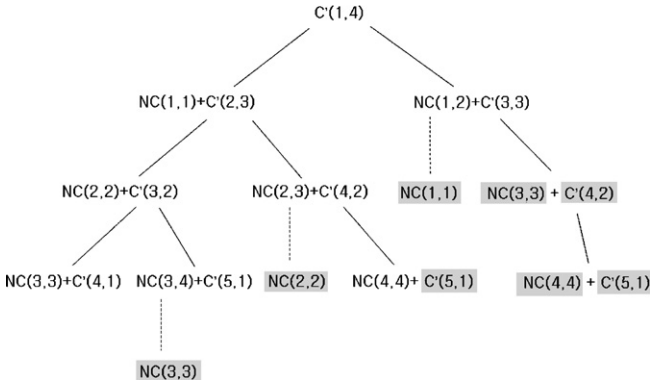

Fig. 3. A process of computing the optimal window sizes ($n = 5$, $m = 4$).

computed *in a bottom-up fashion using dynamic programming*, however, they can be obtained much more efficiently since redundant computations are all removed.

The computation of the optimal cost $O_m(L,F)$ ($=C'(1,m)$) consists of two steps. We first compute array NC in step 1 and then compute the optimal cost $O_m(L,F)$ in step 2.

Step 1. Compute array NC: We show how to compute all $NC(i,j)$'s in $O(n^2)$ time. Since $NC(i,j) = \sum_{k=i}^{j} f_k l_k / l_i$, $NC(i,i) = f_i$ for $1 \leqslant i \leqslant n$. By Lemma 3, $NC(i,j) = NC(i,j-1) + f_j l_j / l_i$ for $1 \leqslant i < j \leqslant n$. It means that we compute $NC(i,i)$ in $O(1)$ time and also compute $NC(i,j)$ in $O(1)$ time from $NC(i,j-1)$. Hence, we can compute all $NC(i,j)$'s in $O(n^2)$ time.

Step 2. We compute the optimal cost $O_m(L,F)$: we show $C'(1,m)$ can be computed in $O(mn^2)$ time. We compute $C'(i,j)$'s for $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant m$ in a bottom-up manner. That is, $C'(i,j)$ can be computed from $NC(i,k-1)$'s and $C'(k,j-1)$'s for $i+1 \leqslant k \leqslant n-j+2$ by Lemma 2, which implies that $C'(i,j)$ can be computed in $O(n)$ time. Since we compute at most mn values of $C'(i,j)$'s, $C'(1,m)$ can be computed in $O(mn^2)$ time. Therefore, we get the following lemma.

Lemma 4. *The optimal cost $O_m(L,F)$ can be computed in $O(mn^2)$ time.*

The pseudocode for computing $C'(1,m)$ is depicted in Fig. 4. In lines 1–4, we compute array NC. In lines 5–10, we initialize the elements of array C'. In lines 11–15, we compute all the elements of array C'.

```
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. 4. Pseudocode for computing $C'(1,m)$.

5. Performance evaluation

In this section, we verify the superiority of the proposed method by performance evaluation with extensive experiments. Section 5.1 explains the experiment environment, and Section 5.2 presents the effect of performance improvement by comparing the proposed and previous methods.

5.1. Experiment environment

For performance evaluation, we conducted experiments with three different kinds of data. The first one is *Stock_Data*, the same one used for our preliminary experiments presented in Section 3, consisting of 620 stock price sequences of length 1024. The second one is *RandomWalk_Data*, a synthetic data set comprising random walk data sequences [1]. The third one is *Periodic_Data*, a synthetic data set comprising pseudo-periodic data sequences [24,25]. For performing our experiments extensively, we generated five sets for each of *RandomWalk_Data* and *Periodic_Data* that comprise 10 data sequences of lengths 100,000, 200,000, 300,000, 400,000, and 500,000, respectively. On all these data sets, we built R^*-trees in the same way as in our preliminary experiments.

Also, we generated query sequences having their lengths of multiples of 32 in the range [64, 1024]. As shown in Fig. 5, the lengths of query sequences have four kinds of distribution: normal, exponential, random, and uniform distribution. The horizontal axis represents a length of query sequences, and the vertical axis does the frequency of query sequences of the length. As a performance factor, we used the average execution time for subsequence matchings with the total of 248 query sequences belonging to each distribution. We also adjusted a tolerance so that 20 final answers are returned.

The hardware platform used in our experiments is a 3 GHz Pentium 4 PC equipped with 1 GB RAM and 80 GB hard disk. The software platform is an MS Windows 2000 Server. The language used in development is Microsoft Visual C++. To minimize the interference from other system and user processes, we set the software platform to a single-user mode. 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 Ref. [22] already verified that *Dual-Match* performs much better than FRM, we only used *Dual-Match* in our experiments.

We evaluated the performance of the four subsequence matching methods: (A) *Dual-Match* with only an index (as in the original approach), (B) *Dual-Match* with multiple indexes whose window sizes are evenly

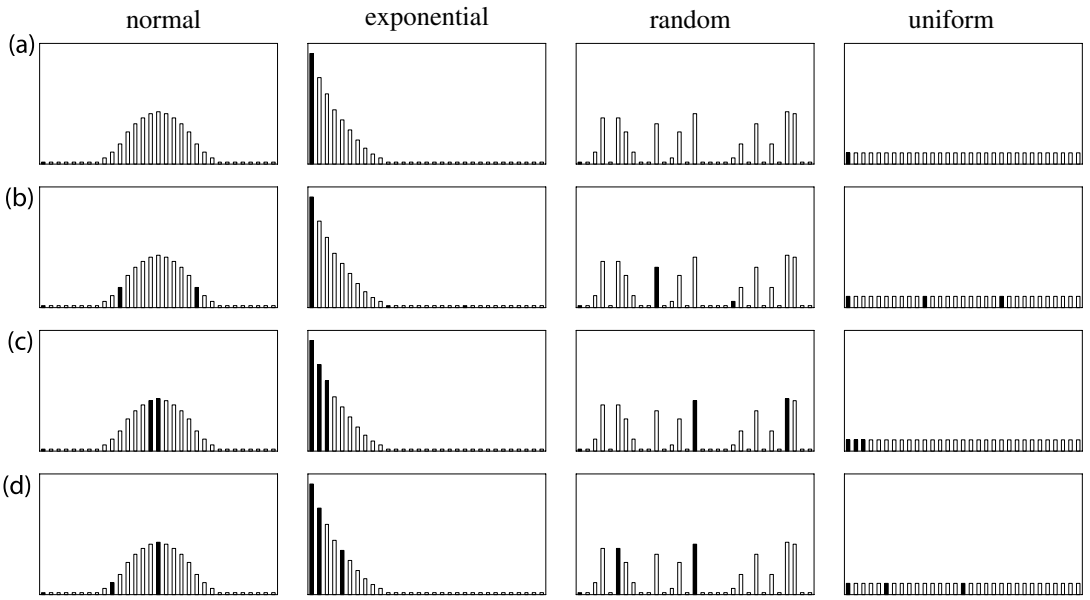

Fig. 5. Distribution of lengths of query sequences and three window sizes chosen.

Table 1
Window sizes chosen by each method

	# of windows	Window sizes chosen			
		Normal	Exponential	Random	Uniform
(A)	1	32	32	32	32
(B)	3	32, 192, 352	32, 192, 352	32, 192, 352	32, 192, 352
	4	32, 144, 272, 384	32, 144, 272, 384	32, 144, 272, 384	32, 144, 272, 384
	5	32, 128, 224, 320, 416	32, 128, 224, 320, 416	32, 128, 224, 320, 416	32, 128, 224, 320, 416
(C)	3	32, 256, 272	32, 48, 64	32, 272, 464	32, 48, 64
	4	32, 256, 272, 288	32, 48, 64, 80	32, 272, 464, 480	32, 48, 64, 80
	5	32, 240, 256, 272, 288	32, 48, 64, 80, 96	32, 80, 272, 464, 480	32, 48, 64, 80, 96
(D)	3	32, 176, 272	32, 48, 96	32, 112, 272	32, 112, 272
	4	32, 160, 224, 272	32, 48, 64, 112	32, 80, 192, 400	32, 80, 176, 320
	5	32, 160, 224, 272, 304	32, 48, 64, 80, 128	32, 80, 192, 272, 464	32, 80, 144, 224, 352

chosen in the range of the minimum and maximum lengths of query sequences, (C) *Dual-Match* with multiple indexes whose window sizes are set to the most frequently occurring lengths of query sequences, (D) *Dual-Match* with multiple indexes whose window sizes are chosen by our approach. Hereafter, we shortly call them methods (A), (B), (C), and (D), respectively.

Table 1 shows the window sizes chosen by each method for each distribution of the length of query sequences. Also, we depict them as black ones in Fig. 5 when we choose three window sizes.

5.2. Results and analyses

We ran three types of experiments for performance evaluation. In Experiment 1, we compared the performance of the four methods (A), (B), (C), and (D) using *Stock_Data* with different numbers of indexes. In Experiments 2 and 3, we compared the performance of the four methods using the two kinds of synthetic *RandomWalk_Data* and *Periodic_Data* while changing the length of data sequences.

In Experiment 1, we measured the performance change of the four methods performed on *Stock_Data* while employing 3, 4, and 5 indexes. Fig. 5 shows the window sizes as black ones chosen by each method in case of employing three indexes. Fig. 6 shows the results of Experiment 1. The horizontal axis represents the number of R^*-trees employed for subsequence matching, and the vertical axis does the average execution time in the unit of seconds. The main purpose of Experiment 1 is to examine the performance tendency caused by different numbers of indexes. Thus, we do not show the result with method (A), which employs just an index, in Fig. 6.

Method (A) shows the worst performance, and method (D) does the best performance in all cases. Let us examine the case of normally distributed lengths of query sequences. When using five R^*-trees, method (D) performs 14.75 times, 1.14 times, and 3.31 times better than methods (A), (B), and (C), respectively. Also, when using three R^*-trees, method (D) performs 10.6 times, 1.15 times, and 3.21 times better than methods (A), (B), and (C), respectively. Next, let us see the case of exponentially distributed lengths of query sequences. With five R^*-trees, method (D) outperforms methods (A), (B), and (C) 3.76 times, 1.22 times, and 1.12 times, respectively. Also, with three R^*-trees, method (D) outperforms methods (A), (B), and (C) 2.94 times, 1.31 times, and 1.33 times, respectively.

Let us investigate the case of randomly distributed lengths of query sequences. With five R^*-trees, method (D) runs faster than methods (A), (B), and (C) 15.69 times, 1.34 times, and 1.35 times, respectively. With three R^*-trees, method (D) runs faster than methods (A), (B), and (C) 10.19 times, 1.16 times, and 2.11 times, respectively. Finally, let us see the case of uniformly distributed lengths of query sequences. When using five R^*-trees, method (D) shows performance 14.65 times, 1.14 times, and 3.42 times better than methods (A), (B), and (C), respectively. With three R^*-trees, method (D) shows performance 10.08 times, 1.12 times, and 4.15 times better than methods (A), (B), and (C), respectively. We also observe that the performance gain of our method tends to get larger with a smaller number of R^*-trees employed in subsequence matching.

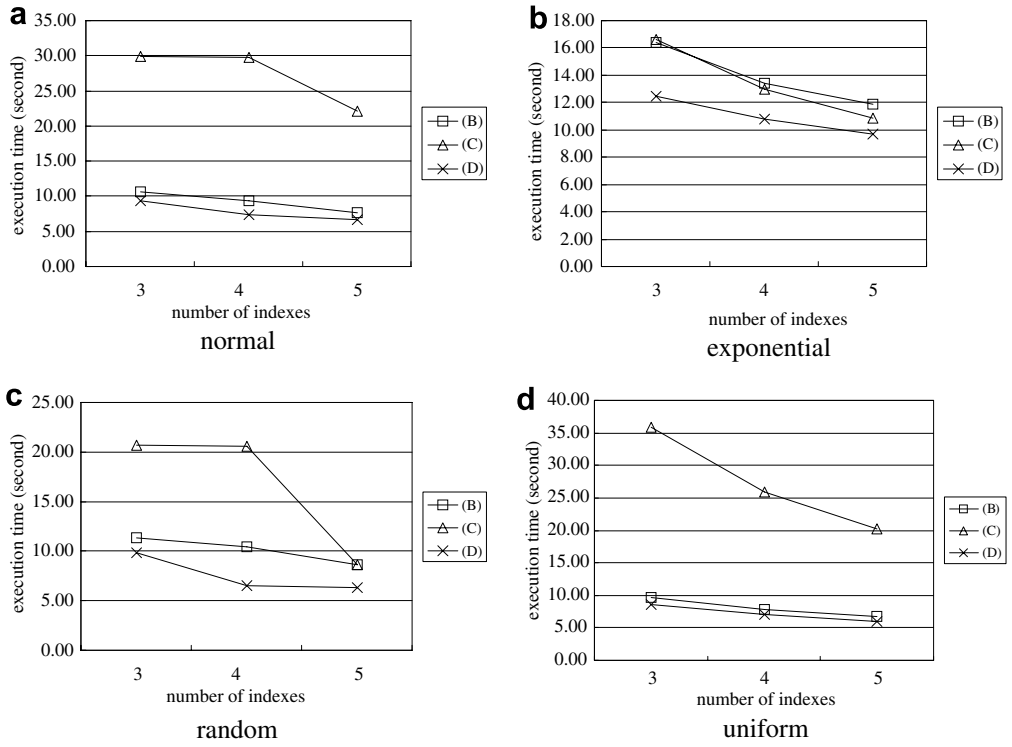

Fig. 6. Performance with different numbers of indexes (*Stock_Data*).

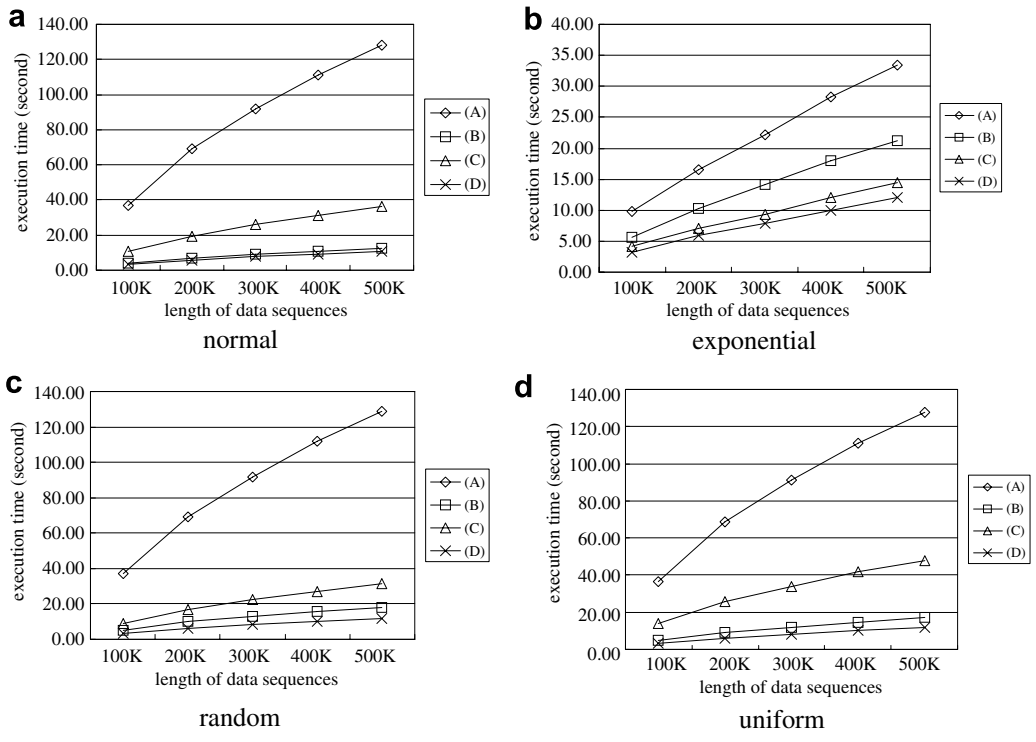

Fig. 7. Performance with different lengths of data sequences (*RandomWalk_Data*).

In Experiment 2, we examined the performance tendency of the four methods performed on 10 *RandomWalk_Data* sequences of 100,000, 200,000, 300,000, 400,000, and 500,000. We built three R^*-trees for methods (B), (C), and (D).

Fig. 7 shows the results of Experiment 2. The horizontal axis represents the length of data sequences, and the vertical axis does the average execution time. In every distribution, method (D) shows the best performance while method (A) does the worst performance. In normal distribution, method (D) performs 11.5, 1.16, and 3.33 times better than methods (A), (B), and (C), respectively, on average. In exponential distribution, method (D) outperforms methods (A), (B), and (C) 2.91, 1.75, and 1.24 times, respectively, on average. In random distribution, method (D) shows performance 10.82, 1.53, and 2.63 times better than methods (A), (B), and (C), respectively, on average. Finally, in uniform distribution, method (D) runs 11.25, 1.46, and 4.23 times faster than methods (A), (B), and (C), respectively, on average.

Method (B), which builds multiple indexes whose windows sizes are selected evenly in the range of the minimum and the maximum of query sequence lengths, performs well in case lengths of query sequences issued are scattered over the entire range. Normal, random, and uniform distribution of query sequence lengths shown in Fig. 5 falls under this case. Therefore, with this distribution, method (B) shows performance better than that of method (C) and comparable to that of method (D).

Method (C) builds multiple indexes within which window sizes are set to the most frequent ones among query sequence lengths. Thus, it performs well when the population of query sequences concentrates upon a few lengths. Exponential distribution in Fig. 5 comes within this case. As a result, with exponential distribution, method (C) significantly outperforms method (B) and also shows performance comparable to that of method (D).

In Experiment 3, we examined the performance tendency of the four methods performed on ten *Periodic_Data* sequences of 100,000, 200,000, 300,000, 400,000, and 500,000. As in Experiment 2, we employed three R^*-trees for methods (B), (C), and (D).

Fig. 8 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 Experi-

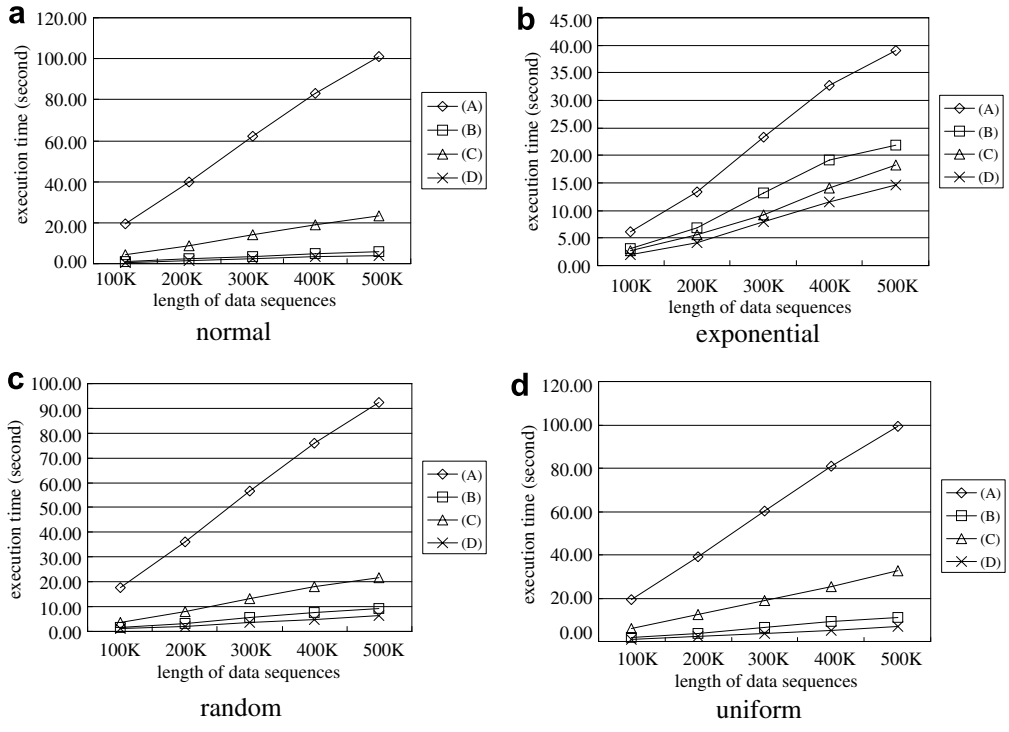

Fig. 8. Performance with different lengths of data sequences (*Periodic_Data*).

ment 2. Regardless of distribution of lengths of query sequences, method (A) performs worst, and method (D) performs best. In case of normal distribution, method (D) achieves 26.01 times, 1.42 times, and 5.83 times speed-up in comparison with methods (A), (B), and (C), respectively, on average. In case of exponential distribution, it performs 2.86, 1.49, and 1.27 times faster than methods (A), (B), and (C), respectively, on average. In case of random distribution, method (D) outperforms methods (A), (B), and (C), 16.06, 1.47, and 3.58 times, respectively, on average. In case of uniform distribution, method (D) runs 15.67, 1.62, and 5.08 times faster than methods (A), (B), and (C), respectively, on average.

In summary, by employing the concept of index interpolation, we could improve the performance of subsequence matching significantly compared with the prior method that uses only a single R^*-tree. Also, our method (D) that selects the optimal window sizes for multiple R^*-trees was shown to be fairly effective when compared with other naive methods (B) and (C).

6. Conclusions

In this paper, we have proposed a novel method for time-series subsequence matching based on index interpolation [20,21] that resolves the performance degradation caused by the window size effect. The basic concept of index interpolation is to build multiple indexes and to perform subsequence matching by selecting the one among them, which is the most appropriate for a given query sequence.

The main contributions can be summarized as follows:

(1) Via preliminary experiments, we have verified that the performance of subsequence matching by using 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 solve this performance problem.
(2) We have derived a formula that estimates the cost for all the subsequence matchings when a set of pairs $\langle \text{length}, \text{frequency} \rangle$ 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.

In experiments with *Stock_Data*, the proposed method (D) shows better performance than methods (A), (B), and (C), up to 15.69, 1.35, 4.15 times, respectively. In experiments with *RandomWalk_Data*, the proposed method (D) outperforms (A), (B), and (C), up to 11.91, 1.77, 4.32 times, respectively. In experiments with *Periodic_Data*, the proposed method (D) runs up to 26.69, 1.65, 5.87 times faster than methods (A), (B), and (C), respectively.

Currently, our 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. We also consider the way to adapt to such environments where the distribution of lengths of query sequences changes dynamically.

Acknowledgement

This work was partially supported by the Korea Research Foundation Grant funded by the Korean Government(KRF-2005-041-D00651), the research fund of Hanyang University (HY-2003-IT), and the ITRC support program supervised by the IITA (IITA-2005-C1090-0502-0009).

References

[1] R. Agrawal, C. Faloutsos, A. Swami, Efficient similarity search in sequence databases, in: Proceedings of the International Conference on Foundations of Data Organization and Algorithms, FODO, 1993, pp. 69–84.

[2] R. Agrawal et al., Fast similarity search in the presence of noise, scaling, and translation in time-series databases, in: Proceedings of the International Conference on Very Large Data Bases, VLDB, 1995, pp. 490–501.
[3] G. Armano et al., A hybrid genetic-neural architecture for stock indexes forecasting, Information Science (2005) 3–33.
[4] N. Beckmann et al., The R^*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of the ACM International Conference on Management of Data, ACM SIGMOD, 1990, pp. 322–331.
[5] D. Berndt, J. Clifford, Finding patterns in time-series: a dynamic programming approach, Advances in Knowledge Discovery and Data Mining (1996) 229–248.
[6] S. Berchtold, D. Keim, H. Kriegel, The X-tree: an index structure for high-dimensional data, in: Proceedings of the International Conference on Very Large Data Bases, VLDB, 1996, pp. 28–39.
[7] C. Chatfield, The Analysis of Time-Series: An Introduction, third ed., Chapman and Hall, 1984.
[8] K. Chan, A. Fu, Efficient time-series matching by wavelets, in: Proceedings of the IEEE International Conference on Data Engineering, IEEE ICDE, 1999, pp. 126–133.
[9] K. Chu, M. Wong, Fast time-series searching with scaling and shifting, in: Proceedings of the ACM Symposium on Principles of Database Systems, ACM PODS, 1999, pp. 237–248.
[10] G. Das, D. Gunopulos, H. Mannila, Finding similar time-series, in: Proceedings of the European Symposium on Principles of Data Mining and Knowledge Discovery, PKDD, 1997, pp. 88–100.
[11] M. Chen et al., Data mining: an overview from database perspective, IEEE Transactions on Knowledge and Data Engineering, IEEE ICDE 8 (6) (1996) 866–883.
[12] Y. Chen et al., Time-series forecasting using flexible neural tree model, Information Science (2005) 219–235.
[13] C. Faloutsos et al., Fast subsequence matching in time-series databases, in: Proceedings of the ACM International Conference on Management of Data, ACM SIGMOD, 1994, pp. 419–429.
[14] A. Fu et al., Scaling and time warping in time-series querying, in: Proceedings of the International Conference on Very Large Data Bases, VLDB, 2005, pp. 649–660.
[15] D. Goldin, P. Kanellakis, On similarity queries for time-series data: constraint specification and implementation, in: Proceedings of the International Conference on Principles and Practice of Constraint Programming, 1995, pp. 137–153.
[16] E. Keogh, C. Ratanamahatana, Exact indexing of dynamic time warping, Knowledge and Information Systems 7 (3) (2005) 358–386.
[17] S. Kim, S. Park, W. Chu, Efficient processing of similarity search under time-warping in sequence databases: an index-based approach, Information Systems 29 (5) (2004) 405–420.
[18] S. Kim, D. Park, H. Lee, Efficient processing of subsequence matching with the Euclidean metric in time-series databases, Information Processing Letters 90 (5) (2004) 253–260.
[19] S. Kim et al., Shape-based retrieval in time-series databases, Journal of Systems and Software 79 (2) (2006) 191–203.
[20] W. Loh, S. Kim, K. Whang, Index interpolation: a subsequence matching algorithm supporting moving average transform of arbitrary order in time-series databases, IEICE Transactions on Information and Systems E84-D (1) (2001) 76–86.
[21] W. Loh, S. Kim, K. Whang, A subsequence matching algorithm that supports normalization transform in time-series databases, Data Mining and Knowledge Discovery Journal 9 (1) (2004) 5–28.
[22] Y. Moon et al., Duality-based subsequence matching in time-series databases, in: Proceedings of the IEEE International Conference on Data Engineering, IEEE ICDE, 2001, pp. 263–272.
[23] Y. Moon, K. Whang, W. Han, General match: a subsequence matching method in time-series databases based on generalized windows, in: Proceedings of the ACM International Conference on Management of Data, ACM SIGMOD, 2002, pp. 382–393.
[24] S. Park et al., Efficient searches for similar subsequences of difference lengths in sequence databases, in: Proceedings of the IEEE International Conference on Data Engineering, IEEE ICDE, 2000, pp. 23–32.
[25] D. Rafiei, A. Mendelzon, Similarity-based queries for time-series data, in: Proceedings of the ACM International Conference on Management of Data, ACM SIGMOD, 1997, pp. 13–24.
[26] D. Rafiei, On similarity-based queries for time-series data, in: Proceedings of the IEEE International Conference on Data Engineering, IEEE ICDE, 1999, pp. 410–417.
[27] R. Weber et al., A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces, in: Proceedings of the International Conference on Very Large Data Bases, VLDB, 1998, pp. 194–205.
[28] H. Wu et al., Subsequence matching on structured time series data, in: Proceedings of the ACM International Conference on Management of Data, ACM SIGMOD, 2005, pp. 682–693.
[29] B. Yi, C. Faloutsos, Fast time sequence indexing for arbitrary L_p norms, in: Proceedings of the International Conference on Very Large Data Bases, VLDB, 2000, pp. 385–394.
[30] B. Yi, H. Jagadish, C. Faloutsos, Efficient retrieval of similar time sequences under time warping, in: Proceedings of the International Conference on Data Engineering, IEEE ICDE, 1998, pp. 201–208.
[31] M. Zhou, M. Wong, A segment-wise time warping method for time scaling searching, Information Sciences 173 (1-3) (2005) 227–254.