Information Sciences 180 (2010) 1458–1476

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Distortion-free predictive streaming time-series matching

Woong-Kee Loh [a,*], Yang-Sae Moon [b], Jaideep Srivastava [c]

[a] Department of Multimedia, Sungkyul University, 400-10, Anyang8-dong, Anyang, Gyeonggi-do 430-742, Republic of Korea
[b] Department of Computer Science, Kangwon National University, 192-1, Hyoja2-dong, Chunchon, Kangwon-do 200-701, Republic of Korea
[c] Department of Computer Science and Engineering, University of Minnesota, 200 Union Street SE, Minneapolis, 55455 MN, USA

ARTICLE INFO

Article history:
Received 24 July 2008
Received in revised form 12 October 2009
Accepted 11 December 2009

Keywords:
Streaming time-series matching
Normalization transform
Multiple indexing
Search cost estimation
Predictive matching

ABSTRACT

Efficient processing of *streaming time-series* generated by remote sensors and mobile devices has become an important research area. As in traditional time-series applications, similarity matching on streaming time-series is also an essential research issue. To obtain more accurate similarity search results in many time-series applications, preprocessing is performed on the time-series before they are compared. The preprocessing removes *distortions* such as offset translation, amplitude scaling, linear trends, and noise inherent in time-series. In this paper, we propose an algorithm for distortion-free predictive streaming time-series matching. Similarity matching on streaming time-series is saliently different from traditional time-series in that it is not feasible to directly apply the traditional algorithms for streaming time-series. Our algorithm is distortion-free in the sense that it performs preprocessing on streaming time-series to remove offset translation and amplitude scaling distortions at the same time. Our algorithm is also predictive, since it performs streaming time-series matching against the predicted most recent subsequences in the near future, and thus improves search performance. To the best of our knowledge, no streaming time-series matching algorithm currently performs preprocessing and predicts future search results simultaneously.

© 2009 Elsevier Inc. All rights reserved.

1. Introduction

A time-series is a sequence of real values sampled at continuous time points. Examples include stock prices, product sales records, medical measurements, and scientific experiments data [1,2,8,14]. For efficient processing of *similar time-series matching*, much research has been performed over many years [1,8,12,15,18,21,25–27,32,38,44]. Given a query sequence Q, similar time-series matching algorithms find the time-series whose distances from Q are within the specified threshold ϵ. These algorithms assume that all the time-series are stored in the database and are intrinsically static. Distance (similarity) measures that are most widely accepted by the existing algorithms are the Euclidean distance [1,8,10–12,25–27] and the dynamic time warping (DTW) distance [15,18,20,21,32,38,44]. However, Keogh identified four kinds of *distortions* that may produce inaccurate search results: offset translation, amplitude scaling, linear trends, and noise [16]. To remove these distortions, preprocessing is performed on the time-series before they are compared. The preprocessing helps find similar time-series more accurately [16].

Due to advances in technologies on remote sensors and mobile devices, efficient processing of time-series generated by these devices, generally called *streaming time-series*, has become an important research area [3,10,11,40]. Each time a new element is appended at the end of the streaming time-series, it is necessary to perform a new similarity matching against the

* Corresponding author. Tel.: +82 31 467 8958; fax: +82 31 467 8058.
E-mail addresses: woong@sungkyul.ac.kr, wkloh2@gmail.com (W.-K. Loh), ysmoon@kangwon.ac.kr (Y.-S. Moon), srivasta@cs.umn.edu (J. Srivastava).

0020-0255/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2009.12.009

most recent subsequence of the streaming time-series. However, similarity matching on streaming time-series is saliently different from traditional time-series matching in the following three aspects: database storage, threshold association, and notion of continuity. First, in traditional time-series matching, a query sequence is given for the similarity search against the time-series in the database. The time-series containing the subsequences similar to the query sequence are returned, and the subsequences have the same length as the query sequence. In contrast, in streaming time-series matching, a number of query sequences of any arbitrary lengths are stored in the database. When a new search is performed, for each of the query sequences Q, the most recent subsequence of the same length as Q is extracted from the streaming time-series and compared with Q. That is, while a search is performed only for a fixed length in traditional matching, the streaming time-series matching is performed for a number of arbitrary lengths. Second, in traditional time-series matching, a fixed threshold ϵ is given with the query sequence. In streaming time-series matching, since the query sequences in the database can have arbitrary lengths, a different threshold is associated with each of the query sequences. Third, there is no notion of *continuity* [11] in traditional time-series matching. Continuity means that the current most recent subsequence is very similar to the next most recent subsequence in the streaming time-series and thus the search results against them should have a large intersection. By using the notion of continuity, a smart streaming time-series matching algorithm could "predict" the future search results, and hence improve search performance.

In this paper, we propose an algorithm for distortion-free predictive streaming time-series matching. Our algorithm is distortion-free in the sense that it performs preprocessing on time-series to remove the offset translation and amplitude scaling distortions at the same time. The preprocessing to remove these two distortions is known as *normalization transform* [12,25], and is useful for finding time-series that have similar fluctuation trends. We explain the importance of normalization transform in streaming time-series matching by an example in Section 3. Our algorithm is also predictive in the sense that it performs streaming time-series matching against the predicted most recent subsequences in the near future, and thus improves search performance.

To the best of our knowledge, no streaming time-series matching algorithm currently performs normalization transform and predicts future search results simultaneously. The contributions of this paper are as follows. (1) We propose a simple algorithm for distortion-free streaming time-series matching, and extend the algorithm to use $k(\geqslant 1)$ indexes to improve search performance. (2) We further extend our algorithm to simultaneously obtain the search results for $m(\geqslant 1)$ time points from the current time point t_0 to a time point $(t_0 + m - 1)$ in the near future by retrieving the index only once. (3) For a given number of indexes k, we present an approximation method to construct k indexes to achieve the optimal search performance of our algorithm. (4) Through a series of experiments, we compare the search performance of our simple and extended algorithms, and show their performance trends according to k and m values. The experiment results show that our algorithms outperform the sequential scan algorithm by up to 13.2 times.

The remainder of this paper is organized as follows. Section 2 briefly introduces previous work on similar time-series matching. Section 3 formally defines the problem of distortion-free streaming time-series matching. Section 4 proposes the distortion-free matching algorithm that uses k indexes. Section 5 proposes the predictive matching algorithm by further extending the algorithm in Section 4. Section 6 presents an approximation method to construct k indexes for optimal search performance of our algorithm. Section 7 evaluates the performance of our algorithm. Finally, Section 8 summarizes and concludes this paper.

2. Related work

A lot of work on time-series matching has been conducted to date. Traditional time-series matching algorithms are classified into two groups: whole matching and subsequence matching algorithms. Whole matching algorithms assume that all time-series in the database and the query sequence have the same length [1]. In contrast, subsequence matching algorithms assume that the time-series and the query sequence can have any arbitrary length, and find the time-series containing the subsequences that are similar to the given query sequence [8,27]. The Euclidean distance has been most widely used in the existing literature to measure the distance between time-series [1,8,10–12,25,27,26], since the measure has the desirable properties in the data mining area such as triangular inequality, which is crucial for efficient indexing. Another widely used distance measure is the DTW distance [15,18,20,21,32,38,44]. Although the DTW measure does not satisfy the triangular inequality, there has been much effort to find efficient indexing based on the measure [15,16,20,32]. Different distance measures previously published include Lp norm [43] and Longest Common SubSequence (LCSS) [37]. Yi and Faloutsos [43] proposed an indexing scheme that supports Lp norm similarity measure for any positive integer $p(p \geqslant 1)$ in a single framework. However, Lp norm has the same weakness as the Euclidean distance in that it is difficult to capture the fluctuation trends in time-series. Perng et al. [30] proposed a new time-series similarity model called the "landmark model," which represents a time-series as a sequence of application-dependent landmarks (important points or events in the time-series). Although their landmark measure is invariant of transforms such as shifting and amplitude/time scaling, they did not focus on performance issues. Instead, their matching algorithm is a simple string matching algorithm without using a multi-dimensional index.

In many applications, similar time-series matching algorithms based on the distance measures explained above may return inaccurate search results. As mentioned in Section 1, Keogh identified four kinds of distortions [16]. For removing the distortions, the time-series are preprocessed before they are compared. The preprocessing helps finding similar time-series

more accurately [16]. There are also publications on similar time-series matching algorithms that deal with preprocessed time-series. The algorithms in [12,25] perform similarity matching after removing the offset translation and amplitude scaling distortions simultaneously, and the algorithm in [26] after removing noise. We claim that preprocessing is independent of the distance measure used for time-series matching. That is, any distance measure can be used regardless of which preprocessing is performed or even whether preprocessing is performed or not.

Streaming time-series matching has different aspects compared to traditional time-series matching as described in Section 1. Hence, it is not feasible to apply the traditional algorithms to solve streaming time-series matching problems. There have also been some algorithms for streaming time-series matching [10,11,40]. In [10], Gao and Wang proposed an algorithm that estimates future elements to be appended to the streaming time-series and performs streaming time-series matching using the estimated elements before the actual elements arrive. However, the algorithm has a drawback that it computes the distance for each of the query sequences in the database without using any index and thus the performance of the algorithm is significantly degraded in disk-based environments. In [11], a k-nearest neighbor (k-NN) search algorithm based on prefetching query sequences from the disk was proposed. The drawback of this algorithm is that the lengths of all the query sequences should be identical and thus the algorithm cannot be applied to general cases when the query sequences have arbitrary lengths. In another approach in [40], a subsequence similarity matching algorithm was proposed for financial data stream such as stock prices. While the algorithm introduces some new ideas such as segmentation of streaming time-series, it has a drawback in that it is specialized to a specific application of Elliott Wave Theory [9].

To the best of our knowledge, no streaming time-series matching algorithm deals with preprocessed time-series. Our algorithm compares the most recent subsequence in the streaming time-series with each of the query sequences of arbitrary lengths after they are normalization-transformed (i.e., after removing the offset translation and amplitude scaling distortions simultaneously). Our algorithm resembles the algorithm in [25] in the sense that k different indexes are used. However, the method for constructing k indexes in our algorithm is far different from the algorithm in [25]. Moreover, the performance of our algorithm is improved by predicting the future search results, which is difficult in the algorithm in [25]. We explain more about these differences in Section 4. The method for constructing optimal k indexes in our algorithm is comparable with that in [22]. However, while our method is an "adaptive" one that captures the performance trend of the matching algorithm for a specific streaming time-series and constructs the optimal indexes based on the performance trend, the method in [22] is a "fixed" one without considering the performance trend against the specific streaming time-series. Moreover, our method is faster for finding optimal k indexes. We explain more in Section 6.

Since time-series data occupy a large size of volume, they need to be transformed to compact representations for better disk storage and processing performance [7]. There have been proposed a number of time-series representations such as discrete Fourier transform (DFT), discrete wavelet transform (DWT), piecewise aggregate approximation (PAA), and adaptive piecewise constant approximation (APCA). Lin et al. [23,24] proposed a new representation called *Symbolic Aggregation approXimation* (SAX). SAX is a kind of symbolic representation that represents time-series with a finite number of symbols instead of countless continuous real numbers. Although SAX is simple and easy to implement, it has been shown [7] that there is very little difference in the tightness of lower bounds between representations, i.e., SAX has no remarkable advantage on matching performance over different representations.

There has been research work on time-series mining based on SAX representation. Mueen et al. [29] proposed an exact algorithm that finds motifs in a given time-series. Motifs are the patterns that appear frequently in a longer time-series. Keogh et al. [19] proposed a discord detection algorithm based on SAX representation. Discords are the subsequences of a time-series that are maximally different from any other subsequences in the same time-series. Shieh and Keogh [33,34] tackled the scalability problem of time-series data volume with an indexing method based on a multi-resolution symbolic representation called *indexable SAX* (iSAX). iSAX is an extension of SAX representation, and the indexing method is based on the idea similar to extensible hashing. Although the above SAX-based algorithms have high search performance, they assume that the entire target time-series should be stored in memory or disk, and they do not deal with online streaming time-series. Moreover, the algorithms do not consider removing distortions before any further processing.

Time-series matching algorithms have been extended for processing two-dimensional shapes. Wei et al. [39] proposed an algorithm for finding shape discords. The algorithm first converts the shapes into one-dimensional time-series, and then orders the shapes using their inter-similarities based on the locality-sensitive hashing function. The most different shapes are recognized as shape discords. Yankov et al. [41] proposed a rotation-invariant algorithm for finding similar shapes. They proposed a shape metric that satisfies triangular inequality and achieved speed-up of their algorithm by pruning needless candidates. Ye and Keogh [42] proposed a decision tree-like shape classification algorithm. The algorithm uses only the shapelets, which are small discriminating subsections of shapes, for shape classification instead of comparing the whole shape.

Time-series forecasting is another interesting research topic. Zhang [45] proposed an ensemble model of neural networks for improving forecasting accuracy. The ensemble is formed by multiple identical network architecture trained with different algorithms, weights, and samples. Hamzacebi [13] adopted neural networks for seasonal or periodic time-series forecasting. The seasonal time-series forecasting problem was regarded as a function approximation problem. Silva [35] presented a nonlinear model for time-series forecasting that combines radial basis functions and the ARMA (p,q) structure. A heuristic algorithm is proposed for finding optimal set of parameters.

3. Problem definition

In this paper, the most recent subsequence of the streaming time-series is compared with each of the query sequences after they are normalization-transformed. The normalization transform is defined as follows [12,25]:

Definition 1. For a sequence $X = (x_0,\ldots,x_{n-1})$ of length n, the elements $\tilde{x}_i (0 \leqslant i < n)$ in the normalization-transformed sequence, which is denoted as $v(X)$, are computed using the following equation:

$$\tilde{x}_i = \frac{x_i - \mu(X)}{\sigma(X)}, \tag{1}$$

where $\mu(X)$ and $\sigma(X)$ are the mean and the standard deviation of all the elements composing the sequence X, respectively.

When the standard deviation $\sigma(X)$ is 0, i.e., the elements in the sequence X are all identical, the case should be processed separately.

As the input of the problem in this paper, the following are given: one streaming time-series S of infinite length, $N(\geqslant 1)$ query sequences $Q_i (0 \leqslant i < N)$ of arbitrary lengths, and search thresholds $\epsilon_i (0 \leqslant i < N)$ associated with each query sequence Q_i. It is not assumed that the lengths of query sequences should follow any specific distribution, and there can be one or more query sequences of the same lengths. The output of the problem is the query sequences Q_i that have similar fluctuation trends with the most recent subsequence X in the streaming time-series S. The most recent subsequence X in S is called *streaming subsequence* in this paper, and should have the same length as Q_i. The length of streaming subsequence X is determined separately according to the query sequence Q_i to be compared with X. The similarity condition can be expressed as the following equation:

$$d(v(X), v(Q_i)) \leqslant \epsilon_i, \tag{2}$$

where $d()$ is the distance function between two sequences. For the distance measure between two sequences, we use the Euclidean distance in this paper, since it is most widely adopted in the literature [1,8,10–12,25,27,26]. The sequences $v(X)$ and $v(Q_i)$ are the normalization-transformed sequences of X and Q_i, respectively. Several reasons make our problem difficult and prohibit us from simply applying the existing time-series matching algorithms. The relative distances among time-series are not preserved before and after the normalization transform. Moreover, since the length of the streaming subsequence X changes depending on the query sequence Q_i to compare with X, the normalization transform should be performed separately for each of the streaming subsequences of different lengths. Table 1 summarizes the notations used in this paper.

We explain the importance of normalization transform in streaming time-series matching by an example. Fig. 1 shows the real stock prices of a wireless communication services company. In the figure, the thin black solid graph shows daily stock prices, and the thick green dotted graph is the result of smoothing (removing the distortion of noise). An element in the smooth graph was obtained by averaging ten stock prices in the past. The smooth graph turns upward at time points (A) and (B), and turns downward at time point (C). If we compare the subsequences around (A), (B), and (C) without performing normalization transform, the subsequences around (B) and (C) would be evaluated more similar than the subsequence around (A), since they have almost the same prices around 180,000. However, if we perform normalization transform on the subsequences before comparing them, the subsequences around (A) and (B) would be evaluated as similar ones, since they have a similar fluctuation trend, i.e., they turn upward. Stock prices are continuously generated while the stock is being traded, and fund managers dealing with stocks and futures must decide whether to buy or sell them in a matter of minutes or even seconds. Therefore, the streaming time-series matching algorithm proposed in this paper will be very helpful in real-world applications.

It was understood that DTW was better than Euclidean distance in finding similar time-series. However, recent work [7,33] has discovered that DTW is advantageous only when dealing with small size datasets. As the dataset gets larger, the advantage of DTW diminishes quickly, and even for moderately large datasets, there is almost no difference in matching accuracy. Since we deal with large datasets in most time-series mining problems, there is no difference in choosing any of DTW and Euclidean distance. The same is true for other similarity measures [7].

Table 1
Summary of notations.

Notation	Description
S	streaming time-series of infinite length
$S[t]$	an element at time point t in the streaming time-series S; current time point is denoted as t_0, and the most recent element in S as $S[t_0]$
X	streaming subsequence in the streaming time-series S; $X = (S[t_0 - Len(X) + 1],\ldots,S[t_0])$
$X^{(t)}$	streaming subsequence at time point t in the streaming time-series S; the most recent streaming subsequence is denoted as $X = X^{(t_0)}$
Q_i	a query sequence $(0 \leqslant i < N)$
$T[s,f]$	a window that is composed of elements $T[s],\ldots,T[f]$ in the sequence T; $T[s,f] = (T[s],\ldots,T[f])(0 \leqslant s \leqslant f < Len(T))$
$Len(T)$	the length of a sequence T
N	the number of the entire query sequences
n	the number of query sequences (those of the same lengths are counted as one)

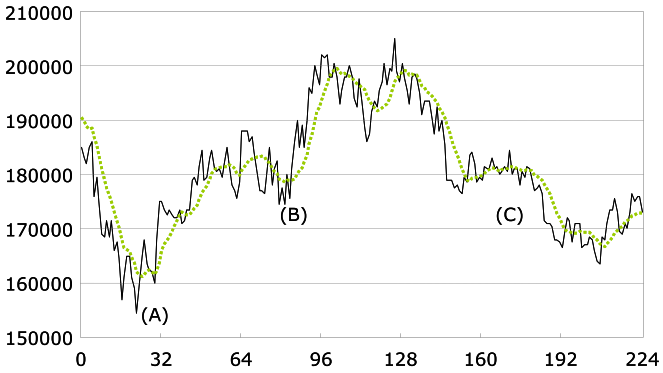


Fig. 1. An example of real stock prices.

One of the strengths of our matching algorithms is that they do not assume any data distributions of query sequences and/or streaming subsequences as stated above. Even though the query sequences or streaming subsequences follow a specific data distribution, the effect of normalization transform is the changes of mean and standard deviation values only, and the data distribution is still preserved. Hence, we believe that there is no simplification of the matching problem by normalization transform.

4. Distortion-free matching algorithm

In this section, we describe our distortion-free streaming time-series matching algorithm for the most recent streaming subsequence. Our algorithm is executed whenever a new element is appended at the end of the streaming time-series S. In Section 4.1, we describe the simple algorithm that uses only one index, and then, in Section 4.2, we present the extended algorithm that uses $k(\geqslant 1)$ indexes for improving search performance.

4.1. Simple algorithm

In our distortion-free matching algorithm, the query sequences Q can have any arbitrary lengths, and the streaming subsequence X should have the same length as the query sequence to which it is being compared. If we manage an index for each of query sequence lengths, it would be significantly burdensome. Hence, we convert the comparison between X and Q of arbitrary lengths into the comparison between windows X_w and Q_w of the fixed length using the following equation [25]:

$$d(v(X), v(Q)) \leqslant \epsilon \Rightarrow d(v(X_w), v(Q_w)) \leqslant \epsilon', \tag{3}$$

where $X_w = (x_s, \ldots, x_f)$ and $Q_w = (q_s, \ldots, q_f)$ are the windows of the same size $w(w = f - s + 1)$ extracted from the same offset s in the sequences X and Q, respectively. The new threshold ϵ' is computed as follows [25]:

$$\epsilon' = \sqrt{2w - 2\sqrt{w^2 - w \cdot \epsilon^2 \cdot \frac{\sigma^2(X)}{\sigma^2(X_w)}}}, \tag{4}$$

where $\sigma(X)$ and $\sigma(X_w)$ are the standard deviations of the elements composing X and X_w, respectively. By using Eq. (3), we can construct only one index of the fixed size, since the sizes of the windows X_w and Q_w are fixed. This equation enables us to efficiently manage the index and to improve the search performance. If the value in the inner square root is less than 0 in Eq. (4), we must perform the sequential scan on the query sequences. However, as a result of our experiments in this paper, such a case rarely occurs unless the query sequences are too short. Nevertheless, the value in the outer square root in Eq. (4) can never be less than 0.

The simple algorithm extracts the rightmost windows of size w_s from all the query sequences, and constructs an index using the windows. The algorithm extracts the most recent window of the same size w_s from the streaming time-series S whenever a new element is appended, and finds similar query sequences using the window. Since the size of windows cannot be greater than the lengths of query sequences, the window size w_s should be the shortest query sequence length. The windows extracted from the query sequences and the streaming time-series are called *query windows* and *streaming window*, respectively, in this paper. The query windows are normalization-transformed before they are stored in the index. The streaming window is also normalization-transformed before it is used to find similar query sequences. Fig. 2 shows the process of similarity matching by the simple algorithm. In the figure, $Q_{i\cdot w}$ $(0 \leqslant i < N)$ is the rightmost window of length w extracted from query sequence Q_i, i.e., $Q_{i\cdot w} = Q_i[Len(Q_i) - w, Len(Q_i) - 1] = (Q_i[Len(Q_i) - w], \ldots, Q_i[Len(Q_i) - 1])$, where $Len(Q_i)$ is the length of Q_i. For example, $Q_{1\cdot w}$ is the rightmost window of length w in Q_1, i.e., $Q_{1\cdot w} = Q_1[Len(Q_1) - w, Len(Q_1) - 1] = (Q_1[Len(Q_1) - w], \ldots, Q_1[Len(Q_1) - 1])$. All the query windows and the streaming window have the same size $Len(Q_2)$, since Q_2 is the shortest query sequence.

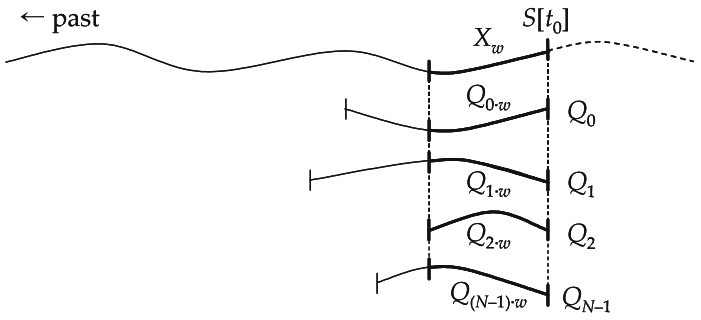


Fig. 2. Simple (distortion-free) algorithm.

1: **for** each query sequence Q_i $(0 \leq i < N)$ **do**
2: Extract rightmost window Q_w of size w_s from Q_i;
3: Normalization-transform Q_w;
4: Store $\nu(Q_w)$ in a multidimensional index;
5: **end for**

(a) Indexing phase.

1: Extract rightmost window X_w of size w_s from the streaming time-series S;
2: Normalization-transform X_w;
3: Find the query windows $\nu(Q_w)$ within distance ϵ' from $\nu(X_w)$ using the index constructed in the indexing phase;
4: **for** each window Q_w found **do**
5: Identify the query sequence Q_i containing the window Q_w;
6: Compute actual distance between $\nu(Q_i)$ and $\nu(X)$, where X is the streaming subsequence of the same length as Q_i in S;
7: Return Q_i if the distance is not larger than the associated threshold ϵ_i;
8: **end for**

(b) Searching phase.

Fig. 3. Indexing and searching phases of simple algorithm.

Fig. 3 shows the indexing and searching phases of the simple algorithm. In line (2) in the indexing phase, the rightmost windows $Q_i[Len(Q_i) - w_s, Len(Q_i) - 1]$ are extracted from the query sequences $Q_i (0 \leqslant i < N)$. In line (4), any multi-dimensional index such as R*-tree [4] can be used. In general, as the dimensionality of the multi-dimensional index increases, the search cost using the index also increases exponentially [5]. Hence, in line (4), as in the traditional time-series matching algorithms, the query windows $\nu(Q_w)$ are converted into $f(\ll w_s)$-dimensional windows through DFT, and then are stored in the f-dimensional index. In the searching phase, the streaming subsequence matching for the current time point t_0 is performed. In line (1), a streaming window $X_w = (S[t_0 - w_s + 1], \ldots, S[t_0])$ of the same size w_s from the streaming time-series S is extracted. In line (3), the normalization-transformed streaming window $\nu(X_w)$ is converted into the f-dimensional window in the same way as the indexing phase.

In line (3) in the searching phase in Fig. 3b, we need the standard deviation $\sigma(X)$ and the threshold ϵ to compute the new threshold ϵ' using Eq. (4). Since ϵ' is used against all the query sequences, $\sigma(X)$ should be computed for all the streaming subsequences X of the same lengths as all the query sequences, and ϵ should be representative of all the thresholds ϵ_i associated with query sequences Q_i. The threshold ϵ' is computed using the maximum standard deviation $\sigma_{\max}(X)$ and the maximum threshold $\epsilon_{\max}$ as in the following equations:

$$\begin{aligned} \sigma_{\max}(X) &= \max\{\sigma(X) | w_s \leqslant Len(X) \leqslant L\}, \\ \epsilon_{\max} &= \max\{\epsilon_i | 0 \leqslant i < N\}, \end{aligned} \tag{5}$$

where L is the largest query sequence length. Since ϵ' becomes the maximum when it is computed with $\sigma_{\max}(X)$ and $\epsilon_{\max}$, there is no query window rejected falsely in the searching phase.

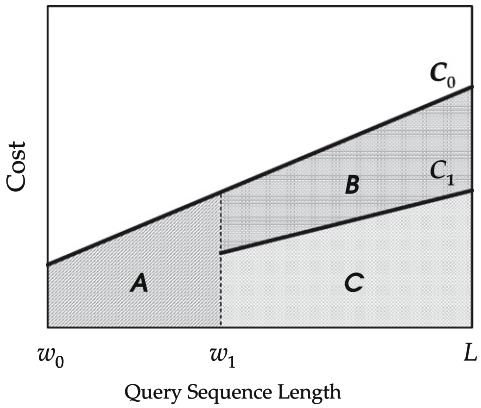
The simple algorithm causes no false dismissal (false negative). The algorithm uses the index constructed in the indexing phase to find all the query windows Q_w that satisfy the consequent part of Eq. (3). The set of query sequences Q_i containing the query windows Q_w found using the index is a superset of the query sequences that satisfy the antecedent part of Eq. (3). That is, the algorithm does not falsely reject any query sequences that should be returned as the search result.

4.2. Extension of simple algorithm

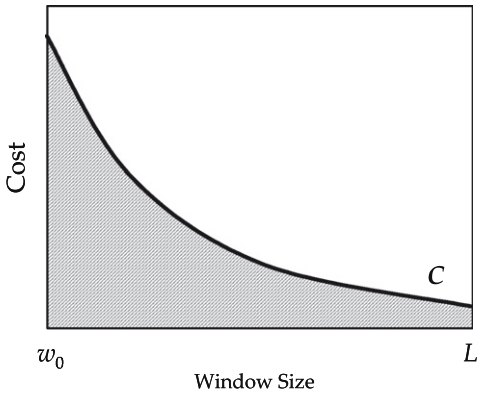
The simple algorithm presented in the previous section has the following weakness. The window size w_s used for constructing the index and performing the search in the simple algorithm is equal to the shortest query sequence length, which can be far less than the lengths of the other query sequences. In general, the search cost of the simple algorithm increases almost linearly to the difference between the query sequence length $Len(Q)$ and the window size w_s. Thus, as the difference becomes larger, the search performance of the algorithm would be significantly degraded.

Fig. 4 shows the search cost trends of the simple algorithm according to query sequence length and window size. Fig. 4a shows the trend according to query sequence length when using the indexes constructed with the windows of sizes $w_0(=w_s)$ and w_1, and Fig. 4b shows the trend according to window size when performing the search against the query sequences of fixed length L. Here, it is assumed that the lengths of query sequences follow a uniform distribution. In Fig. 4a, the graph C_0 represents the search cost when using the index of window size w_0, and the graph C_1 when using the index of window size w_1. The graph C_0 indicates that, when using the index of window size w_0, the search cost for finding query sequence Q increases almost linearly, as the length of Q increases, i.e., the difference $(Len(Q) - w_0)$ increases. The graph C_1 also indicates that the search cost for finding query sequence Q using the index of window size w_1 is almost linear to the length of Q or the difference $(Len(Q) - w_1)$. The graph C_1 appears for query sequence lengths greater than or equal to w_1, since the index of window size w_1 cannot be used to find query sequences whose lengths are less than w_1. To find query sequences whose lengths are somewhere between w_1 and L, it is advantageous to use the index of window size w_1 than w_0, since the search cost is smaller as indicated in Fig. 4a. It should be easily understandable since the difference $(Len(Q) - w_1)$ is smaller than $(Len(Q) - w_0)$.

While the simple algorithm uses only one index, the extended algorithm uses $k(\geqslant 1)$ indexes to improve the search performance. The extended algorithm partitions the whole query sequences into k groups according to their lengths, and constructs an index for each of the groups. The streaming subsequence matching is performed separately using the index for each group. Let $w_0,\ldots,w_{k-1}$ $(w_0 < w_1 < \ldots < w_{k-1} < L)$ be the k query sequence lengths chosen for partitioning the whole query sequences, and $G_j(0 \leqslant j < k)$ be the jth group. A query sequence Q belongs to group G_j according to the following equation:



(a) According to query sequence length.



(b) According to window size.

Fig. 4. Search cost trends.

$$G_j = \{Q | w_j \leqslant Len(Q) < w_{j+1}\}, \tag{6}$$

where it is assumed that $w_k = L + 1$. According to the above equation, w_j is the shortest query sequence length in group G_j. Fig. 5 shows the process of similarity matching by the extended algorithm. In the figure, $Q_{i\cdot w_j}$ $(0 \leqslant i < N, 0 \leqslant j < k)$ is the rightmost window of length w_j extracted from query sequence Q_i, i.e., $Q_{i\cdot w_j} = Q_i[Len(Q_i) - w_j, Len(Q_i) - 1] = (Q_i[Len(Q_i) - w_j], \ldots, Q_i[Len(Q_i) - 1])$, where $Len(Q_i)$ is the length of Q_i. For example, $Q_{1\cdot w_1}$ is the rightmost window of length w_1 in Q_1, i.e., $Q_{1\cdot w_1} = Q_1[Len(Q_1) - w_1, Len(Q_1) - 1] = (Q_1[Len(Q_1) - w_1], \ldots, Q_1[Len(Q_1) - 1])$. The query sequences Q_0 and Q_1 are found using the index of window size w_1, and the query sequences Q_2 and Q_N using the index of window size w_0. The query sequences are normalization-transformed, and then are compared with the normalization-transformed streaming subsequences of the same lengths.

Fig. 6 shows the indexing and searching phases of the extended algorithm. In line (1) in the indexing phase, k query sequence lengths are chosen. We explain how to choose k query sequence lengths for optimal search performance in Section 6. In lines (3)–(5), for each group $G_j (0 \leqslant j < k)$, the algorithm does the same as in the indexing phase of the simple algorithm. The index for group G_j is constructed using the windows of size w_j, and is denoted as I_j in this paper. In the searching phase, the streaming subsequence matching for the current time point t_0 is performed as in the simple algorithm. In line (2), for each group G_j, the streaming window of size w_j is extracted from the streaming time-series S. In line (3), the similar query sequences are found in the same way as in the searching phase in the simple algorithm using the index I_j.

In the searching phase, we compute the threshold $\epsilon'_j (0 \leqslant j < k)$ separately for each of k groups G_j, i.e., the maximum standard deviation $\sigma_{\max\cdot j}(X)$ and the maximum threshold $\epsilon_{\max\cdot j}$ are computed differently for each group G_j. We modify Eq. (5) and obtain the following Eq. (7):

$$\begin{aligned} \sigma_{\max\cdot j}(X) &= \max\{\sigma(X) | w_j \leqslant Len(X) < w_{j+1}\}, \\ \epsilon_{\max\cdot j} &= \max\{\epsilon_i | w_j \leqslant Len(Q_i) < w_{j+1}, 0 \leqslant i < N\}. \end{aligned} \tag{7}$$

The new threshold ϵ'_j is computed using $\sigma_{\max\cdot j}(X)$ and $\epsilon_{\max\cdot j}$ for each group G_j. Similar to the simple algorithm, by using the threshold ϵ'_j for each group, there is no query window rejected falsely in the searching phase. In general, as ϵ' becomes larger, the false alarm (false positive) also becomes larger, which may cause degradation of the search performance of the simple algorithm. However, since the thresholds ϵ'_j are tighter than the general threshold ϵ' in the simple algorithm, there are far fewer false alarms and thus the search performance of the extended algorithm will significantly improve.

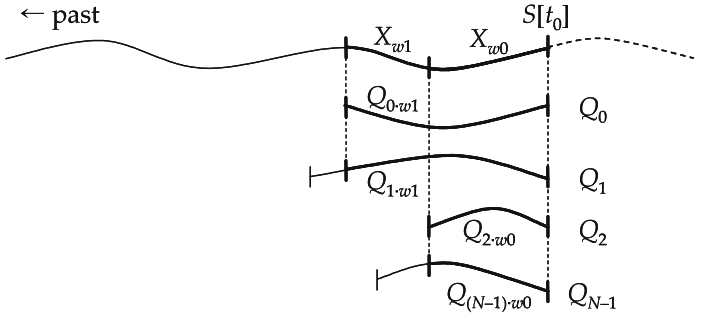


Fig. 5. Extended (distortion-free) algorithm.

1: Choose k window sizes $w_0, \ldots, w_{k-1}$;
2: Partition query sequences into k groups G_j $(0 \leq j < k)$ using Eq. (6);
3: **for** each group G_j **do**
4: Do the same as in the indexing phase in Simple Algorithm (Figure 3(a));
5: **end for**

(a) Indexing phase.

1: **for** each group G_j $(0 \leq j < k)$ **do**
2: Extract rightmost window X_w of size w_j from the streaming time-series S;
3: Do the same as in the searching phase in Simple Algorithm (Figure 3(b)) using the index I_j constructed for the group;
4: **end for**

(b) Searching phase.

Fig. 6. Indexing and searching phases of extended algorithm.

Although the computation cost for normalizing the streaming subsequences is not low, the cost is not a major factor determining the performance of our algorithm for two reasons. First, in most time-series matching algorithms, a great proportion of their execution time is spent accessing the disk for retrieving candidate data sequences or query sequences [6,8,27]. That is, the time for performing normalization transform does not have much influence on the overall execution time of our algorithm. Second, normalization of the streaming subsequences can be performed incrementally. That is, two parameters for normalization, mean and standard deviation, of the most recent streaming subsequence $X^{(t_0)}$ can be obtained by a few simple addition/multiplication from the mean and standard deviation of the previous recent streaming subsequence $X^{(t_0-1)}$. The computation of DFT also has limited influence on the performance of our algorithm. Moreover, there exist numerous Fast Fourier Transform (FFT) algorithms that are applicable to time-series whose lengths are not an integer power of two.

Some researchers may claim that it is beneficial to use the ST-index in our algorithm. The ST-index was used in the subsequence matching algorithm by Faloutsos et al. [8], denoted as *FRM* in this paper, to store minimum bounding rectangles (MBRs) containing multiple windows extracted from data sequences. In the FRM algorithm, the number of windows extracted from a data sequence S is $Len(S) - w + 1$, where $Len(S)$ and w are the lengths of S and a window, respectively. Since the number of MBRs containing the windows is far less than that of the windows themselves, the cost for retrieving the ST-index is far less than that for retrieving an index storing the entire windows. However, the ST-index is not applicable to our algorithm for the following two reasons. First, the number of windows extracted from a query sequence Q is far less than the FRM algorithm, i.e., only 1 in our distortion-free algorithm and m in our predictive algorithm (explained in Section 5), while the number of extracted windows is $Len(Q) - w + 1$ in the FRM algorithm. Second, Moon et al. [27,28] showed that the ST-index can cause many false alarms and thus severe performance degradation.

Our distortion-free matching algorithm resembles the algorithm based on *index interpolation* [25] in the sense that both algorithms use k indexes. However, there are two major differences between the algorithms. First, while the same data sequence is redundantly stored in k indexes in the index interpolation, a query sequence is stored in only one of k indexes in our algorithm. Hence, the total size of indexes in our algorithm is k times smaller than the index interpolation. Second, while only one of k indexes is used for searching in the index interpolation, all the k indexes are used in our algorithm. Since the indexes are used independently of the others in our algorithm, the searching threads using different indexes can be run in parallel. Although Eq. (3) was presented in [25], the way the equation is applied in our distortion-free algorithm is totally different. Moreover, there are two more algorithms to which the equation is applied but they were never dealt with in [25]: the predictive streaming time-series matching algorithm and the algorithm for choosing optimal k window sizes, which will be explained in Sections 5 and 6, respectively.

There is a trade-off on k in our algorithm. When the number of indexes k becomes larger, the false alarms caused by our algorithm in each query sequence group G_j are reduced, and thus the search performance improves. At the same time, the cost of retrieving all the indexes also increases, which causes the degradation of the search performance. When k is very close to N, our algorithm effectively performs the sequential scan. Analytically, k would be affected by numerous factors such as window size, distribution of query sequence lengths, size of the whole time-series database, and size of available physical space. Thus, we believe that finding the optimal k analytically is beyond the scope of this paper.

In the domain of our work, there are two kinds of outliers: (1) an outlier streaming subsequence X in the streaming time-series S and (2) an outlier query sequence Q_i in the database. First, the outlier streaming subsequence X has no negative effect on the performance of our algorithms, since the matching process should finish with almost no result earlier than normal streaming subsequences. Second, the outlier query sequence may affect the performance of our matching algorithms. The structure of underlying R*-tree is optimized following the distribution of query sequences, and sparse distribution may increase the number of R*-tree accesses by our algorithms. However, this is an intrinsic problem of the R*-tree and is beyond the scope of this paper. If there exists a different multi-dimensional index structure which performs better than R*-tree even in presence of outliers, we can just replace the R*-tree with the better index structure.

5. Predictive matching algorithm

In this section, we further extend our distortion-free algorithm in Section 4 to be predictive, i.e., to predict the future search results. The essential idea of our predictive matching algorithm is that we find the candidate query sequences for the m time points in the near future by retrieving the index only once instead of m times. In that way, we can reduce the cost needed to retrieve the index several times, and thus improve the search performance. Fig. 7 shows the idea of our predictive matching algorithm. In the figure, $Q_{f:w}$ $(0 \leqslant f < m)$ is the window of length w whose right offset is f in query sequence Q_f, i.e., $Q_{f:w} = Q_f[Len(Q_f) - (f + w), Len(Q_f) - (f + 1)] = (Q_f[Len(Q_f) - (w + f)],\ldots,Q_f[Len(Q_f) - (f + 1)])$, where $Len(Q_f)$ is the length of Q_f. For example, $Q_{1:w}$ is the window of length w whose right offset is 1 in Q_1, i.e, $Q_{1:w} = Q_1[Len(Q_1) - (w + 1), Len(Q_1) - 2] = (Q_1[Len(Q_1) - (w + 1)],\ldots,Q_1[Len(Q_1) - 2])$. The algorithm described in Section 4 compares the streaming window X_w only with the rightmost windows in the query sequences, e.g., the window $Q_{0:w}$ in query sequence Q_0 in Fig. 7. In contrast, the predictive matching algorithm compares the streaming window X_w not only with the rightmost windows but also with the windows whose right offsets in the query sequences are $1,\ldots,m-1$. If the streaming window X_w is similar to the query window Q_w whose right offset is $f(0 \leqslant f < m)$ in a query sequence Q, i.e., the distance between $v(X_w)$ and $v(Q_w)$ is less than or equal to ϵ', then Q is a candidate query sequence that might be returned by our algorithm as a similar query sequence to the streaming subsequence $X^{(t_0+f)}$ at the future time point $(t_0 + f)$. For example, if the streaming window X_w is similar to the query

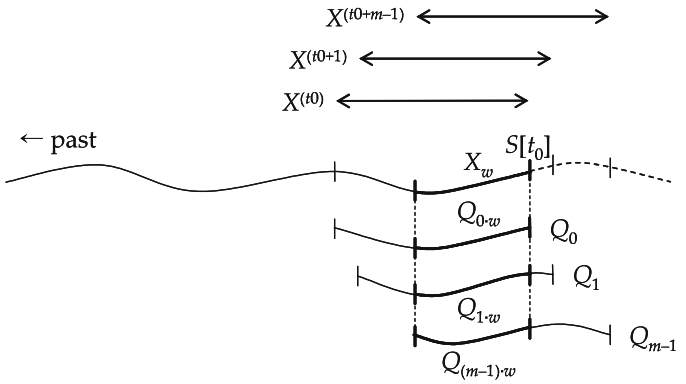


Fig. 7. Predictive matching algorithm.

1: Choose k window sizes $w_0, \ldots, w_{k-1}$;
2: Partition query sequences into k groups G_j $(0 \le j < k)$ using Eq. (6);
3: **for** each group G_j **do**
4: Extract windows Q_w of size w_j whose right offsets are $0, \ldots, m-1$ from Q;
5: Do the same as steps (3) $\sim$ (4) in the indexing phase in Simple Algorithm (Figure 3(a));
6: **end for**

(a) Indexing phase.

1: **for** each group G_j $(0 \le j < k)$ **do**
2: Extract rightmost window X_w of size w_j from the streaming time-series S;
3: Do the same as steps (2) $\sim$ (3) in the searching phase in Simple Algorithm (Figure 3(b)) using the index I_j;
4: **for** $f = 0$ to $m-1$ **do**
5: For each window Q_w whose right offset is f, do the same as steps (5) $\sim$ (7) in the searching phase in Simple Algorithm (Figure 3(b));
6: **end for**
7: **end for**

(b) Searching phase.

Fig. 8. Indexing and searching phases of predictive matching algorithm.

window $Q_{1\cdot w}$ whose right offset is 1 in the query sequence Q_1 in Fig. 7, Q_1 is a candidate query sequence that might be similar to the streaming subsequence $X^{(t_0+1)}$. At time point (t_0+1), the query sequence Q_1 and the streaming subsequence $X^{(t_0+1)}$ are normalization-transformed, and then the actual distance between them is computed. If the distance is not larger than ϵ_1, they are considered similar and the query sequence Q_1 is returned by our algorithm.

Fig. 8 shows the indexing and searching phases of our predictive matching algorithm. In line (1) in the indexing phase, unlike in the previous section, the shortest window size w_0 should be (shortest query sequence length $-m$). In line (4), the windows whose right offsets are $f (0 \leqslant f < m)$ are extracted from each of the query sequences, and the indexes are constructed using the windows. In the indexes, the offset f is stored together with each of the windows. In the searching phase, unlike the algorithm in the previous section, the streaming subsequence matching for a group of time points $t_0, \ldots, (t_0+m-1)$ are performed simultaneously by retrieving the index only once. In line (3), by retrieving the index against the streaming window X_w, all the query windows similar to X_w are obtained regardless of their right offsets in the query sequences. The query windows are partitioned into f groups according to their right offsets. For each of the query windows Q_w retrieved from the index, the distance $e = d(v(Q_w), v(X_w))(\geqslant 0)$ from the streaming window X_w is maintained together with the right offset f. In the for loop in lines (4)–(6), the query windows are processed in the order of their right offsets. At time point (t_0+f), the algorithm compares the streaming subsequence $X^{(t_0+f)}$ only with the query sequences containing the query windows of right offset f.

Since our predictive algorithm performs the streaming subsequence matching in advance against the future streaming subsequences X, the algorithm has a problem in that it cannot determine the standard deviation $\sigma_{\max}(X)$ needed for computing ϵ' in Eq. (4). Thus, we use an estimate $\hat{\sigma}_{\max}(X)$ of $\sigma_{\max}(X)$ to compute the estimated threshold $\hat{\epsilon}'$. As the estimate $\hat{\sigma}_{\max}(X)$, we can use the maximum standard deviation of the past streaming subsequences of the same length as X. The maximum

standard deviation $\hat{\sigma}_{\max\_j}(X)$ for each query sequence group $G_j(0 \leqslant j < k)$ can be obtained in the same manner. The candidate query windows are retrieved from the index using the estimated threshold $\hat{\epsilon}'$. At time point $t(t_0 \leqslant t < t_0 + m - 1)$, the actual threshold ϵ' is computed for the streaming subsequence $X^{(t)}$ and is compared with $\hat{\epsilon}'$. If $\hat{\epsilon}' \geqslant \epsilon'$, there must exist candidate query windows Q_w whose distances e from the streaming window X_w are larger than ϵ', and those query windows are only discarded without any further processing in line (5) in the searching phase (Fig. 8b). If $\hat{\epsilon}' < \epsilon'$, only the query windows whose distances e from X_w are $\hat{\epsilon}' < e \leqslant \epsilon'$ need to be retrieved from the index.

6. Choosing optimal window sizes

In this section, we present a method for choosing k window sizes $w_0,\ldots,w_{k-1}$ for achieving optimal search performance. For a given k,k query sequence lengths are chosen, and the query sequences are partitioned into k groups according to the k query sequence lengths. Then, the k query sequence lengths are used as the window sizes to construct the indexes $I_j(0 \leqslant j < k)$. Since a window size w_j should not be larger than the length of any query sequence belonging to the corresponding group G_j,w_0 becomes the shortest query sequence length. Thus, we explain the method to choose $(k-1)$ window sizes $w_1,\ldots,w_{k-1}$ in this section.

The problem of choosing $(k-1)$ query sequence lengths for the optimal search performance of our algorithm is a difficult problem for the following two reasons. First, although the search cost of our algorithm increases linearly according to the difference between the query sequence length and the window size, the slope and y-intersect change depending on w_j. Second, it is assumed in this paper that the query sequence lengths do not follow any specific distribution, and that there can be an arbitrary number of query sequences of the same lengths. In this paper, we propose an approximation-based method for choosing the optimal $(k-1)$ query sequence lengths. First, a few test indexes are constructed using the windows of arbitrarily chosen sizes, and then the actual search costs of our algorithm are measured through experiments using the test indexes. Next, through the approximation on the measured search costs, a cost formula is generated to estimate the search costs using the indexes of arbitrary window sizes. Interpolation [31] is used for approximation. Finally, using the search cost estimate formula, we choose $(k-1)$ query sequence lengths to construct k indexes for the optimal search performance.

The search cost model of our algorithm might be generated by identifying all or most of the factors that affect the performance of our algorithm. The system environment in which our algorithm is developed could be so complicated that there would be many such factors, e.g., window size, distribution of query sequence lengths, size of the whole time-series database, transactions accessing the database simultaneously, and the size of available physical space. Thus, it is almost impractical or even impossible to consider all such factors. In this paper, the search cost model of our algorithm is generated by identifying only a few dominant factors and observing the performance trend according to the factors. We choose window size and query sequence length as the most dominant factors. We then generate a search cost model according to the chosen dominant factors based on the search results against the earlier streaming subsequences, and then use the search cost model against the later streaming subsequences.

Fig. 4a in Section 4 shows the graphs C_0 and C_1 of search costs by our algorithm using the indexes I_0 and I_1 of window sizes w_0 and w_1, respectively. Since the index I_1 cannot be used to find the query sequences whose lengths are less than w_1, the graph C_1 appears only for the query sequences whose lengths are larger than or equal to w_1. In Fig. 4a, the total search cost of finding all the query sequences using only the index I_0 by our algorithm is equal to the area below the graph C_0 (area $A + B + C$). Here, the probability that a query sequence is returned by the algorithm is assumed to follow a uniform distribution. To find the query sequences whose lengths are larger than or equal to w_1, either the index I_0 or I_1 can be used with the algorithm. However, it is advantageous to use the index I_1 instead of I_0, since the differences between the query sequence lengths and the window size w_1 are smaller and thus it can reduce false alarms. In Fig. 4a, the total search cost to find all the query sequences using both of the indexes I_0 and I_1 with our algorithm is equal to the area below the graphs C_0 and C_1 (area $A + C$). To find the query sequences whose lengths are larger than or equal to w_1, the ratio of the search cost using the index I_1 divided by the search cost using the index I_0 is equal to (area C)/(area $B + C$).

We approximately compute the search cost of our algorithm using an index I_j of a specific window size $w_j(w_0 \leqslant w_j \leqslant L)$ as follows. Let $C_0(w)$ be the search cost for finding a query sequence of length $w(w_j \leqslant w \leqslant L)$ by our algorithm using the index I_0, and $C_j(w)$ using the test index I_j. Here, the search costs $C_0(w)$ and $C_j(w)$ are measured through experiments. For each of the query sequences whose lengths w are larger than or equal to w_j, we compute the ratio $C_j(w)/C_0(w)$ of search costs for finding the query sequence using the indexes I_j and I_0. We then compute the average of the ratios, which is called *search cost ratio* using the test index I_j and is denoted as R_j in this paper. The search cost ratio R_j is computed using the following equation:

$$R_j = \frac{\sum_{w=w_j..L} C_j(w)/C_0(w)}{L - w_j + 1}, \tag{8}$$

where L represents the largest query sequence length. In Fig. 4a, R_1 is obtained by computing the average of the ratios $C_1(w)/C_0(w)$ measured for each of query sequences whose lengths w are larger than or equal to w_1.

Once we obtain the search cost ratios R_j for $(d+1)(d \geqslant 0)$ arbitrary window sizes $w_j(0 \leqslant j \leqslant d)$ using Eq. (8) through experiments, we then derive a formula for estimating the search cost ratios for any arbitrary $w(w_0 \leqslant w \leqslant L)$ as follows.[1] Since

[1] The value d is called *degree* of polynomial in Eq. (9), and can be either larger or smaller than the number of indexes k.

the search cost ratio varies smoothly according to window size w as shown in Fig. 4b, it can be estimated by the following polynomial:

$$\widehat{R}(w) = c_0 + c_1 w + c_2 w^2 + \cdots + c_d w^d, \tag{9}$$

where $\widehat{R}(w)$ is called *estimated search cost ratio* in this paper ($0.0 \leqslant \widehat{R}(w) \leqslant 1.0$), and $c_0,\ldots,c_d$ are constants. According to Eq. (9), $\widehat{R}(w_j)$ returns the estimated search cost ratio that might be obtained by dividing the search cost using the index I_j by the search cost using the index I_0 when our algorithm returns the query sequences whose lengths are larger than or equal to w_j. The constants $c_0,\ldots,c_d$ are determined differently according to the streaming time-series and the query sequences, which is the reason that we call our method "adaptive." The constants are computed by interpolation [31].

Eq. (9) was obtained on the assumption that query sequence lengths follow the uniform distribution and that every query sequence has the same probability to be returned by our algorithm. However, Eq. (9) can also be used without modification even when there are an arbitrary number of query sequences of the same lengths and the query sequences do not have the same probabilities to be returned by the algorithm, which are proven by the following Lemmas 1 and 2:

Lemma 1. Eq. (9) *can be used even when there are an arbitrary number of query sequences of the same lengths.*

Proof. See Appendix A. □

Lemma 2. Eq. (9) *can be used even when the query sequences do not have the same probabilities to be returned by our algorithm.*

Proof. See Appendix B. □

Since Eq. (9) is the estimate formula of search cost ratios compared with the search costs using the index I_0, we should experimentally measure the search costs using the index I_0. Once the estimated search cost ratio formula $\widehat{R}(w)$ and the search costs using the index I_0 are obtained, we can readily choose $(k-1)$ window sizes $w_1,\ldots,w_{k-1}$ for optimal search performance. Fig. 9 shows our heuristic method to choose $(k-1)$ optimal window sizes, where it is assumed that $w_k = L + 1$.

As a data structure for computing the overall search cost in line (2) in Fig. 9, we need an array $CC_0[w_0 \ldots L]$ to save the cumulative search costs of our algorithm with the index I_0. An element $CC_0[w](w_0 \leqslant w \leqslant L)$ of the array contains the total search cost for finding the query sequences whose lengths are from w_0 to w, which can be computed using the following Eq. (10):

$$CC_0[w] = \sum_{i=w_0}^{w} C_0(i). \tag{10}$$

In Fig. 4a, the cost in $CC_0[w_1]$ is equal to the area A.

For given k window sizes $w_j (0 \leqslant j < k)$, we can compute the overall search cost using the array CC_0 as follows. We first obtain the search cost for each j of our algorithm with the index I_0, which can be easily obtained using the array CC_0. We then convert each search cost into that with the index I_j, which can be done using the estimated search cost ratio formula $\widehat{R}(w)$ in Eq. (9). The overall search cost is obtained by summing up the search cost for each j as in the following equation:

$$SC = \sum_{j=0}^{k-1} \widehat{R}(w_j) \cdot \left(CC_0[w_{j+1} - 1] - CC_0[w_j - 1]\right), \tag{11}$$

where it is assumed that $w_k = L + 1, CC_0[w_0 - 1] = 0$, and $\widehat{R}(w_0) = 1.0$. We have time complexity of $O(k)$ for computing the overall search cost using Eq. (11).

The time complexity of our heuristic method in Fig. 9 is as follows. In line (5), the overall search cost is computed for every window size w_j between w_{j-1} and w_{j+1}. The overall search cost is not newly computed using Eq. (11), but is only added by the cost difference according to w_j. Hence, the time complexity of line (5) is $O(n)$. The for loop in lines (4)–(6) is executed k times, and have time complexity of $O(kn)$. Assuming that the repeat loop in lines (3)–(7) is executed r times, the time complexity of the whole method is $O(rkn)$. A dynamic programming method might be used for finding the optimal window sizes $w_0,\ldots,w_{k-1}$, which has time complexity of $O(kn^2)$ [22]. As the result of our experiments, the number of the repeat loop

1: Evenly distribute w_j $(1 \leq j < k)$. That is, for each j, set $w_j = w_0 + (L - w_0) \cdot (j/k)$;
2: Compute the overall search cost;
3: **repeat**
4: **for** $j = 1$ to $k - 1$ **do**
5: Set w_j as the value in the range (w_{j-1}, w_{j+1}) such that the overall search cost becomes minimum;
6: **end for**
7: **until** no change

Fig. 9. Heuristic method for finding optimal window sizes.

executions r was a small number ($r \ll n$). Thus, our heuristic procedure is more efficient than the dynamic programming method.

Our algorithm for choosing optimal window sizes is valuable in a dynamic environment where query sequences are often inserted/deleted for the following reason. When the query sequence database is modified significantly so that the current k indexes do not work well, a new set of k optimal window sizes should be computed and, using the window sizes, new k indexes $I_j (0 \leqslant j < k)$ should be re-generated. It is obvious that, in such cases, a faster algorithm is advantageous, since it minimizes search delay and/or performance degradation. We show that the window sizes computed by our algorithm is comparable to those by the dynamic programming method through experiments in Section 7. Even in the static environment where no insertion/deletion occurs, since our algorithm is faster, it is better than the dynamic programming method.

7. Performance evaluation

In this section, we evaluate the performance of our algorithm proposed in this paper. We explain the experiment settings and data in Section 7.1, and show the evaluation result in Section 7.2.

7.1. Experiment environment

In our experiments, we implemented two types of processes connected via the named pipe [36] to simulate the steaming time-series environment on the Linux Operating System. The process of the first type is *data process* that continuously generates the streaming time-series, and the processes of the second type are *search processes* that actually perform the streaming subsequence matching. The streaming time-series generated by the data process is transferred via the named pipe to the search processes. The search processes are implemented separately according to search algorithms. In this section, we denote the sequential scan algorithm as *SeqScan*, and the simple and the extended distortion-free matching algorithms proposed in Section 4 as *SingleIdx* and *MultiIdx*, respectively. The distortion-free matching algorithm is further classified into three algorithms according to the method to choose the optimal window sizes. We denote the algorithm by the evenly distributing method, the one by our heuristic method proposed in Section 6, and the one by the dynamic programming method as *Multi_Even*, *Multi_Heur*, and *Multi_Dyn*, respectively.[2]

The hardware platform for our experiments is a PC equipped with Intel Pentium 4 2.8 GHz CPU, 512 MB RAM, and 80 GB HDD, and the software platform is the GNU/Linux Version 2.6.6 Operating System. The search processes except *SeqScan* use the R*-tree [4] as the multi-dimensional index structure for storing windows. The number of features is set as 6 for lower-dimensional conversion through the DFT. We use both synthetic and real-world streaming time-series for our experiments. The synthetic streaming time-series is generated using the equation $S[i] = 100 \cdot [\sin(0.1 \cdot RandomWalk[i]) + 1.0 + i/20000] (i = 0..19999)$ as in [10]. The random walk function *RandomWalk* returns 1500.0 as the first value, and generates the $(i+1)$th value $RandomWalk[i+1]$ by adding an arbitrary value in the range $(-1.0, 1.0)$ to the previous value $RandomWalk[i]$. The real-world streaming time-series used in our experiments is the closing prices of Standard and Poor's 500 Index from 1926 to 1993, which consists of 17,610 elements. The real-world data was obtained from the UCR time-series archive built by Keogh [17]. The query sequences are generated by extracting subsequences of length $Len(Q)$ from arbitrary locations of the synthetic and real-world streaming time-series as in [10], where the query sequence lengths are $Len(Q) = 128,160,\ldots,1024$.

We generate four query sets while changing the distribution of query sequence lengths. The first set is denoted as $\mathbf{Q}_{even}$, and the number of query sequences of the same lengths is fixed as 60 in $\mathbf{Q}_{even}$. The other query sets are denoted as $\mathbf{Q}_1$,$\mathbf{Q}_2$, and $\mathbf{Q}_4$, in which the distributions of query sequence lengths are determined using the following equation:

$$QC(w) = Amp \cdot \left[\cos\left(Freq \cdot \pi \cdot \frac{w - w_0}{L - w_0}\right) + 1.0\right] + Min, \tag{12}$$

where $QC(w)$ is the number of query sequences of length w, and Amp, $Freq$, and Min represent amplitude, frequency, and minimum, respectively ($w_0 = 128$, $L = 1024$). In our experiments, we set $Amp = 50$ and $Min = 10$, and generate the query sets $\mathbf{Q}_1$,$\mathbf{Q}_2$, and $\mathbf{Q}_4$ by setting $Freq = 1$, 2, and 4, respectively. We use the query sets $\mathbf{Q}_{even}$,$\mathbf{Q}_1$,$\mathbf{Q}_2$, and $\mathbf{Q}_4$ in our experiments to show that our algorithm executes efficiently on the query sets of diverse distributions of query sequence lengths.

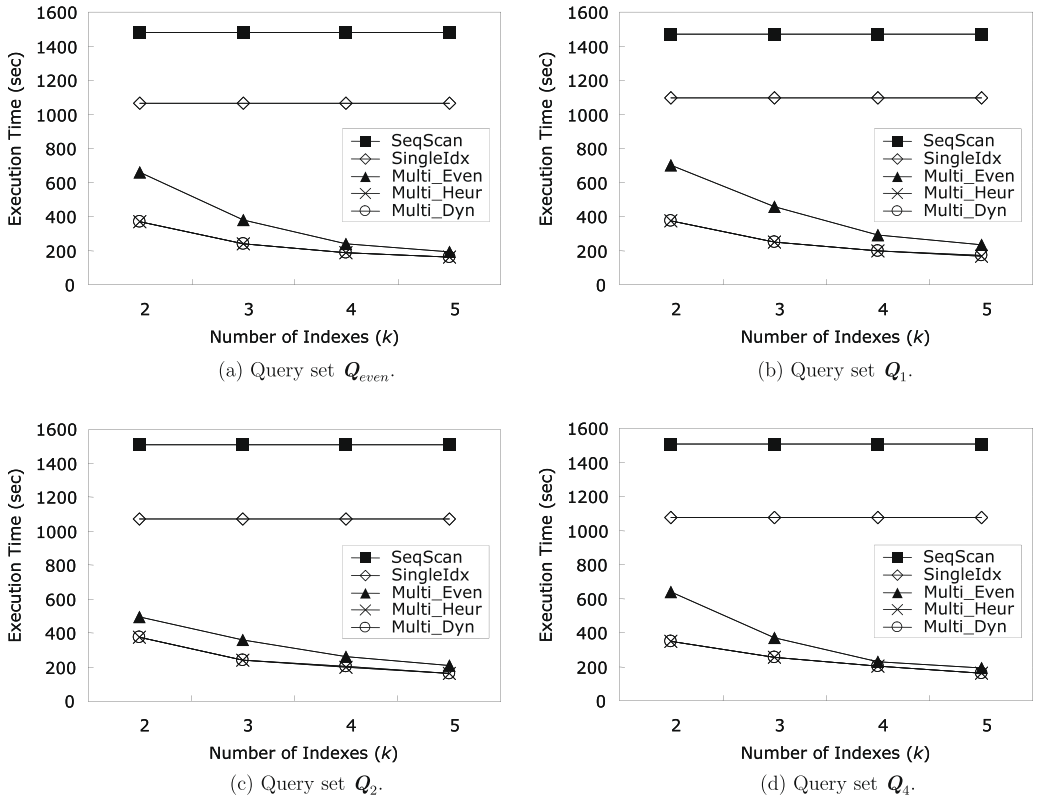
7.2. Performance evaluation results

In this section, we first compare the window sizes chosen by our heuristic method in Section 6 and those chosen by the dynamic programming method. We use the query sets $\mathbf{Q}_{even}$,$\mathbf{Q}_1$,$\mathbf{Q}_2$, and $\mathbf{Q}_4$, and set the number of indexes as $k = 2$, 3, 4, and 5. As a result, the same window sizes are chosen by the two methods for $k = 2$ and 3, and very similar window sizes for $k = 4$. Table 2 shows the window sizes for $k = 5$ chosen by the methods for each of the query sets. As can be seen in the table, similar window sizes are chosen.

[2] The dynamic programming method used in *Multi_Dyn* algorithm exploits the search cost ratio estimate formula $\widehat{R}(w)$ in Eq. (9), and is different from the method in [22].

Table 2
Window sizes chosen by two methods ($k = 5$).

Query set	Method	Window sizes
$\mathbf{Q}_{even}$	Heuristic	128, 256, 416, 576, 800
	Dynamic	128, 288, 448, 608, 800
$\mathbf{Q}_1$	Heuristic	128, 224, 320, 448, 640
	Dynamic	128, 256, 384, 512, 704
$\mathbf{Q}_2$	Heuristic	128, 224, 352, 640, 864
	Dynamic	128, 224, 352, 640, 864
$\mathbf{Q}_4$	Heuristic	128, 224, 416, 544, 864
	Dynamic	128, 256, 448, 576, 864



(a) Query set $\mathbf{Q}_{even}$. (b) Query set $\mathbf{Q}_1$.

(c) Query set $\mathbf{Q}_2$. (d) Query set $\mathbf{Q}_4$.

Fig. 10. Result of first experiment: comparison of performance while changing the number of indexes k.

We perform four experiments in this paper. In the first experiment, we compare the performance of the algorithms while changing the number of indexes k, and in the second experiment, we compare the performance while changing the maximum threshold $\epsilon_{\max}$. The algorithms to be compared in the first and second experiments are the *SeqScan*, *SingleIdx*, *Multi_Even*, *Multi_Heur*, and *Multi_Dyn* algorithms, and they are compared separately for each of the query sets $\mathbf{Q}_{even}$,$\mathbf{Q}_1$,$\mathbf{Q}_2$, and $\mathbf{Q}_4$. We also compare the performance of the algorithms on a very large query set in the third experiment. In the fourth experiment, we observe the performance of *SingleIdx* algorithm while changing the prediction length m for query sets $\mathbf{Q}_{even}$ and $\mathbf{Q}_1$.

In the first experiment, we set $\epsilon_{\max} = 3.5$ and compare the search performance for $k = 2$, 3, 4, and 5. For a query sequence of length w, the associated threshold is set as $\epsilon_{\max} \cdot (w/L)$. Fig. 10a–d show the result of the first experiment using the synthetic data. In the figures, the horizontal axis represents the number of indexes k, and the vertical axis represents the execution time (wall clock time) of the algorithms. As shown in the figures, the *MultiIdx* algorithms outperform *SeqScan* and *SingleIdx*. The performance of the *MultiIdx* algorithms improve more as k increases. Among the *MultiIdx* algorithms, while *Multi_Heur* and *Multi_Dyn* have almost no difference in their performance, they are better than *Multi_Even*. *Multi_Heur* outperforms *SeqScan*, *SingleIdx*, and *Multi_Even* by up to 9.3, 6.7, and 1.9 times, respectively. The maximum performance degradation of *Multi_Heur* compared with *Multi_Dyn* is as small as 2.7%.

In the second experiment, we set $k = 3$ and compare the search performance for $\epsilon_{\max} = 1.5$, 3.5, 5.5, and 7.5. As in the first experiment, for a query sequence of length w, the associated threshold is set as $\epsilon_{\max} \cdot (w/L)$. Fig. 11a–d show the result of the

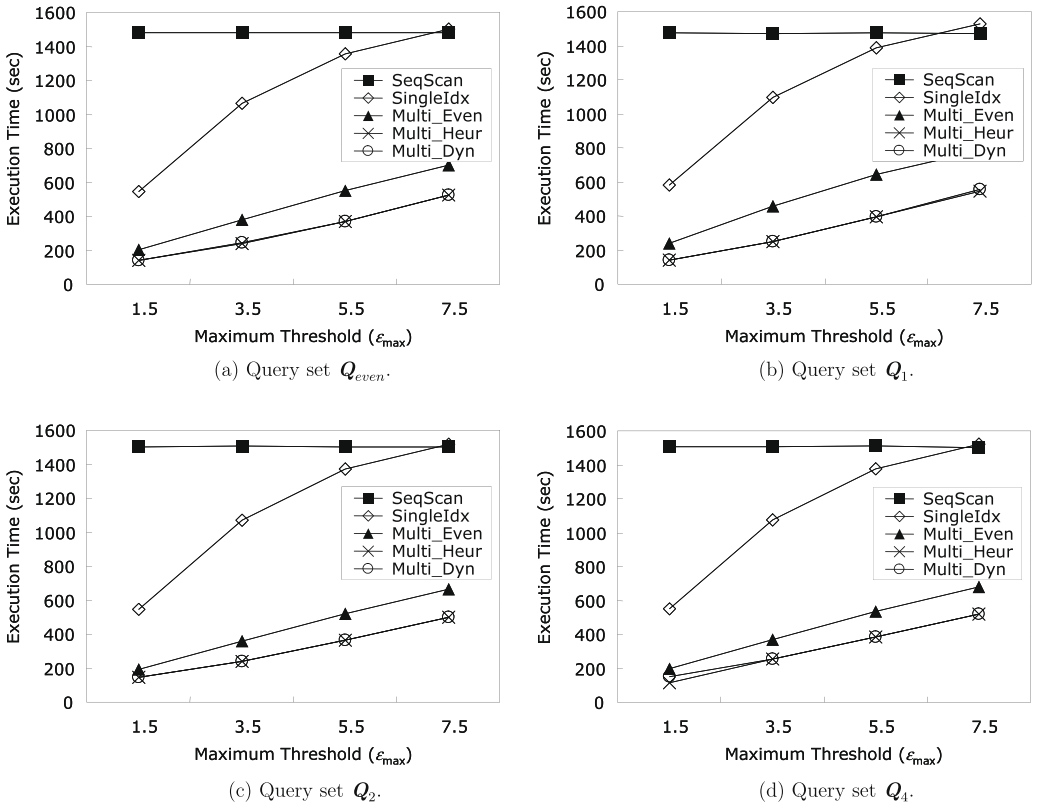


Fig. 11. Result of second experiment: comparison of performance while changing the maximum threshold ϵ_{max}.

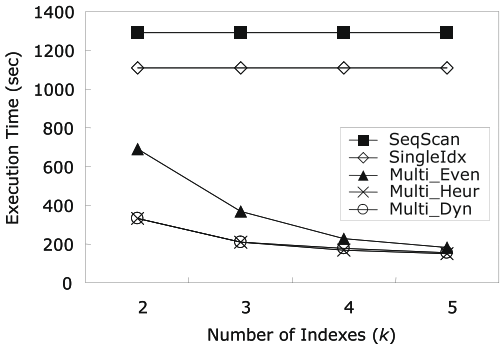
second experiment using the synthetic data. In the figures, the horizontal axis represents the maximum threshold ϵ_{max}, and the vertical axis represents the execution time of the algorithms. As in the first experiment, the *MultiIdx* algorithms outperform *SeqScan* and *SingleIdx*. The performance of the *MultiIdx* algorithms improve more as ϵ_{max} decrease. *Multi_Heur* outperforms *SeqScan*, *SingleIdx*, and *Multi_Even* by up to 13.2, 4.5, and 1.8 times, respectively. The maximum performance degradation of *Multi_Heur* compared with *Multi_Dyn* is as small as 2.4%.

Fig. 12a and b show the results of the first and second experiments using real-world data. Although we used different query sets, the performance trends in the figures are very similar to Figs. 10 and 11a, respectively. We omit the graphs using the other query sets $\boldsymbol{Q}_1$,$\boldsymbol{Q}_2$, and $\boldsymbol{Q}_4$ of real-world data, since they are also very similar to the graphs using synthetic data.

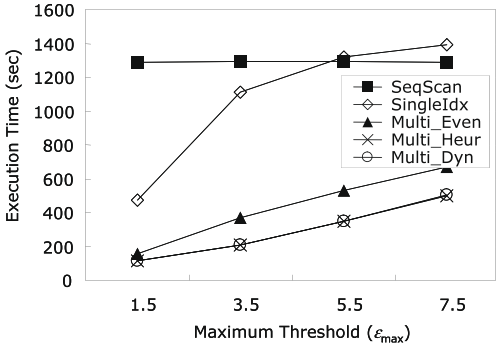
We perform the third experiment using the synthetic dataset consisting of up to 17,400 query sequences (600 query sequences for each of the query lengths). The query sequences are generated in the same way as $\boldsymbol{Q}_{even}$, and the search threshold ϵ_{max} and the number of indexes k are set as 3.5 and 3, respectively. Since the average length of query sequences is 576, the size of the dataset is almost 76 gigabytes ($\sim$17,400 $\times$ 576 $\times$ 8 bytes). Fig. 13 shows the results of the experiment. In the figure, the horizontal axis represents the total number of query sequences, and the vertical axis represents the execution time in the unit of second. Note that the vertical axis is plotted in the log scale. As indicated in the figure, the performance difference between the algorithms becomes larger, as the number of query sequences increases, because, by using multiple indexes, more non-relevant query sequences can be filtered out by our algorithm. According to this result, it can be concluded that our algorithm is suitable for the environment dealing with very large datasets.

In the first three experiments, we tried the early abandoning (or early stopping) technique [18,20] for the *SeqScan* algorithm; however, we could not obtain notable performance improvement due to the following reason. In general, when checking if two sequences Q and X of length n are within r distance from each other, the early abandoning technique compares the squared distance between $Q[0,i]$ and $X[0,i]$ with r^2 for each $i(0 \leqslant i < n)$. If the squared distance is larger than r^2 before i reaches n, it is certain that the distance between Q and X is larger than r, and hence the comparison stops. The earlier the comparison stops, the better the performance that the technique gains. However, in our experiments, since the normalization transform actually narrowed the distance distribution between streaming subsequences and query sequences, the comparison between them rarely stopped early. Along with the intrinsic overhead of comparing the distance for each i, the early abandoning technique did not provide any benefit in our experiments.

In the fourth experiment, we set $k = 1$ and $\epsilon_{max} = 3.5$, and observe the search performance of the *SingleIdx* algorithm while changing the prediction length m to 4,8,...,64. We denote the *SingleIdx* algorithm executed with a specific m value



(a) Query set $\boldsymbol{Q}_{even}$ (first experiment).



(b) Query set $\boldsymbol{Q}_{even}$ (second experiment).

Fig. 12. Result of first and second experiments using real-world data.

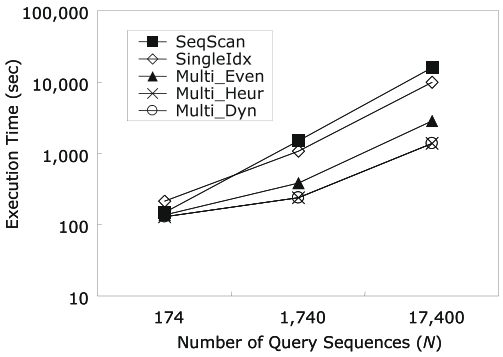
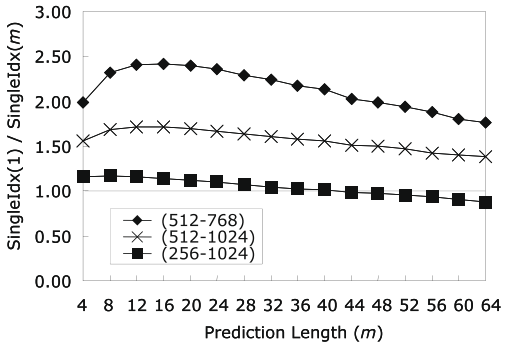
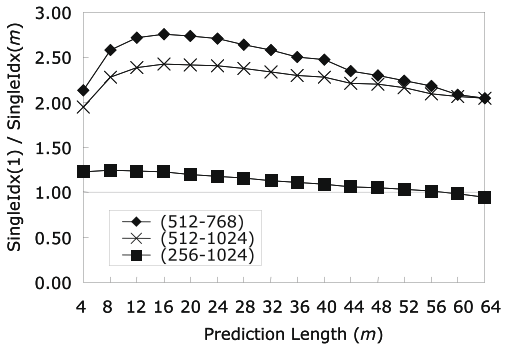


Fig. 13. Result of third experiment: comparison of performance on a very large query set.

as *SingleIdx*(m). We perform the experiment using three subsets of query sets $\boldsymbol{Q}_{even}$ and $\boldsymbol{Q}_1$, and the subsets consist of only the query sequences whose lengths are 256–1024, 512–1024, and 512–768, respectively. Fig. 14 shows the result of the fourth experiment. The horizontal axis represents the prediction length m, and the vertical axis represents the ratio of execution time of *SingleIdx*(1) divided by that of *SingleIdx*(m). As shown in the figure, the ratio increases quickly and then decreases slowly, as the m value increases. For the smaller m values, the search performance of *SingleIdx*(m) improves, since the algorithm retrieves the index only once during m time points. However, for the larger m values above a certain value, the search performance is degraded, since the false alarms increase due to the prediction error and the size of the index also increases. In Fig. 14, we have a smaller ratio for the query sets with the larger ranges of query sequence lengths, because the difference between the largest and the shortest query sequence lengths ($L - w_0$) increases, and it can be readily tackled by using more indexes. In the fourth experiment, *SingleIdx*(m) outperformed *SingleIdx*(1) by up to 2.76 times. We believe that the performance ratio would improve more with a larger number k of indexes.



(a) Query set Q_{even}.



(b) Query set Q_1.

Fig. 14. Result of fourth experiment: comparison of performance while changing the prediction length m.

8. Conclusions

In this paper, we proposed an algorithm for distortion-free predictive streaming time-series matching. Our algorithm is distortion-free in the sense that it deals with the preprocessed time-series to remove the distortions of offset translation and amplitude scaling at the same time. The preprocessing to remove these two distortions is known as normalization transform, and is useful for finding time-series that have similar fluctuation trends. Our algorithm is predictive in the sense that it performs streaming time-series matching against not only the current most recent subsequence but also the predicted most recent subsequences in the near future, and thus improves the search performance. The contributions of this paper are as follows. We proposed a simple algorithm for distortion-free streaming time-series matching, and extended the algorithm to use $k(\geqslant 1)$ indexes for improving the search performance. We further extended our algorithm so it can simultaneously obtain the search results for $m(\geqslant 1)$ time points from current time point t_0 to a time point $(t_0 + m - 1)$ in the near future by retrieving the index only once. Then, for a given number of indexes k, we presented an approximation method to construct k indexes for achieving the optimal search performance of our algorithm. Through a series of experiments, we compared the search performance of our simple and extended algorithms, and showed their performance trends according to k and m values. The experiment results showed that our algorithms outperformed the sequential scan algorithm by up to 13.2 times.

Acknowledgement

This work was partially supported by Defense Acquisition Program Administration and Agency for Defense Development under the contract. We would like to thank Professor Mohamed Mokbel at University of Minnesota for his helping in revising the earlier version of this paper.

Appendix A. Proof of Lemma 1

Proof. Let $f_w(\geqslant 0)$ be the frequency of query sequences of length $w(w_j \leqslant w \leqslant L)$. Then, the total search costs are $C_j(w) \cdot f_w$ and $C_0(w) \cdot f_w$ for our algorithm to find all the query sequences of length w using the indexes I_j and I_0, respectively. The average of search cost ratios is computed as follows:

$$R_j^f = \frac{\sum_{w=w_j \ldots L} (C_j(w) \cdot f_w)/(C_0(w) \cdot f_w)}{L - w_j + 1} = \frac{\sum_{w=w_j \ldots L} C_j(w)/C_0(w)}{L - w_j + 1} = R_j.$$

Since $\widehat{R}(w)$ in Eq. (9) is obtained using R_j, which is equal to R_j^f, $\widehat{R}(w)$ can also be obtained using R_j^f. Thus, Eq. (9) can be used even when there is an arbitrary number of query sequences of the same lengths. □

Appendix B. Proof of Lemma 2

Proof. We prove Lemma 2 in the same way as Lemma 1. Let $p_w (0.0 \leqslant p_w \leqslant 1.0)$ be the probability for query sequences of length $w (w_j \leqslant w \leqslant L)$ to be returned by our algorithm. The search costs are $C_j(w) \cdot p_w$ and $C_0(w) \cdot p_w$ for the algorithm to obtain the query sequences of length w using the indexes I_j and I_0, respectively. The average of the search cost ratios is computed as follows:

$$R_j^p = \frac{\sum_{w=w_j \ldots L} (C_j(w) \cdot p_w)/(C_0(w) \cdot p_w)}{L - w_j + 1} = \frac{\sum_{w=w_j \ldots L} C_j(w)/C_0(w)}{L - w_j + 1} = R_j.$$

Since $\widehat{R}(w)$ in Eq. (9) is obtained using R_j, which is equal to R_j^p, $\widehat{R}(w)$ can also be obtained using R_j^p. Thus, Eq. (9) can be used even when the query sequences do not have the same probabilities to be returned by our algorithm. □

References

[1] R. Agrawal, C. Faloutsos, A. Swami, Efficient similarity search in sequence databases, in: Proceedings of the International Conference on Foundations and Data Organization and Algorithm (FODO), Chicago, IL, 1993, pp. 69–84.
[2] R. Agrawal, K.-I. Lin, H.S. Sawhney, K. Shim, 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), Zurich, Switzerland, 1995, pp. 490–501.
[3] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), Madison, WI, June 2002, pp. 1–16.
[4] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R*-Tree: an efficient and robust access method for points and rectangles, in: Proceedings of the International Conference on Management of Data, ACM SIGMOD, Atlantic City, NJ, 1990, pp. 322–331.
[5] S. Berchtold, D.A. Keim, H.-P. Kriegel, The X-tree: an index structure for high-dimensional data, in: Proceedings of the International Conference on Very Large Data Bases (VLDB), Mumbai, India, 1996, pp. 28–39.
[6] K.-P. Chan, A.W.-C. Fu, Efficient time series matching by wavelets, in: Proceedings of the International Conference on Data Engineering (ICDE), IEEE, Sydney, Australia, 1999, pp. 126–133.
[7] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, E. Keogh, Querying and mining of time series data: experimental comparison of representations and distance measures, in: Proceedings of the VLDB Endowment (PVLDB), vol. 1(1), 2008, pp. 1542–1552.
[8] C. Faloutsos, M. Ranganathan, Y. Manolopoulos, Fast subsequence matching in time-series databases, in: Proceedings of the International Conference on Management of Data, ACM SIGMOD, Minneapolis, MN, 1994, pp. 419–429.
[9] A.J. Frost, R.R. Prechter, C.J. Collins, Elliott Wave Principle: Key to Market Behavior, John Wiley and Sons, 2001.
[10] L. Gao, X.S. Wang, Continually evaluating similarity-based pattern queries on a streaming time series, in: Proceedings of the International Conference on Management of Data, ACM SIGMOD, Madison, WI, 2002, pp. 370–381.
[11] L. Gao, Z. Yao, X.S. Wang, Evaluating continuous nearest neighbor queries for streaming time series via pre-fetching, in: Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM), McLean, VA, 2002, pp. 485–492.
[12] D.Q. Goldin, P.C. Kanellakis, On similarity queries for time-series data: constraint specification and implementation, in: Proceedings of the International Conference on Principles and Practices of Constraint Programming, Cassis, France, 1995, pp. 137–153.
[13] C. Hamzacebi, Improving artificial neural networks' performance in seasonal time series forecasting, Information Sciences 178 (23) (2008) 4550–4559.
[14] J. Han, M. Kamber, Data Mining: Concepts and Techniques, 2nd ed., Morgan Kaufman, 2005.
[15] E.J. Keogh, Exact indexing of dynamic time warping, in: Proceedings of the International Conference on Very Large Data Bases (VLDB), Hong Kong, China, 2002, pp. 406–417.
[16] E.J. Keogh, A decade of progress in indexing and mining large time series databases, in: Proceedings of the International Conference on Very Large Data Bases (VLDB), Tutorial, Seoul, South Korea, 2006, p. 1268.
[17] E.J. Keogh, The UCR Time Series Data Mining Archive, University of California, Riverside, 2006, <http://www.cs.ucr.edu/eamonn/TSDMA/index.html>.
[18] E.J. Keogh, S. Kasetty, On the need for time series data mining benchmarks: a survey and empirical demonstration, in: Proceedings of the International Conference on Knowledge Discovery and Data Mining, ACM SIGKDD, Edmonton, Canada, 2002, pp. 102–111.
[19] E. Keogh, J. Lin, A. Fu, HOT SAX: efficiently finding the most unusual time series subsequence, in: Proceedings of the International Conference on Data Mining (ICDM), Houston, TX, 2005, pp. 226–233.
[20] E.J. Keogh, L. Wei, X. Xi, S.-H. Lee, M. Vlachos, LB_Keogh supports exact indexing of shapes under rotation invariance with arbitrary representations and distance measures, in: Proceedings of the International Conference on Very Large Data Bases (VLDB), Seoul, Korea, 2006, pp. 882–893.
[21] S.-W. Kim, S. Park, W.W. Chu, Efficient processing of similarity search under time warping in sequence databases: an index-based approach, Information Systems 29 (6) (2004) 405–420.
[22] S.-H. Lim, H.-J. Park, S.-W. Kim, Using multiple indexes for efficient subsequence matching in time-series databases, in: Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA), Singapore, 2006, pp. 65–79.
[23] J. Lin, E. Keogh, S. Lonardi, B. Chiu, A symbolic representation of time series, with implications for streaming algorithms, in: Proceedings of the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD), San Diego, CA, 2003, pp. 2–11.
[24] J. Lin, E. Keogh, L. Wei, S. Lonardi, Experiencing SAX: a novel symbolic representation of time series, Data Mining and Knowledge Discovery (DMKD) 15 (1) (2007) 107–144.
[25] W.-K. Loh, S.-W. Kim, K.-Y. Whang, A subsequence matching algorithm that supports normalization transform in time-series databases, Data Mining and Knowledge Discovery (DMKD) 9 (1) (2004) 5–28.
[26] Y.-S. Moon, J. Kim, A single index approach for time-series subsequence matching that supports moving average transform of arbitrary order, in: Proceedings of the Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD), Singapore, 2006, pp. 739–749.
[27] Y.-S. Moon, K.-Y. Whang, W.-K. Loh, Duality-based subsequence matching in time-series databases, in: Proceedings of the International Conference on Data Engineering (ICDE), IEEE, Heidelberg, Germany, 2001, pp. 263–272.

[28] Y.-S. Moon, K.-Y. Whang, W.-S. Han, General match: a subsequence matching method in time-series databases based on generalized windows, in: Proceedings of the International Conference on Management of Data, ACM SIGMOD, Madison, WI, 2002, pp. 382–393.
[29] A. Mueen, E.J. Keogh, Q. Zhu, S. Cash, M.B. Westover, Exact discovery of time series motifs, in: Proceedings of the SIAM International Conference on Data Mining (SDM), Sparks, NV, 2009, pp. 473–484.
[30] C.-S. Perng, H. Wang, S.R. Zhang, D.S. Parker, Landmarks: a new model for similarity-based pattern querying in time series databases, in: Proceedings of the International Conference on Data Engineering (ICDE), San Diego, CA, 2000, pp. 33–42.
[31] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, Numerical Recipes C: The Art of Scientific Computing, second ed., Cambridge University Press, 1992.
[32] C. Ratanamahatana, E.J. Keogh, Three myths about dynamic time warping data mining, in Proceedings of the SIAM International Data Mining Conference (SDM), Newport Beach, CA, 2005, pp. 506–510.
[33] J. Shieh, E. Keogh, iSAX: indexing and mining terabyte sized time series, in: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, NV, 2008, pp. 623–632.
[34] J. Shieh, E.J. Keogh, iSAX: disk-aware mining and indexing of massive time series datasets, Data Mining and Knowledge Discovery (DMKD) 19 (1) (2009) 24–57.
[35] C.G. da Silva, Time series forecasting with a non-linear model and the scatter search meta-heuristic, Information Sciences 178 (16) (2008) 3288–3299.
[36] W.R. Stevens, S.A. Rago, Advanced Programming in the UNIX Environment, second ed., Addison-Wesley, 2005.
[37] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, E.J. Keogh, Indexing multi-dimensional time-series with support for multiple distance measures, in: Proceedings of the International Conference on Knowledge Discovery and Data Mining, ACM SIGKDD, Washington, D.C., 2003, pp. 216–225.
[38] L. Wei, E.J. Keogh, H.V. Herle, A. Mafra-Neto, Atomic Wedgie: efficient query filtering for streaming times series, in: Proceedings of the IEEE International Conference on Data Mining (ICDM), Houston, TX, 2005, pp. 490–497.
[39] L. Wei, E.J. Keogh, X. Xi, M. Yoder, Efficiently finding unusual shapes in large image databases, Data Mining and Knowledge Discovery (DMKD) 17 (3) (2008) 343–376.
[40] H. Wu, B. Salzberg, D. Zhang, Online event-driven subsequence matching over financial data streams, in: Proceedings of the International Conference on Management of Data, ACM SIGMOD, Paris, France, 2004, pp. 23–34.
[41] D. Yankov, E.J. Keogh, L. Wei, X. Xi, W.L. Hodges, Fast best-match shape searching in rotation-invariant metric spaces, IEEE Transactions on Multimedia (TMM) 10 (2) (2008) 230–239.
[42] L. Ye, E.J. Keogh, Time series shapelets: a new primitive for data mining, in: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), Paris, France, 2009, pp. 947–956.
[43] B.-K. Yi, C. Faloutsos, Fast time sequence indexing for arbitrary Lp norms, in: Proceedings of the International Conference on Very Large Data Bases (VLDB), Cairo, Egypt, 2000, pp. 385–394.
[44] B.-K. Yi, H.V. Jagadish, C. Faloutsos, Efficient retrieval of similar time sequences under time warping, in: Proceedings of the International Conference on Data Engineering (ICDE), Orlando, FL, 1998, pp. 201–208.
[45] G.P. Zhang, A neural network ensemble method with jittered training data for time series forecasting, Information Sciences 177 (23) (2007) 5329–5346.