Efficient Subsequence Matching Using the Longest Common Subsequence with a Dual Match Index

Tae Sik Han[1], Seung-Kyu Ko[1,⋆], and Jaewoo Kang[2,⋆⋆]

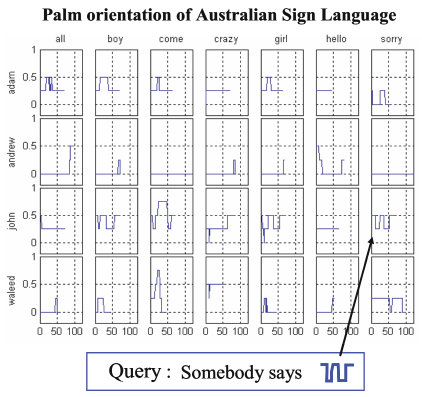
[1] Dept. of Computer Science, North Carolina State University
Raleigh, NC 27569, USA
[2] Dept. of Computer Science and Engineering, Korea University, Seoul 136-705, Korea
kangj@korea.ac.kr

Abstract. The purpose of subsequence matching is to find a query sequence from a long data sequence. Due to the abundance of applications, many solutions have been proposed. Virtually all previous solutions use the Euclidean measure as the basis for measuring distance between sequences. Recent studies, however, suggest that the Euclidean distance often fails to produce proper results due to the irregularity in the data, which is not so uncommon in our problem domain. Addressing this problem, some non-Euclidean measures, such as *Dynamic Time Warping (DTW)* and *Longest Common Subsequence (LCS)*, have been proposed. However, most of the previous work in this direction focused on the whole sequence matching problem where query and data sequences are the same length. In this paper, we propose a novel subsequence matching framework using a non-Euclidean measure, in particular, *LCS*, and a new index query scheme. The proposed framework is based on the Dual Match framework where data sequences are divided into a series of disjoint equi-length subsequences and then indexed in an R-tree. We introduced similarity bound for index matching with *LCS*. The proposed query matching scheme reduces significant numbers of false positives in the match result. Furthermore, we developed an algorithm to skip expensive *LCS* computations through observing the warping paths. We validated our framework through extensive experiments using 48 different time series datasets. The results of the experiments suggest that our approach significantly improves the subsequence matching performance in various metrics.

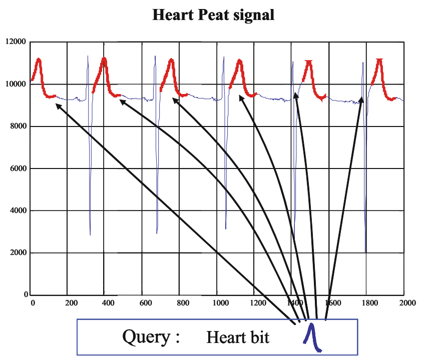
Keywords: Subsequence matching, Longest Common Subsequence, Dual Match.

⋆ He was supported by the IT Scholarship Program supervised by Institute for Information Technology Advancement and Ministry of Information and Communication in Republic of Korea.

⋆⋆ Corresponding author. His work was partially supported by the Microsoft Bioinformatics Award and the Korea University Research Grant.



(a) Whole sequence matching



(b) Subsequence matching

Fig. 1. Whole sequence matching and Subsequence matching

1 Introduction

One of the basic problems in handling time series data is locating a pattern of interest from the long sequence of input data [1,2,7]. The sequence matching problem is largely classified into two categories: whole sequence matching and subsequence matching. Whole sequence matching involves finding, from the dataset, all sequence entries whose lengths are equal to the query and that fall within the similarity threshold specified by the user. For example, Figure 1(a) illustrates the whole sequence matching using the sign language palm orientation example. It shows the palm orientation readings from four different people (rows) using Australian Sign Language saying seven different words (columns)[4]. Each word from different signers has the same length and is searched for a given query.

Subsequence matching finds all subsequences from a longer data sequence that matches to the query. Figure 1(b) shows an example. It shows a short query sequence, one heart beat signal, and all matching regions from the longer data sequence. Subsequence matching is a more general problem than the whole sequence matching problem. However, most of the previous work has focused on the whole sequence matching problem [1,5,11]. While applying whole sequence matching techniques to the subsequence matching can be possible through GEMINI [2] framework, the application is not straightforward when non-Euclidean distance measures are used. Euclidean measure is sensitive to noise and due to the irregular nature of the data in sequence applications (e.g., moving object trajectories, query-by-humming, etc.), non-Euclidean measures are often desirable. The non-Euclidean distance measures such as *DTW*(*Dynamic Time Warping*) and *LCS*(*Longest Common Subsequence*) address some of the problems that Euclidean measure has [5,10].

In this work, we propose an efficient index searching framework for subsequence matching using *LCS*. We choose *LCS* because it is known to be more robust to the noise in the data than DTW [3,9] and yet to the best of our knowledge no previous work has considered it in the context of subsequence matching. We made the following contributions:

– We proposed a subsequence matching framework that employs a non-Euclidean distance measure *LCS*. It is for a more intuitive matching performance.
– We formally introduced the criteria for pruning the search space when using time series index with *LCS* similarity function.
– We introduced a new index query scheme, *multiple window sliding*, where several adjacent windows are queried and aggregated in order to improve the query performance.
– We proposed a new index search scheme that enables us to skip unnecessary similarity computations for the consecutive matching subsequences.

2 Background and Related Work

2.1 Notational Convenience

In order to state the problem and concepts clearly, we define some notations and terminologies in Table 1. In our work, we assume that a time series is a totally ordered set of real numbers and each real number element is collected from a single channel sensor device. A subsequence is a subset of a time series in contiguous time stamps.

Table 1. The basic notation

B	A time series data sequence, $< b_1, b_2, \ldots >$, each b_i is a real number at the i^{th} time stamp.
$\|B\|$	Length of the sequence B
B_i	The i^{th} subsequence of B when B is divided into disjoint subsequences of an equal length
Q	A query sequence, usually $\|Q\| \ll \|B\|$
$B[i:j]$	A subsequence of B from time stamp i to j

2.2 Subsequence Matching Framework (DualMatch vs. FRM)

There are at least two subsequence matching frameworks, FRM [2][1] and Dual Match [7]. Both of the matching processes are illustrated in Figure 2. Let n be the number of data points and w be the size of an index window. In FRM, the data sequence is divided into $n - w + 1$ sliding windows. Figure 2(a) shows the FRM indexing step. Every window is overlapped with the next window except the first data point. Whereas, query Q is divided into disjoint windows (Figure 2(b)), and each window is to be matched against the sliding windows of

[1] It is named after its authors.

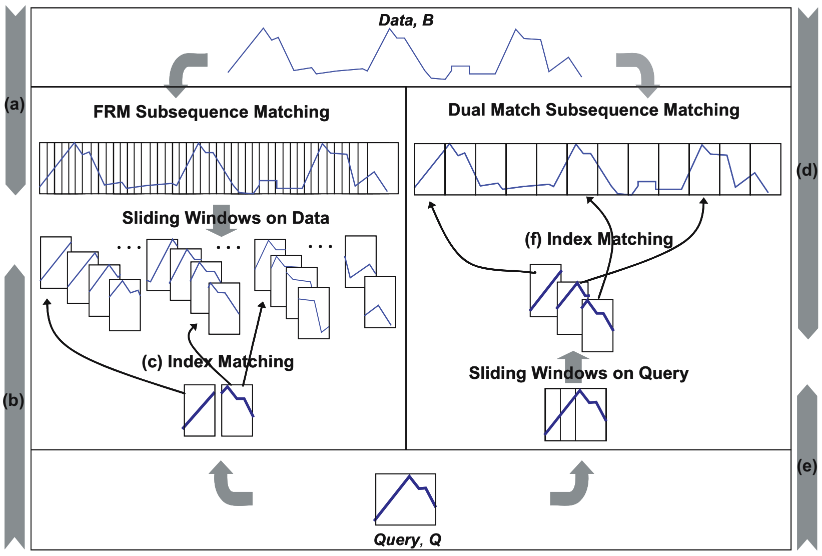
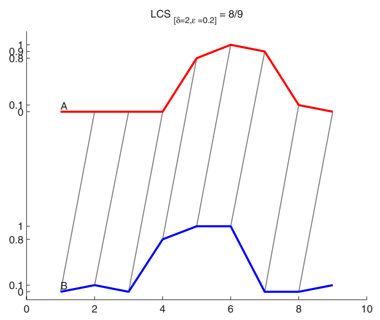


Fig. 2. Two Subsequence Matching Frameworks

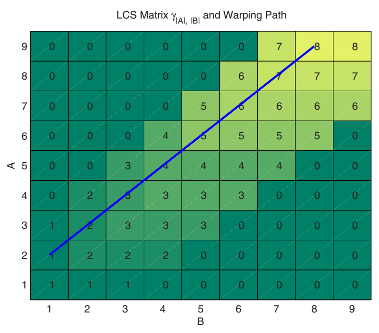
the data sequence (Figure 2(c)). On the other hand, in Dual Match framework, data sequence is divided into disjoint windows (Figure 2(d)), and part of the query in its sliding window is matched to the data indices (Figure 2(e) and 2(f)). Since the Dual Match does not allow any overlap of the index windows, it needs less space for the index and, in consequence, index searching is faster than FRM. Through the index matching, we get a set of candidate matches and the actual similarity or distance is computed for them. Since the length of the data is usually very long, Dual Match framework reduces the indexing efforts. We employ the Dual Match as our indexing scheme.

2.3 Dual Match Subsequence Matching with Euclidean Distance

Dual Match consists of three steps. First, in the indexing step, data is decomposed into disjoint windows and each window is represented by a multi-dimensional vector. They are stored in a spatial index structure like R-tree. Second, query sequence is decomposed into a set of sliding windows and each window is transformed into the same dimensional vector representation as the index window. The size of the sliding window is the same as that of the index window. It is proven that if the length of the query is longer than twice of the index length, one of the sliding windows in the query is guaranteed to match to a data index that belongs to a subsequence that matches to the query [7]. The index matching always returns a super set of the true matching intervals since the similarity of the index and query sliding window is always larger than the similarity of the true match. Lastly, based on the positions of the matching sliding windows, whole matching intervals are decided and actual similarities are computed.



(a) Sequence A, B and warping path



(b) Sakoe-Chiba band in *LCS* warping path martix

Fig. 3. An example of LCS computation

2.4 A Non-euclidean Distance *LCS*

Non-Euclidean similarity measures such as *LCS* and *DTW* are useful to match two time series data when the data has irregularity. The *LCS* is known to be robust to the noise since it does not count the outliers in the sequence that fall out of the range (ϵ). Both use the same dynamic programming procedure to compute the optimal warping path within the time interval (δ). We chose *LCS* as our distance function and its definition is given below.

Definition 1. *[10] Let Q=$< q_1, q_2, ..., q_n >$ be a query and B=$< b_1, b_2, ..., b_n >$ be a data subsequence of time series. Given an integer δ and a real number 0 $< \epsilon <$1, we define the cumulative similarity $\gamma_{i,j}(Q, B)$ or $\gamma_{i,j}$ as*

$$\gamma_{i,j} = \begin{cases} 0, & \text{if } i, j = 0 \\ 1 + \gamma_{i-1,j-1} & \text{if} |q_i - b_j| \leq \epsilon \\ & \text{and } |i - j| \leq \delta \\ max(\gamma_{i,j-1}, \gamma_{i-1,j}) & \text{otherwise} \end{cases}$$

and using that, LCS similarity with δ and ϵ as

$$LCS_{\delta,\epsilon}(Q, B) = \gamma_{|Q|,|B|}$$

$LCS(Q, B)$ returns an integer between 0 and $min(|Q|, |B|)$. δ is the allowable matching interval in the time dimension and ϵ is the allowable error bound in the data value dimension. Here is an example of *LCS* match for the two sequences A and B of the same length where A = <0, 0, 0, 0, 0.8, 1, 0.9, 0.1, 0> and B = <0, 0.1, 0, 0.8, 1, 1, 0, 0, 0.1>. Figure 3(a) shows the *LCS* warping path. Figure 3(b) shows the *LCS* computation process in the *LCS* warping path matrix. It is constructed by dynamic programming of the cumulative similarity $\gamma_{|A|,|B|}$. The non-zero boxes in light color in the *LCS* warping path matrix of Figure 3(b) is called a Sakoe-Chiba band [8].

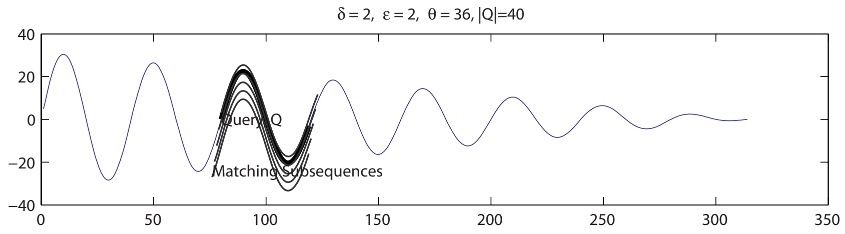


Fig. 4. Matching subsequences in subsequence matching

3 Problem Statement

The purpose of the subsequence matching is to find subsequences similar to the given query sequence. Subsequence matching framework with Euclidean distance has been already developed as we stated in the previous section. However, to the best of our knowledge, many things have not yet been considered when we apply non-Euclidean function to the subsequence matching. We need to improve the index search performance and we need to provide an index matching criteria that avoids expensive computation caused by non-Euclidean measures.

In order to describe what should be the output of the subsequence matching, we define matching subsequences for a query sequence Q in terms of $LCS_{\delta,\epsilon}$.

Definition 2. *Let* $Q = < q_1, q_2, ...q_m >$ *be a query and* $B = < b_1, b_2, ...b_n >$ *be a data subsequence of time series. Given an integer* δ*, a real number* $0 < \epsilon < 1$ *and user defined similarity threshold* θ*, we define the* ***matching subsequences****,* $M = \{B[i : j] \mid LCS_{\delta,\epsilon}(Q, B[i : j]) \geq \theta\}$

There may be many overlapping subsequences in the same region that exceed the similarity threshold θ. We restrict the scope of our work to find only the longest possible matching subsequences of the length $|Q| + 2\delta$. We do not return all matching subsequences that are properly contained in the longest possible one returned. It could be prohibitively expensive to find all matches of all lengths using a non-Euclidean measure. It makes sense to return only the longest matching subsequences since it contains all matching subsequences shorter than $|Q|+2\delta$ in the region. It is possible to search shorter matching subsequences, if needed, after the search process for the longest ones completes. In Figure 4, all the matching subsequences of size $|Q| + 2\delta$ are visualized in grey dotted lines.

Formally, our problem is defined as follows:Find all matching subsequences $B[i : j]$ of length $|Q| + 2\delta$ for data sequence B and query Q such that the similarity $LCS_{\delta,\epsilon}(Q, B[i : j])$ is no less than $s\%$ of the $|Q|$, $\frac{s}{100}|Q|$.

4 Subsequence Matching with *LCS*

4.1 Linear Search and Skipping *LCS* Computation

A straighforward approach to the subsequence matching is comparing the query subsequence Q to all of the candidate subsequences of the data sequence B in

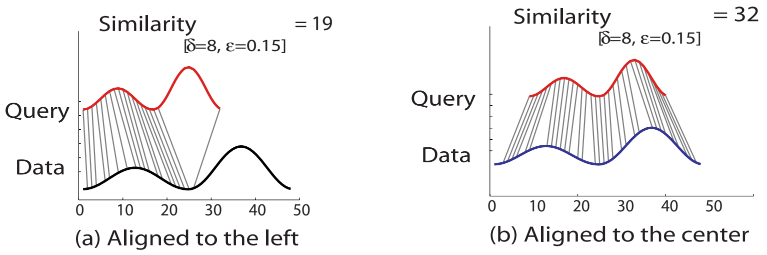


Fig. 5. Alignment with LCS when $|Query| = 32$ and $|Data| = 48$

a sequential manner. All the candidates can be chosen by sliding a fixed size window along the data sequence.

Alignment in *LCS*. When we compare query Q to a candidate data subsequence of length $|Q| + 2\delta$, we align the query in the middle of each candidate as illustrated in Figure 5(b). In the case of the whole sequence matching, alignment is not a problem since the query and data have the same length. However, in our subsequence matching, we need to locate the query in the candidate subsequence. If we align the query to the left side of a candidate, we may find a correct subsequence. In Figure 5(a), shorter query is not matched well to the longer data when aligned to the left. The right side of the query cannot be compared with the data since the δ is not big enough to cover all the matching points in the data. Larger δ increases the computational complexity of the matching process. Figure 5(a) shows that the query is correctly matched with the same δ when properly aligned.

Skipping *LCS* Computation. We can avoid expensive similarity computations of the adjacent subsequences by exploiting the LCS warping path and the local constraint such as the Sakoe-Chiba band. In the subsequence matching, we can think of the computation matrix as a moving window along the data sequence as shown in Figure 6.

Let us take a look at an example. Assume that $|Q| = 4$ and the user wants to find all the subsequences whose similarity is larger than or equal to 3. Figure 6(a) shows the LCS warping path which is represented as a set of arrows. In this case, $LCS(Q, B[1:6]) = 4$. Darker cells represent the Sakoe-Chiba band.

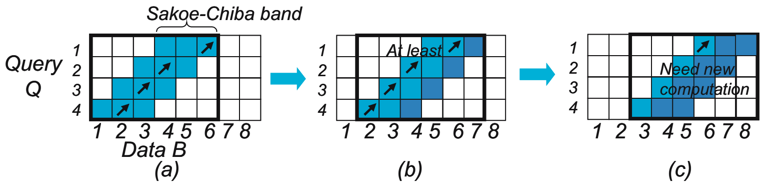


Fig. 6. An example of skipping LCS computation when $|Q| = 4$ and $\delta = 1$

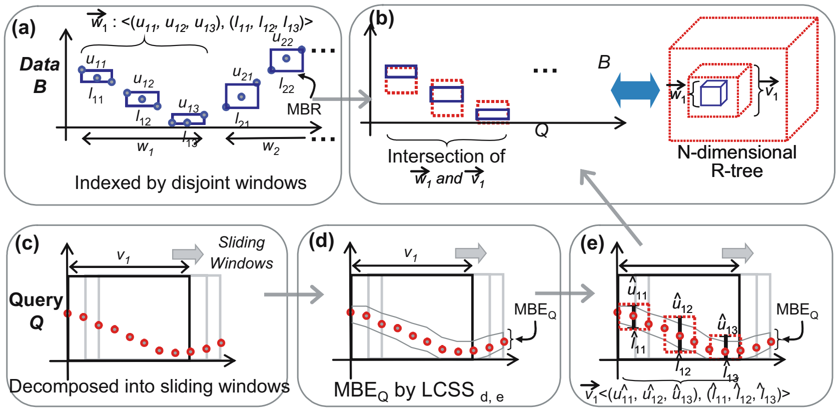

(a)
Data B
$\vec{w}_1$: <(u_{11}, u_{12}, u_{13}), (l_{11}, l_{12}, l_{13})>
MBR
w_1
w_2
Indexed by disjoint windows
(b)
B
Q
Intersection of $\vec{w}_1$ and $\vec{v}_1$
N-dimensional R-tree
(c)
Query Q
Sliding Windows
v_1
Decomposed into sliding windows
(d)
MBE$_Q$
MBE$_Q$ by LCSS$_{d, e}$
(e)
$\vec{v}_1$<($\hat{u}_{11}$, $\hat{u}_{12}$, $\hat{u}_{13}$), ($\hat{l}_{11}$, $\hat{l}_{12}$, $\hat{l}_{13}$)>

Fig. 7. Indexing and Index Matching where w=9 and N=3

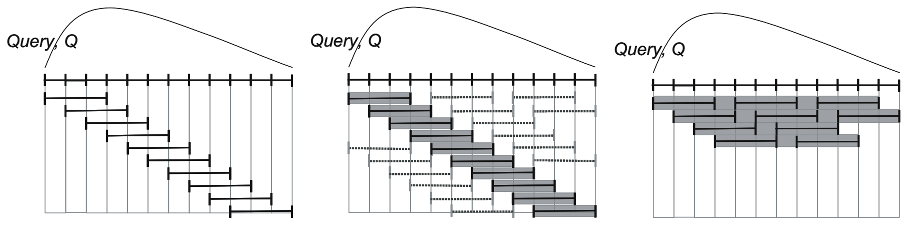
In Figure 6(b), we move a sliding window by a time stamp. The Sakoe-Chiba band still includes the warping path. In this case, we don't have to compute the $LCS(Q, B[2:7])$ since the dynamic programming finds a maximum warping path in the Sakoe-Chiba band and $LCS(Q, B[2:7])$ must be larger than or equal to 4. In Figure 6(c),we need to compute $LCS(Q, B[3:8])$ since the first three warping steps now became invalid.

We can skip the computation of a sliding window by tracing the warping path. If we find that the Sakoe-Chiba band of the current LCS matrix includes the previous warping path more than or equal to the user defined threshold, then we can skip the LCS computation. The skipping goes until a Sakoe-Chiba band includes warping path less than the user defined threshold. It is a useful property to reduce the expensive similarity computation in the subsequence matching where the adjacent window usually has a similar similarity value.

4.2 Index Match

Indexing enables us to avoid unecessary similarity computations for true-negative candidates for subsequence matching. In order to do that, we compute the pruning criteria to choose candidate matching subsequences with LCS. We also introduce in this section a new framework to efficiently search the index.

Indexing. Data is divided into equi-length disjoint windows for indexing. Each window is then represented as a multi-dimensional vector. That is, data sequence B is divided into equi-length disjoint windows $< w_i >$. Let N be the dimensionality of the space we want to have indexed. An MBR, $Minimum Bounding Rectangle$, represents a dimension. N $MBRs$ for a w_i, are transformed into $\vec{w_i} =< (u_{i1}, \ldots, u_{iN}), (l_{i1}, \ldots, l_{iN}) >$,where u_{ij} and l_{ij} represent the maximum and minimum values in the j^{th} interval of w_i. $\vec{w_i}$ is stored in an N dimensional R-tree. An example is illustrated in Figure 7(a). In the figure, the data in the first window, $w_1 =< b_1, ..., b_9 >$ is transformed into $\vec{w_1} =< (u_{11}, u_{12}, u_{13}), (l_{11}, l_{12}, l_{13}) >$. It is stored in an R-tree as showin in Figure 7(b).



(a) Naive Single Window (b) Single Window (c) Multiple Window

Fig. 8. Window sliding schemes when $|v|$=4

Index Matching with *LCS*. Query Q is compared first to the index. Q is transformed into an *MBE, Minimum Bounding Envelope,* with $LCS_{\delta,\epsilon}$ function as illustrated in Figure 7(d). Let MBE_Q be an MBE for Q. Let the i^{th} sliding window of Q be v_i. It is transformed into $\overrightarrow{v_i} =< (\hat{u}_{i1}, \ldots, \hat{u}_{iN}), (\hat{l}_{i1}, \ldots, \hat{l}_{iN}) >$, where $\hat{u}_{ij}$ and $\hat{l}_{ij}$ are the maximum and minimum values respectively in MBE_Q of the j^{th} MBR of the v_i. This is illustrated in Figure 7(e). Since MBE_Q covers the whole possible matching area, any point that lies outside the MBE_Q is not counted for the similarity. The number of intersecting points between B and MBE_Q provides the upperbound for $LCS_{\delta,\epsilon}(B, Q)$ [10]. The number of intersections is counted through the R-tree operation as shown in Figure 7(b), which is the intersection of Figure 7(a) and Figure 7(e).

4.3 Window Sliding Schemes in Index Matching

There are three ways to slide query windows and choose the candidate matching subsequences: Naive Single Window Sliding, Single Window Sliding and Multiple Window Sliding. We explain each window sliding scheme and show how the the bounding similarity is computed.

Naive Single Window Sliding. In this scheme, as illustrated in Figure 8(a), we compare a sliding window of a query to index, which is first introduced in the Dual Match [6]. This overestimation method cannot be applied to the LCS based subsequence matching since it is based on the Euclidean distance. We should consider δ on both ends of the query sliding window. In Figure 9 (a), a sliding window v of a query Q is matched to a window w of the data sequence B. In actual index matching, near the ends of the point of the Q cannot be matched to the points of w as in Figure 9 (b). The data is just indexed by MBR that does not consider δ time shift.

We compute the similarity threshold for the naive single window sliding method.

Let v be a sliding window of Q. The minimum similarity, θ is

$$\theta = |v| - (|Q| - \frac{s}{100}|Q|) - 2\delta \quad (1)$$

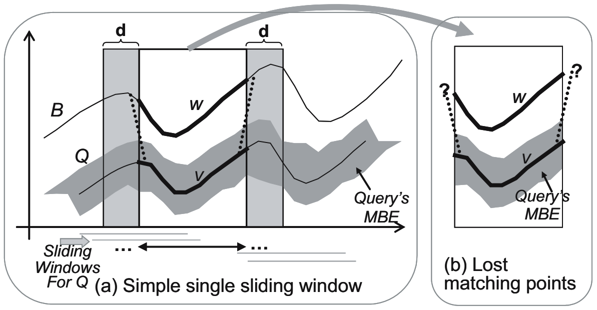


Fig. 9. Matching points not captured in the index matching using *LCS*

The term, $(|Q| - \frac{s}{100}|Q|)$ for the Equation (1) is subtracted from $|v|$ when all the mismatches can be found in the current window v. The last term 2δ is the maximum possible number of the lost matching points.

Single Window Sliding. When the query length is long enough to contain more than one sliding window, we can use the consecutive matching information as in Figure 8(b). Assume query Q and matching data subsequence B has M consecutive disjoint windows, B_i's and Q_i's. If some Q_i and B_i pairs are not similar, then the other Q_j and B_j pairs should be similar and we can recognize the B and Q pair is a candidate through B_j and Q_j. When all B_i and Q_i pairs have the same similarities, we should have the minimum value to decide the candidate for comparison. The *multiPiece* search [2] is proposed to choose candidates through this process. The same applies for the Euclidean distance measure. In the *multiPiece*, the two subsequences, B and Q, of the same length are given and each can be divided into p subsequences each of which has length l. $d(B,Q) < \epsilon \Rightarrow d(B_i,Q_i) < \frac{\epsilon}{\sqrt{p}}$ for some $1 \leq i \leq p$ where B_i, Q_i are i^{th} subsequence of the length l and $\epsilon > 0$. In the case of the Dual Match using Euclidean distance, we can count a candidate if the distance is less than or equal to $\frac{\epsilon}{\sqrt{p}}$.

Similarly, in the case of LCS, $LCS_{\delta,\epsilon}(B,Q) > \frac{s}{100}|Q| \Rightarrow LCS_{\delta,\epsilon}(v, Q[i:j]) > \frac{M|v|-(|Q|-\frac{s}{100}|Q|)-2\delta}{M}$ for some $j-i+1 = |v|$. So the similarity threshold for single window sliding, θ_s is

$$\theta_s = |v| - \frac{(|Q| - \frac{s}{100}|Q|) + 2\delta}{M} \tag{2}$$

As illustrated in Figure 8(b), M consecutive sliding windows are thought to be one big sliding window that might lose warping path at both ends. The threshold for the M sliding windows is $M|v| - (|Q| - \frac{s}{100}|Q|) - 2\delta$ and it is divided by M for one sliding window. If one of the sliding windows among consecutive M sliding windows in Q is larger than or equal to θ_s, we can get a candidate and we don't have to do index matching for the remaining consecutive sliding windows at the same candidate location.

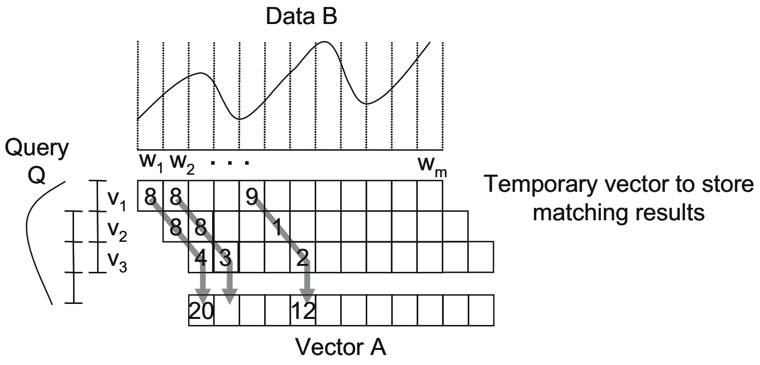


Fig. 10. Index matching result

Multiple Window Sliding. In this new window sliding scheme, as illustrated in Figure 8(c), the matching results of consecutive sliding windows in a query are aggregated. If we sum up the index matching result from M consecutive sliding windows, we can further reduce false positives. Let M be the number of consecutive windows fitted in a query Q. We vary M to contain the maximum number of sliding windows depending on the left most window.

The index matching results of each sliding window for all disjoint data windows are added up to get M consecutive sliding windows. In Figure 10, the aggregation is done by accumulating the results in a vector A of the size $\frac{|B|}{w}$. B is the data sequence and w is the length of an index window. Assume that $< v_1, \ldots, v_M >$ is a series of consecutive windows in the query Q. The index matching results of a query window v_j is placed in a temporary row vector in Figure 10. It is added to A and A is shifted to the right. The next matching result for v_{j+1} is placed in the temporary row vector. It is added to A and A is shifted right. In Figure 10, we get A such that

$$\begin{aligned}
&A[1] = LCS_{\delta,\epsilon}(\overrightarrow{v_1}, \overrightarrow{w_1}) + LCS_{\delta,\epsilon}(\overrightarrow{v}_2, \overrightarrow{w}_2) + LCS_{\delta,\epsilon}(\overrightarrow{v}_3, \overrightarrow{w}_3),\\
&A[2] = LCS_{\delta,\epsilon}(\overrightarrow{v_1}, \overrightarrow{w_2}) + LCS_{\delta,\epsilon}(\overrightarrow{v}_2, \overrightarrow{w}_3) + LCS_{\delta,\epsilon}(\overrightarrow{v}_3, \overrightarrow{w}_4), \ldots\\
&A[m] = LCS_{\delta,\epsilon}(\overrightarrow{v_1}, \overrightarrow{w_{m-2}}) + LCS_{\delta,\epsilon}(\overrightarrow{v}_2, \overrightarrow{w}_{m-1}) + LCS_{\delta,\epsilon}(\overrightarrow{v}_3, \overrightarrow{w}_m).
\end{aligned}$$

The shift operations aggregate the consecutive index matching results.

The similarity threshold for multiple sliding windows, θ_m, is computed as if the consecutive M windows move together like one big window.

$$\theta_m = M|v| - (|Q| - \frac{s}{100}|Q|) - 2\delta \tag{3}$$

θ_m is for an aggregate comparison of M consecutive sliding windows while θ_s is for one sliding window.

Through the aggregation of the consecutive index matching information, we can enhance the pruning power of the index. That is, we have less false alarms than the single window sliding scheme. In Figure 10, the diagonal sum illustrates the aggregatation of the consecutive index matching results. If $\theta_s = 8$, the first,

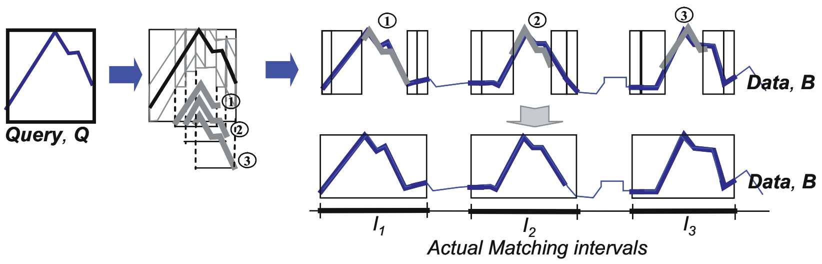


Fig. 11. Postprocessing to find whole length of the candidate matching subsequences

second and the fifth diagonals are selected as the candidates since one of the matches is greater than or equal to 8. However, in case of the multiple window sliding, if the $\theta_m = 20$, the fifth diagonal is not a candidate since the sum 12 is less than 20, so it has less false alarms than the single window sliding scheme.

Post-Processing. Post-processing is the final procedure to decide the whole length of the matching subsequence depending on the position of the matching sliding window and matching index. The actual similarity computation is done for the whole interval of the subsequence against the query. Figure 11 demonstrates the postprocessing. We intensionally omit the adjacent matching subsequences and show only one matching. Through the index matching process, matching indexes for each sliding window ①, ②, ③ are to be found and then whole length of the candidate subsequence is computed including 2δ area. In Figure 11, one candidate subsequence has an index matching area and a four box area.

Skipping *LCS* computation. After deciding the whole length of the candidate subsequences, skipping *LCS* computation is applied to reduce the computational load. Subsequence matching cannot avoid many adjacent matching subsequences where one subsequence is found. By tracing the warping path of the matching subsequences in its *LCS* warping path matrix, we can reduce the *LCS* computation.

5 Experiment

Experiments were conducted on a machine with 2.8 GHz pentium 4 processor and 2GB Memory using Matlab 2006a and Java. Here are the parameters to run the tests.

– **Dataset.** We used 48 different time series datasets[2] for evaluation. Each dataset has a different length of data and a different number of channels.

[2] http://www.cs.ucr.edu/ eamonn/TSDMA/UCR, The UCR Time Series Data Mining Archive.

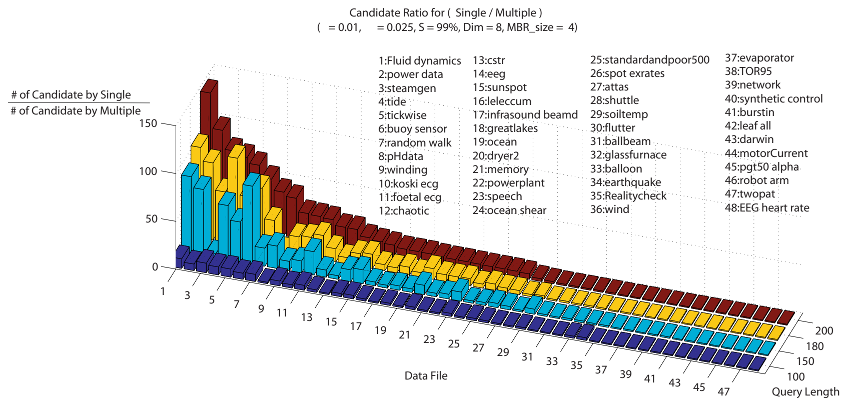


Fig. 12. Candidates generated by single window sliding and multiple window sliding

We set the length of each to 100,000 by attaching the beginning to the end so that all the datasets have the same length.

– **Index.** We set the dimension to 8 and *MBR* size to 4. Regarding the parameters to index dataset such as dimension, *MBR* and R-tree size need domain knowledge.
– **Query.** We choose 4 fixed length of queries, 100, 150, 180 and 200 so that each length includes 3,4,5 and 6 windows. 10 queries for each length are randomly selected from the data sequence.
– **Similarity.** ϵ is set to 1 % of the data range, δ is 2.5 % of the $|Q|$. Similarity threshold s is set to 99%.

5.1 Different Sliding Schemes and Candidates

We compare the performance of the two different index sliding schemes : single window sliding and multiple window sliding scheme. Figure 12 shows that the ratios, $\frac{\text{\# of candidates by single windows sliding}}{\text{\# of candidates by multiple windows sliding}}$ for different lengths of queries of each dataset. Ratios greater than one means that the multiple window sliding scheme generates less candidates than those of the single window sliding scheme. The multiple window sliding scheme has less false alarms than the single window sliding scheme in the tests. The ratio varies from 1 to 140. Multiple sliding window generates only $\frac{1}{140}$ of the single window sliding scheme in the Fluid dynamics dataset. Figure 13 shows the median values from the Figure 12 for each length of the queries. Figure 13 summarizes how much the performance is improved as the length of query gets longer in all of the datasets. It demonstrates that as the length of a query gets longer to include more index windows, we have less false alarms in the multiple window sliding than in the single window sliding.

However in the datasets such as EEG heart rate, two pat or robot arm, there is not much difference between the two methods. We can explain it in terms of the index. For these datasets, all of the disjoint data windows are very similar to each other. Figure 14 shows the first 500 points index of the best and the worst

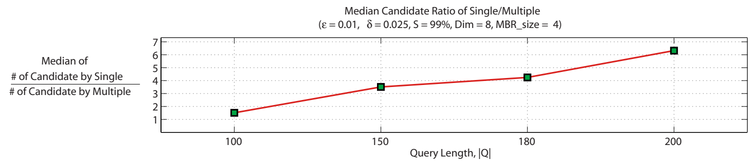


Fig. 13. Summary of Candidate generated in Figure 12

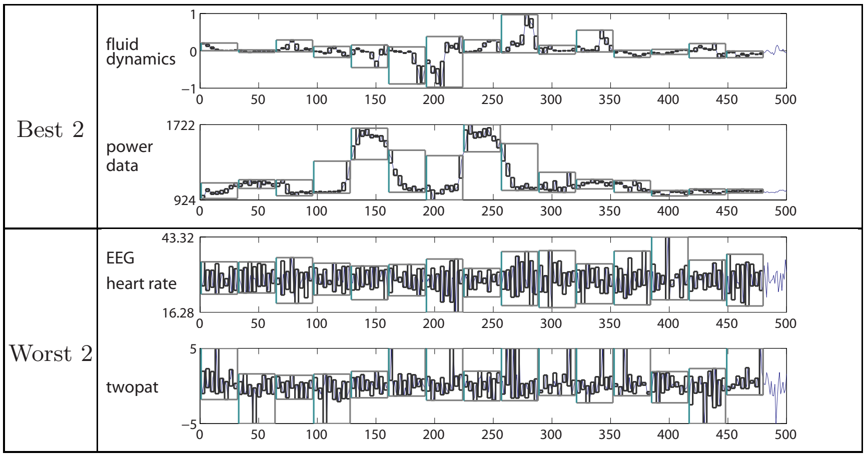


Fig. 14. Index

three datasets regarding the candidate generation. Comparing the index of the top three datasets to the bottom three, we cannot easily distinguish one window from another. It makes hard to search the index quickly even though multiple index information is used.

5.2 Goodness and Tightness

Goodness and tightness are metrics that shows how well the index works [5].

$$Goodness = \frac{\#\text{ of all true matches}}{\#\text{ of all candidates}}\ ,Tightness = \frac{\text{Sum of all true similarity}}{\text{Sum of all estimated similarity}} \tag{4}$$

Goodness shows how much the index reduces the expensive computations. Tightness shows how the estimated values are close to the actual values in indexing [5]. If the tightness is 1.0 then it means estimation is perfect. In Figure 15, the multiple sliding window scheme shows higher goodness and tightness than that of the single window sliding scheme.

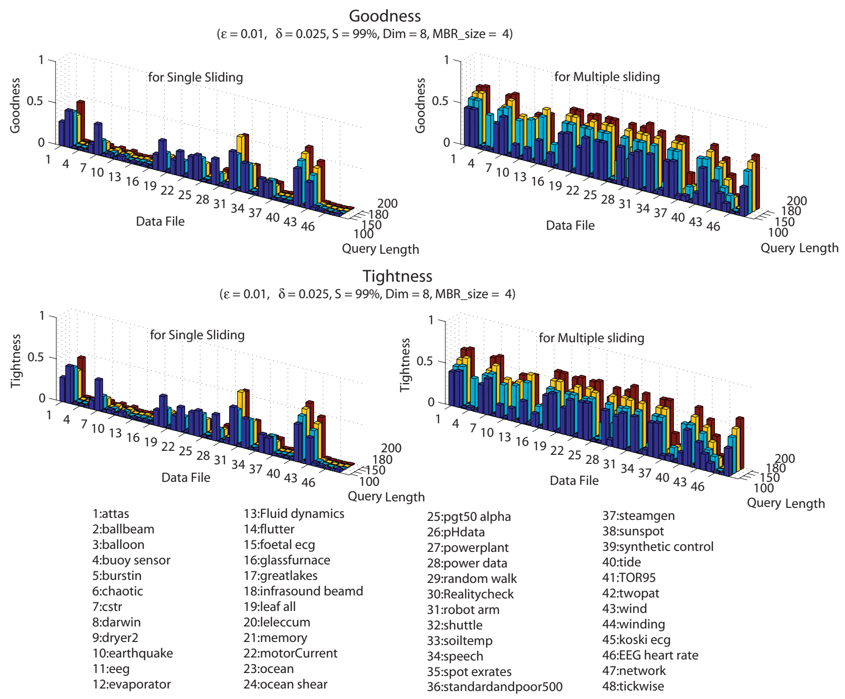


Fig. 15. Goodness and Tightness

5.3 Improving Performance by Skipping Similarity Computations

Figure 16 shows how the skipping of the similarity computation is effective. The chart shows that we can avoid many similarity computations as the length of the query gets longer.

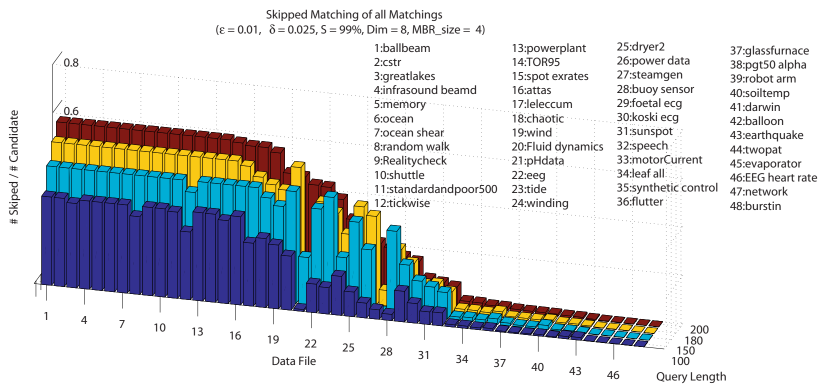


Fig. 16. Skipping Similarity Computations

However it also shows that the skipping mechanism does not work well for the datasets that cannot be properly indexed, since the index parameter captures all of the windows in the data as well as the ones similar to the *LCS* matrix.

6 Conclusion

We proposed a novel subsequence matching framework that employs a non-Euclidean distance, a multiple window sliding scheme and a similarity skipping idea. As validated through experiments with various datasets, proposed methods enable us to have more intuitive and efficient subsequence matching algorithms. The multiple window sliding scheme was more efficient than the single window sliding scheme for the longer query in candidate generation, goodness and tightness. In addition, skipping the *LCS* computation greatly reduces expensive similarity computations.

References

1. Agrawal, R., Faloutsos, C., Swami, A.N.: Efficient similarity search in sequence databases. In: Lomet, D.B. (ed.) FODO 1993. LNCS, vol. 730, pp. 69–84. Springer, Heidelberg (1993)
2. Faloutsos, C., Ranganathan, M., Manolopoulos, Y.: Fast subsequence matching in time-series databases. In: Proceedings 1994 ACM SIGMOD Conference, Mineapolis, MN, ACM Press, New York (1994)
3. Gunopoulos, D.: Discovering similar multidimensional trajectories. In: ICDE '02. Proceedings of the 18th International Conference on Data Engineering, p. 673. IEEE Computer Society Press, Los Alamitos (2002)
4. Kadous, M.: Grasp: Recognition of australian sign language using instrumented gloves (1995)
5. Keogh, E.J.: Exact indexing of dynamic time warping. In: VLDB, pp. 406–417 (2002)
6. Moon, Y.-S., Whang, K.-Y., Loh, W.-K.: Duality-based subsequence matching in time-series databases. In: Proceedings of the 17th ICDE, Washington, DC, pp. 263–272. IEEE Computer Society Press, Los Alamitos (2001)
7. Moon, Y.-S., Whang, K.-Y., Loh, W.-K.: Efficient time-series subsequence matching using duality in constructing window. Information Systems 26(4), 279–293 (2001)
8. Sakoe, H., Chiba, S.: Dynamic programming algorithm optimization for spoken word recognition, pp. 159–165 (1990)
9. Sankoff, D., Kruskal, J.: Time warps, string edits, and macromolecules: the theory and practice of sequence comparison. Addison-Wesley, Reading (1983)
10. Vlachos, M., Hadjieleftheriou, M., Gunopulos, D., Keogh, E.: Indexing multidimensional time-series with support for multiple distance measures. In: KDD '03, pp. 216–225. ACM Press, New York (2003)
11. Zhu, Y., Shasha, D.: Warping indexes with envelope transforms for query by humming. In: SIGMOD '03, pp. 181–192. ACM Press, New York (2003)