

# Efficient Processing of Warping Time Series Join of Motion Capture Data

Yueguo Chen <sup>#1</sup>, Gang Chen <sup>§2</sup>, Ke Chen <sup>\*3</sup>, Beng Chin Ooi <sup>#4</sup>

<sup>#</sup>*School of Computing, National University of Singapore, Singapore*  
{<sup>1</sup>chenyueg, <sup>4</sup>ooibc}@comp.nus.edu.sg

<sup>§</sup>*College of Computer Science, Zhejiang University, China*  
<sup>2</sup>cg@zju.edu.cn

<sup>\*</sup>*School of Aeronautics and Astronautics, Zhejiang University, China*  
<sup>3</sup>chenk@zju.edu.cn

**Abstract**—Discovering non-trivial matching subsequences from two time series is very useful in synthesizing novel time series. This can be applied to applications such as motion synthesis where smooth and natural motion sequences are often required to be generated from existing motion sequences. We first address this problem by defining it as a problem of  $l$ - $\varepsilon$ -join over two time series. Given two time series, the goal of  $l$ - $\varepsilon$ -join is to find those non-trivial matching subsequences by detecting maximal  $l$ -connections from the  $\varepsilon$ -matching matrix of the two time series. Given a querying motion sequence, the  $l$ - $\varepsilon$ -join can be applied to retrieve all connectable motion sequences from a database of motion sequences.

To support efficient  $l$ - $\varepsilon$ -join of time series, we propose a two-step filter-and-refine algorithm, called Warping Time Series Join (WTSJ) algorithm. The filtering step serves to prune those sparse regions of the  $\varepsilon$ -matching matrix where there are no maximal  $l$ -connections without incurring costly computation. The refinement step serves to detect closed  $l$ -connections within regions that cannot be pruned by the filtering step. To speed up the computation of  $\varepsilon$ -matching matrix, we propose a block-based time series summarization method, based on which the block-wise  $\varepsilon$ -matching matrix is first computed. Lots of pairwise distance computation of elements can then be avoided by applying the filtering algorithm on the block-wise  $\varepsilon$ -matching matrix. Extensive experiments on  $l$ - $\varepsilon$ -join of motion capture sequences are conducted. The results confirm the efficiency and effectiveness of our proposed algorithm in processing  $l$ - $\varepsilon$ -join of motion capture time series.

## I. INTRODUCTION

Time series can be widely applied in representing trajectories, shapes, human motion, etc. Given two time series, we may want to find matching subsequences within the two series. Figure 1 shows an example of a pair of matching subsequences within two 2D trajectories. The matching subsequences must be similar enough under some distance measures. They should also be long enough to exclude the trivial matching subsequences. In our definition, the join of two time series serves to find those non-trivial matching subsequences within them.

Such a definition of time series join is quite useful in applications such as motion synthesis [1], [2], [3] where users may want to find a smooth and nature translation from one motion sequence to another without knowing where to connect them. A fictional motion sequence can be created by

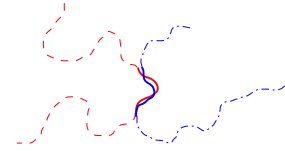


Fig. 1. An example of time series join

connecting two sequences within the regions where matching subsequences exist. The cost of such a synthesis will be low as two matching subsequences are quite similar to each other. Figure 2 shows an example of synthesizing a novel trajectory from the two trajectories in Figure 1.

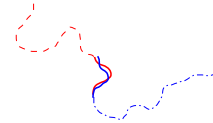


Fig. 2. Synthesis of time series using time series join

In motion synthesis applications, given a querying motion sequence, users want to retrieve some natural subsequent movements (connected at some local regions of the querying sequence) from a database of motion sequences by using time series join. Those subsequent movements can then be used to synthesize new motions extended to the querying motion sequence. Such motion reuse requires efficient processing of time series join to fully exploit a large motion database. This paper seeks to address the time series join problem. Specifically, we focus on time series join of human motion data. However, the proposed techniques can also be applicable to other time series such as trajectories.

There have been some recent works on subsequence matching [4], [5], [6], [7]. However, they are different from the time series join problem as subsequence matching actually matches given querying subsequences to long time series, i.e., one subsequence of matching subsequence pairs has been given. There are also studies that focus on motif discovery within time series [8], [9]. They are different from the time series join problem in three ways. First, they actually try to discover

frequent similar subsequences within one time series. Second, all matching subsequences of a motif are of the same length. Third, motifs of different lengths are discovered by checking them with different lengths one by one.

There have some studies [10], [11], [12] on trajectory join, which tries to find pairs of trajectories from two trajectory data sets such that each pair of two trajectories have matching subsequences in some way. However, in these studies, there is no time shifting between two matching subsequences, i.e., a subsequence of time  $[t_s : t_e]$  can only match with another subsequence having the same time slot  $[t_s : t_e]$ . There is one work [13] which allows matching subsequences to be anywhere in trajectories. However, two subsequences are matched only if they are of the same length, and every element in one subsequence must be matched to the element at the same relative position of the other subsequence. We call this form of subsequence matching between two time series, equal-size join. An example is shown in Figure 3(a).

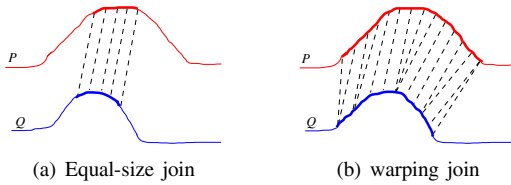


Fig. 3. Two ways of time series join (matching subsequences are bold)

However, it is quite common for similar motion sequences to have different time scalings [14] due to difference in the moving velocity of human bodies. Such scaling variance can be easily handled by motion synthesis techniques [1], [2]. However, the join algorithms must support time warping [15] between matching subsequences before the scaled matching subsequences can be fully exploited through time series join. We call such kind of time series join, warping join. Figure 3(b) shows an example of warping join which finds longer matching subsequences (of different time scalings) than equal-size join.

In this paper, we focus on the warping time series join problem, in which we want to identify the arbitrary lengths of matching subsequences whose elements are continuously matched with time warping supports. This problem is much more difficult than the equal-size join for two reasons. First, the positions and lengths of matching subsequences are arbitrary within two time series. Second, the matches of elements within two matching subsequences can be warped. To the best of our knowledge, ours is the first work to address the warping time series join problem. In our work, we define the warping time series join as an  $l$ - $\varepsilon$ -join problem, and propose an efficient Warping Time Series Join algorithm. The proposed sequence join algorithm is applied in our PIPA system (a database engine for interactive media, for supporting effective matches of multimedia objects). Our contributions can be summarized as follows:

- We formally define the problem of  $l$ - $\varepsilon$ -join over two time series as the problem of maximal  $l$ -connection detection from the  $\varepsilon$ -matching matrix of two time series.

- We propose a two-step filter-and-refine WTSJ algorithm, to efficiently identify possible sub-matrices containing  $l$ -connections and detect closed  $l$ -connections from the  $\varepsilon$ -matching matrix.
- We propose summarizing a time series as a number of blocks, with each block represented as a sphere of radius  $\delta$ .  $\varepsilon$ -matching matrix can be efficiently computed through the pruning effects of block-wise distances.
- Extensive experimental study using motion capture data was conducted. The results indicate the efficiency of our proposed algorithms on  $l$ - $\varepsilon$ -join of time series.

The rest of the paper is organized as follows: Section II introduces some related work. Section III provides the problem definition of warping time series join. We propose the WTSJ algorithm for efficient maximal  $l$ -connection detection in Section IV, and introduce the time series summarization algorithm for efficient  $\varepsilon$ -matching matrix computation in Section V. The experimental studies are shown in Section VI, and conclusions are drawn in Section VII.

## II. RELATED WORK

### A. Distances Measures of Time Series

The distance between any two time series is essentially computed from the aggregation of pair-wise difference of elements within them. Traditionally, the Euclidean distance is used to measure the distances between time series of the same length. However, it is common for the shapes of time series to be shifted in time domain. Therefore, warping distances such as DTW [15] and EDR [16] have been proposed for measuring distances of time series of arbitrary lengths. The optimal alignments of elements between two time sequences are obtained by repeating some elements so that the lengths of two sequences can be the same. Such a warping mechanism can well match similar time series with different time scaling [14]. A comparison of warping distances is given in [16].

A warping distance is computed by dynamic programming over distance matrix, which has a time complexity of  $O(mn)$  ( $m$  and  $n$  being the lengths of two time series). We use an example to illustrate how warping distance is computed. All pair-wise distances of entries in 1D time series  $P$  and  $Q$  are shown Figure 4(a). The DTW distance is aggregated over the distance matrix (Figure 4(b)) in a dynamic programming manner as  $M[i, j] = \min(M[i, j-1], M[i-1, j], M[i-1, j-1]) + d(P[i], Q[j])$ , where  $M[i, j]$  is an entry of the distance matrix, and  $d(P[i], Q[j])$  is the pair-wise distance between elements  $P[i]$  and  $Q[j]$  (in this case,  $d(P[i], Q[j]) = |P[i] - Q[j]|$ ). The DTW distance of  $P$  and  $Q$  obtained is 1.0. The warping path can be traced from the distance matrix  $M$ .

### B. Subsequence Matching

Subsequence matching [4], [5], [6], [7] has been widely studied in recent years. Given a short query pattern, subsequence matching attempts to locate matching subsequences of the query pattern within a long time series. Several dynamic programming based subsequence matching algorithms [5], [7] have recently been proposed to detect matching subsequences.

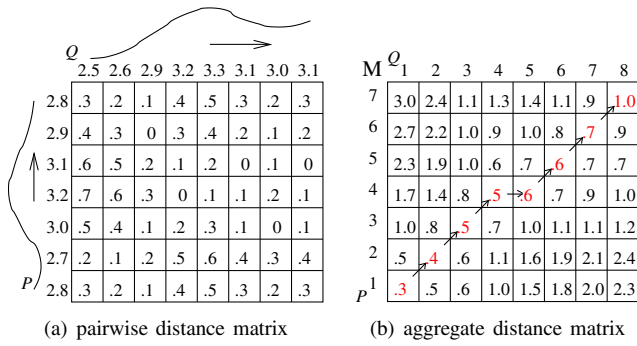


Fig. 4. An example of DTW distance aggregation over distance matrix

However, they cannot be directly applied to time series join as the lengths and shapes of querying subsequences are not given in the time series join problem.

Motif discovery, which is aimed at finding frequent similar subsequences within one time series, is also studied in [8], [9]. It is more difficult than the subsequence matching problem because there are no given querying subsequences for matching. However, motif discovery is less complex than our proposed time series join problem because a motif determines the length of matching subsequences, i.e., matching subsequences of a motif are assumed to be of the same length.

As we have stated, matching subsequences in trajectory join are either without time shifting [10], [11], [12] or without time warping (equal-size join [13]). In these studies, two elements within two trajectories are matched if their distance is close enough (e.g., no more than given threshold  $\varepsilon$ ). Two subsequences are matched only if elements within them are continuously one-to-one matched (as the way in Figure 3(a)).

Sequence join is also studied in string and DNA sequences [17], [18], [19], [20], where  $q$ -grams are typically used to index and match subsequences. However, these studies are focused on one-dimensional categorical sequences where two elements are matched only they are exactly the same. They tolerate the existence of unmatched elements within matching subsequences by using edit distances. Algorithms in these studies are not designed for handling time scaling difference of matching subsequences. As a result, warps in temporal dimension will be significantly punished [18]. Our sequence join problem focuses on high-dimensional motion data. Therefore, matching elements are not required to be exactly the same. Moreover, we do not tolerate unmatched elements within matching subsequences because there is often no noise in human motion capture data.

### C. Motion Capture Data

Human motion can be recorded by attaching dozens of optical marks to human bodies. The 3D positions of the marks are captured in fine granularity (e.g., 120Hz) using multiple cameras. The obtained motion capture data are time series of high-dimensional data (recording the positions of marks or joints). They are widely used in computer animation or gait/gesture analysis. However, a motion database can only

hold a limited number of motion sequences because: 1) Motion capture so far is expensive, and it is difficult to collect a database of motion sequences including different styles of motions because they vary in many ways. 2) Motion capture data require a large storage space due to the high-dimensional time series of fine granularity. For example, the original CMU motion capture database [21] contains around 2000 motion sequences, and it consumes a storage space of 2.3G.

Human motion sequences are stored in a large motion database. Content-based human motion data retrieval has been widely studied [14], [22], [23], [24]. Logic-oriented geometric features were proposed [23] for extracting effective and compact features from original human motion data. These features are extracted from some local dimensions and are robust to variations of human movements. Normalization is necessary to transform features of different magnitudes to  $[0, 1]$ .

Because practical motion databases only collect a limited number of motion sequences, motion synthesis [1], [2], [3] has been widely used to generate more novel and realistic motion instances by connecting and modifying existing motion sequences. Time series join is quite useful in such applications because it helps users retrieve connectable matching subsequences. Techniques such as blending [2], interpolation [25], style translation [3] and constraint graph [1] can then be applied to create smooth transition of motion sequences at the local regions of matching subsequences.

## III. WARPING TIME SERIES JOIN

To facilitate the understanding of concepts and algorithms introduced in this paper, we first list some frequently used notations in Table I:

TABLE I  
NOTATIONS

$P, Q$	<b>a time series</b>
$P[i]$	<b>the <math>i^{th}</math> element of a time series <math>P</math></b>
$P[s : e]$	<b>a subsequence of <math>P</math> from <math>P[s]</math> to <math>P[e]</math></b>
$d(\cdot, \cdot)$	<b>a metric distance between two elements</b>
$M$	<b>the <math>\varepsilon</math>-matching matrix</b>
$u = (i, j)$	<b>a node in the <math>\varepsilon</math>-matching matrix</b>
$u.i, u.j$	<b>the coordinate of a node</b>
$o = u_1 \triangleright u_2$	<b>a connection from node <math>u_1</math> to node <math>u_2</math></b>
$u_1 \prec u_2$	<b>node <math>u_1</math> dominates <math>u_2</math></b>
$o_1 \prec o_2$	<b>connection <math>o_1</math> dominates <math>o_2</math></b>
$u.sky$	<b>the list of skyline nodes of a node <math>u</math></b>

### A. Nodes and Connections

Basically, a time series consists of a sequence of elements. We denote the  $i^{th}$  element of a time series  $P$  as  $P[i]$ . A subsequence of a time series  $P$  consists of a subset of contiguous elements in the time series  $P$ , and it is denoted as  $P[s : e]$ , whose starting and ending entries are  $P[s]$  and  $P[e]$  respectively. Therefore, the length of a subsequence  $P[s : e]$  is measured as  $len(P[s : e]) = e - s + 1$ .

Given two elements (entries)  $P[i]$  and  $Q[j]$ , they are  $\varepsilon$ -matching if  $d(P[i], Q[j]) \leq \varepsilon$ . Given two time series  $P$  and  $Q$  of length  $m$  and  $n$  respectively, we can build a  $\varepsilon$ -matching matrix  $M^{m \times n}$  describing the matches of elements between

$P$  and  $Q$ . The entry  $(i, j)$  of  $M$  is a node if  $P[i]$  and  $Q[j]$  are  $\varepsilon$ -matching. We mark  $M[i, j]$  with 1 if  $(i, j)$  is a node. Otherwise, it is 0. Figure 5 shows an example of the  $\varepsilon$ -matching matrix of two time series  $P$  and  $Q$  in Figure 4(a).

	Q							
M	1	2	3	4	5	6	7	8
7			1					
6			1				1	
5				1		1	1	1
4				1	1	1		1
3			1			1	1	1
2		1						
P	1		1					

Fig. 5. The  $\varepsilon$ -matching matrix of a 5-0.15-join of two time series

A node  $u = (i, j)$  can be a direct predecessor of any of the three nodes  $(i+1, j)$ ,  $(i, j+1)$  and  $(i+1, j+1)$  (note that  $(i, j)$  is a node only if  $M[i, j] = 1$ ). If a node  $u_1$  is a predecessor of  $u_2$ , we say that there is a connection between  $u_1$  and  $u_2$ , denoted as  $u_1 \triangleright u_2$ . In  $u_1 \triangleright u_2$ ,  $u_1$  is a predecessor of  $u_2$ , and  $u_2$  is also a successor of  $u_1$ . The connections among nodes have the transitive property, i.e., if  $u_1 \triangleright u_2$  and  $u_2 \triangleright u_3$ , then  $u_1 \triangleright u_3$ . For example, in Figure 5,  $(4, 4) \triangleright (4, 5)$ ,  $(3, 6) \triangleright (4, 8)$ .

A node is an open node if it has no predecessor, and a closed node if it has no successor. If a node has both successor(s) and predecessor(s), it is called an intermediate node. Examples of an open node, an intermediate node and a closed node in Figure 5 are  $(2, 2)$ ,  $(4, 5)$  and  $(6, 7)$  respectively.

Given a connection  $u_1 \triangleright u_2$  where  $u_1 = (i_1, j_1)$  and  $u_2 = (i_2, j_2)$ , there must be at least one connecting chain from  $u_1$  to  $u_2$  linked by a number of intermediate nodes  $(i_1, j_1), (i_{x_1}, j_{x_1}), \dots, (i_{x_t}, j_{x_t}), (i_2, j_2)$ . Each node in a connecting chain must be a direct predecessor of the next node in the chain. If there is no such connecting chain from a node  $u_1$  to another node  $u_2$ , then  $u_1$  cannot be a predecessor of  $u_2$  according to the transitive property of connections. For the example in Figure 5, we show one connecting chain from  $(2, 2)$  to  $(6, 7)$ .

The connection  $o = u_1 \triangleright u_2$  ( $u_1 = (i_1, j_1)$  and  $u_2 = (i_2, j_2)$ ) describes the existence of a match between subsequences  $P[i_1 : i_2]$  and  $Q[j_1 : j_2]$ . Similar to the warping paths of warping distances, the connecting chains of a connection provide warping mechanisms for the two matching subsequences so that they can be of arbitrary lengths. For each connecting chain between node  $u_1$  and node  $u_2$ , we measure the matching score of the connection  $u_1 \triangleright u_2$  as  $s(u_1 \triangleright u_2) = \min(i_2 - i_1 + 1, j_2 - j_1 + 1)$ . The matching score measures the minimal lengths of the two matching subsequences determined by the connection. The larger the matching score is, the longer the matching subsequences between the two time series are.

Note that there may be many trivial matching subsequences where the lengths of the matching subsequences are not large. Therefore, we propose using a parameter  $l$  to filter those trivial matching subsequences. A connection  $o$  is said to be an  $l$ -connection if  $s(o) \geq l$ .

## B. Dominating-ships of Nodes and Connections

Given two nodes  $u_1$  and  $u_2$ ,  $u_1$  dominates  $u_2$ , denoted as  $u_1 \prec u_2$ , if  $u_1.i \leq u_2.i$  and  $u_1.j \leq u_2.j$ .

**Lemma 3.1:** *If  $u_1 \triangleright u_2$ , then  $u_1 \prec u_2$ .*

*Proof:* Let  $u_1 = (i_1, j_1)$  and  $u_2 = (i_2, j_2)$ . According to the definition of a connection, if  $u_1$  is the direct predecessor of  $u_2$ , it is obvious that  $i_1 \leq i_2$  and  $j_1 \leq j_2$ . Otherwise, there must be a connecting chain  $u_1 = (i_1, j_1) \triangleright (i_{x_1}, j_{x_1}) \triangleright \dots \triangleright (i_{x_t}, j_{x_t}) \triangleright (i_2, j_2) = u_2$ . Therefore, we still have  $i_1 \leq i_{x_1} \leq \dots \leq i_{x_t} \leq i_2$  and  $j_1 \leq j_{x_1} \leq \dots \leq j_{x_t} \leq j_2$ . Then we have  $u_1 \prec u_2$ . ■

However,  $u_1 \prec u_2$  does not imply  $u_1 \triangleright u_2$  because there may be no connecting chain from  $u_1$  to  $u_2$  even when  $u_1 \prec u_2$ . For example, in Figure 5, there is no connection between  $(2, 2)$  and  $(3, 6)$  even when  $(2, 2) \prec (3, 6)$ .

Given a connection  $o = u_1 \triangleright u_2$ , if  $u_1$  is an open node and  $u_2$  is a closed node, the connection  $o$  is called a closed connection. The connecting chains of a closed connection (e.g.,  $(2, 2) \triangleright (6, 7)$  in Figure 5) cannot be extended. However, the connecting chains of a non-closed connection (e.g.,  $(2, 2) \triangleright (5, 6)$  in Figure 5) can be extended at either the starting side or the ending side since either  $u_1$  or  $u_2$  is an intermediate node. In the time series join problem, we care more about closed connections because they maximize the lengths of local matching subsequences.

Given two connections  $o_1 = (i_1, j_1) \triangleright (i_2, j_2)$  and  $o_2 = (i_3, j_3) \triangleright (i_4, j_4)$ , we say  $o_1$  dominates  $o_2$ , denoted as  $o_1 \prec o_2$ , if  $[i_1 : i_2] \supseteq [i_3 : i_4]$  and  $[j_1 : j_2] \supseteq [j_3 : j_4]$ . For the example in Figure 5, the closed connection  $(2, 2) \triangleright (6, 7)$  dominates another closed connection  $(3, 6) \triangleright (6, 7)$ . If  $o_1 \prec o_2$ , it is obvious that  $o_1$  covers longer matching subsequences within time series  $P$  and  $Q$  than  $o_2$  does. A connection is a maximal connection if it is not dominated by any other connections. For example, the connection  $(2, 2) \triangleright (6, 7)$  in Figure 5 is a maximal connection.

**Lemma 3.2:** *A maximal connection is a closed connection.*

*Proof:* We prove it by contradiction, i.e., a non-closed connection cannot be a maximal connection. Suppose a connection  $o_1 = (i_1, j_1) \triangleright (i_2, j_2)$  is a non-closed connection. Without loss of generality, suppose  $(i_2, j_2)$  has a successor  $(i_3, j_3)$ , i.e.,  $(i_2, j_2) \triangleright (i_3, j_3)$ . According to Lemma 3.1,  $i_3 \geq i_2$  and  $j_3 \geq j_2$ . Then we have  $[i_1 : i_3] \supseteq [i_1 : i_2]$  and  $[j_1 : j_3] \supseteq [j_1 : j_2]$ . Therefore,  $o_2 = (i_1, j_1) \triangleright (i_3, j_3) \prec o_1$ , and  $o_1$  is not a maximal connection. ■

## C. Warping Time Series Join

We define warping time series join as a  $l$ - $\varepsilon$ -join problem. A  $l$ - $\varepsilon$ -join over two time series is aimed at retrieving all those maximal  $l$ -connections in the  $\varepsilon$ -matching matrix of the two given time series.

In the above definition of  $l$ - $\varepsilon$ -join, the connections guarantee that elements within the matching subsequences of two time series are continuously  $\varepsilon$ -matching in a warping manner. Such a requirement of continuity over matching elements are quite

reasonable because human motion data are often smooth and sampled in fine granularity. They are different from the signal types of time series collected from devices where noise is quite common.

The parameter  $\varepsilon$  controls the similarity of matching elements. It can be adjusted based on applied motion synthesis approaches. The parameter  $l$  helps filter trivial matching subsequences. The number of matching subsequences between two time series can be controlled by adjusting the two parameters during time series join.

Figure 5 illustrates an example of  $l$ - $\varepsilon$ -join where  $l = 5$ ,  $\varepsilon = 0.15$ . The distances of all pairs of elements are shown in Figure 4(a). We filter those unmatched pairs of elements and show the  $\varepsilon$ -matching matrix in Figure 5. The only maximal 5-connection in our running example is  $(2, 2) \triangleright (6, 7)$ , which describes a match of subsequences  $P[2 : 6]$  and  $Q[2 : 7]$ .

#### IV. THE WARPING TIME SERIES JOIN ALGORITHM

In this section, we introduce the Warping Time Series Join (WTSJ) algorithm for processing the  $l$ - $\varepsilon$ -join of two given time series  $P$  and  $Q$ . Before we proceed to describe the algorithm, we shall illustrate what is involved in the process.

To facilitate  $l$ - $\varepsilon$ -join, the  $\varepsilon$ -matching matrix of two time series has to be constructed. As illustrated in Figure 5, all pairwise distances of elements within  $P$  and  $Q$  are computed (in Figure 4(a)) and filtered by  $\varepsilon$ . Those  $\varepsilon$ -matching elements are marked as 1 in the corresponding positions of the  $\varepsilon$ -matching matrix. Note that after  $\varepsilon$  filtering, the  $\varepsilon$ -matching matrix will be a sparse matrix unless  $P$  and  $Q$  are time series of constant values. The problem of  $l$ - $\varepsilon$ -join is then simplified to find the maximal  $l$ -connections within the sparse matching matrix.

To facilitate expression, we define the aggregate score of a node  $u$  as  $a(u) = \max_{u' \triangleright u} s(u' \triangleright u)$ . Maximal  $l$ -connections cannot be detected simply by the dynamic programming way (e.g., DTW [15] or the Smith-Waterman algorithm [18]) of computing aggregate scores of nodes in the  $\varepsilon$ -matching matrix. This is because the aggregate score of a node  $u$  cannot be computed from the aggregate scores of its direct predecessors as there is no information about from which node the node  $u$  achieves the aggregate score.

Even the open node  $u'$  from which a node  $u$  gets its aggregate score  $a(u) = s(u' \triangleright u)$  is recorded by  $u$ , the aggregate scores still cannot be used for detecting maximal  $l$ -connections. We use an example in Figure 6 to show this. The node  $u$  (in Figure 6(a)) has two direct predecessors  $u_1$  and  $u_2$  who record two open nodes  $u'_1$  and  $u'_2$  respectively. As a result of dynamic programming, node  $u$  records  $u' = u'_1$  from which it gets the aggregate score  $a(u) = 3$  (in Figure 6(b)), based on the two open nodes recorded by  $u_1$  and  $u_2$ . In this way, there will be no 5-connection after the dynamic programming way of computing aggregate scores because all aggregate scores in Figure 6(b) are less than 5. However, there is actually a connection  $u'_2 \triangleright u_3$  whose score is 5. This is because the open node  $u'_2$  has been pruned when computing  $a(u)$ . Therefore, maximal  $l$ -connections cannot be detected by using the simple dynamic programming approach.

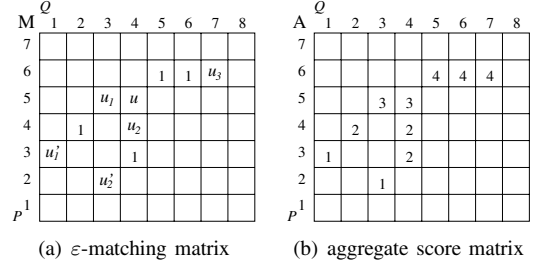


Fig. 6. Maximal  $l$ -connections by the simple dynamic programming approach

We propose to solve the maximal  $l$ -connection detection problem using a two-step filter-and-refine algorithm. In the filtering step, we propose to extract sub-regions within the  $\varepsilon$ -matching matrix that possibly hold closed  $l$ -connections. In the refinement step, we propose two algorithms – Flooding and Skywave – to detect closed  $l$ -connections from candidate matching matrices extracted from the filtering step. Detected closed  $l$ -connections are then filtered so that only maximal  $l$ -connections remain as the results of  $l$ - $\varepsilon$ -join. The whole WTSJ algorithm is outlined as a flow chart in Figure 7.

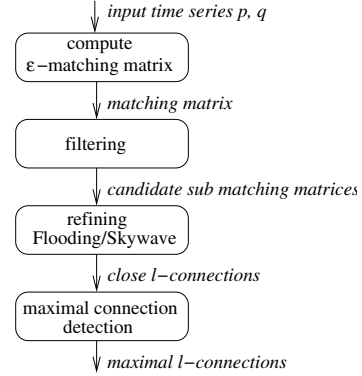


Fig. 7. The flow chart of the WTSJ algorithm

#### A. The Filtering Step

To avoid scanning closed and maximal connections over the whole  $\varepsilon$ -matching matrix, the continuity property of the connecting chains of connections is applied to prune some search regions where it is guaranteed that no  $l$ -connections exist.

**Lemma 4.1:** *Given a connection  $u_1 \triangleright u_2$ , within the sub-matrix cornered by  $u_1$  and  $u_2$ , each row and each column must have at least one node.*

*Proof:* *Given the connection  $u_1 \triangleright u_2$ , there must be a connecting chain between  $u_1$  and  $u_2$ . Given any node  $u_k$  in the connecting chain, we have  $u_1 \triangleright u_k \triangleright u_2$ . Based on Lemma 3.1, we have  $u_{1.i} \leq u_{k.i} \leq u_{2.i}$  and  $u_{1.j} \leq u_{k.j} \leq u_{2.j}$ . Therefore, all the nodes in the connecting chain must be within the sub-matrix cornered by  $u_1$  and  $u_2$ . To guarantee the continuity of the connecting chain, for each row between  $u_1$  and  $u_2$ , there must be one node in the connecting chain. Since all nodes of the connecting chain must be in the sub-matrix, each row in the sub-matrix will then have at least one*

node. Similarly, each column in the sub-matrix also has at least one node. ■

Based on Lemma 4.1, there will be no connections between node  $u_1$  and  $u_2$ , if any row or any column of the sub-matrix cornered by  $u_1$  and  $u_2$  has no nodes. This motivates us to propose a filtering algorithm to prune regions that have no  $l$ -connections. In the filtering algorithm, two summarizing vectors  $R$  and  $C$  are maintained to record whether there are nodes within each row (for  $R$ ) and each column (for  $C$ ) of the  $\varepsilon$ -matching matrix  $M$  respectively. For example, in Figure 8, there are no nodes in the 4<sup>th</sup> row and the 5<sup>th</sup> column of  $M$ . Therefore,  $R[4] = 0$  and  $C[5] = 0$ .

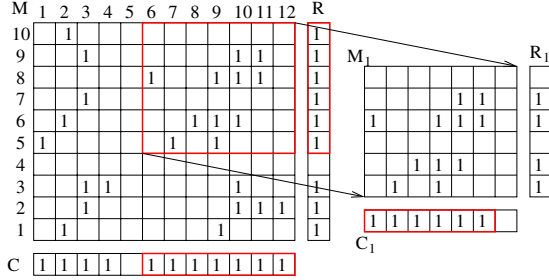


Fig. 8. Illustration of the filtering algorithm

We can apply the filtering algorithm (Algorithm 1) to extract candidate regions in  $M$  that may contain  $l$ -connections, using the summarizing vectors  $R$  and  $C$ . The filtering algorithm first detects local patterns which consist of contiguous 1s within  $R$  and  $C$ . The lengths of these local patterns should be no less than  $l$ . Based on these local patterns, candidate sub-matrices can be identified. They are further checked by applying the filtering algorithm recursively. The pruning stops if the summarizing vectors of a candidate sub-matrix do not have zero entries.

The filtering algorithm does not perform actual  $l$ -connections detection. The linear scan of summarizing vectors consume  $O(m + n)$  time complexity. Therefore, the filtering algorithm is quite efficient. Those candidate sub-matrices can be quickly converged and identified. Many sparse regions where there are no good matches of subsequences can be efficiently pruned, without performing any expensive  $l$ -connection detection. In the example of Figure 8, the parameter  $l = 5$ . Only one local pattern is detected from  $R$  and  $C$  respectively. Therefore, only one candidate sub-matrix  $M_1$  exists within  $M$ . The sub-matrix  $M_1$  is further checked by the filtering algorithm.  $M_1$  can be directly pruned as there is no local pattern in  $R_1$ . Therefore, the filtering algorithm is able to ensure that there are no 5-connections in the resultant  $\varepsilon$ -matching matrix  $M$ .

### B. The Refinement Step

For matrices that cannot be pruned by the filtering step of the WTSJ algorithm, actual closed  $l$ -connection detection have to be performed in the refinement step. In this step, the algorithm finds closed  $l$ -connections which start at open nodes and end at closed nodes within the candidate  $\varepsilon$ -matching

### Algorithm 1 The filtering algorithm

**Input:**  $M^{m \times n}$ , a matching matrix.

**Input:**  $l$ , the specified parameter in the  $l$ - $\varepsilon$ -join algorithm.

**Input:**  $candMList$ , recording all candidate matrices that cannot be pruned by the Filtering algorithm.

1. compute  $R$  //the row summarizing vector of  $M$ .
2. compute  $C$  //the column summarizing vector of  $M$ .
3. **if**  $countOnes(R) = n$  and  $countOnes(C) = m$  **then**
4.    $candMList.add(M)$  //a matrix cannot be pruned
5.   **return**
6. initialize  $listR$  and  $listC$
7.  $t \leftarrow 1$
8. **while**  $t \leq n - l$  **do**
9.   **if**  $R[t] = 1$  **then**
10.      $s \leftarrow t$
11.     **while**  $R[t] = 1$  and  $t \leq n$  **do**
12.        $t++$
13.     **if**  $t - s > l$  **then**
14.        $listR.append((s, t - 1))$
15.      $t++$
16.    $t \leftarrow 1$
17. **while**  $t \leq m - l$  **do**
18.   **if**  $C[t] = 1$  **then**
19.      $s \leftarrow t$
20.     **while**  $C[t] = 1$  and  $t \leq m$  **do**
21.        $t++$
22.     **if**  $t - s > l$  **then**
23.        $listC.append((s, t - 1))$
24.      $t++$
25. **for**  $(s_i, e_i) \in listR$  **do**
26.   **for**  $(s_j, e_j) \in listC$  **do**
27.      $M_1 \leftarrow M[s_i : e_i, s_j : e_j]$
28.     recursively apply filtering algorithm on  $M_1$

matrices. The closed  $l$ -connections can be detected in two ways: 1) from an open node, find all closed nodes to which it connects; 2) at a closed node, find all open nodes connecting to it. Correspondingly, we propose two algorithms, namely Flooding and Skywave algorithms.

1) *The Flooding algorithm:* The Flooding algorithm (Algorithm 2) searches for all closed nodes to which an open node connects. It first scans the open nodes within the  $\varepsilon$ -matching matrix. Once an open node is detected, a flooding process is executed to search all closed nodes connected with the open node. During the flooding process, each expanded node checks whether it has direct successor(s) from its top, right and top-right neighbors. The flooding is further extended to the successors if the expanded node has direct successors. Otherwise, it stops at the expanded node because it is a closed node. A closed connection then is detected when the flooding reaches a closed node. The scores of the detected closed connections are then computed and filtered over  $l$  to distill those  $l$ -connections.

In the Flooding algorithm, each open node initiates a

---

**Algorithm 2 The Flooding algorithm**


---

**Input:**  $M^{m \times n}$ , a matching matrix.

**Input:**  $l$ , the specified parameter in the  $l$ - $\varepsilon$ -join algorithm.

**Output:**  $O = \{o_i\}$  all closed  $l$ -connections in  $M$ .

```

1.  $scanRound = 0$ ;
2.  $E^{m \times n}$  //indicates the round of flooding
3. for  $i = 1 : m$  do
4.   for  $j = 1 : n$  do
5.      $E[i][j] \leftarrow scanRound$ 
6.   for  $i = 1 : m - l + 1$  do
7.     for  $j = 1 : n - l + 1$  do
8.       if  $M[i, j] = 1$  and  $E[i, j] = 0$  then
9.          $(i, j)$  is an open node
10.         $toExpandNodes((i, j))$  //initiate a queue with
             $(i, j)$  as the first element.
11.         $scanRound + +$  //initiate a flooding
12.        while  $toExpandNodes.hasElements()$  do
13.           $(x, y) \leftarrow toExpandNodes.pop()$ 
14.           $successors \leftarrow \emptyset$ 
15.          for  $p = (x, y + 1), (x + 1, y), (x + 1, y + 1)$  do
16.            if  $M[p.x, p.y] = 1$  then
17.               $successors + +$ 
18.              if  $E[p.x][p.y] < scanRound$  then
19.                 $toExpandNodes.push((p.x, p.y))$ 
20.                 $E[p.x, p.y] \leftarrow scanRound$ 
21.            if  $successors = \emptyset$  then
22.               $o \leftarrow (i, j) \triangleright (x, y)$  //  $o$  is a closed connection
23.              if  $s(o) \geq l$  then
24.                insert  $o$  into  $O$  //  $o$  is a closed  $l$ -connection

```

---

flooding process. The expanding areas of multiple floodings may overlap with each other if the initial open nodes of these floodings are close to each other (e.g.,  $(5, 7)$  and  $(5, 9)$  in Figure 8). As a result, those intermediate nodes and closed nodes can be expanded for multiple times by different floodings.

To efficiently identify the expanded nodes of a flooding, we use a labelling matrix  $E$  in the Flooding algorithm. Each flooding is assigned an identified label  $scanRound$ , which automatically increases with different floodings. All nodes expanded by flooding is labelled with the label of that flooding. In this way, an open node can be detected when the Flooding scans to a node whose label in  $E$  is never changed (i.e.,  $E[i, j] = 0$ ). Those unexpanded nodes are easily identified by comparing the corresponding labels of those nodes in  $E$  with the label of the flooding, so that there is no repetitive expansion of nodes within one flooding. The complexity of the Flooding algorithm is  $O(rmn)$  where  $r$  is the average number of nodes connected to an open node. In worst cases,  $r$  can be as large as  $O(mn)$ . However, it means that most of the nodes will be intermediate nodes from which no flooding is required.

2) *The Skywave algorithm:* The Skywave algorithm (Algorithm 3) detects all skyline nodes (specific open nodes) connecting to a closed node. Like the computation of warping distances, the Skywave algorithm scans nodes in the  $\varepsilon$ -

matching matrix row by row (bottom up). Within each row, nodes are scanned from left to right. For each scanned node, a sky list (denoted as  $u.sky$ ) is maintained to record all skyline nodes connecting to it. The sky list of a node can be dynamically computed from the sky lists of its direct predecessors.

A node  $u_1$  is a skyline node of  $u$  if: 1)  $u_1 \triangleright u$ ; and 2) there is not any other node  $u_2 \triangleright u$  such that  $u_2 \prec u_1$ . It is obvious that a skyline node is also an open node. Otherwise, the skyline node will be dominated by an open node connecting to it. The sky list of a node includes all skyline nodes of that node. For example, in Figure 9,  $(6, 7).sky = \{(3, 2)\}$ ,  $(5, 8).sky = \{(3, 5), (1, 6)\}$ .

M	1	2	3	4	5	6	7	8	9	10
8									1	
7						1		1	1	1
6	1				1	1	$u_1$	$u$		
5		1	1	1				$u_2$	1	
4			1				1	1		
3		1			1	1				
2						1				
1					1					

Fig. 9. Illustration of the skyline nodes and sky lists

The sky list of a node  $u.sky$  is computed by combining the sky lists of its direct predecessors. Two sky lists  $u_1.sky$  and  $u_2.sky$  are combined ( $u_1.sky \uplus u_2.sky$ ) by unifying the skyline nodes of these two lists and removing those skyline nodes that are dominated by some other skyline nodes in the resultant sky list. In the example of Figure 9,  $u = (6, 8)$ ,  $u_1 = (6, 7)$  and  $u_2 = (5, 8)$ .  $u_1.sky = \{(3, 2)\}$ ,  $u_2.sky = \{(3, 5), (1, 6)\}$ .  $u.sky = u_1.sky \uplus u_2.sky = \{(3, 2), (1, 6)\}$  because  $(3, 2) \prec (3, 5)$ . Some other open nodes (e.g.,  $(5, 2)$  and  $(3, 8)$ ) connecting to  $u$  are also dominated by some skyline nodes of  $u$ . Therefore, they do not appear in  $u.sky$ .

Closed connections are detected when the algorithm scans to a closed node  $u$ . Each skyline node of  $u$  forms a closed connection with  $u$ . The detected closed connections are further filtered by the parameter  $l$  based on their matching scores. This way of maintaining sky lists guarantees that all maximal  $l$ -connections are returned by the Skywave algorithm. This is because the pruned open nodes in the sky lists cannot form maximal connections with any closed node.

The complexity of the Skywave algorithm is  $O(r'mn)$  where  $r'$  is the average number of skyline nodes of a node which can be bounded by  $O(m + n)$ . Compared to the Flooding algorithm, the Skywave algorithm scans the  $\varepsilon$ -matching matrix only in one pass. However, each node in the Skywave algorithm has to maintain a sky list. The union of the sky lists is a costly operation. A merit of the Flooding algorithm is that it is memoryless, i.e., each node does not need to remember the open nodes connected to it because each flooding is destined for the open node initiating that flooding. We shall compare the performance of these two algorithms in our experimental study later.

---

**Algorithm 3 The Skywave algorithm**

---

**Input:**  $M^{m \times n}$ , a matching matrix.

**Input:**  $l$ , the specified parameter in the  $l$ - $\varepsilon$ -join algorithm.

**Output:**  $O = \{o_i\}$  all closed  $l$ -connections in  $M$ .

```
1. for  $i = 1 : m$  do
2.   for  $j = 1 : n$  do
3.     if  $M[i, j] = 1$  then
4.        $u \leftarrow (i, j)$ 
5.       if  $M[i, j - 1] = 1$  or  $M[i - 1, j] = 1$  then
6.          $u.sky \leftarrow (i, j - 1).sky \uplus (i - 1, j).sky$ 
7.       else
8.         if  $M[i - 1, j - 1] = 1$  then
9.            $u.sky \leftarrow (i - 1, j - 1).sky$ 
10.      else
11.         $u.sky \leftarrow \{u\}$ 
12.      if  $u$  is a closed node then
13.        for  $u' \in u.sky$  do
14.           $o \leftarrow u' \triangleright u$  //  $o$  is a closed connection
15.          if  $s(o) \geq l$  then
16.            insert  $o$  into  $O$  //  $o$  is a closed  $l$ -connection
```

---

### C. Extracting Maximal $l$ -Connections

The closed  $l$ -connections detected from the refinement step are not all maximal connections. Some of them may be dominated by some other closed  $l$ -connections. The  $l$ - $\varepsilon$ -join only retrieves those maximal  $l$ -connections which are not dominated by any other connections. Extracting maximal  $l$ -connections from closed  $l$ -connections is not costly. We simply use Algorithm 4 to extract maximal  $l$ -connection from those closed  $l$ -connections, as a final step of the WTSJ algorithm.

---

**Algorithm 4 The maximal  $l$ -connection checking algorithm**

---

**Input:**  $\{o_1, \dots, o_n\}$ ,  $n$  closed  $l$ -connections. A connection  $o$  is represented as  $(i_1, j_1) \triangleright (i_2, j_2)$ .

**Output:**  $A^{n \times 1}$ ,  $A[t] = 1$  if  $o_t$  is a maximal  $l$ -connection.

```
1.  $A = 1^{n \times 1}$ 
2.  $R_a$  ranks all connections based on  $o.i_1$  in decreasing order
3.  $R_b$  ranks all connections based on  $o.i_2$  in increasing order
4.  $R_c$  ranks all connections based on  $o.j_1$  in decreasing order
5.  $R_d$  ranks all connections based on  $o.j_2$  in increasing order
6. for  $t = 1 : n$  do
7.    $r_a$  the smallest rank of  $o$  whose  $o.i_1 < o_t.i_1$  in  $R_a$ 
8.    $r_b$  the smallest rank of  $o$  whose  $o.i_2 > o_t.i_2$  in  $R_b$ 
9.    $r_c$  the smallest rank of  $o$  whose  $o.j_1 < o_t.j_1$  in  $R_c$ 
10.   $r_d$  the smallest rank of  $o$  whose  $o.j_2 > o_t.j_2$  in  $R_d$ 
11.   $r_x = \min(r_a, r_b, r_c, r_d)$ 
12.  for  $o_k \in R_x$  and ranked less than  $r_x$  do
13.    if  $o_t \prec o_k$  and  $t \neq k$  then
14.       $A[k] \leftarrow 0$ 
```

---

The WTSJ algorithm introduced so far requires all pair-wise distances of elements within two time series to be computed because any node may contribute to a closed connection. The computation of the  $\varepsilon$ -matching matrix is very costly if all pair-

wise distances of elements are computed. Subsequently, we therefore examine techniques to reduce the  $\varepsilon$ -matching matrix computational overhead in the next section.

## V. WTSJ WITH SEQUENCE SUMMARIZATION

In this section, we introduce a summarization technique on time series to bound the distances of elements so that most pair-wise distance computation can be pruned. We then generalize the problem of  $l$ - $\varepsilon$ -join over two time series to the  $l$ - $\varepsilon$ -join of a given querying time series with a database of time series. Some indexing techniques can be applied to index the summaries of time series within a database.

### A. Time Series Summarization

A major computational cost of time series join lies in computing the pair-wise distances of elements, whose complexity is  $O(mn)$  for the join of two time series with lengths of  $m$  and  $n$ . We stand to benefit if the computation of pair-wise distances can be substantially reduced.

We observe that for many time series sampled with fine granularity, especially for motion capture sequences, the consecutive elements of a time series are often quite similar. This motivates us to summarize similar consecutive elements as a sphere with a small radius. A time series is then partitioned into a number of blocks, with each block summarizing a bunch of sequential entries as a sphere. The distance of an element to any element summarized by the block can be lower-bounded by the closest distance of the element to the sphere representing the block. Therefore, for computing the  $\varepsilon$ -matching matrix, reduction over actual computation of pair-wise distances of elements (entries) can then be achieved.

Figure 10 uses an example to show how a time series is summarized into a number of blocks. Given a time series  $P$ , it is partitioned into a number of blocks  $\{\hat{P}[k]\}$ . Each block is represented as  $\hat{P}[k] = (P[c_k], s_k, e_k)$ , where  $s_k$  and  $e_k$  are starting and ending entries of the block,  $P[c_k]$  is the reference element (or reference point) of the block  $\hat{P}[k]$ . The partition should guarantee that: 1)  $s_k \leq c_k \leq e_k$ ; 2) for each element  $P[i]$  in block  $\hat{P}[k]$ ,  $d(P[i], P[c_k]) \leq \delta$ , where  $\delta$  is a given distance threshold for time series summarization; and 3)  $s_{k+1} = e_k + 1$ .

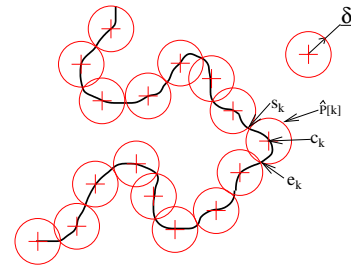


Fig. 10. Block-wise time series summarization

The partition of a time series starts from the starting element of the time series. For each block  $\hat{P}[k]$ , the reference element is first located by sequentially scanning from the starting

element  $s_k$  of the block. The last element satisfying that the distances from this element to the other scanned elements (from  $s_k$  to this element) of this block are all within  $\delta$ , is treated as the reference element  $c_k$ . Then, the scan continues until it reaches the first element whose distance to the reference element is larger than  $\delta$ . The element before this terminating element is treated as the ending element  $e_k$  of the block. In this way, all elements between elements  $s_k$  and  $e_k$  are within a distance  $\delta$  to the reference element  $c_k$ . The block scanning is done incrementally. The starting element of the next block is next to the ending element of the last block. The algorithm of time series summarization is outlined in Algorithm 5.

---

**Algorithm 5 Time series summarization**


---

**Input:**  $P$ , a time series of length  $m$

**Input:**  $\delta$ , the error bound of a block

**Output:**  $\{\hat{P}[k]\}$ , a number of blocks extracted from  $P$ , with each block  $\hat{P}[k] = (P[c_k], s_k, e_k)$

```

1.  $k \leftarrow 1; i \leftarrow 1$ 
2. while  $i \leq m$  do
3.    $s \leftarrow i; i++$ ;  $oneblockdone \leftarrow false$ 
4.   while  $i \leq m$  do
5.      $j \leftarrow s$ 
6.     while  $j < i$  and  $d(P[j], P[i]) \leq \delta$  do
7.        $j++$ 
8.     if  $j = i$  then
9.        $i++$ 
10.    else
11.       $c \leftarrow i - 1; j \leftarrow c + 1$ 
12.      while  $j \leq m$  and  $d(P[j], P[c]) \leq \delta$  do
13.         $j++$ 
14.       $e_k \leftarrow j - 1; \hat{P}[k] \leftarrow (P[c], s, e)$ 
15.       $k++$ ;  $i \leftarrow j$ ;  $oneblockdone \leftarrow true$ ;
16.      break
17.    if  $i > m$  and  $!oneblockdone$  then
18.       $\hat{P}[k] = (P[m], s, m)$ 

```

---

### B. Time Series Join on Block-wise Sequences

The summarization of time series as a number of blocks brings the benefit that the distances of the other elements or blocks to the elements within a block are lower-bounded because a metric distance function  $d(\cdot, \cdot)$  is applied to measure the distances between elements. Given a block  $\hat{P}[k] = (P[c_k], s_k, e_k)$  and an element  $Q[i]$ , the distance of an element  $P[j]$  within the block  $\hat{P}[k]$  is bounded as  $d(Q[i], P[c_k]) - \delta \leq d(Q[i], P[j]) \leq d(Q[i], P[c_k]) + \delta$ . Based on this, as long as  $d(Q[i], P[c_k]) > \delta + \varepsilon$ , we can ensure that  $Q[i]$  will not be  $\varepsilon$ -matching to any elements within  $\hat{P}[k]$ . Therefore, a number of pair-wise distance computations between  $Q[i]$  and those elements within  $\hat{P}[k]$  are pruned.

The lower bounding techniques can also be extended to distances between two blocks. Given two blocks  $\hat{Q}[r]$  and  $\hat{P}[k]$ , the distance between an element  $Q[i]$  in  $\hat{Q}[r]$  and another element  $P[j]$  in  $\hat{P}[k]$  is bounded as  $d(Q[c_r], P[c_k]) - 2\delta \leq d(Q[i], P[j]) \leq d(Q[c_r], P[c_k]) + 2\delta$ . We can ensure that no

elements in  $\hat{Q}[r]$  are  $\varepsilon$ -matching with any elements in  $\hat{P}[k]$  if  $d(Q[c_r], P[c_k]) > \varepsilon + 2\delta$ . Therefore, we save lots of pair-wise distance computation between elements in  $\hat{Q}[r]$  and those in  $\hat{P}[k]$ .

With the support of lower bounding distances between blocks, we can partition two time series into blocks, and compute block-wise distances before we conduct actual distance computation between elements of the two time series. Given two blocks  $\hat{Q}[r]$  and  $\hat{P}[k]$ , we define the block-wise distance of these two blocks as the lower bounding distance of elements between  $\hat{Q}[r]$  and  $\hat{P}[k]$ , i.e.,  $d(\hat{Q}[r], \hat{P}[k]) = \min(d(Q[c_r], P[c_k]) - 2\delta, 0)$ . Two blocks  $\hat{Q}[r]$  and  $\hat{P}[k]$  are  $\varepsilon$ -matching if  $d(\hat{Q}[r], \hat{P}[k]) \leq \varepsilon$ .

Suppose two time series  $P$  and  $Q$  are partitioned into  $\hat{m}$  and  $\hat{n}$  blocks respectively, we can build a block-wise  $\varepsilon$ -matching matrix  $\hat{M}^{\hat{m} \times \hat{n}}$ . The entry  $\hat{M}[i, j]$  is marked as 1 if  $\hat{P}[i]$  and  $\hat{Q}[j]$  are  $\varepsilon$ -matching. Figure 11 shows an example of block-wise  $\varepsilon$ -matching matrix. Due to the summarization of time series, the size of the block-wise  $\varepsilon$ -matching matrix  $\hat{M}$  should be much smaller than that of the pair-wise  $\varepsilon$ -matching matrix  $M$ . Therefore,  $\hat{M}$  can be efficiently computed.

$\hat{M}$	$\hat{Q}$	1	2	3	4	5	6	7
6								
5								
4								
3						1		
2				1	1			
$\hat{P}$	1							

Fig. 11. The block-wise  $\varepsilon$ -matching matrix

The filtering algorithm can also be applied to  $\hat{M}$  by taking the length of blocks into consideration when detecting local patterns within the summarizing vectors of the  $\varepsilon$ -matching matrix. For those candidate matching matrices extracted from  $\hat{M}$ , actual distances between elements are computed only within those  $\varepsilon$ -matching blocks of the candidate matching matrices. They are then processed as common  $l$ - $\varepsilon$ -joins of time series.

### C. Join over Multiple Time Series

The above discussion of time series join focuses on the join of two time series only. Two time series are connectable if some maximal  $l$ -connection can be detected from  $l$ - $\varepsilon$  join. Our time series join problem can also be extended to the join of two time series sets (as the trajectory join problem defined in [13]), in which pairs of connectable time series (with each series in a pair from one dataset) are retrieved. For the motion synthesis purpose, we define the connectable sequence retrieval problem as: given a querying time series  $Q$  and a threshold  $\varepsilon$ , retrieve all connectable sequences that have maximal  $l$ -connections with  $Q$  from a database  $D = \{P_i\}$ .

The baseline solution of the connectable sequence retrieval problem is to compute  $l$ - $\varepsilon$ -join of  $Q$  and  $P_i$  one by one. However, as time series in the database can be partitioned into blocks, it will be beneficial if all the blocks can be

indexed, so that  $\varepsilon$ -matching blocks of the blocks extracted from the querying time series can be efficiently identified using range queries over some spatial access methods. A common spatial index such as R-tree [26] can be applied to index the reference points of blocks of low dimensionality. For time series of high-dimensional data such as human motion data, high-dimensional indexing techniques such as iDistance [27] can be applied. The querying time series  $Q$  is also partitioned into a number of blocks. For each block  $\hat{Q}[k]$ , the  $\varepsilon$ -matching block of  $\hat{Q}[k]$  can be retrieved by issuing a range query over the spatial index. All those blocks whose reference points within a distance of  $\varepsilon + 2\delta$  to  $Q[c_k]$  are retrieved as  $\varepsilon$ -matching blocks of  $\hat{Q}[k]$ .

## VI. EXPERIMENTAL STUDY

### A. Experiment Settings

Our experimental platform was a PC with a Pentium 2.33G CPU and 3.25G RAM. We used the CMU motion capture database [21] which has around two thousand motion sequences of more than 100 subjects, with the lengths of sequences varying from 2 to 22,948. In our experiments, we selected 1136 human motion sequences whose lengths were from around 500 to 3000 from the CMU database as our test cases. The original motion sequences were three-dimensional trajectories of 31 joints of human bodies. All the sequences were sampled with a frequency of 120Hz.

Based on the geometric features proposed in [23], 30 numeric geometric relation features were extracted in our experiments. Among them, there were 10 pair-wise distance features (e.g., hand-to-hand distance), nine angle features (e.g., the angle between femur and tibia) and 11 plane features (e.g., whether left hand is before the body). In our experiments, each geometric feature was normalized into a real value within [0, 1]. Therefore, the time series data we used were actually time series of 30-dimensional data.

### B. Setting Effective Parameters

We first studied the properties of the used database by plotting the distribution of pair-wise distances of randomly chosen elements within the motion sequences in Figure 15. Most of the pairwise Euclidean distances were less than 2.0. The average pairwise distance of elements was 1.12.

To better understand the best region of parameter  $\varepsilon$  in  $l$ - $\varepsilon$  join, we measured the average distance of two elements with a fixed shifting step within the same sequences. The average distances over the shifting steps were plotted in Figure 12. It is obvious that the less the shifting step between two elements was, the better they were matched. We observed that two elements were generally similar if they were within a sliding step of 20 (i.e., two snapshots with a shift of around 0.17 seconds in the same motion sequence). According to the results, the average distance of elements with a shifting step of 20 was around 0.3. Therefore, if we chose an  $\varepsilon$  between 0.2 and 0.3, it would correspond to an edit distance of elements with a shift of 0.1 to 0.17 seconds in a motion sequence, which was applicable to those motion synthesis approaches.

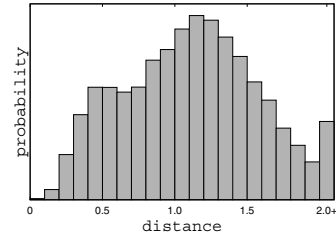


Fig. 15. Distribution of pair-wise distances of elements

Most of our experiments were done with the random selection of 10,000 pairs of motion sequences. We measured the average performance of the WTSJ algorithm over the 10,000 joins of two time series. We first studied the probability that two time series were connectable by varying parameters  $\varepsilon$  and  $l$ . The results are shown in Figure 13 and 14.

In Figure 13, parameter  $l$  is fixed as 60. The results show that setting the parameter  $\varepsilon$  as a value of [0.2, 0.3] can averagely detect 5 to 60 connectable sequences from our motion database of 1136 sequences. If  $\varepsilon$  is too large (e.g.,  $\varepsilon > 0.3$ ), the number of connectable sequences becomes quite large, and there will be lots of false positive matching subsequences. If  $\varepsilon$  is set too small (e.g.,  $\varepsilon < 0.2$ ), there will be rare connectable sequences retrieved from the database. In our experiments, we use a default value of  $\varepsilon = 0.25$  for those experiments where  $\varepsilon$  was not specified.

For the parameter  $l$ , a value of 40-80 is common as it specifies a matching of two motion subsequences lasting for at least 0.33 to 0.67 seconds. In Figure 14, we can see that a large  $l$  (e.g.,  $l > 100$ ) makes connectable sequences quite scarce. Conversely, a small  $l$  (e.g.,  $l < 30$ ) generates lots of connectable sequences from many trivial matching subsequences. The probability of finding connectable sequences from our database for  $l \in [40, 80]$  is 1.3%-5.1%. In our experiments, we use a default value of  $l = 60$  for those experiments without specifying  $l$ . Note that parameters  $\varepsilon$  and  $l$  are tunable. For different query sequences, users can choose different parameters for finding an appropriate number of connectable sequences from the motion database.

### C. Computing $\varepsilon$ -Matching Matrix

In this experiment, we compared the cost of pair-wise  $\varepsilon$ -matching matrix computation and that of block-wise  $\varepsilon$ -matching matrix computation. Figure 16 and 17 show the average cost of computing one  $\varepsilon$ -matching matrix and conducting one  $l$ - $\varepsilon$ -join under different values of parameters  $\varepsilon$  and  $l$  respectively. The results clearly show that the block-wise approach saves lots of computational cost for computing the  $\varepsilon$ -matching matrix. This is because the pair-wise approach computes all pair-wise distances of elements to obtain the  $\varepsilon$ -matching matrix. In the block-wise approach, most pair-wise distances are pruned with the use of block-wise distances. In the pair-wise approach, matching matrix computation accounts for the majority of the overall cost of time series join. In contrast, matching matrix computation in the block-wise approach only takes up a small part.

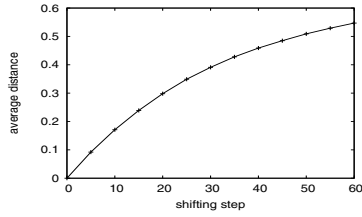


Fig. 12. Average distances of elements with a certain shift within the same sequences

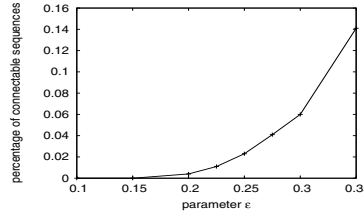


Fig. 13. The probability of two randomly selected time series are connectable over  $\epsilon$

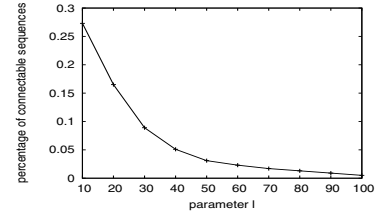


Fig. 14. The probability of two randomly selected time series are connectable over  $l$

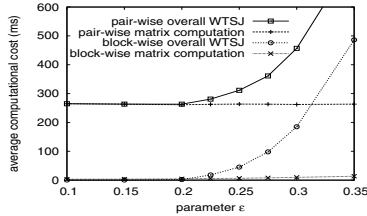


Fig. 16. Pair-wise vs. block-wise  $\epsilon$ -matching matrix computation over  $\epsilon$

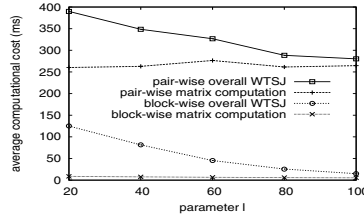


Fig. 17. Pair-wise vs. block-wise  $\epsilon$ -matching matrix computation over  $l$

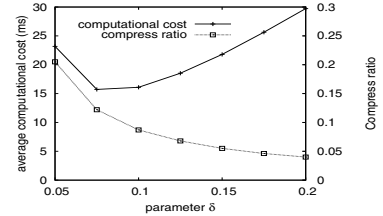


Fig. 18. The impact of parameter  $\delta$  on block-wise  $\epsilon$ -matching matrix computation

Figure 16 also shows that the cost of block-wise  $\epsilon$ -matching matrix computation increases slightly when  $\epsilon$  is enlarged. This is because there will be more  $\epsilon$ -matching blocks which cannot help to prune pair-wise distances when  $\epsilon$  is enlarged. The results also show that block-wise  $\epsilon$ -matching matrix computation is quite scalable in terms of various values of the parameters  $\epsilon$  and  $l$  because the major computational cost of time series join is always taken up by join algorithms rather than by distance computation.

Note that the parameter  $\delta$  affects the compression ratio of time series. Therefore, it will impact the computational cost of the  $\epsilon$ -matching matrix (shown in Figure 18). When  $\delta$  is enlarged, the compression ratio drops because one block can summarize more elements. However, computational cost does not always drop when  $\delta$  is enlarged. This is because the lower bounds provided by blocks become looser when parameter  $\delta$  is enlarged. As a result, more  $\epsilon$ -matching blocks are detected. Less pruning power will be achieved by block-wise distance computation. In our experiments, we use  $\delta = 0.1$  where lower computational cost of  $\epsilon$ -matching matrix was achieved.

#### D. Warping Time Series Join

In the WTSJ algorithm, the filtering step and the refinement step run in a pipeline manner, and do not interfere with each other. Therefore, we study the performance of the filtering step (the filtering algorithm) and that of the refinement step (the Flooding and Skywave algorithms) respectively.

1) *The filtering step:* The filtering algorithm seeks to prune those regions within the  $\epsilon$ -matching matrix where it is guaranteed that no closed  $l$ -connections exist. Figure 19 and 20 show the overall computational cost of two time series join with and without the filtering algorithm under different values of the parameters  $\epsilon$  and  $l$ . Note that the cost of computing the  $\epsilon$ -matching matrix is negligible and the dominant costs are the filter-and-refine steps of the WTSJ algorithm, which has been shown in Figure 16 and 17. Figure 19 shows that the filtering algorithm is quite effective when  $\epsilon$  is small (e.g.,

$\epsilon \leq 0.2$ ). It can even prune all computations of the refinement step. The performance of the filtering algorithm drops when  $\epsilon$  is enlarged because there are more nodes in the  $\epsilon$ -matching matrix. Figure 20 shows that the filtering algorithm achieves better pruning power when  $l$  is large because less local patterns will be detected in the filtering algorithm when  $l$  is enlarged. As a result, there will be fewer candidate matching matrices detected from the filtering step.

2) *The refinement step: Flooding vs. Skywave:* Finally, we compare the performance of the two different solutions of the refinement step: the Flooding algorithm and the Skywave algorithm under various values of the parameters  $\epsilon$  and  $l$  in Figures 21 and 22 respectively. It is obvious that the Flooding algorithm performs better than the Skywave algorithm in all our test cases. This is because operations of skyline union in the Skywave algorithm are quite costly when the number of nodes in sky lists is large (e.g., around 10). The Flooding algorithm works better because it is memoryless, i.e., it does not need to dynamically maintain a data structure recording skyline nodes even when many intermediate nodes are scanned multiple times.

3) *A querying example:* Considering the overall computational cost of warping time series join, one join of two time series using the Flooding algorithm consumes around 52ms in average. Given a querying motion sequence, retrieving all connectable sequences from our motion database will consume around one minute in average using the WTSJ algorithm. Note that the connectable sequences can be retrieved in an online manner. Those connectable sequences detected early can be immediately fed to users while the join of motion sequences is still running.

For simplicity, we use the querying example in [22] to show the effectiveness of retrieving connectable motion sequences with WTSJ. The querying sequence is a swing action in golf (sequence 64\_01<sup>1</sup>). We retrieve the connectable sequences of

<sup>1</sup>Videos of all motion sequences are available at <http://mocap.cs.cmu.edu>

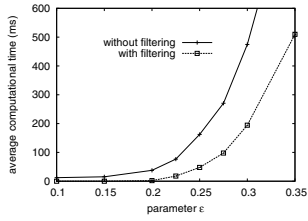


Fig. 19. The effectiveness of the filtering algorithm when  $\varepsilon$  is varied

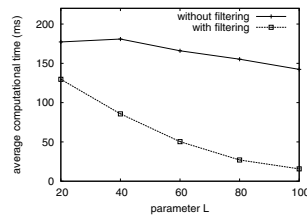


Fig. 20. The effectiveness of the filtering algorithm when  $l$  is varied

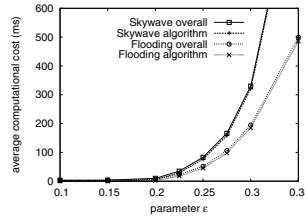


Fig. 21. Skywave vs. Flood when  $\varepsilon$  is varied

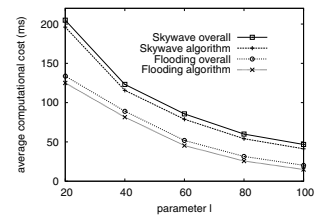


Fig. 22. Skywave vs. Flood when  $l$  is varied

this sequence using  $l$ - $\varepsilon$ -join with default parameter settings. All maximal  $l$ -connections and connectable sequences are shown in Table II. Motion sequences containing a swing action in our database are all retrieved as connectable sequences of the querying sequence. As we can see, the scaling difference of matching subsequences can be up to 0.724. The time cost for answering this query is quite small (only 2.0 seconds) because most sequences in the database are not connectable to the querying sequence.

TABLE II

RESULTS OF A CONNECTABLE SEQUENCE RETRIEVAL USING  $l$ - $\varepsilon$ -JOIN

subseq. in Q	subseq. in P	seq. P	time scale	subseq. in Q	subseq. in P	seq. P	time scale
[163:225]	[196:274]	64_02	0.797	[163:225]	[104:170]	64_07	0.940
[293:447]	[329:492]	64_02	0.945	[293:447]	[207:375]	64_07	0.917
[163:225]	[105:191]	64_03	0.724	[163:225]	[083:148]	64_08	0.955
[293:447]	[245:404]	64_03	0.969	[293:447]	[194:358]	64_08	0.939
[163:225]	[121:188]	64_04	0.926	[163:225]	[161:233]	64_09	0.863
[293:447]	[224:374]	64_04	1.026	[293:447]	[259:433]	64_09	0.886
[163:225]	[094:172]	64_05	0.797	[163:225]	[202:279]	64_10	0.808
[293:447]	[208:382]	64_05	0.886	[293:370]	[298:385]	64_10	0.886
[163:225]	[102:171]	64_06	0.900	[384:447]	[398:462]	64_10	0.985
[293:447]	[214:368]	64_06	1.000				

## VII. CONCLUSION

In this paper, we have first addressed the warping time series join problem by defining it as the  $l$ - $\varepsilon$ -join problem of sequences, which seeks to detect maximal  $l$ -connections from the  $\varepsilon$ -matching matrix of two time series. The major computational cost for processing  $l$ - $\varepsilon$ -join of time series arises from two aspects. One is the computation of the  $\varepsilon$ -matching matrix. The other is the maximal  $l$ -connections over the  $\varepsilon$ -matching matrix. To address the first aspect, we have proposed the time series summarization approach to reduce the size of the time series and to prune much of the pair-wise distance computation. To address the second aspect, we have proposed a warping time series join algorithm called WTSJ, which includes two major steps: filtering and refinement. Our experiments show the effectiveness of our proposed techniques for time series join of human motion sequences. With our solution, retrieving connectable motion sequences from a large human motion database can be efficiently handled.

## REFERENCES

- [1] O. Arikan and D. A. Forsyth, "Interactive motion generation from examples," in *SIGGRAPH*, 2002, pp. 483–490.
- [2] R. Heck and M. Gleicher, "Parametric motion graphs," in *SI3D*, 2007, pp. 129–136.
- [3] E. Hsu, K. Pulli, and J. Popović, "Style translation for human motion," in *SIGGRAPH '05*. New York, NY, USA: ACM, 2005, pp. 1082–1089.
- [4] V. Athitsos, P. Papapetrou, M. Potamias, G. Kollios, and D. Gunopulos, "Approximate embedding-based subsequence matching of time series," in *SIGMOD Conference*, 2008.
- [5] Y. Chen, M. A. Nascimento, B. C. Ooi, and A. K. H. Tung, "Spade: On shape-based pattern detection in streaming time series," in *ICDE*, 2007, pp. 786–795.
- [6] W.-S. Han, J. Lee, Y.-S. Moon, and H. Jiang, "Ranked subsequence matching in time-series databases," in *VLDB*, 2007, pp. 423–434.
- [7] Y. Sakurai, C. Faloutsos, and M. Yamamuro, "Stream monitoring under the time warping distance," in *ICDE*, 2007, pp. 1046–1055.
- [8] B. Chiu, E. J. Keogh, and S. Lonardi, "Probabilistic discovery of time series motifs," in *KDD*, 2003, pp. 493–498.
- [9] S. Papadimitriou and P. S. Yu, "Optimal multi-scale patterns in time series streams," in *SIGMOD Conference*, 2006, pp. 647–658.
- [10] P. Bakalov, M. Hadjieleftheriou, E. J. Keogh, and V. J. Tsotras, "Efficient trajectory joins using symbolic representations," in *Mobile Data Management*, 2005, pp. 86–93.
- [11] Y. Chen, "Towards a unified framework for efficient access methods and query operations in spatio-temporal databases," Ph.D. dissertation, Univ. of Michigan, 2008. [Online]. Available: [http://deepblue.lib.umich.edu/bitstream/2027.42/58540/1/yunc\\_1.pdf](http://deepblue.lib.umich.edu/bitstream/2027.42/58540/1/yunc_1.pdf)
- [12] J. Sun, Y. Tao, D. Papadias, and G. Kollios, "Spatio-temporal join selectivity," *Inf. Syst.*, vol. 31, no. 8, pp. 793–813, 2006.
- [13] P. Bakalov, M. Hadjieleftheriou, and V. J. Tsotras, "Time relaxed spatiotemporal trajectory joins," in *GIS*, 2005, pp. 182–191.
- [14] E. J. Keogh, T. Palpanas, V. B. Zordan, D. Gunopulos, and M. Cardle, "Indexing large human-motion databases," in *VLDB*, 2004, pp. 780–791.
- [15] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series," in *KDD Workshop*, 1994, pp. 359–370.
- [16] L. Chen, M. T. Özsu, and V. Oria, "Robust and fast similarity search for moving object trajectories," in *SIGMOD Conference*, 2005, pp. 491–502.
- [17] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Approximate string joins in a database (almost) for free," in *VLDB*, 2001, pp. 491–500.
- [18] T. Smith and M. Waterman, "The identification of common molecular," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.
- [19] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, pp. 403–410, 1990.
- [20] X. Cao, A. K. H. Tung, B. C. Ooi, K.-L. Tan, and S. C. Li, "String join using precedence count matrix," in *SSDBM*, 2004, pp. 345–348.
- [21] *CMU Graphics Lab Motion Capture Database*, <http://mocap.cs.cmu.edu/>.
- [22] Y. Chen, S. Jiang, B. C. Ooi, and A. K. H. Tung, "Querying complex spatio-temporal sequences in human motion databases," in *ICDE*, 2008, pp. 90–99.
- [23] M. Müller, T. Röder, and M. Clausen, "Efficient content-based retrieval of motion capture data," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 677–685, 2005.
- [24] C. Li and B. Prabhakaran, "Indexing of motion capture data for efficient and fast similarity search," *Journal of Computers*, vol. 1, no. 3, pp. 35–42, 2006.
- [25] T. Mukai and S. Kuriyama, "Geostatistical motion interpolation," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 1062–1070, 2005.
- [26] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD Conference*, 1984, pp. 47–57.
- [27] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "iDistance: An adaptive b<sup>+</sup>-tree based indexing method for nearest neighbor search," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 364–397, 2005.