

Indexing Large Human-Motion Databases

Please note that this is only a rough draft of the paper that will appear in VLDB 2004. If you are going to reference this work, please ask us for the final version of the paper.

Eamonn Keogh Themistoklis Palpanas Victor B. Zordan Dimitrios Gunopoulos

Department of Computer Science
University of California, Riverside
{eamonn, themis, vbz, dg}@cs.ucr.edu

Marc Cardle
Computer Laboratory
University of Cambridge
mpc33@cl.cam.ac.uk

ABSTRACT

Data-driven animation has become the industry standard for computer games and many animated movies and special effects. In particular, motion capture data recorded from live actors, is the most promising approach thus far for animating realistic human characters. However, the manipulation of such data for general use and re-use is not yet a solved problem. Many of the existing techniques dealing with editing motion rely on indexing for annotation, segmentation, and re-ordering of the data. Euclidean distance is inappropriate for solving these indexing problems because of the inherent variability found in human motion. The limitations of Euclidean distance stems from the fact that it is very sensitive to distortions in the time axis. A partial solution to this problem, Dynamic Time Warping (DTW), aligns the time axis before calculating the Euclidean distance. However, DTW can only address the problem of *local* scaling. As we demonstrate in this paper, *global* or *uniform scaling* is just as important in the indexing of human motion. We propose a novel technique to speed up similarity search under uniform scaling, based on bounding envelopes. Our technique is intuitive and simple to implement. We describe algorithms that make use of this technique, we perform an experimental analysis with real datasets, and we evaluate it in the context of a motion capture processing system. The results demonstrate the utility of our approach, and show that we can achieve orders of magnitude of speedup over the brute force approach, the only alternative solution.

Keywords

Motion Capture, Animation, Time Series, Indexing

1. INTRODUCTION

Data-driven animation has now become the industry standard for the production of computer games and many animated movies and special effects. The most promising and widely applied approach so far is the use of motion capture data. These

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

are motion data recorded from live actors, which can subsequently be used for animating realistic human characters. Nevertheless, the manipulation of such data for general use and re-use is still an open problem. Among the issues at hand, the semi-automatic annotation [3] and re-ordering of motion data [4][21][22] are appearing in animation research conferences and slowly trickling their way into games where realism, interactivity, and speed drive innovation.

Motion capture data, in its rawest form, is recorded with a few technologies, the most popular of which appears to be optical (see Vicon [35] and Motion Analysis [36] products) in which digital cameras record small reflective markers fixed to the human actor as he/she moves. Through multiple cameras and triangulation, three dimensional position traces for the markers are resolved faithfully. The markers can then be identified (as outer left knee, for example) and filtered. Motion capture allows the animation of a 3D model, where the data is mapped to the skeleton of the desired character and body orientations are determined (Figure 1).

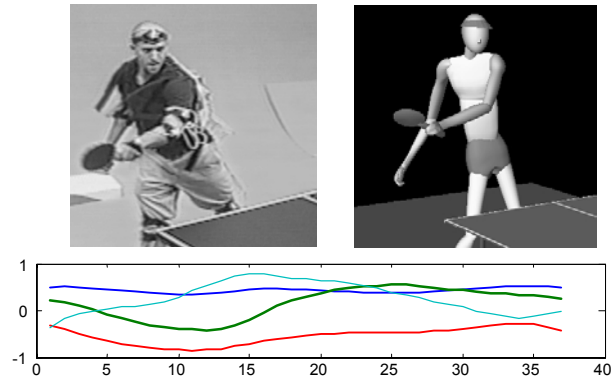


Figure 1: (Top Left) An actor being recorded using an Ascension magnetic system while playing table tennis. In post-processing, the data recorded from the actor's motion is manually segmented into motion time series (Bottom) and placed in a library that is later used to animate the simulated player shown (Top Right).

In practical applications, most motion capture data is stored in segmented sequences in a motion library, for example a modern sports game may contain thousands of motion data “clips”. The system, i.e. game engine in this case, selects and plays motions from the database [24]. Our approach aids in the creation and manipulation of such libraries by quickly finding instances of a given motion segment in the complete raw-data repository, e.g., *kicks* or *punches* in the case of a hand-to-hand

combat game. In addition to speeding up brute force searches, our main contribution is finding examples independent of the speed in which the actor performed these behaviors.

A major difficulty in indexing and matching motion streams (hereafter used interchangeably with “time series”) is the variability in the speed of human motion. For example, an actor may perform a fast or slow punch. Such variability can manifest itself as *uniform scaling*, a global stretching or shrinking of the time series (i.e., with respect to the time axis). In this work we introduce the first indexing technique to support uniform scaling. Our contributions can be summarized as follows.

- We motivate the need for similarity search under uniform scaling, and differentiate it from Dynamic Time Warping (DTW). Although the superiority of DTW over Euclidean distance is becoming increasingly apparent [1][8][17][33], the need for similarity search which is invariant to uniform scaling is not well understood.
- We introduce the first known lower bounding technique for uniform scaling. This technique allows us to index the time series in order to achieve fast similarity search under uniform scaling.
- We demonstrate the efficiency and effectiveness of our techniques with a comprehensive empirical evaluation on real datasets. We also evaluate our techniques using a motion capture processing system. These experiments validate the utility of the approach we propose.

The rest of this paper is organized as follows. In Section 2 we motivate the need to index motion streams under uniform scaling. Section 3 considers related work on indexing time series. We introduce the problem at hand formally in Section 4, before introducing our solution in Section 5. In Section 6 we describe algorithms that solve the problem in secondary storage. Section 7 offers a comprehensive empirical evaluation of our technique, and we conclude in Section 8.

2. MOTIVATING THE NEED FOR UNIFORM SCALING

In addition to the classic Euclidean and Dynamic Time Warping distance measures, the last decade has seen the introduction of dozens of new similarity measures for time series. Recent empirical studies, however, suggest that the majority of these measures are of dubious utility for real world problems [20]. We will therefore take the time to motivate the absolute need for uniform scaling in our domain.

A classic task in motion editing is the concatenation of short motion clips into a longer, plausible motion. For clarity let us consider a concrete example [29][34]. Imagine we have a motion sequence that contains several distinct motions, and then ends with a particular action, the drawing and aiming of a gun. We would like to append to this action sequence another sequence where the actor falls to the ground. While we have a library of perhaps thousands of sequences labeled “falling”, we must decide to which of these we should append our current sequence. The challenge in this case is to make sure that the transition is as smooth and natural as possible. For example, we

do not want the character’s left arm to instantaneously move from his/her side to above his/her head.

A simple way to guarantee natural plausible motion is to ensure that the suffix of the first motion, let’s say the last n data points, is an approximate match to the prefix of the candidate sequence, the first n data points. This way, instead of concatenating the two sequences end to end, they are allowed to overlap by n data points. Averaging or time warping can be used to smooth out any slight inconsistencies within the n overlapping data points. Figure 2 illustrates the basic idea with a simple problem, taken from a video segment. Although this example considers a one-dimensional time series, it can easily be extended to multi-dimensional time series, by combining the results for each degree of freedom, possibly weighted by their perceptual importance (i.e., arm motion may be more important than leg motion in some situations).

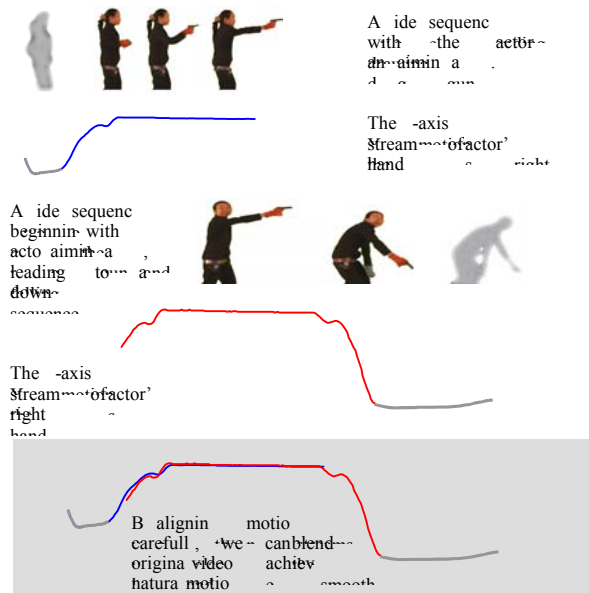


Figure 2: We can create a smooth transition between two video clips (Top and Center), by ensuring the prefix of one approximately matches the suffix of the other (Bottom).

In Figure 2, our contrived example happens to have a closely matching prefix/suffix pair. More generally however, the motion streams may occur at different speeds. Although the animator can trivially recognize this when he/she see it, human inspection does not scale to large databases. The importance of (time) scale invariance stems from the fact that small differences in scaling can greatly confuse distance calculations. This is well known anecdotally in the motion capture domain, and has also been observed in other similar domains. For example, in music retrieval has been reported [8]: “To achieve tempo invariance, the targets are stretched by 19 different scaling factors from 0.5 to 2.0.” Similar remarks can be found in the literature of gait analysis [13], handwritten archive indexing [27], bioinformatics [1] and data mining [7].

We can reiterate here the utility of uniform scaling with a simple experiment. We created 3 pairs of time series, where each pair was created using one of 3 functions, sine wave,

sawtooth I, and sawtooth II. Within each pair, the only difference between the time series is that we allow their length to vary in the range 256 ± 16 . We clustered them using two different distance measures, the classic Euclidean distance [6][11][15][18][19][23], and using uniform scaling, where we search over the best possible scaling, truncating off any unmatched suffix (see Section 4 for more formal details). The results are shown in Figure 3.

If these synthetic time series had been, examples of an actor’s gait, then using the Euclidean distance, a video clip corresponding to sequence 2 would be concatenated to its closest match, sequence 5. This would be a very abrupt and noticeable transition. In contrast, under uniform scaling, sequence 2 would be concatenated to sequence 1. In this case, the only difference in the resulting animation would be a slight change of pace (if we chose not to permanently rescale one of the sequences). This is why automatic matching of motion capture data *must* consider uniform scaling.

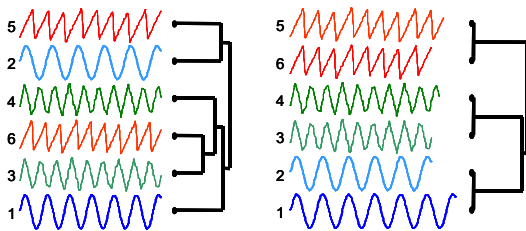


Figure 3: (Left) A clustering of 6 synthetic time series using Euclidean distance, and (Right) uniform scaling. Subjectively and objectively, the clustering on the right is correct.

Note that the generally useful tool of DTW is *not* the answer to this problem [12][17][33]. On the dataset above DTW is about 200 times slower than uniform scaling, and returns a dendrogram of $((2,((4,(6,3)),5)),1)$, which is no better than the Euclidean distance dendrogram. The problem is that DTW is designed to consider only local adjustments of the time axis, whereas it is global adjustments that are required to solve our problem.

To further demonstrate the difference between DTW and uniform scaling, we perform the following experiment. We record twice a one second snippet of an individual’s electrocardiogram. On the first occasion, the time series captures two heartbeats, while on the second (during exercise), captures three heartbeats. When we use DTW to measure the similarity of the two sequences, we get meaningless results, because DTW must match every point, and there is simply no sensible way to map two heartbeats to three (Figure 4(left)). In contrast, uniform scaling can stretch the faster heartbeat until finding a near perfect alignment (Figure 4(right)). As mentioned above, DTW can be useful to remove subtle local differences *after* uniform scaling has located the best global match [8].

3. RELATED WORK

The problem of indexing large time series databases has attracted great interest in the database community, and, at least for the Euclidean distance measure, may now be

regarded as a solved problem [6][11][15][18][19][28]. However, in recent years, there has been an increasing awareness that Euclidean distance is inappropriate for many real world applications [1][8][33]. The limitations of Euclidean distance stems from the fact that it is very sensitive to distortions in the time axis. A partial solution to this problem, DTW, essentially aligns the time axis before calculating the Euclidean distance [17]. However DTW can only address the problem of *local* scaling, and as we demonstrated above, uniform scaling is just as important in the motion capture editing domain. Similar observations hold for the Longest Common SubSequence measure [9][31].

The utility of uniform scaling has been noted before [1][2][7][26]. However, all previous work has focused on speeding up similarity search, when the scaling factor is *known* [7][16][25]. The feature that differentiates our work from all the rest is that we allow a user to issue a single query, and find the best match at *any* scaling.

Note that although we don’t generally know the scaling factor in advance, we may know upper and lower limits on the scaling factor, based on limitations of human biomechanics. For example, most people can only speed up their natural walk about 20% before changing their gait into a run [13].

There is exactly one other technique in the literature that allows similarity search under uniform scaling, while guaranteeing no false dismissals, the “*CD-criterion*” technique of [2]. While pioneering, we do not see this work as a complete solution to our problem for the following reasons. The algorithm can only test if sequences are within a user-supplied epsilon, and thus cannot be used for ranking, classification or clustering. The algorithm requires a parameter to be set; this parameter does not affect the accuracy, but it can affect the speedup. Finally the real weakness of the approach is that it only speeds up main memory search, and cannot be indexed. In fact, the authors suggest that indexing of uniform scaling “*appears infeasible*” [2], although this is exactly the contribution of this work.

In spite of these limitations, we empirically compare this work to our approach in Section 7.1.

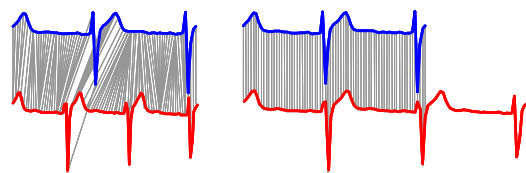


Figure 4: A visual contrast of DTW (left) and Uniform Scaling (right). Using DTW we cannot achieve an intuitive alignment between 2 heartbeats and 3 heartbeats, even if the sequences happen to be the same length. However uniform scaling, by stretching the bottom sequence can achieve a meaningful alignment.

4. THE UNIFORM SCALING PROBLEM

We begin by formally defining the uniform scaling problem.

Suppose we have two time series, a query Q , and a candidate match C , of length n and m respectively, where:

$$Q = q_1, q_2, \dots, q_i, \dots, q_n \quad (1)$$

$$C = c_1, c_2, \dots, c_j, \dots, c_m \quad (2)$$

For clarity of presentation we will assume that $n \leq m$, that is, C is always longer than or equal to Q . Thus, we are only interested in stretching the query to match some prefix of C . This assumption is only to simplify notation and does not preclude matching a time series by shrinking. For example, if the user wishes to perform a query of length 100, with the flexibility to shrink or stretch by 10%, the system simply interpolates the data down to 90 data points, and then searches for matches *stretched* by up to 22%.

If we wish to compare the two time series, and it happens that $n = m$, we can use the ubiquitous Euclidean distance [6][11][15][19]:

$$D(Q, C) \equiv \sqrt{\sum_{i=1}^n (q_i - c_i)^2} \quad (3)$$

Since the square root function is monotonic and concave, we can remove the square root step to get the squared Euclidean distance which gives identical rankings, clustering and classifications [20].

$$D(Q, C) \equiv \sum_{i=1}^n (q_i - c_i)^2 \quad (4)$$

In addition to the utility of slightly speeding up the calculations, working with this distance measure makes other optimizations possible, as well [20].

If n is smaller than m , then the distance measures introduced above are not defined. To compare the two time series in this case, we have several choices; we can truncate C , and compare Q to $[c_1, c_2, \dots, c_n]$, we can *stretch* Q to be of length m , or more generally, we can *stretch* Q to be of length p , ($n \leq p \leq m$), truncate off the last $m-p$ values of C , and then use the squared Euclidean distance. The informal idea behind *stretching* can be captured in the more formal definition of scaling. In order to scale time series Q to produce a new time series QP of length p , we use the formula:

$$QP_j = Q_{\lceil j * n/p \rceil}, 1 \leq j \leq p \quad (5)$$

Note that we can quickly obtain any scaling in $O(p)$ time. We call the ratio p/n the *scaling factor* or *sf*. Similarly, we use sf_{max} to denote the ratio m/n , which can be thought of as the *maximum* scaling factor. Figure 5 visually summarizes the above definitions.

If we wish to find the best scaled match between Q and C , we can simply test all possible scalings, as illustrated in Table 1.

The algorithm takes $O(p*(m-n))$ time and seems unworthy of any optimization effort. However, for real world datasets, rather than having a single candidate time series C , we are typically confronted with massive collections of possible candidate time series, which will denote as \mathbf{D} . In order to find

the best scaled match to a query Q in database \mathbf{D} , we can use a brute force algorithm as shown in Table 2.

Note that the time complexity for this algorithm is $O(|\mathbf{D}|*(m-n))$, which is simply untenable for large datasets.

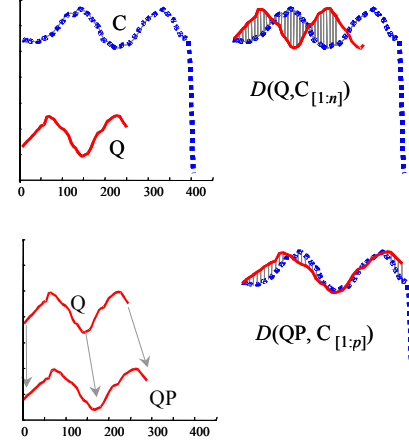


Figure 5. A visual summary of the notation introduced in this section. A) From (left) to (right) A candidate time series C , and a shorter query Q . The squared Euclidean distance between Q and the first n datapoints in C can be visualized as the sum of the squared lengths of the gray hatch lines. B) From (left) to (right) The query Q can be stretched to length p , producing a new time series QP . In this case, QP is a good match to the first p datapoints in C .

Table 1. An algorithm to find the best scaled match between two time series

```

procedure TestAllScalings( $Q, C$ )
     $BestMatchVal$  = inf;
     $BestScalingFactor$  = null;
    for  $p = n$  to  $m$ 
         $QP = \text{rescale}(Q, p)$ ;
         $Distance = \text{SquaredEuclideanDistance}(QP, C[1..p])$ ;
        if  $distance < BestMatchVal$ 
             $BestMatchVal = distance$ ;
             $BestScalingFactor = p/n$ ;
    return( $BestMatchVal, BestScalingFactor$ )
    
```

Table 2. An algorithm to find the best scaled match to query from a set of possible matches.

```

procedure SearchDatabaseforScaledMatch( $Q$ )
     $OverallBestTimeSeries$  = null;
     $OverallBestMatchVal$  = inf;
     $OverallBestScaling$  = null;
    for  $i = 1$  to number of time series in ( $\mathbf{D}$ )
        [ $dist, scale$ ] = TestAllScalings( $Q, C_i$ )
        if  $dist < OverallBestMatchVal$ 
             $OverallBestTimeSeries = i$ ;
             $OverallBestMatchVal = dist$ ;
             $OverallBestScaling = scale$ ;
    return( $OverallBestTimeSeries, OverallBestMatchVal, OverallBestScaling$ )
    
```

4.1 Speeding up Search with Lower Bounding

To speed up matching under uniform scaling we will rely on the classic idea of lower bounding. The intuition is the following. Given some technique for quickly calculating the minimum possible distance between the query and a candidate sequence at any possible scaling, we can prune off many calculations. Before calling the subroutine *TestAllScalings()*, we first perform the quick lower bounding test. If the lower bound distance between the candidate and the query is greater than the distance of the best-scaled match already seen, we can simply discard the candidate from consideration. There are two important properties that a lower bounding measure should have.

- It must be fast to compute. A measure that takes as long to compute as *TestAllScalings()* is of little use. We would like the time complexity to be at most linear in the length of the time series.
- It must be a relatively tight lower bound. A lower bound that is not tight, will not prune enough of the search space.

The idea of speeding up search using lower bounding is not new. In fact, it is the cornerstone of virtually every time series similarity search algorithm. However, while dozens of lower bounding measures are known for Euclidean distance [6][11][15][18][19][28], and three lower bounding measures are known for DTW [17][33], only one, recently introduced measure, the *CD-criterion*, is known for uniform scaling [2]. As we mentioned above, the original authors believe this technique is non-indexable, and in any case, as we will show in Section 7, the bounds are quite weak. Therefore, we propose a novel, indexable, and tight lower bounding measure for uniform scaling.

It is important to note here that the lower bounding technique and all the algorithms we describe in this study for the efficient solution of the uniform scaling problem are exact. This means that we are guaranteed to find all the solutions we are looking for, with no false dismissals. The essence of the techniques we propose is that they can effectively prune the search space, by excluding candidate time series that cannot be part of the solution. The result is considerable savings in computation time, since we do not have to perform the expensive distance calculations for every time series in the database.

5. OUR SOLUTION

We will begin by showing how we can lower bound sequences of arbitrary lengths in main memory. Since indexing structures degrade with dimensionality, we will further show how we can lower bound dimensionality reduced representations of the time series.

5.1 Lower Bounding in Main Memory

In order to create a lower bounding distance measure for uniform scaling, we will generate a bounding envelope. Bounding envelopes were introduced in [17] to lower bound DTW, and since then they have sparked a flurry of research

activity [12][23][27][31][33]. While the principle is the same here, the definitions of the envelope are *very* different. In particular, we create two sequences U and L, such that:

$$U_i = \max(c_{\lfloor (i-1)*m/n \rfloor + 1}, \dots, c_{\lfloor i*m/n \rfloor}) \quad (6)$$

$$L_i = \min(c_{\lfloor (i-1)*m/n \rfloor + 1}, \dots, c_{\lfloor i*m/n \rfloor}) \quad (7)$$

These sequences can be visualized as bounding the first n points of the time series C. Figure 6 shows some examples.

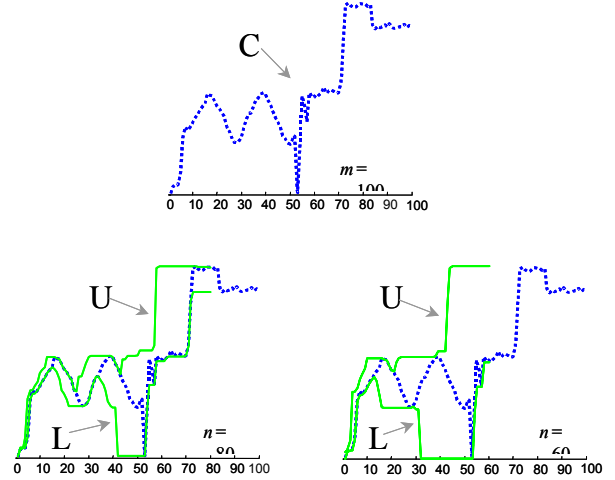


Figure 6. (Top) A time series C of length 100. (Bottom Left) The time series shrouded by upper and lower envelopes U and L with lengths 80. (Bottom Right) The same time series shrouded by upper and lower envelopes U and L with lengths 60.

Having defined U and L, we can now introduce the lower bounding function, LB_US , which lower bounds the distance between Q and C for any scaling factor sf , $1 < sf \leq sf_{max}$.

$$LB_US(Q,C) = \sum_{i=1}^n \begin{cases} (q_i - U_i)^2 & \text{if } q_i > U_i \\ (q_i - L_i)^2 & \text{if } q_i < L_i \\ 0 & \text{otherwise} \end{cases} \quad (8)$$

This function can be visualized as the squared Euclidean distance between any part of the query time series not falling within the envelope and the nearest (orthogonal) corresponding section of the envelope. Figure 7 illustrates this idea.

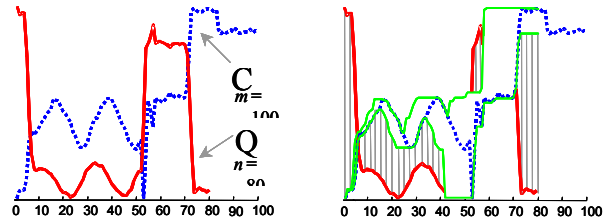


Figure 7. (Left) A time series C and a shorter query Q. (Right) A visualization of the lower-bounding function $LB_US(Q,C)$. Note that any part of query time series Q that falls inside the bounding envelope is ignored. Otherwise the distance corresponds to the sum of the squared straight line distances from the query to the nearest point in the envelope (the gray hatch lines).

We can now prove that $LB_US(Q,C)$ is a lower bound for the distance between Q and C under uniform scaling.

Lemma 1 The distance $LB_US(Q,C)$ lower bounds the squared Euclidean distance between any scaling of Q , and the appropriate prefix of C .

Proof: We want to prove

$$LB_US(Q,C) \leq D(Q,C) \cdot$$

We begin by introducing the definitions:

$$\sum_{i=1}^n \begin{cases} (q_i - U_i)^2 & \text{if } q_i > U_i \\ (q_i - L_i)^2 & \text{if } q_i < L_i \\ 0 & \text{otherwise} \end{cases} \leq \sum_{j=1}^p (q_j - c_j)^2$$

We will show that every term on the LHS is matched by a larger or equal term on the RHS. In addition, there are some additional, positive terms on the RHS that are unmatched. For the moment let us just consider the case where $q_i > U_i$.

$$(q_i - U_i)^2 \leq (q_j - c_j)^2$$

Since both sides are positive by definition, we can square both sides.

$$q_i - U_i \leq q_j - c_j$$

The terms on each side have different subscripts, however the *scaling factor* can be used to relate both sides.

$$q_i - U_i \leq q_{\lfloor i \cdot \text{sf} \rfloor} - c_{\lfloor i \cdot \text{sf} \rfloor}$$

By Eq. 5 we can see that q_i and $q_{\lfloor i \cdot \text{sf} \rfloor}$ have the same value, so we have

$$q_{\lfloor i \cdot \text{sf} \rfloor} - U_i \leq q_{\lfloor i \cdot \text{sf} \rfloor} - c_{\lfloor i \cdot \text{sf} \rfloor} \quad \text{which simplifies to}$$

$$U_i \geq c_{\lfloor i \cdot \text{sf} \rfloor} \quad \text{We can replace } U_i \text{ with its Eq. 6 definition}$$

$$\max(c_{\lfloor (i-1) \cdot \text{sf} \rfloor + 1}, \dots, c_{\lfloor i \cdot \text{sf} \rfloor}) \geq c_{\lfloor i \cdot \text{sf} \rfloor}$$

The above is true, since the maximum of any set is clearly larger than or equal to any element of the set. The case where $q_i < L_i$ yields to a similar argument, and the final case is trivial, since 0 is always less than or equal to any squared real number.

Therefore, every term on the LHS is matched by a larger or equal term on the RHS, and $LB_US(Q,C) \leq D(Q,C)$. ■

5.2 Lower Bounding in Index Space

As noted in Section 4.1, if we have a distance measure that is expensive in terms of CPU time, we can dramatically speed up similarity search using a tight lower bound. However, if the majority of the data exists on secondary storage, the CPU costs may be dwarfed by the disk (or tape) access time. The solution is to *index* the data. Having defined the bounding envelopes, we proceed in a manner similar to previous work [12][17][23][27][31][33].

We have previously denoted a time series as $Q = q_1, \dots, q_n$. We assume each sequence in our database is n units long. Let N be the dimensionality of the space we wish to index ($1 \leq N \leq n$). For convenience, we assume that N is a factor of n .

A time series Q of length n can be represented in N dimensional space by a vector $\bar{Q} = \bar{q}_1, \dots, \bar{q}_N$. The i^{th} element of \bar{Q} is calculated using the following equation:

$$\bar{q}_i = \frac{N}{n} \sum_{j=\frac{n}{N}(i-1)+1}^{\frac{n}{N}i} q_j \quad (9)$$

In other words, to reduce the time series from n dimensions to N dimensions, the data is divided into N equal sized “frames”. The mean value of the data falling within a frame is calculated and a vector of these values becomes the data reduced representation. The representation can best be visualized as an attempt to model the original time series with a linear combination of box basis functions as shown in Figure 8.

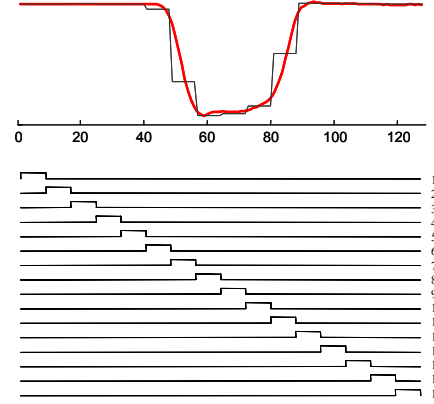


Figure 8. The PAA representation can be visualized as an attempt to model a sequence with a linear combination of box basis functions. In this example, the first 128 data points of a motion stream time series are reduced to 16 dimensions.

In Section 5.1, we discussed the lowering bounding function LB_US . However, calculating this function requires n values. Since n may be in the order of hundreds for realistic human motion, and multi-dimensional index structures begin to degrade rapidly somewhere above 16 dimensions, we need a way to create a lower, N -dimensional version of the function, where N is a number that can be reasonably handled by a multi-dimensional index structure. We also need this lower dimension version of the function to lower bound LB_US (and therefore, by transitivity, uniform scaling).

We begin by creating special *Piecewise Constant Approximations* (PAAs) of U and L , which we will denote as \hat{U} and \hat{L} . Although they are piecewise constant approximations, the definitions of \hat{U} and \hat{L} differ from those we have seen in Eq. 6 and 7. In particular, we have

$$\hat{U}_i = \max\left(U_{\frac{n}{N}(i-1)+1}, \dots, U_{\frac{n}{N}i}\right) \quad (10)$$

$$\hat{L}_i = \min\left(L_{\frac{n}{N}(i-1)+1}, \dots, L_{\frac{n}{N}i}\right) \quad (11)$$

We can visualize \hat{U} and \hat{L} as the piecewise constant functions which bound, without intersecting, U and L , respectively. Figure 9 illustrates this intuition.

We are now able to define the low dimension, lower bounding function, which we denote as $MINDIST()$. Given a query sequence Q , transformed to \bar{Q} by Eq. 9, and a candidate sequence C , with its companion PAA functions $\hat{R} = \{\hat{U}, \hat{L}\}$, the following function lower bounds LB_US

$$MINDIST(\bar{Q}, \hat{R}) = \sqrt{\sum_{i=1}^N \frac{n}{N} \begin{cases} (\bar{q}_i - \hat{U}_i)^2 & \text{if } \bar{q}_i > \hat{U}_i \\ (\bar{q}_i - \hat{L}_i)^2 & \text{if } \bar{q}_i < \hat{L}_i \\ 0 & \text{otherwise} \end{cases}} \quad (12)$$

This function is visualized in Figure 10.

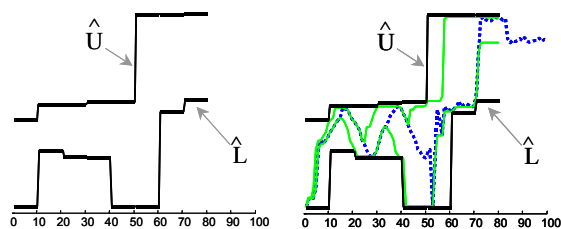


Figure 9. We can readily visualize \hat{U} and \hat{L} as the piecewise constant functions which bound, without intersecting, U and L , respectively. (Left) The \hat{U} and \hat{L} for the time series shown in Figure 5. (Right) The \hat{U} and \hat{L} shown overlaid on top of the generating time series.

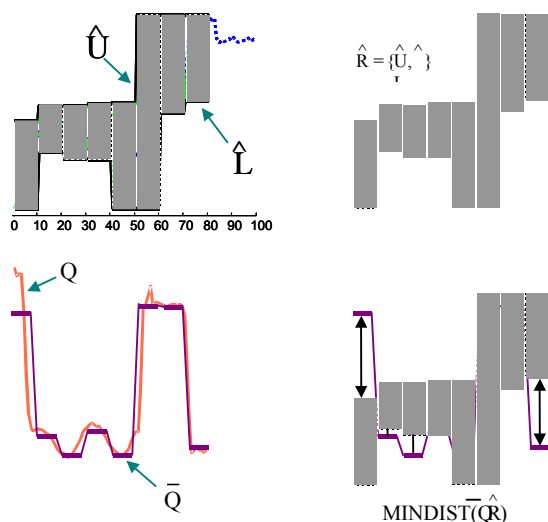


Figure 10. (Top Left) The time series C , and its bounding envelopes. (Top Right) The set of bounding envelopes $\hat{R} = \{\hat{U}, \hat{L}\}$. (Bottom Left) The query Q , and its approximation \bar{Q} . (Bottom Right) Illustration of the $MINDIST(\bar{Q}, \hat{R})$ function.

Similarly to $MINDIST(\bar{Q}, \hat{R})$, we can also define $MAXDIST(\bar{Q}, \hat{R})$ (illustrated in Figure 11), which serves as an upper bound for the distance between a query Q and \hat{R} .

$$MAXDIST(\bar{Q}, \hat{R}) = \sqrt{\sum_{i=1}^N \frac{n}{N} \begin{cases} (\bar{q}_i - \hat{U}_i)^2, & \text{if } (\bar{q}_i < \hat{L}_i) \text{ or } (\bar{q}_i - \hat{L}_i < \hat{U}_i - \bar{q}_i) \\ (\bar{q}_i - \hat{L}_i)^2, & \text{if } (\bar{q}_i > \hat{U}_i) \text{ or } (\hat{U}_i - \bar{q}_i < \bar{q}_i - \hat{L}_i) \end{cases}} \quad (13)$$

The use of this upper bound will become apparent in Section 6, when we discuss algorithms for fast similarity search under uniform scaling.

Note that there is no inherent restriction in our framework that would prevent us from incorporating into it DTW indexing, as well. DTW can be used in conjunction with uniform scaling to allow small local adjustments to the time axis after uniform scaling has found the best global scaling. In this case, we simply have to compute the upper and lower bounding envelopes for the DTW transform, and then, apply the techniques described in this study on these envelopes. Since

this is a straightforward extension, we do not pursue it any further in this work.

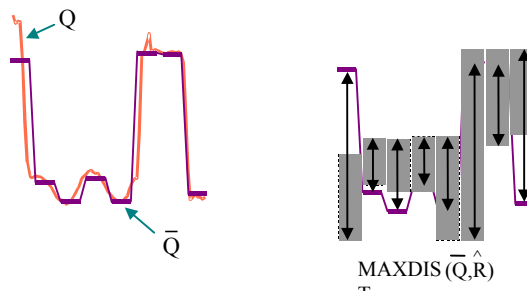


Figure 11. (Left) The query Q , and its approximation \bar{Q} . (Right) Illustration of the $MAXDIST(\bar{Q}, \hat{R})$ function.

6. ALGORITHMS FOR SECONDARY STORAGE

Based on the discussion of the previous section, we can now present algorithms that solve the time series similarity problem under uniform scaling, when the time series database does not fit in main memory. In the following discussion, as well as in the experiments, we assume that all the time series data and their bounding envelopes are disk resident. For ease of exposition, we also make the simplifying assumption that we are only interested in the single best match to our query. The extensions to more general cases are straightforward, and we omit them for brevity.

6.1 Linear Scan of the Time Series

Linear scan is a brute force approach, where we do not make use of our lower bounding technique. This algorithm sequentially reads each time series C from the disk, and computes its minimum distance to the query series by trying all possible rescalings. We store the minimum distance value in variable *OverallBestMatchVal*. The only optimization we can apply in this case, is to stop the computation of the Euclidean distance function as soon as it becomes larger than the currently minimum distance, that is, *OverallBestMatchVal*¹. We call the above algorithm *LinearScan*, and we present it only as a baseline against which we compare our proposed algorithms.

In order to improve the performance of *LinearScan*, we incorporate the use of the bounding envelopes as follows. Along with each time series C , we also store on disk its corresponding envelopes R . As before, we keep track of the minimum distance between the query and a candidate time series in variable *OverallBestMatchVal*. The algorithm starts by reading the envelopes \hat{R} of some time series C and computing the lower bound $MINDIST(\bar{Q}, \hat{R})$. If this lower bound is less than *OverallBestMatchVal*, then we have to call *TestAllScalings()* to compute the best match for all possible

¹ We apply the same optimization to all the algorithms we present in this paper. However, the performance benefits are in all cases minimal.

rescalings, and update *OverallBestMatchVal* accordingly. Otherwise, we know that C cannot have a distance to Q less than *OverallBestMatchVal*, and we can simply discard C . We refer to this algorithm as *LinearScanLB* (*Linear Scan Lower Bound*), and we expect it to run faster than *LinearScan*, since it avoids using *TestAllScalings()* for all the time series in the database. Table 3 (in the Appendix) depicts the pseudocode for the *LinearScanLB* algorithm.

The rest of the algorithms we describe make a more informed use of the bounding envelopes. The premise is that these algorithms will more effectively prune the search space, and result in superior performance.

6.2 Linear Scan of the Bounding Envelopes

The intuition behind our next algorithm, *FastScan*, is the following. Instead of retrieving a time series from disk every time that the lower bound distance is less than *OverallBestMatchVal*, start by retrieving from disk the time series in increasing order of their lower bound value. If the lower bounds we compute are tight enough, then best match to the query time series will be among the first time series to be retrieved.

The *FastScan* algorithm computes the solution in two phases. First, it performs a linear disk scan of the bounding envelopes of all the time series in the database \mathbf{D} . Note that this process is relatively fast, since the size of the bounding envelopes is much smaller than the size of the time series themselves. While scanning the envelopes, it calculates the lower bound on the distance between the corresponding candidate time series and the query, and inserts this value in a minimum priority queue.

During the second phase, the algorithm dequeues elements from the priority queue, reads from the disk the relevant candidate time series, and calls *TestAllScalings()* to determine the best match. The algorithm stops when the lower bound value for the dequeued element is larger than *OverallBestMatchVal*, or when the priority queue empties. It is easy to see that *FastScan* performs less work than *LinearScanLB*, and only in the worst case does it perform an equal amount of work. The pseudocode for the *FastScan* algorithm (which is similar in nature to the VA-file method [32]) is shown in Table 4 (in the Appendix).

The *FastScan* algorithm requires inserting in the priority queue as many values as there are time series in the database. However, this is not a concern, even though it is a main memory data structure. For example, a heap would require 20 bytes² per entry, a small amount even for databases with one million objects. The processing time required for the heap, $O(\log|\mathbf{D}|)$, is not of concern either, since it is insignificant compared to the time required for disk I/O and for computing the distances between time series. Note that we can incorporate in the algorithm the use of the upper bound (*MAXDIST*) as well, which can help maintain the size of the priority queue

small; usually less than 15% of the original size. However, this comes at a cost in time. In the version of *FastScan* we used in the experiments, we did not take into account this optimization.

6.3 Algorithms based on R-trees

Having the bounding envelopes \hat{R} in a sufficiently low dimensionality allows us to use an R-tree for indexing them. The goal is to use the R-tree index, and avoid reading all the bounding envelopes, which is what *FastScan* does.

The use of the R-tree is straightforward. We associate each bounding envelope $\hat{R} = \{\hat{U}, \hat{L}\}$ to an MBR(\mathbf{l}, \mathbf{h}) as follows: $\mathbf{l} = (\hat{L}_1, \hat{L}_2, \dots, \hat{L}_N)$, and $\mathbf{h} = (\hat{U}_1, \hat{U}_2, \dots, \hat{U}_N)$. This allows to compute the lower bound distance $MINDIST(\bar{Q}, \hat{R})$ as usual. Along with each MBR that represents a set of bounding envelopes \hat{R} , we also store a pointer to the corresponding time series C in the file system. Then, we build the R-tree using one of the traditional construction algorithms [5][14].

The R-tree search procedure needs only minor modifications. The algorithm starts by reading an MBR r , and computing the lower bound $MINDIST(\bar{Q}, r)$. If this lower bound is less than *OverallBestMatchVal*, and r is an inner node, then we process r recursively. If r is a leaf node, that is, it refers to bounding envelopes \hat{R} of candidate time series C , then we have to read C from disk, call *TestAllScalings()* to compute the best match for all possible rescalings, and update *OverallBestMatchVal*. If the lower bound is larger than *OverallBestMatchVal*, then we simply disregard r . We refer to this algorithm as *RtreeBF* (*R-tree Brute Force*).

An optimization that we can apply to the above algorithm is to simply use the R-tree index to get a small number of candidate time series with the lowest lower bound values, and then test each of these candidates to determine the true best match. This is the same idea that *FastScan* uses.

Now, the R-tree search algorithm needs to maintain a minimum priority queue on the lower bound values. However, unlike *FastScan*, we can also make use of the upper bound distance between the query and an MBR, $MAXDIST(\bar{Q}, r)$, in order to further prune the search space (this optimization is not applicable to *RtreeBF* that only needs to compare the lower bound to *OverallBestMatchVal*).

For each MBR r , we compute the lower and upper bounds, and maintain the lowest value for the upper bounds in variable *LowestUpper*. We recursively search the R-tree, following only the MBR nodes whose lower bound is less than *LowestUpper*. If this condition is true for a leaf MBR, then we insert an entry for the corresponding candidate time series in the priority queue. When we have finished processing the R-tree, we start calling *TestAllScalings()* to determine the best match among all the candidates stored in the priority queue. This processing follows the order specified by the queue. We call this algorithm *RtreeProbe*, and Table 5 (in the Appendix) shows its outline.

² The elements we need to store are the lower bound value (8 bytes), a pointer to the time series in the file system (4 bytes), and two pointers to the children nodes (8 bytes).

7. EXPERIMENTAL RESULTS

In this section we will empirically evaluate our approach. Although we are particularly interested in motion capture data, as we noted above, our algorithm may have utility in domains as diverse as music retrieval [7] and space telemetry [8]. We will therefore perform all experiments on the following two datasets.

- **Motion Capture:** This dataset was distilled from several hours of recording with Vicon (an optical motion capture system), using 124 sensors. The data for our experiments are drawn uniformly at random from a pool of 250,000 subsequences.
- **Mixed Bag:** This dataset was created by concatenating 10 diverse datasets from the UCR time series archive. The 10 datasets are foetal ecg, steam generator, space shuttle, Photon Burst, Standard and Poor 500, ocean, power demand, leleccum, Koski ECG, and infrasound_beamd. The subsequences we use for our experiments are drawn at random from this pool, making sure that all 10 seed time series contribute equally.

In both cases, the queries are random subsequences not present in the database of candidate time series.

7.1 Main Memory Experiments

In the first set of experiments, we evaluate the effectiveness of our lower bounding technique when the datasets fit in main memory. We use the main memory version of algorithm *LinearScanLB*, which we call *LB_US*. We compare only to the brute force search algorithm defined in Table 2, and to the recently introduced *CD-criterion* technique [2], because there are no other techniques in existence that support uniform scaling queries. To eliminate the possibility of implementation bias [18], and because *CD-criterion* does not support indexing, we consider the speedup obtained in main memory. To compare the three competing techniques we report the *Pruning Power*, i.e., the fraction of times that each approach must call the squared Euclidean distance function.

$$\text{Pruning Power} = \frac{\text{Number of calls to distance function by proposed approach}}{\text{Number of calls to distance function by brute force search}} \quad (13)$$

This measure depends only on the tightness of the lower bounds, and is independent of language, platform, caching or any other implementation details. As an additional sanity check we also measured the CPU time. However, since it is almost perfectly correlated with the *Pruning Power*, we omit it for brevity. Note that by definition, the pruning power of brute force is always 1.

As noted earlier, the *CD-criterion* algorithm can only test whether the distance of two sequences is within a user-supplied epsilon. In order to allow direct comparison with the two other approaches, we supply to the algorithm the exact epsilon that will return the single nearest neighbor.

Since the speed-up obtained for our approach clearly depends on the range of scaling factors and the length of the time series, we test our approach for the cross product of maximum scaling factors $sf_{max} = \{1.05, 1.10, 1.20\}$ and candidate time series lengths of $\{64, 128, 256\}$. These

represent realistic parameters in our motion capture domain, and seem representative of other domains as well [8][13][33].

We conduct our experiments as follows. We randomly remove a subsequence of the appropriate length from the data to use as a query, and randomly choose 10,000 other subsequences to act as the database. We then search for the best scaled match. We repeated this 500 times for every combination of scaling factors and candidate lengths. Figure 12 shows the results.

The results are quite impressive for the proposed approach, which essentially needs to perform an order of magnitude less work than *CD-criterion*, over all parameter settings on both datasets. In addition, our approach is two to three orders of magnitude more efficient than the brute force algorithm. The above experiments demonstrate that our technique can effectively reduce the amount of required effort by avoiding a considerable number of unnecessary computations.

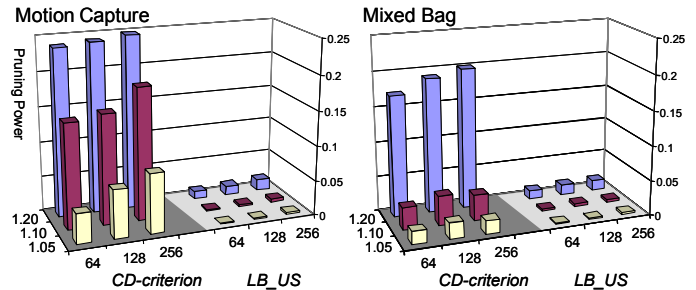


Figure 12. The pruning power of *CD-criterion* and *LB_US* algorithm on two datasets, over a range of scaling factors and candidate lengths.

7.2 Secondary Storage Experiments

In this section, we evaluate the performance of our techniques when the time series database \mathbf{D} does not fit in main memory. The experiments were performed on an Athlon 1.6GHz Linux machine, with 1GB of main memory. All the data were stored on its local disk. In the interest of space, we only report the results for the motion capture dataset; the results for the mixed bag dataset exhibit similar trends.

In all the following experiments, we measure the time (in seconds) it takes each algorithm to process a single query. We repeated each experiment 500 times, and report the average time. Note that we only need to compare the algorithms in terms of their running-time performance. Since all algorithms are exact, they return in every case the same (correct, based on the distance definition) answer.

7.2.1 Exploring the Properties of the Algorithm

In the first set of experiments, we compare the performance of the algorithms *LinearScan*, *LinearScanLB*, *FastScan*, *RtreeBF*, and *RtreeProbe*, for the cross product of maximum scaling factors $sf_{max} = \{1.05, 1.10, 1.20\}$ and candidate time series lengths of $\{64, 128, 256\}$ (Figure 13). The dimensionality of the approximated bounding envelopes is fixed to 16, and the number of candidate time series to 10,000.

The results clearly show that *LinearScan* cannot compete with the other alternatives, which can actually offer interactive response times, a crucial factor for the real-life applications we have in mind. Note that all the other algorithms outperform *LinearScan* by up to two orders of magnitude, exactly because they make use of the techniques we introduce in this paper. For the rest of the discussion, we will disregard *LinearScan*, and focus on the comparison among the other algorithms.

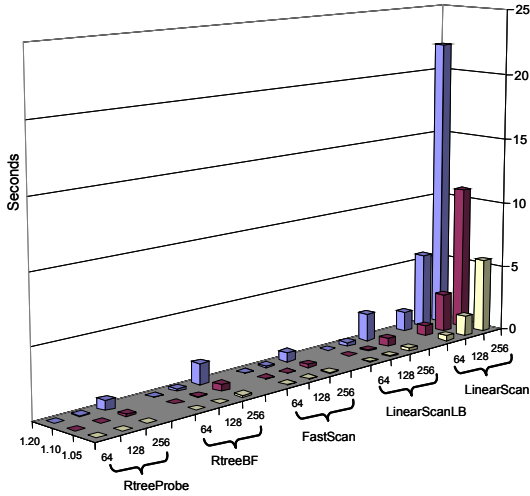


Figure 13. Average time to answer a query for algorithms *LinearScan*, *LinearScanLB*, *FastScan*, *RtreeBF*, and *RtreeProbe*, over a range of scaling factors and candidate lengths.

In Figure 14, we zoom in on the results of the four algorithms that use our proposed technique. As expected, the trends are that the query response time increases with both the length of the candidate time series and the scaling factor. Out of the two, the latter seems to be the dominant factor. All four algorithms exhibit similar trends.

In terms of absolute numbers, *FastScan* and *RtreeProbe* perform better than the other two algorithms, executing up to 3.5 times faster. Note that these two are the algorithms that first determine the most promising order in which to access the candidate time series, and only then start accessing them.

7.2.2 Scalability Experiments

We now turn our attention to the scalability issues, and examine the behavior of the algorithms when we vary the size of the time series database. The default values we use are, unless otherwise noted, 10,000 for the number of candidate time series, 128 for their length, and 16 for the dimensionality of the approximated representations of the bounding envelopes.

In the first set of experiments we vary the number of candidate time series from 5,000 to 80,000. The results are shown in Figure 15. We observe that the query response time for *FastScan* is better than *RtreeBF* for the small database sizes, but then deteriorates faster as the database size increases. This is due to the fact that it still needs to read from disk the bounding envelopes for all the series in the database. *RtreeBF* can effectively prune the search space. Yet, for the smaller database sizes, it pays the price of not determining in advance the most promising order in which to test the candidate time

series. The performance of *RtreeProbe* is consistently the best among the proposed approaches (2 to 9 times faster), combining the advantages of both *FastScan* and *RtreeBF*.

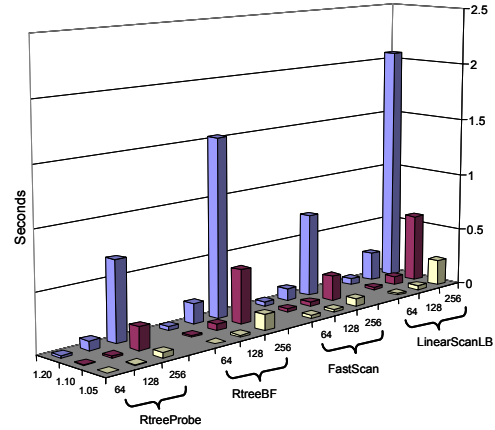


Figure 14. Average time to answer a query for algorithms *LinearScanLB*, *FastScan*, *RtreeBF*, and *RtreeProbe*, over a range of scaling factors and candidate lengths.

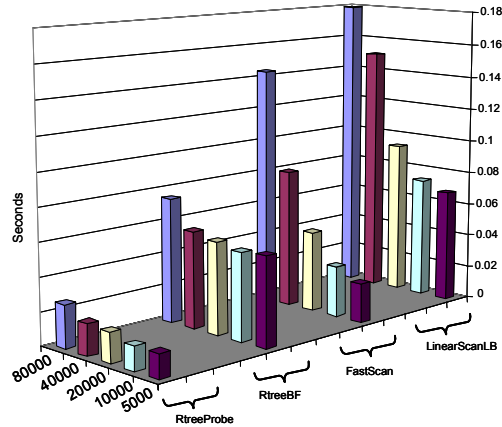


Figure 15. Average time to answer a query for algorithms *LinearScanLB*, *FastScan*, *RtreeBF*, and *RtreeProbe*, when varying the number of candidate time series.

Figure 16 depicts the results of the experiments where we varied the length of the candidate time series (from 64 to 1024). At the high end of the length spectrum, with the exception of *RtreeProbe*, all the algorithms tend to behave the same. The experimental results indicate that the algorithms fail to prune a significant amount of the candidate time series. On the other hand, *RtreeProbe* manages to do a slightly better job in this respect, which results in significant savings in terms of time. Its performance is 2.5 to 4 times better than the best alternative.

In the last set of experiments (Figure 17), we evaluate the effect of the dimensionality of the approximated bounding envelopes on the performance of the algorithms. An increase in the dimensionality results in more accurate representations, and consequently, in more precise lower and upper bounds. The experiments show that this is, in general, beneficial for all the algorithms. However, the benefits are diminishing as the dimensionality increases. In fact, we may also get the reverse effect, as is the case with *RtreeBF* for dimensionality 16. In this case, the bounding envelopes of the candidate time series do

not form tight clusters in the high-dimensional R-tree, and consequently, lead to poor performance. *RtreeProbe* avoids the cost of this problem, because it postpones the expensive *TestAllScalings()* operation till the end.

The experimental evaluation indicates that it is beneficial to first identify the most promising candidates entirely in the reduced dimensionality space (i.e., using only \hat{R}), and then test the query against the candidate time series, despite the fact that these calculations are not as precise. Though, the above may not be true for large databases (see Figure 15). It is also interesting to note that the behavior of the R-tree algorithms is not always predictable. Consider, for example, the non-intuitive *RtreeBF* results for database sizes 5,000 and 10,000 (Figure 15), and dimensionalities of 12 and 16 (Figure 17). In both cases, the results worsen when the parameters become seemingly more favorable. Nevertheless, one of the R-tree-based algorithms, *RtreeProbe*, exhibits a consistently superior performance across all experiments, outperforming the best alternative by up to 4 times.

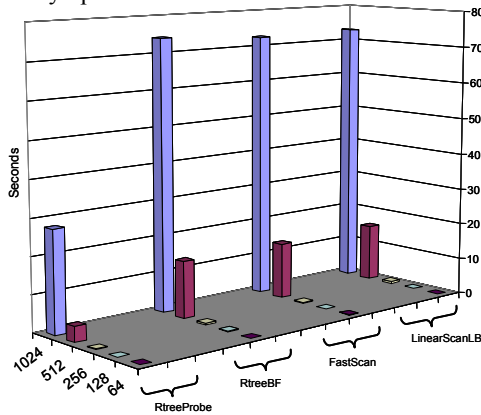


Figure 16. Average time to answer a query for algorithms *LinearScanLB*, *FastScan*, *RtreeBF*, and *RtreeProbe*, when varying the length of the candidate time series.

7.3 Case Study

In addition to the comprehensive experiments on the efficiency of indexing discussed above, we also used a motion capture processing system to evaluate our techniques. We conducted numerous experiments to test the quality of matches obtained by uniform scaling, allowing both stretching and shrinking.

The experimental setup was as follows. We randomly selected a query motion from our motion capture database (sitting up, stretching arms, etc.), and searched the database for the closest (non-self) match. We evaluated the returned sequences in two ways. Objectively, by measuring the Euclidean distance of the sequences (which express the orientations of the joints in the body), and subjectively, by creating and reviewing animations based on the matched sequences. These animations show the query sequence and its best matches superimposed on each other.

The results of the experiments validate the utility and effectiveness of our approach, and also demonstrate that uniform scaling produces better results than DTW. Since such experiments do not lend themselves to a text and graphic

exposition, we have created a website with full video examples [37], and only present here a single figure to give the flavor of these experiments.

In Figure 18, we show the best match under uniform scaling (top) and the best match under DTW (bottom) for a query involving a motion of the arms. In the figures and graphs, we have superimposed the query with each of the best match sequences. It is obvious that when we allow uniform scaling we are able to find a match that is much closer to the query sequence (Euclidean distance 51.83) than when we use DTW (Euclidean distance 154.16).

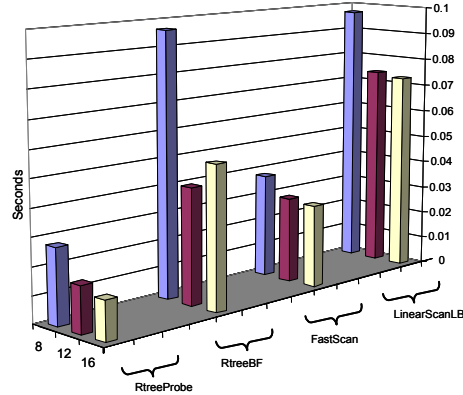


Figure 17. Average time to answer a query for algorithms *LinearScanLB*, *FastScan*, *RtreeBF*, and *RtreeProbe*, when varying the dimensionality of the approximated representations.

8. CONCLUSIONS

In this work, we motivated the need for uniform scaling similarity matching, which has applications in several domains where human variability necessitates this type of matching flexibility (e.g., motion libraries, music retrieval, and historical handwritten archives). We introduced the first technique for indexing time series with invariance to uniform scaling, based on bounding envelopes. This technique enables fast similarity searching in large time series databases. We presented several algorithms that make use of the above technique, and evaluated our proposed approaches with a comprehensive set of experiments on real data, over realistic parameter choices suggested by domain experts. The experimental results demonstrate the significant advantages of our indexing technique, and evaluate the relative benefits of the different alternatives. Finally, we describe the application of our method in a motion capture processing system, and illustrate its usefulness and the superiority of the results it returns when compared to other alternatives.

Reproducible Research Statement: All datasets, animations and code used in this work are freely available by emailing the authors or at [37].

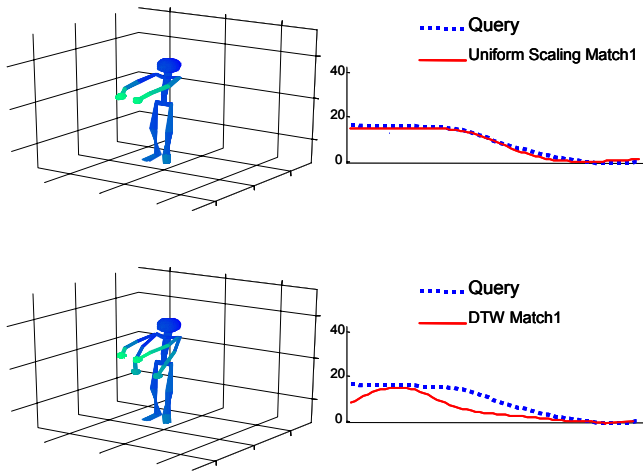


Figure 18: Sample results from a motion-capture similarity search experiment. A query motion (upper body only) was submitted to a motion capture database. (Top Left) The stick figure is a superposition of both the query and the best matching sequence under uniform scaling. Since both figures are so similar (after scaling the match back to the query length) we can only see a single figure. (Bottom Left) The superposition of the query and the best DTW match show considerable mismatch. (Right) The corresponding time series (only the Z-axis is shown) illustrates the source of the difference.

9. REFERENCES

- [1] Aach, J. and Church, G. (2001). Aligning gene expression time series with time warping algorithms. *Bioinformatics*. Volume 17, pp 495-508.
- [2] Argyros, T & Ermopoulos, C. (2003). Efficient subsequence matching in time series databases under time and amplitude transformations. *IEEE ICDM* pp 481-484.
- [3] Arikan, O., Forsyth, D. & O'Brien, J. (2003). Motion synthesis from annotations. *ACM Transactions on Graphics*. V22(3). pp. 402-408.
- [4] Arikan, O. & Forsyth, D. (2002). Synthesizing constrained motions from examples. *ACM Transactions on Graphics*, 21(3):483-490. ISSN 0730-0301.
- [5] Beckmann, N., Kriegel, H.-P., Schneider, R., & Seeger, B. (1990). The R*-tree: an efficient and robust access method for points and rectangles. In *Proc of ACM SIGMOD*, pp. 220-231.
- [6] Chan, K. & Fu, A. W. (1999). Efficient time series matching by wavelets. In *proceedings of the 15th IEEE Int'l Conference on Data Engineering*. Sydney, Australia. pp 126-133.
- [7] Chu, K., Lam, S. & Wong, M. (1998). An efficient hash-based algorithm for sequence data searching. *The Computer Journal* 41 (6): 402-415.
- [8] Dannenberg, R., Birmingham, W., Tzanetakis, G., Meek, C., Hu, N., and Pardo, B. (2003). The MUSART testbed for query-by-humming evaluation. *ISMIR 2003*, 4th International Conference on Music Information Retrieval Baltimore, Maryland.
- [9] Das, G., Gunopulos, D., Mannila, H. (1997). Finding similar time series. In *Proc. of the First PKDD Symp*, pp. 88-100.
- [10] DeCoste, D. and Levine, M (2000). Automated event detection in space instruments: a case study using IPEX-2 data and support vector machines. *SPIE Conference Astronomical Telescopes and Instrumentation*.
- [11] Faloutsos, C., Ranganathan, M., & Manolopoulos, Y. (1994). Fast subsequence matching in time-series databases. In *Proc. ACM SIGMOD Conf.*, Minneapolis. pp. 419-429.
- [12] Fung, W & Wong, M. (2003). Efficient subsequence matching for sequences databases under time warping. The 7th International Database Engineering and Application Symposium.
- [13] Gavrilu, D. M. & Davis, L. S. (1995). Towards 3-d model-based tracking and recognition of human movement: a multi-view approach. In *International Workshop on Automatic Face- and Gesture-Recognition*.
- [14] Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proceedings ACM SIGMOD Conference*. pp 47-57.
- [15] Hetland, M. (2003). A survey of recent methods for efficient retrieval of similar time sequences. To appear in an Edited Volume, *Data Mining in Time Series Databases*. Published by the World Scientific Publishing Company.
- [16] Kahveci, T. & Singh, A. (2001). Variable length queries for time series data. In *proceedings of the 17th Int'l Conference on Data Engineering*. Heidelberg, Germany, pp 273-282.
- [17] Keogh, E. (2002). Exact indexing of dynamic time warping. In *28th International Conference on Very Large Data Bases*. Hong Kong. pp 406-417.
- [18] Keogh, E., Chakrabarti, K., Pazzani, M. & Mehrotra (2000). Dimensionality reduction for fast similarity search in large time series databases. *Journal of Knowledge and Information Systems*. pp 263-286.
- [19] Keogh, E., Chakrabarti, K., Pazzani, M. & Mehrotra (2001). Locally adaptive dimensionality reduction for indexing large time series databases. In *Proc of ACM SIGMOD Conference on Management of Data*. pp 151-162.
- [20] Keogh, E. and Kasetty, S. (2002). On the need for time series data mining benchmarks: a survey and empirical demonstration. In *the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Edmonton, Canada. pp 102-111.
- [21] Kovar, L., Gleicher, M., & Pighin, F (2002). Motion graphs. *Proceedings of ACM SIGGRAPH*.
- [22] Lee, J., J. Chai, P.S.A. Reitsma, J. K. Hodgins, & N. S. Pollard. (2002). Interactive control of avatars animated with human motion data. *ACM Transactions on Graphics*. V21(3) pp. 491-500.
- [23] Li, Q., Lopez, I, & Moon, B. (2004). Skyline index for time series data. To appear in *TKDE*.
- [24] Li, Y., Wang, T. & Shum, H.-Y. (2002). Motion texture: a two level statistical model for character motion synthesis. *ACM Transactions on Graphics*, 21(3):465-472.
- [25] Park, S., Chu, W. W., Yoon, J. & Hsu, C. (2000). Efficient searches for similar subsequences of different lengths in sequence databases. In *proceedings of the 16th Int'l Conference on Data Engineering*. San Diego, CA, pp 23-32.
- [26] Perng, C., Wang, H., Zhang, S., & Parker, S. (2000). Landmarks: a new model for similarity-based pattern querying in time series databases. In *proceedings of 16th International Conference on Data Engineering*. pp 33-42.
- [27] Rath, T. & Manmatha, R. (2002). Lower-bounding of dynamic time warping distances for multivariate time Series. Tech Report MM-40, University of Massachusetts Amherst.
- [28] Roddick, J. F. and Spiliopoulou, M. (2001). A survey of temporal knowledge discovery paradigms and methods. *IEEE Tran's on Knowledge and Data Engineering*. pp. 750-767.
- [29] Rose, C., Guenter, B., Bodenheimer, B., & Cohen, M.F. (1996). Efficient generation of motion transitions using spacetime constraints. In *Proc of ACM SIGGRAPH*, New Orleans, USA, pp. 147-154.
- [30] Vlachos, M., Kollios, G., & Gunopulos, G. (2002). Discovering similar multidimensional trajectories. In *Proc 18th International Conference on Data Engineering*.
- [31] Vlachos, M., Hadjieleftheriou, M., Gunopulos, D., & Keogh, E. (2003). Indexing multi-dimensional time-series with support for multiple distance measures. In *Proc ACM SIGKDD*, Washington DC, USA.
- [32] Weber, R., Schek, H & Blott S. (1998) A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. *VLDB* pp. 194-205.
- [33] Zhu, Y. & Shasha, D. (2003). Query by humming: a time series database approach. *SIGMOD 2003*.
- [34] Zordan, V. B., Hodgins, J. K., (2002). Motion capture-driven simulations that hit and react, *ACM SIGGRAPH Symposium on Computer Animation*.
- [35] Vicon, <http://www.vicon.com/>
- [36] Motion Analysis, <http://www.motionanalysis.com/>
- [37] Videos from Case Study, <http://www.cs.ucr.edu/~eamonn/VLDB2004/VLDB.htm>

APPENDIX

Table 3. The *LinearScanLB* algorithm.

```

procedure LinearScanLB( $\bar{Q}$ )
  OverallBestTimeSeries = null;
  OverallBestMatchVal = inf;
  OverallBestScaling = null;
  for ( $i = 1$  to number of time series in  $D$ )
    read from disk  $\hat{R}_i$ ; /* needed for lower bound distance */
    if ( $MINDIST(\bar{Q}, \hat{R}_i) < OverallBestMatchVal$ )
      read from disk  $C_i$ ; /* needed for TestAllScalings() */
      [ $dist, scale$ ] = TestAllScalings( $Q, C_i$ );
      if ( $dist < OverallBestMatchVal$ )
        OverallBestTimeSeries =  $i$ ;
        OverallBestMatchVal =  $dist$ ;
        OverallBestScaling =  $scale$ ;

  return ( $OverallBestTimeSeries, OverallBestMatchVal,$ 
     $OverallBestScaling$ );

```

Table 4. The *FastScan* algorithm.

```

procedure FastScan( $Q$ )
  let  $H$  be a minimum priority queue;
  OverallBestTimeSeries = null;
  OverallBestMatchVal = inf;
  OverallBestScaling = null;
  for ( $i = 1$  to number of time series in  $D$ )
    read from disk  $\hat{R}_i$ ; /* needed for lower bound distance */
    insert in  $H$   $MINDIST(\bar{Q}, \hat{R}_i)$ , and pointer to time series  $i$ ;

  while ( $H$  not empty)
    dequeue minimum element from  $H$ ;
    let this element correspond to time series  $j$ ;
    if ( $MINDIST(\bar{Q}, \hat{R}_j) > OverallBestMatchVal$ )
      return ( $OverallBestTimeSeries, OverallBestMatchVal,$ 
         $OverallBestScaling$ );
    read from disk  $C_j$ ; /* needed for TestAllScalings() */
    [ $dist, scale$ ] = TestAllScalings( $Q, C_j$ );
    if ( $dist < OverallBestMatchVal$ )
      OverallBestTimeSeries =  $j$ ;
      OverallBestMatchVal =  $dist$ ;
      OverallBestScaling =  $scale$ ;

  return ( $OverallBestTimeSeries, OverallBestMatchVal,$ 
     $OverallBestScaling$ );

```

Table 5. The *RtreeProbe* algorithm.

```

procedure RtreeProbe( $Q$ )
  let  $H$  be a minimum priority queue;
  LowestUpper = inf;
  OverallBestTimeSeries = null;
  OverallBestMatchVal = inf;
  OverallBestScaling = null;
  [ $H$ ] = ProcessRtreeNode(root of Rtree,  $Q, LowestUpper$ );
  while ( $H$  not empty)
    dequeue minimum element from  $H$ ;
    let this element correspond to time series  $j$ ;
    if ( $MINDIST(\bar{Q}, \hat{R}_j) > OverallBestMatchVal$ )
      return ( $OverallBestTimeSeries, OverallBestMatchVal, OverallBestScaling$ );
    read from disk  $C_j$ ; /* needed for TestAllScalings() */
    [ $dist, scale$ ] = TestAllScalings( $Q, C_j$ );
    if ( $dist < OverallBestMatchVal$ )
      OverallBestTimeSeries =  $j$ ;
      OverallBestMatchVal =  $dist$ ;
      OverallBestScaling =  $scale$ ;
  return ( $OverallBestTimeSeries, OverallBestMatchVal, OverallBestScaling$ );

procedure ProcessRtreeNode(Rtree node  $M, Q, LowestUpper$ )
  for (all nodes  $r$ , children of  $M$ )
    if ( $MAXDIST(\bar{Q}, r) < LowestUpper$ )
      LowestUpper =  $MAXDIST(\bar{Q}, r)$ ;
    if ( $MINDIST(\bar{Q}, r) < LowestUpper$ )
      if ( $r$  is a leaf node)
        insert  $\hat{R}_i$  in  $H$ , along with pointer to corresponding time series;
        /*  $\hat{R}_i$  is the same as the MBR values */
      else [ $H$ ] = ProcessRtreeNode(node  $r, Q, LowestUpper$ )
  return  $H$ ;

```