An Efficient and Accurate Method for Evaluating Time Series Similarity

Michael Morse Jignesh M. Patel
University of Michigan
Ann Arbor, MI
{mmorse, jignesh}@eecs.umich.edu

ABSTRACT

A variety of techniques currently exist for measuring the similarity between time series datasets. Of these techniques, the methods whose matching criteria is bounded by a specified ϵ threshold value, such as the LCSS and the EDR techniques, have been shown to be robust in the presence of noise, time shifts, and data scaling. Our work proposes a new algorithm, called the Fast Time Series Evaluation (FTSE) method, which can be used to evaluate such threshold value techniques, including LCSS and EDR. Using FTSE, we show that these techniques can be evaluated faster than using either traditional dynamic programming or even warp-restricting methods such as the Sakoe-Chiba band and the Itakura Parallelogram.

We also show that FTSE can be used in a framework that can evaluate a richer range of ϵ threshold-based scoring techniques, of which EDR and LCSS are just two examples. This framework, called Swale, extends the ϵ threshold-based scoring techniques to include arbitrary match rewards and gap penalties. Through extensive empirical evaluation, we show that Swale can obtain greater accuracy than existing methods.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data Mining, Spatial Databases and GIS

General Terms

Algorithms, Measurement, Performance, Experimentation

Keywords

Time Series, Trajectory Similarity, Clustering

1. INTRODUCTION

Techniques for evaluating the similarity between time series datasets have long been of interest to the database community. New location-based applications that generate time series location trails (called trajectories) have also fueled interest in this topic since time series simularity methods can be used for computing trajectory similarity. One of the critical research issues with time series analysis is the choice of distance function to capture the notion of similarity between two sequences. Past research in this area has produced a number of distance measures, which can be divided into two classes. The first class includes functions based on the L1 and L2 norms. Examples of functions in this class are Dynamic Time Warping (DTW) [2] and Edit Distance with Real Penalty (ERP) [4]. The second class of distance functions includes methods that compute a similarity score based on a matching threshold ϵ. Examples of this class of functions are the Longest Common Subsequence (LCSS) [30], and the Edit Distance on Real Sequence (EDR) [5]. Previous research [5, 30] has demonstrated that this second class of methods is robust in the presence of noise and time shifting.

All of the advanced similarity techniques mentioned above rely on dynamic programming for their evaluation. Dynamic programming requires that each element of one time series be compared with each element of the other; this evaluation is slow. The research community has thus developed indexing techniques such as [5, 7, 14, 18, 32] that use an index to quickly produce a superset of the desired results. However, these indexing techniques still require a refinement step that must perform the dynamic programming evaluation on elements of the superset. Furthermore, time series clustering has also been studied [5, 15, 30], and these clustering techniques require pairwise comparison of *all* time series in the dataset, which means that indexing methods cannot be used to speed up clustering applications.

To address this problem, a number of techniques have been developed that impose restrictions on the warping length of the dynamic programming evaluation. The Sakoe-Chiba band, studied in [26], uses a sliding window of fixed length to narrow the number of elements that are compared between two time series. The Itakura Parallelogram, studied in [11], also limits the number of comparisons to accomplish a similar effect as the Sakoe-Chiba band. These techniques that constrain the warping factor are faster, but at the expense of ignoring sequence matches that fall outside of the sliding window. If the best sequence match between two time series falls outside of the restricted search area, then these techniques will not find it.

In this paper, we propose a novel technique to evaluate the second class of time series comparison functions that compute a similarity score based on an ϵ matching threshold. The popular LCSS and EDR comparison functions belong to this class and can directly benefit from our new evaluation technique. This technique, called the **F**ast **T**ime **S**eries **E**valuation (**FTSE**), is not based around the dynamic programming paradigm nor is it an approximation (i.e. it computes the actual exact similarity measure). Using a number of experiments on real datasets, we show that FTSE is nearly an order of magnitude faster than the traditional dynamic programming-style of similarity computation. In addition, we show that *FTSE is*

SIGMOD'07, June 12–14, 2007, Beijing, China.

Symbol	Definition
$R,\ S$	Time series $(r_1, ..., r_m)$ and $(s_1, ..., s_n)$.
r_i	The i^{th} element of R.
$Rest(R)$	R with the first element removed.
M^d	d dimensional MBR.
$M^{d1},\ M^{d2}$	Lower and upper bounds of M

Table 1: Symbols and definitions.

also faster than popular warp-restricting techniques by a factor of 2-3, while providing an exact answer.

We show that FTSE can evaluate a broader range of ϵ threshold-based scoring techniques and not just LCSS and EDR. Motivated by FTSE's broader ability, we propose the **S**equence **W**eighted **AL**ignm**E**nt (**Swale**) scoring model that extends ϵ threshold based scoring techniques to include arbitrary match rewards and gap penalties. We also conduct an extensive evaluation comparing Swale with popular existing methods, including DTW, ERP, LCSS, and EDR and show that *Swale is generally more accurate than these existing methods*.

The remainder of this paper is organized as follows: Section 2 discusses the terminology that is used in the rest of the paper. Section 3 discusses related work and Section 4 describes the FTSE algorithm. Section 5 introduces the Swale similarity scoring method and Section 6 presents experimental results. Finally, Section 7 presents our conclusions.

2. TERMINOLOGY

Existing similarity measures such as LCSS, DTW, and EDR assume that time is discrete. For simplicity and without loss of generality, we make these same assumptions here. Formally, the time series data type T is defined as a sequence of pairs $T = (p_1, t_1), (p_2, t_2), ... , (p_n, t_n)$, where each p_i is a data point in a d-dimensional data space, and each t_i is the time at which p_i occurs. Each t_i is strictly greater than each t_{i-1}, and the sampling rate of any two time series is equivalent. Other symbols and definitions used in this paper are shown in Table 1.

Time series datasets are usually normalized before being compared. We follow the normalization scheme for time series data described in [9]. Specifically, for S of length n, let the mean of the data in dimension d be μ_d and let the standard deviation be σ_d. Then, to obtain the normalized data $N(S)$, we can evaluate $\forall\ i\ \in\ n:\ s_{i,d} = (s_{i,d} - \mu_d)/\sigma_d$ on all elements of S. This process is repeated for all dimensions. In this paper, all data is normalized, and we use S to stand for $N(S)$, unless stated otherwise.

3. RELATED WORK

There are several existing techniques for measuring the similarity between different time series. The Euclidean measure sums the Euclidean distance between points in each time series. For example, in two dimensions the Euclidean distance is computed as: $\sqrt{\sum_{i=1}^{n}((r_{i,x} - s_{i,x})^2 + (r_{i,y} - s_{i,y})^2)}$. This measure can be used only if the two time series are of equal length, or if some length normalization technique is applied. More sophisticated similarity measures include Dynamic Time Warping (DTW) [2], Edit distance with Real Penalty (ERP) [4], the Longest Common Subsequence (LCSS) [30], and Edit Distance on Real sequences (EDR) [5]. These measures are summarized in Table 2.

DTW was first introduced to the database community in [2]. DTW between two time series does not require the two series to be of the same length, and it allows for time shifting between the two time series by repeating elements. ERP [4] creates g, a constant value for the cost of a gap in the time series, and uses the L1 distance norm as the cost between elements. The LCSS technique introduces a threshold value, ϵ, that allows the scoring technique to handle noise. If two data elements are within a distance of ϵ in each dimension, then the two elements are considered to match, and are given a match reward of 1. If they exceed the ϵ threshold in some dimension, then they fail to match, and no reward is issued. The EDR [5] technique uses gap and mismatch penalties. It also seeks to minimize the score (so that a score closer to 0 represents a better match).

In [1], the authors use the Euclidean distance to measure similarity in time series datasets. The Discrete Fourier Transform is used to produce features that are then indexed in an R-tree. Dimensionality reduction is also studied in [3, 14, 18, 20, 23, 32]. Indexing is also studied in [7], which proposes a generic method built around lower bounding to guarantee no false dismissals. Indexing methods for DTW have been the focus of several papers including [13, 19, 27, 33, 34]. Indexing for LCSS [29] and EDR [5] has also been studied. In this paper, our focus is not on specific indexing methods, but on the design of robust similarity measures, and efficient evaluation of the similarity function. We note that our work is complementary to these indexing methods, since the indexing methods still need to perform a refinement step that must evaluate the similarity function. Traditionally, previous work has not focused on this refinement cost, which can be substantial. Previous works employ a dynamic programming (DP) method for evaluating the similarity function, which is expensive, especially for long sequences. In other words, FTSE can be used to boost the performance of existing LCSS or EDR-based indexing methods since it is faster than traditional DP methods for the refinement step.

The Sakoe-Chiba Band [26] and Itakura Parallelogram [11] are both estimation techniques for restricting the amount of time warping to estimate the DTW score between two sequences. A restriction technique similar to the Sakoe-Chiba Band is described for LCSS in [30] and the R-K Band estimate is described in [24].

Time series may be clustered using compression techniques [6, 16]. We do not compare our algorithms with these techniques because of their inapplicability for clustering short time series.

The FTSE algorithm that we propose bears similarity to the Hunt-Szymanski algorithm [10, 21] for finding the longest common subsequence between two sequences. However, Hunt-Syzmanski is only concerned with string sequences (and not time series), supports only a limited string edit-distance model, and does none of the grid matching that FTSE does to identify matching elements between time series (see Section 4).

A more closely set of related work is concerned with clustering of trajectory datasets (such as [5, 15, 30, 31]). In fact, a commonly established way of evaluating the effectiveness of trajectory similarity measures is to use it for clustering, and then evaluate the quality of the clusters that are generated [5, 15, 30]. Common clustering methods such as complete linkage are often used for trajectory data analysis [5, 15, 30], and these methods require that each trajectory in the dataset be compared to every other trajectory. Essentially, for a dataset of size s, this requires approximately s^2 comparisons. As we show in this paper for such problems, not only is the Swale scoring method more effective, but the FTSE technique is also faster than the existing methods.

4. FAST TIME SERIES EVALUATION

In this section, we introduce the FTSE algorithm. In order to better understand why FTSE is faster than dynamic programming, we first discuss dynamic programming and its shortcomings for evaluating ϵ based comparison functions. We then provide an overview

Definition		
DTW(R,S)	=	$\begin{cases} 0 & \text{if } m = n = 0 \\ \infty & \text{if } m = 0 \text{ or } n = 0 \\ dist(r_1, s_1) + min\{DTW(Rest(R), Rest(S)), & \text{otherwise} \\ DTW(Rest(R), S), DTW(R, Rest(S))\} & \end{cases}$
ERP(R,S)	=	$\begin{cases} \sum_1^n dist(s_i, g),\ \sum_1^m dist(r_i, g) & \text{if } m = 0, \text{ if } n = 0 \\ min\{ERP(Rest(R), Rest(S)) + dist(r_1, s_1) & \\ ERP(Rest(R), S) + dist(r_1, g), & \text{otherwise} \\ ERP(R, Rest(S)) + dist(s_1, g)\} & \end{cases}$
LCSS(R,S)	=	$\begin{cases} 0 & \text{if } m = 0 \text{ or } n = 0 \\ LCSS(Rest(R), Rest(S)) + 1 & \text{if } \forall d, \lvert r_{d,1} - s_{d,1} \rvert \leq \epsilon \\ \max\{LCSS(Rest(R), S), LCSS(R, Rest(S))\} & \text{otherwise} \end{cases}$
EDR(R,S)	=	$\begin{cases} n,\ m & \text{if } m = 0, \text{ if } n = 0 \\ \min\{EDR(Rest(R), Rest(S)) + \text{subcost}, & \text{otherwise} \\ EDR(Rest(R), S) + 1, EDR(R, Rest(S)) + 1\} & \end{cases}$

Table 2: Distance Functions: $dist(r_i, s_i)$ = L1 or L2 norm; subcost= 0 if $|r_{1,t} - s_{1,t}| \leq \epsilon$, else subcost= 1.

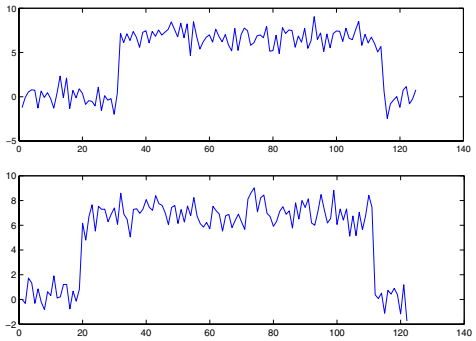

Figure 1: Two time series examples of the cylinder class from the Cylinder-Bell-Funnel Dataset.

of the FTSE algorithm. We also discuss its operation for LCSS and EDR, provide an example for each, and analyze the cost for each.

4.1 Dynamic Programming Overview

Time series comparison techniques such as those shown in Table 2 are typically evaluated using dynamic programming. Two time series R and S of length m and n, respectively, are compared using dynamic programming in the following way: First, an m x n two dimensional array A is constructed. Next, each element r_i of R is compared with each element s_j of S for all $1 \leq i \leq m$ and $1 \leq j \leq n$. The result of the comparison of r_i and s_j is added to the best cumulative score between $(r_1, ..., r_{i-1})$ and $(s_1, ..., s_{j-1})$ and stored in A at position (i, j). Once all the mn comparisons have been made and the elements of A are filled in, the final score is stored in $A(m, n)$.

For a concrete example, consider finding the LCSS score between the two time series shown in Figure 1. These two time series are from the popular Cylinder-Bell-Funnel (CBF) synthetic dataset [15]. The CBF dataset consists of time series from three classes, cylinders, bells, and funnels. Elements from the same class in the dataset are usually similar to each other. The two time series shown in Figure 1 are both from the cylinders class.

The two dimensional array used by the dynamic programming method for the LCSS comparison between R and S is shown in Figure 2, where the ϵ matching criteria is chosen as one-quarter the standard deviation of the normalized time series (a common ϵ value, also chosen in [5]). In Figure 2, black entries in the array at position (i, j) indicate mismatches between r_i and s_j. Gray entries in the array indicate matches between r_i and s_j that do not contribute to the LCSS score of R and S, and light gray colored entries indicate matches between r_i and s_j that are chosen by LCSS as the best alignment between R and S. Notice that the light gray colored entries run approximately from $(0, 0)$ to (m, n) along the grid diagonal. This makes intuitive sense – the alignment between two similar time series should match similar parts of each series (i.e. the front portion of R should not match the final portion of S).

4.1.1 *Shortcomings of Dynamic Programming*

When evaluating the LCSS of R and S, many of the comparisons made by dynamic programming when filling in the $m \times n$ two-dimensional array are between components of R and S that do not match, and therefore cannot positively impact the score between R and S. Much of the computation can be saved by finding only those elements r_i and s_j of R and S that match. An example of the positive matches between R and S is given in Figure 3. This is the same two-dimensional array that is shown in Figure 2, but the mismatching portions are no longer shown in black. The number of squares in this figure is much smaller than before. Since each square in the array represents work that must be done by the algorithm as well as space that must be used, the evaluation of ϵ threshold scoring techniques can be made more efficient. The main observation that we make is that if only those matches in Figure 3 are considered when comparing two time series, considerable computation can be saved since mismatching pairs are ignored.

4.2 Overview of FTSE

FTSE identifies the matching elements r_i and s_j between time series R and S *without* using a large two-dimensional array, such as that shown in Figure 2. This is done by treating R and S nonuniformly, rather than treating them in the same way as in dynamic programming. In dynamic programming, both R and S are treated the same (each is lined up on one edge of the two-dimensional array to be compared with the elements of the other sequence).

To find the matching pairs between R and S without comparing each r_i with every s_j, FTSE indexes the elements of R on-the-fly into a grid. Each element of R is placed into a grid cell. Now, to find the elements of R that s_j matches, the grid is probed with s_j. Only the elements of R that reside in the same grid cell as s_j need to be compared with it to see if they match.

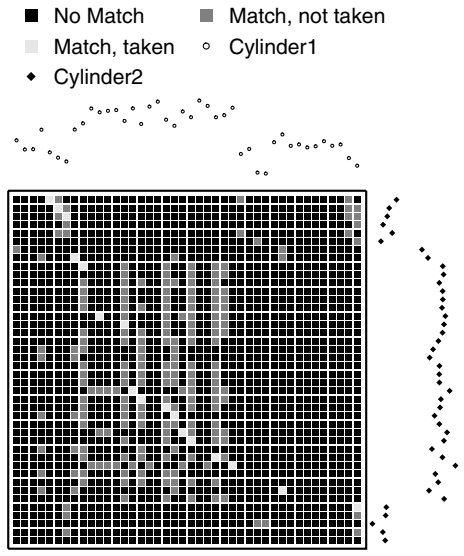

Figure 2: The dynamic programming computations necessary to evaluate LCSS between the two cylinder examples from Figure 1. The first time series is above the computation array and the second time series is on its right.

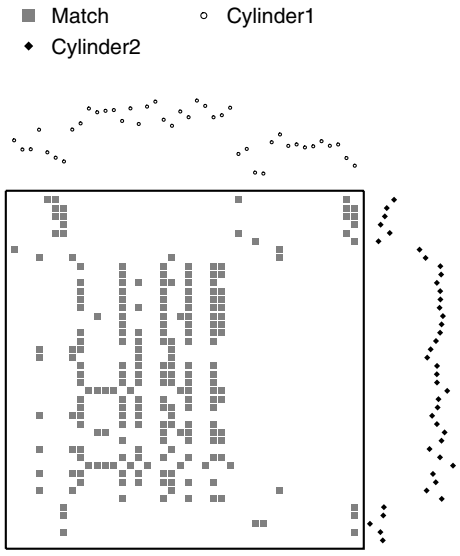

Figure 3: The matching elements as determined by LCSS between the two time series shown in Figure 1. This is what is needed by FTSE to perform the evaluation, in contrast to the larger number of comparisons needed by dynamic programming.

Once the matching pairs of R and S are found, the score of LCSS, EDR, or of the more general Swale ϵ scoring functions for R and S and the best alignment between them can be found using only an array of size n and a list containing the matching pairs between the two sequences (in contrast to the mn size array of dynamic programming). This is accomplished by noting that the grid can be probed by order of increasing S position. Hence, when the grid is probed with s_j to find the matching pairs between R and s_j, the matching pairs between the preceding $j-1$ elements of S with R have already been found. Therefore, when considering previous matches between $(s_1, ..., s_{j-1})$ and R for the best cumulative score for a match between r_i and s_j, there is no restriction on the previous matches from s_j. Any of the previous matches that contribute to the best cumulative score for r_i and s_j simply must be between elements of R before position i because the previous matches are inherently before s_j. Thus, high scores by position in R can be indexed into a one dimensional array of size n. The best alignment between R and S can be stored using a list containing matching pairs of elements derived from the grid.

One crucial requirement must be met for the index strategy of FTSE to win over the dynamic programming paradigm: the number of cells in the grid must be less than mn. Since the data is normalized, most elements fall between -3σ and 3σ. If epsilon is chosen as 0.5σ as is done in [29], then the grid contains 6/0.5=12 entries. Since time series are not usually more than 2 dimensional and typically of length considerably greater than 12, the grid size is typically much smaller than mn.

4.3 Finding Matches

In this section, we describe how the novel Fast Time Series Evaluation method finds matching pairs between elements of R and elements of S. FTSE measures the similarity between time series R and S with threshold value ϵ. Using ϵ, each pair of elements $r_i \in R$ and $s_j \in S$ can be classified as either a match or a mismatch. The elements r_i and s_j are said to match if $|r_i - s_j| < \epsilon$ in all dimensions. Otherwise, these two elements of R and S are a mismatch.

The first step in the FTSE algorithm is to find all intersecting pairs between elements of R and elements of S. The technique used to obtain these intersecting pairs is shown in Algorithm 1. First, a grid of dimensionality d is constructed (line 4 of the algorithm). The edge length of each element of the grid is ϵ.

In lines 6 to 8 of the algorithm, a Minimum Bounding Rectangle (MBR) is constructed for each element r_i of R. This MBR has a side length of 2ϵ in each dimension, and its center is the point r_i. This construction method ensures that r_i overlaps with no more than 3^d elements in the grid.

The MBR construction is illustrated in Figure 4 for one and two dimensions. In one dimension, the MBR of r_i is flattened into a line and intersects with 3 grid elements, as shown in Figure 4a. In two dimensions, the MBR of r_i intersects with 9 grid elements, as shown in Figure 4b.

A FIFO queue is associated with each cell g of the grid. The queue for each g is used to maintain a reference to all r_i that are within ϵ of g, in order of increasing i. This is done in line 9 of Algorithm 1.

The intersections between R and S are found in lines 11-18 of Algorithm 1. The grid cell g that contains each $s_j \in S$ is located. The elements of R in the queue associated with g are compared with s_j to see if they are within ϵ of one another. For each element r_k of R that is within ϵ of s_j, the index of r_k, i.e. k, is inserted into the intersection list L_j of s_j. The entries of L_j are also maintained in order of increasing k.

Note that the size of the grid is likely to be small for the following reason: Since data is normalized with mean zero and standard deviation $\sigma = 1$, most data will fall between -3 and 3. If the ϵ value is not exceptionally small relative to σ (which is common – for example, [29] uses 0.5σ), the size of the grid is reasonably small. Outliers beyond -3 or 3 are rare and can be captured into an additional grid cell.

Algorithm 1 Build Intersection List.

```
1: Input: R, m, S, n, ε
2: Output: Intersection List L
3: Local Variables: Grid G, MBR M
4: Initialize G: each grid element contains a queue that stores references
   to all intersecting elements.
5: for i = 1 to m do
6:    for k = 1 to d do
7:       M_i^k=(r_i^k-ε, r_i^k+ε)
8:    end for
9:    Insert M_i into the queue associated with each grid square g of G
      which M_i intersects.
10: end for
11: for i = 1 to n do
12:    Obtain queue q_g for grid square g in which s_i lies.
13:    for k ∈ q_g do
14:       if |s_i − r_k| < ε in all dimensions then
15:          insert k into L_i
16:       end if
17:    end for
18: end for
```

If the dimensionality is unusually high, the grid may be built on a subset of the overall dimensions since, as is shown in the next section, the number of matching pairs between time series R and S decreases quickly as the dimensionality grows. This way, the technique can still be applicable in higher dimensional spaces.

4.3.1 Cost Analysis of Match Finding

The cost of finding the matches using the grid technique of FTSE is $O(P + m + n)$, where P is the total number of comparison operations between the elements of R and the elements of S made when probing the grid, m is the length of R, and n is the length of S. The cost to insert each element of R into a grid is $O(m)$, and the cost to probe the grid with each element of S is $O(n)$. There are $O(P)$ total comparisons between R and S.

The total number of probe comparisons between R and S will be similar to the total number of matches M, both of which are determined by the size of ϵ. (An element of S will match all elements that are within ϵ in each dimension. It will be compared in the probe phase with elements that are up to 2ϵ away from it in each dimension, and on average within 1.5ϵ in each dimension, since the element will be mapped to 3 grid cells in each dimension that are each of size ϵ.) While this cost is $O(mn)$ in the worst case, in the general case, P will be much less than mn for common ϵ values.

To obtain an average case analysis, we consider two 1 dimensional sequences R and S whose elements are chosen uniformly from the unit space in 1 dimension. Once R and S are normalized, they will be distributed normally with mean 0 and variance 1. The conditional density function of the standard normal random variable Z is provided in Equation 1. Since S is normalized, the values of its elements follow a normal distribution and we can consider the value of s_j to be a normal random variable. The probability that a standard normal random variable Z lies between two values a and b, where $a < b$, is given by Equation 2. Hence, the probability that the normalized value of r_i, $N(r_i)$, lies within ϵ of the normalized value of s_j, $N(s_j)$, is given in Equation 3.

$$\Phi(z) = \frac{1}{\sqrt[2]{2\pi}} \int_{-\infty}^{z} e^{-u^2/2}\, du \quad (1)$$

$$P[a < Z \leq b] = \Phi(b) - \Phi(a) \quad (2)$$

$$P[N(r_i) - \epsilon < N(s_j) \leq N(r_i) + \epsilon] = \Phi(N(r_i) + \epsilon) - \Phi(N(r_i) - \epsilon) \quad (3)$$

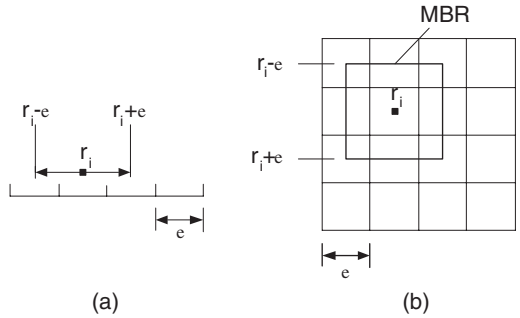

Figure 4: A depiction of R sequence elements with MBRs mapped to a (a) one-dimensional and (b) two-dimensional grid.

The expected number of matches M between r_i and the n elements of S is equal to the probability that a particular element of $N(S)$ matches with a value $N(r_i)$ multiplied by n. This is shown in Equation 4. We can then find the expected number of matches between R and S by summing over all m, in Equation 5. To obtain a solution in terms of mn, we can multiply by 1 (m/m) in Equation 6. We can approximate this summation (since we want an estimate in terms of mn) numerically by picking m values uniformly from the unit space for each r_i. We use two values for ϵ, 0.25σ and 0.50σ, which are commonly used ϵ values in [5] and [29], respectively. For $\epsilon = 0.50$, we obtain between $0.26mn$ and $0.27mn$ matches between R and S, and for $\epsilon = 0.25$, we obtain about $0.13mn$ matches between R and S. This is approximately a 4-7X improvement over dynamic programming.

$$E[M|r_i] = n(\Phi(N(r_i) + \epsilon) - \Phi(N(r_i) - \epsilon)) \quad (4)$$

$$E[M] = n \sum_{i=1}^{m} (\Phi(N(r_i) + \epsilon) - \Phi(N(r_i) - \epsilon)) \quad (5)$$

$$E[M] = mn \sum_{i=1}^{m} \frac{1}{m} (\Phi(N(r_i) + \epsilon) - \Phi(N(r_i) - \epsilon)) \quad (6)$$

The expected number of probes P can be found by replacing ϵ in Equation 6 with 1.5ϵ, the average maximum distance away from s_j that elements in R can be and still be compared with s_j in the probe phase. Doing so produces about $0.4mn$ probe comparisons when $\epsilon = 0.50$ and about $0.2mn$ probe comparisons when $\epsilon = 0.25$. This is an improvement of 2.5-5X over dynamic programming.

To obtain the average case analysis for 2 dimensions, we consider 2 dimensional time series R and S whose elements are drawn independently from the unit space. The analysis then is similar to the analysis above. The main difference is that $N(r_i)$ must match $N(s_j)$ in both dimensions. Since the values of r_i and s_j in each dimension are independent, we can arrive at Equation 7 by performing the same analysis as we did in the 1 dimensional case. If we approximate the number of matches in two dimensions numerically, we obtain between $0.06mn$ and $0.07mn$ matches when $\epsilon = 0.50$ and about $0.02mn$ matches when $\epsilon = 0.25$. This is about a 14-50X improvement over the dynamic programming, which produces mn comparisons.

$$E[M] = mn \sum_{i=1}^{m} \frac{1}{m} [(\Phi(N(r_i^1) + \epsilon) - \Phi(N(r_i^1) - \epsilon)) * (\Phi(N(r_i^2) + \epsilon) - \Phi(N(r_i^2) - \epsilon))] \quad (7)$$

Algorithm 2 LCSS Computation.

```
1: Input: R, m, S, n, ε, Intersections L
2: Output: score
3: Local Variables: Array matches
4: Initialize matches[0] = 0 and matches[1 to n] = m + 1.
5: max=0;
6: for j = 1 to n do
7:    Let c, a pointer into the matches array, =0.
8:    Let temp store an overwritten value from matches.
9:    temp=matches[0].
10:   for k ∈ L_j do
11:      if temp < k then
12:         while matches[c] < k do
13:            c = c + 1.
14:         end while
15:         temp=matches[c].
16:         matches[c] = k.
17:         if c > max then
18:            max = c
19:         end if
20:      end if
21:   end for
22: end for
23: score = max.
```

The expected number of probes P in 2 dimensions can be found by replacing ϵ in Equation 7 with 1.5ϵ, the average distance away from s_j that elements in R can be and still be compared with it in each dimension in the probe phase. Doing so produces about $0.16mn$ probe comparisons for R and S when $\epsilon = 0.50$ and about $0.05mn$ probe comparisons when $\epsilon = 0.25$. This is an improvement of 6-20X over dynamic programming for 2 dimensions.

4.4 Computing LCSS using FTSE

Once the intersections are found, the LCSS score for the pair R and S can be evaluated using Algorithm 2. An array called $matches$ is maintained that stores at position $matches[i]$ the smallest value k for which i matches exist between the elements of S and r_1, ... , r_k (line 4).

The values in $matches$ are filled by iterating through the elements of S (line 6). Variable c is an index into $matches$ and $temp$ stores an overwritten value from matches. For each of the intersections between r_k and s_j (line 10), k is checked against the value of $temp$ (line 11). Initially, $temp$ is 0 (line 9), so the algorithm proceeds to line 12. Next, c is incremented until the value of $matches[c]$ is not less than k. This indicates that there are $c-1$ matches between s_1, ... ,s_{j-1} and r_1, ..., $r_{matches[c-1]}$. Adding the match between s_j and r_k makes c matches.

The old value of $matches[c]$ is stored to $temp$ (line 15) and $matches[c]$ is updated to k (line 16). The maximum possible number of matches is stored in max and updated if c is greater than it (lines 17-19). The value of $temp$ is updated because subsequent intersections between R and s_j cannot make use of the intersection between r_k and s_j. This is because the $LCSS$ technique only allows s_j to be paired with one r_k so the previous value is retained as a stand in for the old $matches[c]$ for the next loop iteration. At the end of the algorithm, the $LCSS$ score is stored in max (line 23).

4.4.1 Example for LCSS

To demonstrate the operation of FTSE for LCSS, let R be $r_1 = 2.0$, $r_2 = -0.5$, $r_3 = 1.0$, $r_4 = -2.2$, and $r_5 = -0.4$, and let S be $s_1 = -0.4$, $s_2 = -2.1$, $s_3 = 1.4$, $s_4 = -1.8$. Let $\epsilon = 0.5$.

The matching phase of Algorithm 1 progresses by generating a one dimensional grid in which each grid cell has a side length of 0.5 (the ϵ value). Assume that grid boundaries occur at -2.5, -2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2, and 2.5 (line 4 of Algorithm 1). Next, the algorithm generates MBRs for each element of R (lines 5 to 8). The MBRs for each r_i are (1.5, 2.5) for r_1, (-1, 0) for r_2, (0.5, 1.5) for r_3, (-2.7, -1.7) for r_4, and (-0.9, 0.1) for r_5,

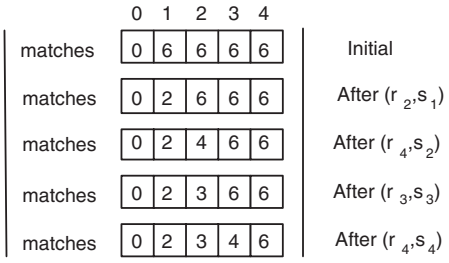

Figure 5: The $matches$ array during FTSE LCSS evaluation.

Next, the algorithm inserts each r_i into the grid (line 9). For example, the grid cell with boundaries $(-0.5, 0)$ contains both r_2 and r_5. The grid is then probed with each S value (lines 11-18). First, the grid is probed with s_1. The cell in which it lies, (-0.5,0), contains two MBRs – namely r_2 and r_5. Both elements of R are compared with s_1. Since they are both within ϵ of s_1, 2 and 5 are inserted into intersection list L_1, in that order.

Then, the grid is probed with s_2. The grid in which it is located, $(-2, -2.5)$, contains only one element, r_4. Since r_4 and s_2 are within 0.5 of one another, 4 is inserted into L_2. In a similar way, the grid is probed with s_3 and s_4 to produce a match with r_3 for s_3 and with r_4 for s_4.

Next, the operation of Algorithm 2 progresses. The initial state of the $matches$ array is shown in Figure 5. The algorithm begins processing the intersection list of s_1. The first value in the intersection list for s_1 is 2 (line 10 of the algorithm), since s_1 intersects with r_2.

Since $matches[0] < 2 < matches[1]$ (lines 12-14), the greatest number of matches possible so far is 1, so the c pointer is set to 1. Hence, the value of $temp$ is updated to the old value of $matches[1]$ (line 15), which is 6 and $matches[1]$ is updated to 2 (line 16). The value of max is updated to 1 (lines 17-18). The new status of $matches$ is shown in Figure 5. The next value in the intersection list for s_1 is 5. Since 5 is less than $temp$ (line 11), this intersection cannot be used.

After the processing of the s_1 intersection list, c and $temp$ are reset for the intersections of s_2 (lines 7-9). The first and only value in the intersection list for s_2 is 4 (line 10). Since $matches[1] < 4 < matches[2]$ (lines 12-14), c is set to 2. The value of $temp$ is updated to $matches[2]$ (line 15), and $matches[2]$ is updated to 4 (line 16). The value of max is also updated to 2 (lines 17-18).

The intersection list for s_3 is processed in the same way. Since its only match is with r_3, and because $matches[1] < 3 < matches[2]$, the value of $matches[2]$ is overwritten with 3 (see Figure 5). The intersection list of s_4 is also processed, and since s_4 intersects with r_4, and $matches[2] < 4 < matches[3]$, the value of $matches[3]$ is updated to 4, and the max value becomes 3.

Since all the S points have been processed, the algorithm terminates. The best possible number of matches between R and S is stored in max, which is 3. This is the LCSS score.

4.4.2 Cost Analysis of FTSE computing LCSS

The cost of FTSE for computing LCSS is $O(M + Ln)$, where M is the number of matches (discussed in Section 4.3.1) and L is the length of the longest matching sequence between R and S (i.e. the LCSS score). The proof of this is straightforward and hence is omitted (essentially, each matching pair is considered, and the

length of the longest match is stored in the array and is iterated over for each element n of S). In the worst case, this length will be equal to the length of $min(m, n)$ (since the LCSS score cannot exceed the length of either sequence), which could be as long as m, making the overall cost $O(mn)$. However, this worst case occurs only when all elements of R and S are matched in the LCSS score, which is not expected to happen often, even for sequences that are quite similar.

To obtain an average case analysis for the size of L in 1 dimension, we again assume time series R and S have their elements drawn uniformly from the unit space. We numerically approximate L by generating one thousand random versions of R and S, each of length one thousand. We then measure the average, maximum, and minimum length of L. For $\epsilon = 0.25$, the average size of L is $0.52m$, the maximum size is $0.54m$, and the minimum size is $0.51m$. For $\epsilon = 0.50$, the average size of L is $0.66m$, the maximum size is $0.68m$, and the minimum size is $0.64m$. The small variation in the sizes of L show that this average case analysis produces repeatable results. It also shows a 1.5-2X improvement over dynamic programming's mn computation to find the best alignment of R and S.

We obtain an average case analysis for 2 dimensions through numerical approximation as well. For $\epsilon = 0.25$, the average size of L is $0.23m$, the maximum size is $0.24m$, and the minimum size is $0.22m$. For $\epsilon = 0.50$, the average size of L is $0.41m$, the maximum size is $0.43m$, and the minimum size is $0.39m$. The smaller size of L in two dimensions is because r_i must match s_j in two dimensions instead of just 1, which produces fewer matches between each r_i and the elements of S (see Section 4.3.1). This analysis shows a 2.5-4X improvement over dynamic programming.

4.5 Computing EDR using FTSE

Unlike LCSS, EDR does not reward matches, but rather penalizes gaps and mismatches, so the FTSE algorithm changes slightly. The maximum possible score for EDR(R,S,ϵ) is 0 if R and S are nearly identical. The worst possible score is $-1 * (m+n)$, if all m elements of R and all n elements of S incur a gap penalty. A mismatch penalty of -1 between elements r_i of R and s_j of S can thus be viewed as a savings of 1 over two mismatches (which together have a cost of -2 versus the -1 mismatch cost). A match between r_i and s_j has a score of 0, which is a savings of 2 over the gap penalty costs. FTSE for EDR thus scores a match with a $+2$ reward and a mismatch with a $+1$ reward, and considers the baseline score to be $-1 * (m + n)$ instead of zero.

The FTSE algorithm for EDR is presented in Algorithm 3. The $matches$ array is initialized (line 4 of Algorithm 3) similar to Algorithm 2. Since match rewards are being scored with a 2, the array needs to be twice as long. Variables max (line 5), c (line 7), and $temp$ (line 8) are the same as before. Variable $temp2$ stores an overwritten value of the $matches$ array, similar to $temp$. A second such temporary holder is needed because match rewards are scored with a $+2$, hence two values can be overwritten on an iteration.

Most of FTSE for EDR is the same as FTSE for LCSS, such as iterating through the elements of S (line 6) and checking each element of the intersection list for the appropriate $matches$ value (lines 10-12).

Mismatches are handled by lines 13-19. Variable $temp$ stores the value of $matches[c-1]$. Since s_j can obtain a mismatch with any element of R, each value of $matches$ must be incremented (line 15). The overwritten value of $matches$ is stored back into $temp$ (lines 14, 16, 18). Line 13 checks that a previous element has not matched at position c of $matches$ (producing a higher score than a potential mismatch) and that the length of R has not been exceeded.

Algorithm 3 EDR Computation.

```
 1: Input: R, m, S, n, ε, Intersections L
 2: Output: score
 3: Local Variables: Array matches
 4: Initialize matches[0] = 0 and matches[1 to 2n] = m + 1.
 5: max=0;
 6: for j = 1 to n do
 7:    Let c, a pointer into the matches array, =0.
 8:    Let temp store an old value from matches,=matches[0]
 9:    Let temp2 store an old value from matches,=matches[0]
10:    for k ∈ L_j do
11:       if temp < k then
12:          while matches[c] < k do
13:             if temp < matches[c] − 1 and temp < m − 1 then
14:                temp2 = matches[c]
15:                matches[c] = temp + 1
16:                temp = temp2
17:             else
18:                temp = matches[c]
19:             end if
20:             c = c + 1.
21:          end while
22:          temp2=matches[c].
23:          matches[c]=temp + 1.
24:          temp=matches[c + 1].
25:          if matches[c + 1] > k, then matches[c + 1] = k
26:          if max < c + 1, then max = c + 1
27:          c = c + 2.
28:       else if temp2 < k and k < matches[c] then
29:          temp2 = temp
30:          temp = matches[c]
31:          matches[c] = k
32:          if max < c, then max = c
33:          c = c + 1
34:       end if
35:    end for
36:    for j = c to max + 1 do
37:       if temp < matches[j] − 1 and temp < m − 1 then
38:          temp2 = matches[j]
39:          matches[j] = temp + 1
40:          temp = temp2
41:          if max < j, then max = j
42:       else
43:          temp = matches[j]
44:       end if
45:    end for
46: end for
47: score = max − (m + n).
```

Lines 22-27 handle a match. The previous value of $matches[c]$ is stored in $temp2$ (line 22) since $matches[c]$ will be updated with a mismatch score (line 23); $matches[c+1]$ is stored in $temp$ (line 24) since a match is recorded at $matches[c + 1]$ (line 25). The maximum score and c counter are updated in lines 26 and 27 respectively.

Lines 28-34 handle the case when the next matching element in intersection list L_j is greater than the previous element by exactly 1. For example, if s_j matches elements r_k and r_{k+1}. In this case, the match with r_{k+1} will not necessarily exceed $temp$, the previously updated $c-1$ value, but might exceed $temp2$, the previously updated $c - 2$ value. The update code is similar to lines 22-27 already described.

Lines 36-45 handle the case when either s_j has no matches in R or when s_j matches elements only near the beginning of R. In this case, s_j could obtain mismatch scores with the remaining portions of R. This section of Algorithm 3 is similar to the already described lines 13-19.

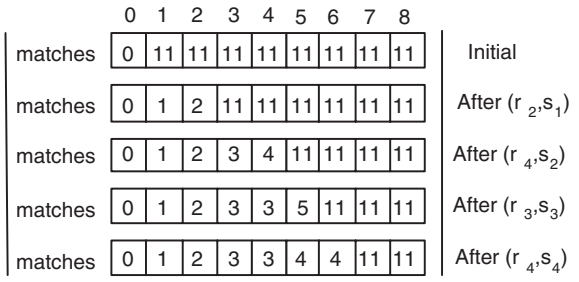

Figure 6: The $matches$ array during FTSE EDR evaluation.

4.5.1 Example for EDR

We show the operation of FTSE evaluating EDR with the same example as was used for LCSS. Following the intersection list generation of Algorithm 1 already discussed, Algorithm 3 begins by initializing $matches$. This initialized state is seen in Figure 6.

The first match is obtained from the intersection list (line 10 of the algorithm). This is the intersection between r_2 and s_1, hence $k = 2$. Since $matches[0] < 2 < matches[1]$, c is set to 1 in lines 13-20. $temp$ and $temp2$ are both set to 11 (lines 22 and 24). $matches[1]$ is set to 1 because s_1 can mismatch with r_1. $matches[2]$ is set to 2 because r_2 matches with s_1. Nothing is done for the match between r_5 and s_1. The updated $matches$ array is shown in Figure 6.

The intersection list for s_1 is now empty, so FTSE proceeds to line 36. c is 3 and $max + 1$ is 3, so the loop is taken exactly once. The if condition at line 37 fails, so no changes are made to $matches$.

Next, the intersection between r_4 and s_2 is processed, so $k = 4$. Since $matches[2] < 4 < matches[3]$, c is set to 3 by lines 12-20. No changes are made to the $matches$ array by lines 14-16. Hence, the else condition (line 18) is taken for both $c = 1$ and 2 and $temp = 2$. $matches[3]$ is set to $temp + 1 = 3$ (line 23) and $matches[4]$ is set to 4 (line 25) since $k = 4$. max is updated to 4 (line 26) and c is set to 5 (line 33). Again, lines 36-45 make no changes to $matches$.

The intersection between s_3 and r_3 is next considered. As shown in Figure 6, The match between s_3 and r_3 can use the match between s_1 and r_2 at position 2 of $matches$. So, the value of k (3) is recorded at position 4 of $matches$. When processing for s_3 reaches line 36, $temp$ is 4, c is 5, and max is 4. Hence, lines 37-40 record a value of 5 in position $matches[5]$. This is because s_3 builds upon the r_4 and s_2 match with a mismatch between itself and r_5.

Finally, the intersection between r_4 and s_4 is processed. Since the intersection between r_3 and s_3 has resulted in a $match[4]$ value of 3, line 23 will set $match[5]$ to 4, and line 25 will set $match[6]$ to a value of 4. This means that max is also set to 6 (line 27). The final score achieved (line 47) is $-1 * (5 + 4) + 6 = -3$.

4.5.2 Cost Analysis of FTSE computing EDR

The cost of FTSE when evaluating EDR is $O(M + Tn)$, where M is the number of matches between R and S, n is the length of time series S, and T is the value of max in Algorithm 3. This complexity results from iterating over the $matches$ array for each of the n elements of S up to max places in the array. The value of max is bounded between $min(m, n)$ and $2min(m, n)$. This is because the value of max is increased once for each mismatch and two times for each match that occurs in the final alignment between R and S. While this is still $O(mn)$ in the worst case, FTSE for EDR still achieves efficiencies relative to dynamic programming since it only needs to store the number of matching elements M between R and S. This leads to better performance, which is later quantified experimentally in Section 6.2.

4.6 Maintaining the Best Matching Pairs

The FTSE algorithm for either LCSS or EDR can be easily modified to find not only the best score between two time series, but also the best sequence of matching pairs that produce that score. Maintaining the best sequence of matching pairs is useful for applications that seek to compute the best alignment between two time series. We now discuss how to modify FTSE for LCSS; a similar discussion for EDR is omitted.

The matching pairs found in Algorithm 1 are maintained in a list of intersections. The list element that contains a particular match can be linked to the previous best set of list elements when the match is considered in line 10 of Algorithm 2 since each match contributes to the best score in at most one position. The best alignment can be found by maintaining an array of the list elements that contain the matching pairs. Each array position corresponds to the last match in the sequence, with the remaining matches chained out behind it.

The following three lines can be added to Algorithm 2 between lines 16 and 17 to maintain the record of the best alignment (where l_k is the list element for match k):

$alignment[c] = l_k$.
if $c > 0$ *then* $l_k.next = alignment[c - 1]$.
else $l_k.next = 0$.

The $alignment$ array is of length n, similar to $matches$. It is initialized to all null entries. At the end of the algorithm, the best sequence is maintained in the $alignment$ array, and it can be returned to the user.

5. THE SWALE SCORING MODEL

The FTSE algorithm can be used to evaluate a broad class of ϵ threshold value based scoring models, of which LCSS and EDR are two examples. This broader class of scoring models includes a new Swale scoring model, which we present below. The Swale scoring model improves over previous approaches in several ways. First, it allows for a sequence similarity score to be based on both match rewards and mismatch penalties. Second, it allows for the match reward and gap penalties to be weighted relative to one another. These weights also allow a user or domain expert with particular knowledge of a certain area to tune the distance function for optimal performance instead of having only one technique for all data domains. If the user has no such domain-specific knowledge, a training dataset can be used to automatically learn the weights (as we do in all the experiments presented in this paper).

More formally, the Swale distance function is defined as:

DEFINITION 5.1. *Let R and S be two time series of length m and n, respectively. Let the gap cost be gap_c and let the match reward be $reward_m$. Then $Swale(R, S)$ =*

$$\begin{cases} n * gap_c, & \text{if } m = 0 \\ m * gap_c, & \text{if } n = 0 \\ reward_m + & \text{if } \forall d, |r_{d,1} - s_{d,1}| \leq \epsilon \\ \quad Swale(Rest(R), Rest(S)), & \\ max\{gap_c + Swale(Rest(R), S), & \text{otherwise} \\ gap_c + Swale(R, Rest(S))\} & \end{cases}$$

Next we explain why Swale offers a better similarity measure compared to the best existing ϵ methods, namely LCSS and EDR

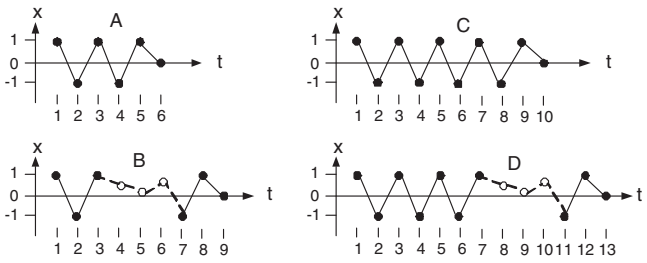

Figure 7: Time Series Examples

[5, 30]. For this illustration, consider the sequences shown in Figure 7. Sequence A contains six elements. Sequence B has the same six elements as A, but has three additional "noise" elements embedded in it. Sequence C contains ten elements, and sequence D has the same ten elements with three additional "noise" elements in it. Note that the number of mismatched elements between C and D is the same as that between A and B.

Both EDR and LCSS lose some information when scoring these sequences. EDR scores gaps and mismatches, but does not reward matches. In this sense, it only measures dissimilarity between two sequences. For example, A and B receive the same score as C and D even though C and D have nearly twice as many matching elements.

LCSS rewards matches, but does not capture any measure of dissimilarity between two sequences. For example, the LCSS technique scores C and D identically to C scored with itself, which is not intuitive.

Swale is similar to LCSS because it rewards matches between sequences, but it also captures a measure of their dissimilarity by penalizing gap elements. Swale allows C and D to obtain a higher score than A and B because they have more matching elements while still penalizing them for gap costs.

The Swale scoring function can be evaluated with the same FTSE algorithm described for LCSS by simply changing the last line of Algorithm 2 to $score = max * reward_m + gap_c * (m + n - 2max)$.

6. EXPERIMENTS

In this section, we experimentally evaluate the performance of FTSE, and the accuracy of Swale.

6.1 FTSE Experimental Evaluation

In this section, we evaluate the effectiveness of the FTSE technique evaluating both LCSS and EDR. Since Swale is evaluated with only a small modification to FTSE for LCSS, its performance is identical to LCSS with FTSE. All experiments are run on a machine with a 1.7 GHz Intel Xeon, with 512MB of memory and a 40GB Fujitsu SCSI hard drive, running Debian Linux 2.6.0. We compare the performance of FTSE against DTW, ERP, LCSS, and EDR. Each technique is evaluated using a traditional, iterative dynamic programming-style algorithm.

The performance of FTSE is dependant on the ϵ value, since this value determines which elements of R and S are close enough to one another to be matched. The emphasis of our work is not on describing how to pick an ϵ value for either LCSS or EDR, but to demonstrate the effectiveness of FTSE for reasonable choices of ϵ. Consequently, we show results with an ϵ value of 0.5σ, where σ is the standard deviation of the data (since we are dealing with normalized data, σ is 1). We have chosen this ϵ value since it was shown to produce good results in [29].

Method	CM	ASL	CBF	Trace
DTW	53.23	1.31	1.94	521.93
ERP	77.43	1.76	2.68	553.73
LCSS	42.74	0.93	1.41	386.09
EDR	43.69	1.01	1.41	390.87
SC-B$_{DTW}$	10.55	0.78	0.71	104.91
SC-B$_{LCSS}$	14.61	0.80	0.88	132.43
I-Par	15.44	0.86	0.90	141.05
FTSE$_{LCSS}$	5.13	0.78	0.74	80.80
FTSE$_{EDR}$	6.27	0.82	0.85	99.17

Table 3: Time in seconds to cluster a given dataset, using techniques that compute the actual alignment.

Method	CM	ASL	CBF	Trace
DTW	35.23	1.20	1.78	329.17
LCSS	14.24	0.84	1.16	129.42
SC-B$_{DTW}$	7.05	0.76	0.74	72.91
SC-B$_{LCSS}$	6.40	0.74	0.72	66.95
FTSE$_{LCSS}$	2.69	0.72	0.61	48.28
FTSE$_{SC-B}$	2.26	0.70	0.60	40.74

Table 4: Time in seconds to cluster a given dataset, using $O(n)$ storage techniques that do not compute the alignment.

In our first experiment we show the average time to perform the similarity comparisons for a complete linkage clustering evaluation. Complete linkage clustering of time series was used in both [30] for LCSS and in [5] for EDR. For a dataset with k time series, each clustering run involves computing approximately $k \times (k-1)$ time series similarity scores.

To perform the complete linkage clustering, our evaluation uses the same datasets used in [30] and in [5], which includes the Camer-amouse (**CM**) dataset [8] and the Australian Sign Language (**ASL**) dataset from the UCI KDD archive [28]. Since both of these datasets are two dimensional, we also experiment with the popular Cyliner-Bell-Funnel (**CBF**) dataset of [15] and the **Trace** dataset of [24]. The CBF dataset contains three classes (one each for the cylinder, bell, and funnel shapes) and is synthetically generated. We use 10 examples from each class in the clustering. The Trace dataset is a four class synthetic dataset that simulates instrument failures inside of a nuclear power plant. There are fifty examples for each class.

The CM dataset consists of 15 different time series obtained from tracking the fingertips of people in two dimensions as they write words. Three different people wrote out five different words. This gives a total of five distinct class labels (one for each word) and three members for each class.

The ASL dataset contains examples of Australian Sign Language signs. The dataset contains time series in two dimensions for words that are signed by the user, and each word is signed five different times. We choose to use the same 10 word examples of [30]. This gives us a dataset with 10 classes with 5 time series per class.

We also compare with the Sakoe-Chiba Band (SC Band) and Itakura Parallelogram techniques for warp-restricting DTW. A restriction value of 10 percent is used in [34], so we also use this value. A similar technique to the SC Band for LCSS is described in [30], which sets the restriction value to 20 percent. The Itakura Parallelogram is referred to as (I-Par).

The results for the complete linkage clustering test is shown in Table 3. For the CM data set, FTSE is faster than the dynamic

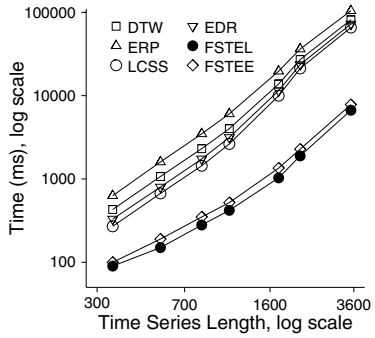

Figure 8: Cost of computing similarity scores v/s time series length. (CLIVAR dataset)

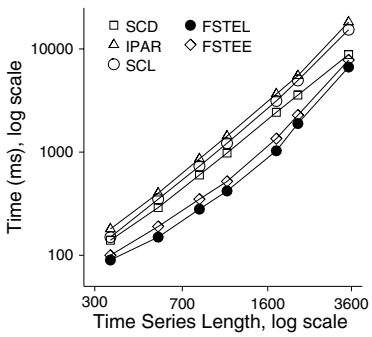

Figure 9: Cost of computing similarity scores v/s time series length; comparison with warp-restricting methods. (CLIVAR dataset)

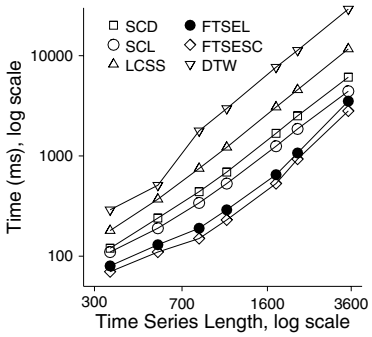

Figure 10: Cost of computing similarity scores v/s time series length; comparison with methods that do not compute the actual alignment. (CLIVAR dataset)

programming methods by a factor of 7-8 and faster than the warp-restricting techniques by a factor of 2-3. FTSE is faster than DP by a factor of 4-5 and is nearly twice as fast as the SC Band evaluating LCSS for the Trace dataset. FTSE also consistently performs better than dynamic programming on the other datasets. Note that the performance advantage achieved using FTSE relative to the various DP techniques is not as large for ASL and CBF as it is for the CM and Trace datasets. This is because the average sequence length of the ASL and CBF sequences are 45 and 128 respectively, while the average length of the CM is 1151 and the Trace is 255. This indicates that FTSE performs better than DP as length increases, which we also show in the next experiment. The Trace dataset also takes longer to evaluate than others datasets because it contains many more sequence examples (200) than CM (15), ASL (50), or CBF (30).

We also show results for both the DP and SC Band techniques using $O(n)$ storage techniques that produce the best score but do not yield the best alignment between the two sequences in Table 4. Essentially, since the i^{th} column of the mn matrix depends only on the $i - 1^{th}$ column, 2 column arrays can be used. Similarly, it is a simple extension for FTSE to show that the list of intersections need not be materialized if an alignment between the two time series is not required; we have also implemented this version of FTSE ($FTSE_{LCSS}$ in the table). FTSE can also be implemented with a warp-restriction (in essence, to only consider matches in the same narrow band as the SC Band technique). We have also implemented this version with a restriction value of 20 percent to show that FTSE ($FTSE_{SC-B}$ in the table) can obtain the same score as the SC Band, if desired. In these tests, we limit results to LCSS and DTW evaluation In these tests, FTSE is faster than DP for LCSS by a factor of 7 and by more than 2X when restricting the warping window for the 2 dimensional CM dataset and by a factor of 2.5 over exact DP for LCSS and by a factor of 1.5 when restricting the warping window when evaluating the 1 dimensional Trace dataset.

The second experiment evaluates the effectiveness of the FTSE algorithm as the time series length varies. For this experiment, we use the CLIVAR-Surface Drifters trajectory dataset from [22], which contains climate data obtained in 2003 from free-floating buoys on the surface of the Pacific Ocean. This data contains the longitude and latitude coordinates for each buoy. The time series in this data set vary in length from 4 to 7466 data points.

From the CLIVAR-Surface Drifters dataset, subsets of data are produced such that each subset contains time series of similar length (all time series in a subset are within 10% of the average). For experimentation, subsets of 5 time series each are chosen with the following average time series lengths: 349, 554, 826, 1079, 1739, 2142, and 3500. As before, we report the time needed to perform $k \times (k-1)$ comparisons (the same as was done in the clustering experiments). Since each subset contains 5 time series, this is the time to perform 20 time series comparisons. The results for this experiment are shown in Figures 8, 9, and 10.

Figure 8 shows the results for FTSE evaluating LCSS (labeled FTSEL) and EDR (FTSEE). It also shows DTW, ERP, LCSS, and EDR evaluated using dynamic programming. As can be seen in this figure, FTSE is nearly an order of magnitude faster than the dynamic programming techniques. The figure also shows that the performance advantage of FTSE over the DP techniques increases with sequence length.

Figure 9 presents results for FTSE and the Sakoe-Chiba Band (SCD evaluating DTW and SCL evaluating LCSS) and Itakura Parallelogram (IPAR) warp-restricting techniques. FTSE is about twice as fast as the Sakoe-Chiba Band and 2-3 times faster than the Itakura Parallelogram technique. SC for LCSS is slower than for DTW because the warping restriction needed for good results (20%) for LCSS is larger than for DTW (10%).

Figure 10 presents results for the $O(n)$ storage techniques already discussed. FTSE (FTSEL in the figure) is generally about 3 times faster than the DP methods (LCSS and DTW) and almost twice as fast when the warp-restricted version of FTSE (FTSESC) is compared with the SC Band technique (SCL and SCD).

In summary, compared to existing methods that compute the actual alignment, *FTSE is up to 7-8 times faster than popular dynamic programming techniques for long sequences and 2-3 faster than warp-restricting techniques, while providing an exact answer*.

6.2 Experimental Cost Analysis of FTSE

The complexity and average case cost of FTSE have already been analyzed in Sections 4.3.1 and 4.4.2. In this section, we analyze the experimental cost of FTSE to show why it performs better than the other techniques that produce the best alignment, using the CM dataset as an example.

FTSE is more efficient for two reasons: it performs fewer operations than the competing techniques and it requires less space, which improves the algorithm's memory and cache performance.

The number of operations performed by FTSE is dependent on two principle components: the number of matches between elements of R and elements of S obtained by Algorithm 1 and the number of reads or writes to the $matches$ array in Algorithm 2.

For the CM dataset, there are about 120 thousand matching pairs on average between any two sequences R and S (since the average length of each time series is 1151 elements, there are a total possibility of $1151*1151 = 1.32$ million) and about 300 thousand reads and writes to the $matches$ array. This means that FTSE performs about 420 thousand operations on the CM dataset versus the 1.32 million for DP, which is less than one-third.

The amount of space used by FTSE is dependant on the number of matching pairs generated by Algorithm 1. The $matches$ array and the grid (which contains fewer than 200 grid cells for CM) are of negligible size. For the CM dataset, the number of matching pairs is approximately 120 thousand. The equivalent DP algorithm writes approximately 1.32 million elements.

To test that this considerable space difference actually results in cost savings, we modified Algorithm 2 by allocating an amount of space equivalent to that of the DP algorithm and adding a line between lines 13 and 14 of Algorithm 2 that randomly writes to an element of the allocated space. The new algorithm attains improved performance only from the saved operations, not from memory or cache efficiency. The time this new FTSE takes to cluster the CM dataset is 12.12 seconds (before it was 5.13). This is expected, since DP for LCSS takes 42.74 seconds and the ratio of operations between FTSE and DP is $420/1320$ and $42.74*420/1320 = 13.59$ seconds.

6.3 Evaluation of the Swale Scoring Model

In this section, we evaluate the effectiveness of the Swale scoring model compared to existing similarity models. For this evaluation, we test the ability of the model to produce high-quality clusters. (Following well-established methodology [5, 15, 30].)

For our evaluation, we used the Cameramouse (CM) dataset [8], and the Australian Sign Language (ASL) dataset (described in Section 5.1). In addition, we also obtained an additional dataset from the UCI KDD archive [28] called the High Quality ASL. This dataset differs from the ASL dataset in the following way: In the ASL dataset, several different subjects performed the signing, and lower quality test gloves were used. The High Quality ASL (HASL) dataset consists of one person performing each sign 27 times using higher quality test gloves. Details regarding these differences can be found at [28]. We do not provide detailed results for the Trace and CBF datasets because these datasets have a small number of classes (4 and 3, respectively) and hence do not offer as much room for differentiation as the ASL datasets (all techniques tested on Trace and CBF performed nearly identically).

In the evaluation we perform hierarchical clustering using Swale, DTW, ERP, LCSS, and EDR. (We omit a comparison with the Euclidean distance, since it has been generally shown to be less robust than DTW [5, 13, 30].) Following previous established methods [5,15,30], for each dataset, we take all possible pairs of classes and use the complete linkage algorithm [12], which is shown in [30] to produce the best clustering results.

Since DTW can be used with both the L1-norm [4] and the L2-norm [17] distances, we implement and test both these approaches. The results for both are similar. For brevity, we present the L1-norm results.

The Swale match reward and mismatch penalty are computed using training datasets. The ASL dataset in the UCI KDD archive contains time series datasets from several different signers placed into directories labeled by the signer's name and trial run number. We selected the datasets labeled adam2, john3, john4, stephen2, and stephen4 for test datasets 1-5, respectively, and datasets andrew2 and john2 for training. For the HASL, each word has 27 examples, so we are able to group them into 5 different collections of data, each with 5 examples, with 2 examples left over. The first such dataset is used for training, and the others are used for testing.

	1	2	3	4	5	total
DTW	**40**	**32**	34	37	41	184
ERP	38	**32**	39	40	41	190
LCSS	**40**	30	38	39	41	188
EDR	38	27	39	37	**43**	184
Swale	39	29	**41**	**42**	42	**193**

Table 5: Number of correct clusterings (each out of 45) for the ASL dataset. The best performers are highlighted in bold.

	1	2	3	4	total
DTW	8	8	2	5	23
ERP	9	5	4	**7**	25
LCSS	8	**10**	**6**	**7**	31
EDR	13	2	3	6	24
Swale	**18**	**10**	5	**7**	**40**

Table 6: Number of correct clusterings (each out of 45) for the HASL dataset. The best performers are highlighted in bold.

For the training algorithm, we use the random restart method [25]. Since the relative weight of the match reward and gap cost is what is important (i.e. the ratio between them), we fix the match reward to 50 and use the training method to pick various gap costs. The computed mismatch cost for ASL is -8 and for HASL is -21.

The CM dataset does not have enough data to produce a training and a test set. We therefore chose the ASL weight as the default. All techniques correctly clustered the dataset (10 out of 10 correct).

The total number of correct clusterings for each of the five different ASL datasets (out of 45 for each dataset) are shown in Table 5. As can be seen in the table, Swale has the overall best performance for the tests. There is a high degree of variability for all the similarity functions from one ASL dataset to the next, but some general trends do emerge. For example, all of the techniques perform well on dataset 5, averaging over 40 correct clusterings out of 45 possible. All of the techniques do relatively poorly on dataset 2, averaging only about 30 correct clusterings out of 45. These two datasets emphasize the variability of data for multi-dimensional time series; two datasets in the same ASL clustering framework produce very different results for all of the tested similarity measures.

The results for the HASL datasets are shown in Table 6 and are once again out of a possible 45 for each technique on each test. Overall, DTW, ERP, LCSS, EDR, and Swale obtain fewer correct clusterings on the HASL datasets than they do on the ASL datasets. There is also high variability in accuracy across the datasets, just as in the ASL data presented in Table 5. Swale performs much better on the classifications for the HASL datasets than the alternative techniques, obtaining a total of 40 correct total classifications. The closest competitor is the LCSS technique with 31. This dataset also highlights how Swale leverages the combination of the match reward and gap penalty on real datasets for improved accuracy. On HASL dataset 1, EDR, which also uses gap penalties, performs much better than the LCSS technique. Swale also performs very well on this dataset. On HASL dataset 2, the LCSS technique performs better than EDR. Swale performs as well as the LCSS technique on this dataset, and is thus able to obtain the best of both worlds - it does well when EDR does well, and also does well when LCSS does well!

In summary, the results presented in this section demonstrate that Swale is consistently a more effective similarity measuring method compared to existing methods.

7. CONCLUSIONS

In this paper, we have presented a novel algorithm called FTSE to speed up the evaluation of ϵ threshold-based scoring functions for time series datasets. We have shown that FTSE is faster than the traditionally used dynamic programming methods by a factor of 7-8, and is even faster than approximation techniques such as the Sakoe-Chiba Band by a factor of 2-3. In addition, we also presented a flexible new scoring model for comparing the similarity between time series. This new model, called Swale, combines the notions of gap penalties and match rewards of previous models, and also improves on these models. Using extensive experimental evaluation on a number of real datasets, we show that Swale is more accurate compared to existing methods.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. IIS-0414510. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We would also like to thank the anonymous reviewers for their comments that have improved our paper.

9. REFERENCES

[1] R. Agarwal, C. Faloutsos, and A. R. Swami. Efficient Similarity Search in Sequence Databases. In *FODO*, pages 69–84, 1993.

[2] D. Berndt and J. Clifford. Using Dynamic Time Warping to Find Patterns in Time Series. In *AAAI-94 Workshop on Knowledge Discovery in Databases*, pages 359–370, 1994.

[3] K. Chan and A.-C. Fu. Efficient Time Series Matching by Wavelets. In *ICDE*, pages 126–133, 1999.

[4] L. Chen and R. Ng. On the Marriage of Lp-norms and Edit Distance. In *VLDB*, pages 792–803, 2004.

[5] L. Chen, M. T. Özsu, and V. Oria. Robust and Fast Similarity Search for Moving Object Trajectories. In *SIGMOD*, pages 491–502, 2005.

[6] R. Cilibrasi and P. M. B. Vitanyi. Clustering by Compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[7] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast Subsequence Matching in Time-Series Databases. In *SIGMOD*, pages 419–429, 1994.

[8] J. Gips, M. Betke, and P. Fleming. The Camera Mouse: Preliminary Investigation of Automated Visual Tracking for Computer Access. In *Proc. of Rehabilitation Engineering and Assistive Technology Society of North America (RESNA)*, pages 98–100, 2000.

[9] D. Goldin and P. Kanellakis. On Similarity Queries for Time-Series Data: Constraint Specification and Implementation. In *Constraint Programming*, pages 137–153, 1995.

[10] J. W. Hunt and T. G. Szymanski. A Fast Algorithm for Computing Longest Common Subsequences. *CACM*, pages 350–353, 1977.

[11] F. Itakura. Minimum Prediction Residual Principle Applied to Speech Recognition. *IEEE Trans. Acoustics, Speech, and Signal Proc.*, Vol. ASSP-23:52–72, 1975.

[12] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[13] E. Keogh. Exact Indexing of Dynamic Time Warping. In *VLDB*, pages 406–417, 2002.

[14] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *KAIS*, 3(3):263–286, 2000.

[15] E. Keogh and S. Kasetty. The Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration. In *SIGKDD*, pages 102–111, 2002.

[16] E. Keogh, S. Lonardi, and C. A. Ratanamahatana. Towards Parameter-Free Data Mining. In *SIGKDD*, pages 206–215, 2004.

[17] E. Keogh and M. Pazzani. Scaling Up Dynamic Time Warping to Massive Datasets. In *PKDD*, pages 1–11, 1999.

[18] E. J. Keogh, K. Chakrabarti, S. Mehrotra, and M. J. Pazzani. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. In *SIGMOD*, pages 151–162, 2001.

[19] S.-W. Kim, S. Park, and W. W. Chu. An Index-Based Approach for Similarity Search Supporting Time Warping in Large Sequence Databases. In *ICDE*, pages 607–614, 2001.

[20] F. Korn, H. Jagadish, and C. Faloutsos. Efficiently Supporting Ad Hoc Queries in Large Datasets of Time Sequences. In *SIGMOD*, pages 289–300, 1997.

[21] S. Kuo and G. R. Cross. An Improved Algorithm to Find the Length of the Longest Common Subsequence of Two Strings. *ACM SIGIR Forum*, 23(3-4):89–99, 1989.

[22] Climate variability and predictability website. http://www.clivar.org.

[23] I. Popivanov and R. Miller. Similarity Search Over Time Series Data Using Wavelets. In *ICDE*, page 212, 2001.

[24] C. Ratanamahatana and E. Keogh. Making Time-series Classification More Accurate Using Learned Constraints. In *SIAM International Conference on Data Mining*, 2004.

[25] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[26] H. Sakoe and S. Chiba. Dynamic Programming Algorithm Optimization for Spoken Word Recognition. *IEEE Trans. Acoustics, Speech, and Signal Proc.*, Vol. ASSP-26(1):43–49, 1978.

[27] Y. Sakurai, M. Yoshikawa, and C. Faloutsos. FTW: Fast Similarity Search under the Time Warping Distance. In *PODS*, pages 326–337, 2005.

[28] S. Hettich and S. D. Bay. The UCI KDD Archive [http://kdd.ics.uci.edu].

[29] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing Multi-Dimensional Time-Series with Support for Multiple Distance Measures. In *SIGKDD*, pages 216–225, 2003.

[30] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering Similar Multidimensional Trajectories. In *ICDE*, pages 673–684, 2002.

[31] J. Yang and W. Wang. CLUSEQ: Efficient and Effective Sequence Clustering. In *ICDE*, pages 101–112, 2003.

[32] B.-K. Yi and C. Faloutsos. Fast Time Sequence Indexing for Arbitrary Lp Norms. In *VLDB*, pages 385–394, 2000.

[33] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient Retrieval of Similar Time Sequences Under Time Warping. In *ICDE*, pages 201–208, 1998.

[34] Y. Zhu and D. Shasha. Warping Indexes with Envelope Transforms for Query by Humming. In *SIGMOD*, pages 181–192, 2003.