Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets

9 authors, including:

Chin-Chia Michael Yeh
University of California, Riverside
27 PUBLICATIONS   521 CITATIONS

SEE PROFILE

Yan Zhu
University of California, Riverside
23 PUBLICATIONS   426 CITATIONS

SEE PROFILE

Nurjahan Begum
University of California, Riverside
11 PUBLICATIONS   365 CITATIONS

SEE PROFILE

Anh Dau
University of California, Riverside
19 PUBLICATIONS   402 CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Learning and performance assessment with class-imbalanced datasets View project

Time Series Segmentation View project

Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets

Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding,
Hoang Anh Dau, †Diego Furtado Silva, ‡Abdullah Mueen, and Eamonn Keogh

University of California, Riverside, †Universidade de São Paulo, ‡University of New Mexico
{myeh003, yzhu015, lulan001, nbegu001, yding007, hdau001}@ucr.edu, diegofsilva@icmc.usp.br, mueen@unm.edu, eamonn@cs.ucr.edu

Abstract— The *all-pairs-similarity-search* (or *similarity join*) problem has been extensively studied for text and a handful of other datatypes. However, surprisingly little progress has been made on similarity joins for time series subsequences. The lack of progress probably stems from the daunting nature of the problem. For even modest sized datasets the obvious nested-loop algorithm can take months, and the typical speed-up techniques in this domain (i.e., indexing, lower-bounding, triangular-inequality pruning and early abandoning) at best produce one or two orders of magnitude speedup. In this work we introduce a novel scalable algorithm for time series subsequence all-pairs-similarity-search. For exceptionally large datasets, the algorithm can be trivially cast as an anytime algorithm and produce high-quality approximate solutions in reasonable time. The exact similarity join algorithm computes the answer to the *time series motif* and *time series discord* problem as a side-effect, and our algorithm incidentally provides the fastest known algorithm for both these extensively-studied problems. We demonstrate the utility of our ideas for many time series data mining problems, including motif discovery, novelty discovery, shapelet discovery, semantic segmentation, density estimation, and contrast set mining.

Keywords—Time Series; Similarity Joins; Motif Discovery

I. INTRODUCTION

The *all-pairs-similarity-search* (also known as *similarity join*) problem comes in several variants. The basic task is this: *Given a collection of data objects, retrieve the nearest neighbor for each object*. In the text domain the algorithm has applications in a host of problems, including community discovery, duplicate detection, collaborative filtering, clustering, and query refinement [1]. While virtually all text processing algorithms have analogues in time series data mining, there has been surprisingly little progress on Time Series subsequences All-Pairs-Similarity-Search (TSAPSS).

We believe that this lack of progress stems not from a lack of interest in this useful primitive, but from the daunting nature of the problem. Consider the following example that reflects the needs of an industrial collaborator. A boiler at a chemical refinery reports pressure once a minute. After a year, we have a time series of length 525,600. A plant manager may wish to do a similarity self-join on this data with week-long subsequences (10,080) to discover operating regimes (summer vs. winter or light distillate vs. heavy distillate etc.) The obvious nested loop algorithm requires 132,880,692,960 Euclidean distance computations. If we assume each one takes 0.0001 seconds, then the join will take 153.8 days. The core contribution of this work is to show that we can reduce this time to 6.3 hours, using an off-the-shelf desktop computer. Moreover, we show that this join can be computed and/or updated incrementally. Thus we could maintain this join essentially forever on a standard desktop, even if the data arrival frequency was much faster than once a minute.

Our algorithm uses an ultra-fast similarity search algorithm under z-normalized Euclidean distance as a subroutine, exploiting the overlap between subsequences using the classic Fast Fourier Transform (FFT) algorithm.

Our method has the following advantages/features:

- It is *exact*, providing no false positives or false dismissals.
- It is simple and parameter-free. In contrast, the more general metric space APSS algorithms require building and tuning spatial access methods and/or hash functions.
- Our algorithm requires an inconsequential space overhead, just O(n) with a small constant factor.
- While our *exact* algorithm is extremely scalable, for extremely large datasets we can compute the results in an anytime fashion, allowing ultra-fast *approximate* solutions.
- Having computed the similarity join for a dataset, we can incrementally update it very efficiently. In many domains this means we can effectively maintain exact joins on streaming data forever.
- Our method provides *full* joins, eliminating the need to specify a similarity *threshold*, which as we will show, is a near impossible task in this domain.
- Our algorithm is embarrassingly parallelizable, both on multicore processors and in distributed systems.

Given all these features, our algorithm has implications for many time series data mining tasks [5][18][28].

The rest of the paper is organized as follows. Section II reviews related work and introduces the necessary background materials and definitions. In Section III we introduce our algorithm and its anytime and incremental variants. Section IV sees a detailed empirical evaluation of our algorithm and shows its implications for many data mining tasks. Finally, in Section V we offer conclusions and directions for future work.

II. RELATED WORK AND BACKGROUND

The basic variant of *similarity join* problem we are interested in is as follows: Given a collection of data objects, retrieve the nearest neighbor for *every* object.

Other common variants include retrieving the top-K nearest neighbors or the nearest neighbor for each object if that neighbor is within a user-supplied threshold, τ. (Such variations are trivial generalizations of our proposed algorithm, so we omit them from further discussion). The latter variant results in a much easier problem, provided that the threshold is small. For example, [1] notes that virtually all research efforts "*exploit a similarity threshold more aggressively in order to limit the set*

of candidate pairs that are considered.. [or] ...*to reduce the amount of information indexed in the first place*."

This critical dependence on τ is a major issue for text joins, as it is known that "*join size can change dramatically depending on the input similarity threshold*" [10]. However, this issue is even more critical for time series for two reasons. First, unlike *similarity* (which is bounded between zero and one), the Euclidean distance is effectively unbounded, and generally not intuitive. For example, if two heartbeats have a Euclidean distance of 17.1, are they similar? Even for a domain expert that knows the sampling rate and the noise level of the data, this is not obvious. Second, a single threshold can produce radically different output sizes, even for datasets that are very similar. Consider Figure 1 which shows the output size vs. threshold setting for the first and second halves of a ten-day period monitoring data center chillers [21]. For the first five days a threshold of 0.6 would return zero items, but for the second five days the same setting would return 108 items. This shows the difficulty in selecting an appropriate threshold. Our solution is to have *no* threshold, and do a *full* join.
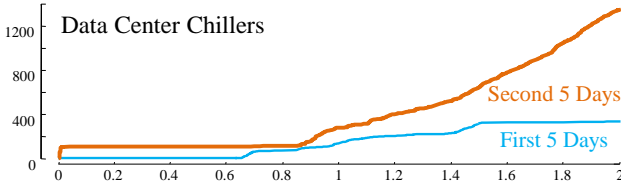

Figure 1. Output size vs. threshold for data center chillers [21]. Values beyond 2.0 are truncated for clarity (but archived at [24]).

A handful of efforts have considered joins on time series, achieving speedup by (in addition to the use of MapReduce) converting the data to lower-dimensional representations such as PAA [11] or SAX [12] and exploiting lower bounds and/or Locality Sensitive Hashing (LSH) to prune some calculations. However, the methods are very complex, with many (10-plus) parameters to adjust. As [11] acknowledges with admirable candor, "*Reasoning about the optimal settings is not trivial*." In contrast, our proposed algorithm has zero parameters to set.

A very recent research effort [28] has tackled the scalability issue by converting the real-valued time series into discrete "fingerprints" before using a LSH approach, much like the text retrieval community [1]. They produced impressive speedup, but they also experienced false negatives. Moreover, the approach has several parameters that need to be set; for example, they need to set the threshold to a very precise 0.818.

As we shall show, our algorithm allows both *anytime* and *incremental* (i.e. streaming) versions. While a streaming join algorithm for *text* was recently introduced [15], we are not aware of any such algorithms for time series data or general metric spaces. More generally, there is a large amount of literature on joins for text processing [1]. Such work is interesting, but of little utility given our constraints, data type and problem setting. We require *full* joins, not *threshold* joins, and we are unwilling to allow the possibility of false negatives.

A. Definitions and Notation

We begin by defining the data type of interest, *time series*:

Definition 1: A *time series T* is a sequence of real-valued numbers t_i: $T = t_1, t_2, ..., t_n$ where n is the length of T.

We are not interested in the *global* properties of time series, but in the similarity between local *subsequences*:

Definition 2: A *subsequence* $T_{i,m}$ of a T is a continuous subset of the values from T of length m starting from position i. $T_{i,m} = t_i, t_{i+1},..., t_{i+m-1}$, where $1 \leq i \leq n-m+1$.

We can take any subsequence from a time series and compute its distance to *all* sequences. We call an ordered vector of such distances a *distance profile*:

Definition 3: A *distance profile D* is a vector of the Euclidean distances between a given query and each subsequence in an all-subsequences set (see **Definition 4**).

Note that we are assuming that the distance is measured using the Euclidean distance between the z-normalized subsequences [8]. The distance profile can be considered a *meta* time series that annotates the time series T that was used to generate it. The first three definitions are illustrated in Figure 2.
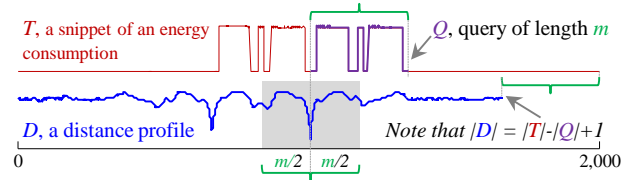

Figure 2. A subsequence Q extracted from a time series T is used as a query to every subsequence in T. The vector of all distances is a distance profile.

Note that if the query and all-subsequences set belong to the same time series, the distance profile must be zero at the location of the query, and close to zero just before and just after. Such matches are called *trivial matches* in the literature [18], and are avoided by ignoring an exclusion zone (shown as a gray region) of $m/2$ before and after the location of the query.

We are interested in similarity join of all subsequences of a given time series. We define an *all-subsequences set* of a given time series as a set that contains all possible subsequences from the time series. The notion of all-subsequences set is purely for notational purposes. In our implementation, we do not actually *extract* the subsequences in this form as it would require significant time and space overhead.

Definition 4: An *all-subsequences set A* of a time series T is an ordered set of all possible subsequences of T obtained by sliding a window of length m across T: $A = \{T_{1,m}, T_{2,m},..., T_{n-m+1,m}\}$, where m is a user-defined subsequence length. We use $A[i]$ to denote $T_{i,m}$.

We are interested in the nearest neighbor (i.e., *1NN*) relation between subsequences; therefore, we define a *1NN-join function* which indicates the nearest neighbor relation between the two input subsequences.

Definition 5: *1NN-join function*: given two all-subsequences sets A and B and two subsequences $A[i]$ and $B[j]$, a *1NN-join function* θ_{1nn} ($A[i]$, $B[j]$) is a Boolean function which returns "true" only if $B[j]$ is the nearest neighbor of $A[i]$ in the set B.

With the defined join function, a *similarity join set* can be generated by applying the similarity join operator on two input all-subsequences sets.

Definition 6: *Similarity join set*: given all-subsequences sets A and B, a *similarity join set* J_{AB} of A and B is a set containing pairs of each subsequence in A with its nearest neighbor in B: $J_{AB} = \{\langle A[i], B[j] \rangle | \theta_{1nn} (A[i], B[j])\}$. We denote this formally as $J_{AB} = A \bowtie_{\theta_{1nn}} B$.

We measure the Euclidean distance between each pair within a similarity join set and store the resultants into an ordered vector. We call the result vector the *matrix profile*.

Definition 7: A *matrix profile* (or just *profile*) P_{AB} is a vector of the Euclidean distances between each pair in J_{AB}.

We call this vector the <u>matrix profile</u> because one (inefficient) way to compute it would be to compute the full distance matrix of all the subsequences in one time series with all the subsequence in another time series and extract the smallest value in each row (the smallest *non-diagonal* value for the self-join case). In Figure 3 we show the matrix profile of our running example.
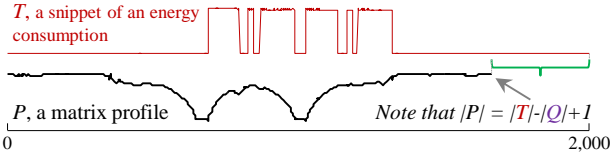

Figure 3. A time series *T*, and its self-join matrix profile *P*.

Like the distance profile, the matrix profile can be considered a *meta* time series annotating the time series *T* if the matrix profile is generated by joining *T* with itself. The profile has a host of interesting and exploitable properties. For example, the highest point on the profile corresponds to the time series discord [5], the (tied) lowest points correspond to the locations of the best time series motif pair [18], and the variance can be seen as a measure of the *T*'s complexity. Moreover, the histogram of the values in the matrix profile is the *exact* answer to the time series density estimation [4].

We name this special case of the similarity join set (**Definition 6**) as *self-similarity join set*, and the corresponding profile as *self-similarity join profile*.

Definition 8: A *self-similarity join set* J_{AA} is a result of similarity join of the set *A* with itself**.** We denote this formally as $J_{AA} = A \bowtie_{\theta 1nn} A$. We denote the corresponding matrix profile or *self-similarity join profile* as P_{AA}.

Note that we exclude trivial matches when self-similarity join is performed, i.e., if $A[i]$ and $A[j]$ are subsequences from the same all-subsequences set *A*, θ_{1nn} ($A[i]$, $B[j]$) is "false" when $A[i]$ and $A[j]$ are a trivially matched pair.

The ith element in the matrix profile tells us the Euclidean distance to the nearest neighbor of the subsequence of *T*, starting at *i*. However, it does not tell us *where* that neighbor is located. This information is recorded in *matrix profile index*.

Definition 9: A *matrix profile index* I_{AB} of a similarity join set J_{AB} is a vector of integers where $I_{AB}[i] = j$ if $\{A[i], B[j]\} \in J_{AB}$.

By storing the neighboring information this way, we can efficiently retrieve the nearest neighbor of $A[i]$ by accessing the ith element in the matrix profile index.

Note that the function which computes the similarity join set of two input time series is not symmetric; therefore, $J_{AB} \neq J_{BA}$, $P_{AB} \neq P_{BA}$, and $I_{AB} \neq I_{BA}$.

For ease of presentation, we have confined this work to the single dimensional case; however, nothing intrinsically precludes generalizations to multidimensional data.

Summary of the Previous Section

The previous section was rather dense, so before moving on we summarize the main takeaway points. We can create two meta time series, the *matrix profile* and the *matrix profile index*, to annotate a time series T_A with the distance and location of all its subsequences nearest neighbors in itself or another time series T_B. These two data objects explicitly or implicitly contain the answers to many time series data mining tasks. However, they appear to be too expensive to compute to be practical. In the next section we will show an algorithm that *can* compute these efficiently.

III. ALGORITHMS

We are finally in a position to explain our algorithms. We begin by stating the fundamental intuition, which stems from the relationship between distance profiles and the matrix profile. As Figure 2 and Figure 3 visually suggest, all distance profiles (excluding the trivial match region) are upper bound approximations to the matrix profile. More critically, if we compute *all* the distance profiles, and take the minimum value at each location, the result *is* the matrix profile!

This tells us that if we have a fast way to compute the distance profiles, then we also have a fast way to compute the matrix profile. As we shall show in the next section, we have an ultra-fast way to compute the distance profiles.

A. Mueen's ultra-fast Algorithm for Similarity Search (MASS)

We begin by introducing a novel Euclidean distance similarity search algorithm for time series data. The algorithm does not just find the nearest neighbor to a query and return its distance; it returns the distance to *every* subsequence. In particular, it computes the distance profile, as shown in Figure 2. The algorithm requires just O($n$log$n$) time by exploiting the FFT to calculate the dot products between the query and all subsequences of the time series.

We need to carefully qualify the claim of "*ultra-fast*". There are dozens of algorithms for time series similarity search that utilize index structures to efficiently locate neighbors [8]. While such algorithms can be faster in the *best case*, all these algorithms degenerate to brute force search in the worst case[1] (actually, much worse than brute force search due to the overhead of the index). Likewise, there are index-free methods that achieve speed-up using various early abandoning tricks [22], but they too degrade to brute force search in the worst case. In contrast, the performance of the algorithms outlined in TABLE I and TABLE II is completely *independent* of the data.

TABLE I.   CALCULATION OF SLIDING DOT PRODUCTS

Procedure SlidingDotProduct(Q, T)
Input: A query Q, and a user provided time series T
Output: The dot product between Q and all subsequences in T
1 $n \leftarrow$ Length(T), m $\leftarrow$ Length(Q)
2 $T_a \leftarrow$ Append T with n zeros
3 $Q_r \leftarrow$ Reverse(Q)
4 $Q_{ra} \leftarrow$ Append Q_r with $2n$-m zeros
5 $Q_{raf} \leftarrow$ FFT(Q_{ra}), $T_{af} \leftarrow$ FFT(T_a)
6 $QT \leftarrow$ InverseFFT(ElementwiseMultiplication(Q_{raf}, T_{af}))
7 **return** QT

<hr>

[1] There are *many* such worse case scenarios, including high levels of noise blurring the distinction between closest and furthest neighbors, and rendering triangular-inequality pruning and early abandoning worthless.

Line 1 determines the length of both the time series T and the query Q. In line 2, we use that information to append T with an equal number of zeros. In line 3, we obtain the mirror image of the original query. This reversing ensures that a convolution (i.e. "crisscrossed" multiplication) essentially produces in-order alignment. Because we require both vectors to be the same length, in line 4 we append enough zeros to the (now reversed) query so that, like T_a, it is also of length $2n$. In line 5, the algorithm calculates Fourier transforms of the appended-reversed query (Q_{ra}) and the appended time series T_a. Note that we use FFT algorithm which is an O($n\log n$) algorithm. The Q_{raf} and the T_{af} produced in line 5 are vectors of complex numbers representing frequency components of the two time series. The algorithm calculates the element-wise multiplication of the two complex vectors and performs inverse FFT on the product. Lines 5-6 are the classic convolution operation on two vectors [7]. Figure 4 shows a toy example of the sliding dot product function in work. The algorithm time complexity does *not* depend on the length of the query (m).
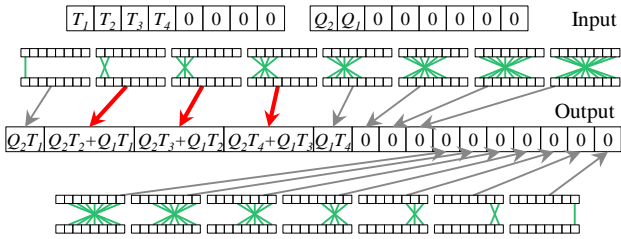

Figure 4. A toy example of convolution operation being used to calculate sliding dot products for time series data. Note the *reverse* and *append* operation on T and q in the input. Fifteen dot products are calculated for every slide. The cells $m = 2$ to $n = 4$ from left (red/bold arrows) contain valid products. TABLE II takes this subroutine and uses it to create a distance profile (see **Definition 3**).

In line 1 of TABLE II, we invoke the dot products code outlined in TABLE I. The formula to calculate the z-normalized Euclidean distance $D[i]$ between two time series subsequence Q and $T_{i,m}$ using their dot product, $QT[i]$ is (see [24] for derivation):

$$D[i] = \sqrt{2m\left(1 - \frac{QT[i] - m\mu_Q M_T[i]}{m\sigma_Q \Sigma_T[i]}\right)}$$

where m is the subsequence length, μ_Q is the mean of Q, $M_T[i]$ is the mean of $T_{i,m}$, σ_Q is the standard deviation of Q, and $\Sigma_T[i]$ is the standard deviation of $T_{i,m}$. Normally, it takes O(m) time to calculate the mean and standard deviation for every subsequence of a long time series. However, here we exploit a technique noted in [22] in a different context. We cache cumulative sums of the values and square of the values in the time series. At any stage the two cumulative sum vectors are sufficient to calculate the mean and the standard deviation of any subsequence of any length. See [14] for an elaborate description and variations of MASS.

TABLE II. MUEEN'S ALGORITHM FOR SIMILARITY SEARCH (MASS)

Procedure MASS(Q, T)
Input: A query Q, *and* a user provided time series T
Output: A distance profile D of the query Q
1   $QT \leftarrow$ SlidingDotProducts(Q, T)
2   $\mu_Q, \sigma_Q, M_T, \Sigma_T \leftarrow$ ComputeMeanStd(Q, T)   // see [22]
3   $D \leftarrow$ CalculateDistanceProfile(Q, T, QT, μ_Q, σ_Q, M_T, Σ_T)
4   **return** D

Unlike the dozens of time series KNN search algorithms in the literature [8], this algorithm calculates the distance to *every*

subsequence, i.e. the *distance profile* of time series T. Alternatively, in join nomenclature, the algorithm produces one full row of the all-pair similarity matrix. Thus, as we show in the next section, our join algorithm is simply a loop that computes each full row of the all-pair similarity matrix and updates the current "*best-so-far*" matrix profile when needed.

B. The STAMP Algorithm

We call our join algorithm STAMP, **S**calable **T**ime series **A**nytime **M**atrix **P**rofile. The algorithm is outlined in TABLE III. In line 1, we extract the length of T_B. In line 2, we allocate memory and initial matrix profile P_{AB} and matrix profile index I_{AB}. From lines 3 to line 6, we calculate the distance profiles D using each subsequence $B[idx]$ in the time series T_B and the time series T_A. Then, we perform pairwise minimum for each element in D with the paired element in P_{AB} (i.e., min($D[i]$, $P_{AB}[i]$) for $i = 0$ to length(D) - 1.) We also update $I_{AB}[i]$ with idx when $D[i] \leq P_{AB}[i]$ as we perform the pairwise minimum operation. Finally, we return the result P_{AB} and I_{AB} in line 7.

Note that the algorithm presented in TABLE III computes the matrix profile for the general similarity join. To modify the current algorithm to compute the self-similarity join matrix profile of a time series T_A, we simply replace T_B in line 1 with T_A, replace B with A in line 4, and ignore trivial match in D when performing *ElementWiseMin* in line 5.

TABLE III. THE STAMP ALGORITHM

Procedure STAMP(T_A, T_B, m)
Input: Two user provided time series, T_A and T_B, interested subsequence length m
Output: A matrix profile P_{AB} and associated matrix profile index I_{AB} of T_A join T_B, $J_{AB} = A \bowtie_{\theta 1nn} B$
1   $n_B \leftarrow$ Length(T_B)
2   $P_{AB} \leftarrow$ infs, $I_{AB} \leftarrow$ zeros, $idxes \leftarrow 1:n_B-m+1$
3   **for each** idx **in** $idxes$ // In any order, but random for anytime algorithm
4       $D \leftarrow$ **MASS**($B[idx]$, T_A)
5   P_{AB}, $I_{AB} \leftarrow$ ElementWiseMin(P_{AB}, I_{AB}, D, idx)
6   **end for**
7   **return** P_{AB}, I_{AB}

To parallelize the STAMP algorithm for multicore machines, we simply distribute the indexes to secondary process run in each core, and the secondary processes use the indexes they received to update their own P_{AB} and I_{AB}. Once the main process returns from all secondary processes, we use *ElementWiseMin* to merge the received P_{AB} and I_{AB}.

C. An Anytime Algorithm for TSAPSS

While the exact algorithm introduced in the previous section is extremely scalable, there will always be datasets for which time needed for an *exact* solution is untenable. We can mitigate this by computing the results in an *anytime* fashion, allowing fast *approximate* solutions [30]. To add the anytime nature to the STAMP algorithm, we simply ensure a randomized search order in line 2 of TABLE III.

We can compute a (post-hoc) measurement of the quality of an anytime solution by measuring the Root-Mean-Squared-Error (RMSE) between the true matrix profile and the current best-so-far matrix profile. As Figure 5 suggests, with an experiment on random walk data, the algorithm converges very quickly.
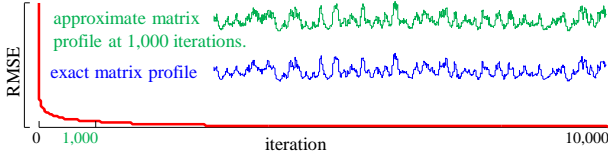
Figure 5. *main)* The decrease in RMSE as the STAMP algorithm updates matrix profile with the distance profile calculated at each iteration. *inset)* The approximate matrix profile at the 10% mark is visually indistinguishable from the final matrix profile.

Zilberstein [30] gives a number of desirable properties of anytime algorithms, including *Low Overhead*, *Interruptibility*, *Monotonicity*, *Recognizable Quality, Diminishing Returns* and *Preemptability* (these properties are mostly obvious from their names, but full definitions are at [30]).

Because each subsequence's distance profile is bounded below by the exact matrix profile, updating an approximate matrix profile with a distance profile with pairwise minimum operation either drives the approximate solution closer the exact solution or retains the current approximate solution. Thus, we have guaranteed *Monotonicity*. From Figure 5, the approximate matrix profile converges to the exact matrix profile superlinearly; therefore, we have strong *Diminishing Returns*. We can easily achieve *Interruptibility* and *Preemptability* by simply inserting a few lines of code between lines 5 and 6 of TABLE III that read:

5^{new}	**if** CheckForUserInterrupt = **TRUE**
6^{new}	Report({P_{AB}, I_{AB}}, 'Here is an approximate answer.')
7^{new}	**if** GetUserChoice = 'further refine', **CONTINUE**, **else BREAK**

The space and time overhead for the anytime property is effectively zero, thus we have *Low Overhead*. This leaves only the property of *Recognizable Quality*. Here we must resort to a probabilistic argument. The convergence curve shown in Figure 5 is very typical, so we could use past convergence curves to predict the quality of solution when interrupted on similar data.

D. Time and Space Complexity

The overall complexity of the proposed algorithm is $O(n^2 \log n)$ where n is the length of the time series. However, our experiments (see Section 4.1) empirically suggest that the runtime of STAMP's growth rate is roughly $O(n^2)$ instead of $O(n^2 \log n)$. One possible explanation for this is that the $n \log n$ factor comes from the FFT subroutine. Because FFT is so important in many applications, it is *extraordinarily* well optimized. Thus, the empirical runtime is very close to linear.

In contrast to the above, the brute force nested loop approach has a time complexity of $O(n^2 m)$. Recall the industrial example in the introduction section. We have $m = 10,080$, but $\log(n) = 5.7$, so we would expect our approach to be about 1,768 times faster. In fact, we are empirically *even faster*. The complexity analysis downplays the details of important constant factors. The nested loop algorithm must also z-normalize the subsequences. This either requires $O(nm)$ time, but with an untenable $O(nm)$ space overhead, or an $O(n^2 m)$ time overhead. And recall that this is *before* a single Euclidean distance calculation is performed.

Finally, we mention one quirk of our algorithm which we inherit from using the highly optimized FFT subroutine. Our algorithm is fastest when n is an integer power of two, slower for non-power of two but composite numbers, and slowest when n is prime. The difference (for otherwise similar values of

n) can approach a factor of 1.6x. Thus, where possible, it is worth contriving the best case by truncation or zero-padding to the nearest power of two.

E. Incrementally Maintaining TSAPSS

Up to this point we have discussed the *batch version* of TSAPSS. By *batch*, we mean that the STAMP algorithm needs to see the *entire* time series T_A and T_B (or just T_A if we are calculating the self-similarity join matrix profile) before creating the matrix profile. However, it would be advantageous if we could build the matrix profile incrementally. Given that we have performed a batch construction of matrix profile, if new data arrives, it would clearly be preferable to incrementally *adjust* the current profile, rather than start from scratch.

Because the matrix profile solves both the times series motif and the time series discord problems, an incremental version of STAMP would automatically provide the first incremental versions of both algorithms. We call such an algorithm the STAMP*I* (STAMP *I*ncremental) algorithm.

We will demonstrate our ability to incrementally maintain the matrix profile in this section. For simplicity and brevity TABLE IV only shows the algorithm to maintain the self-similarity join. The generalizations are obvious.

TABLE IV.   THE STAMP*I* ALGORITHM

Procedure STAMP*I*(T_A, t, P_{AA}, I_{AA})
Input: The original time series T_A, a new data point t following T_A, the matrix profile P_{AA} and its associated matrix profile index I_{AA} of T_A.
Output: The incrementally updated matrix profile $P_{AA,new}$ and its matrix profile index $I_{AA,new}$ of the current time series $T_{A,new}$= T_A, t.
1    $T_{A,new}$ = [T_A, t]
2   S ← last subsequence in $T_{A,new}$, idx ← index of S in $T_{A,new}$
3   D ← **MASS** (S, T_A)
4   P_{AA}, I_{AA} ← ElementWiseMin(P_{AA}, I_{AA}, D, idx)
5    $p_{AA,last}$, $i_{AA,last}$ ← FindMin(D)
6    $P_{AA,new}$ ← [P_{AA}, $p_{AA,last}$], $I_{AA,new}$ ← [I_{AA}, $i_{AA,last}$]
7   **return** $P_{AA,new}$, $I_{AA,new}$

For clarity, we denote the updated time series as $T_{A,new}$, the updated matrix profile as $P_{AA,new}$ and the associated matrix profile index as $I_{AA,new}$. As each additional data point t arrives, the size of the time series T_A increases by one, and a new subsequence S is generated at the end of $T_{A,new}$. In line 3 we obtain the distance profile of S with regard to T_A. Then, as in the original STAMP algorithm, in line 4 we perform a pairwise comparison between every element in D with the corresponding element in P_{AA} to see if the corresponding element in P_{AA} needs to be updated. In line 5, we find the nearest neighbor of S and the associated index by evaluating the minimum value of D. Finally, in line 6, we obtain the new matrix profile and associated matrix profile index by concatenating the results in line 4 and line 5.

The time complexity of the STAMP*I* algorithm is $O(n \log n)$ where n is the length of size of the current time series T_A. Note that as we maintain the profile, each incremental call of STAMP*I* requires invoking the FFT subroutine with a slightly longer time series (*n becomes $n+1$*). Thus it gets very slightly slower at each time step. Therefore, the best way to measure the performance is to ask for the Maximum Time Horizon (MTH), in essence the answer to this question: "*Given this arrival rate, how long can we maintain the profile before we can no longer update fast enough?*"

Note that the subsequence length m is not considered in the MTH evaluation. Recall that overall time complexity of the

algorithm is determined by the efficiency of the FFT subroutine, which is independent of m. We have computed the MTH for two common scenarios of interest to the community.

- **House Electrical Demand** [19]: This dataset is updated every eight seconds. By iteratively calling the STAMP*I* algorithm, we can maintain the profile for 5.8 years.
- **Oil Refinery**: Most telemetry in oil refineries is sampled at once a minute [26]. The relatively low sampling rate reflects the "inertia" of massive boilers/condensers. Even if we maintain the profile for 40 years, the update time is only around 6.78 seconds. Moreover, the raw data, matrix profile and index would only require 0.5 gigabytes of main memory. Thus the MTH here is forty plus years. Given projected improvements in hardware, this effectively means we can maintain the matrix profile forever.

As impressive as these numbers are, they are actually quite pessimistic. For simplicity we assume that *every* value in the matrix profile index will be updated at each time step. However, empirically, much less than 0.1% of them need to be updated. If it is possible to *prove* an upper bound on the number of changes to the matrix profile index per update, then we could greatly extend the MTH or handle much faster sampling rates. We leave such considerations for future work.

IV. Experimental Evaluation

We begin by stating our experimental philosophy. We have designed all experiments such that they are *easily* reproducible. To this end, we have built a webpage [24] which contains all datasets and code used in this work, together with spreadsheets which contain the raw numbers and some supporting videos.

Given page limits, and the utility of our algorithm for a host of existing (and several new) data mining tasks, we have chosen to conduct exceptionally broad but shallow experiments. We *have* conducted such deep detailed experiments and placed them at [24]. Unless otherwise stated we measure wall clock time on an Intel i7@4GHz with 4 cores.

A. Scalability of Profile-Based Self-Join

Because the time performance of STAMP is independent of the data quality or any user inputs (there are none except the choice of m, which does not affect the speed), our scalability experiments are unusually brief. In TABLE V we show the time required for a self-join with m fixed to 256, for increasingly long time series.

TABLE V.   Time required for a self-join with $M = 256$, varying N

Value of n	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
Time Required	15.1 min	70.4 min	5.4 hours	24.4 hours	4.2 days

In TABLE VI, we show the time required for a self-join with n fixed to 2^{17}, for increasing long m. Again recall that unlike virtually all other time series data mining algorithms in the literature whose performance degrades for longer subsequences [8][18], the running time of STAMP does not depend on m.

TABLE VI.   Time required for a self-join with $N = 2^{17}$, varying M

Value of m	64	128	256	512	1,024
Time Required	15.1 min	15.1 min	15.1 min	15.0 min	14.5 min

Finally, we further exploit the simple parallelizability of the algorithm by using four 16-core virtual machines on Microsoft Azure to redo the two-million join ($n = 2^{21}$ and $m = 256$)

experiment. By scaling up the computational power, we have reduced the running time from 4.2 days to just 14.1 hours. This use of cloud computing required writing just few dozen lines of simple additional code [24].

It is difficult to find good baselines to compare to. We believe STAMP is the only algorithm that does *full*, *exact* joins on time series subsequences. Most algorithms do only *threshold* joins and/or do *approximate* joins, and are not specialized for time series subsequences. However, we searched for the best baselines and made the following concessions to the rival algorithms to allow comparisons.

- $TSFR_{DAA}$ [11]: While STAMP returns *all* nearest neighbors, we adjust the threshold (the *selectively*) of $TSFR_{DAA}$ such that only needs to return the top 1% nearest neighbors, a much easier task.
- $HDSJ_{I-SAX}$ [12]: This method allows false negatives; we ignore this. It needs time to build indexes; we do not count this time, and as above, we allow it to return the only the top 1% nearest neighbors (as before, not the 100% the STAMP returns, and therefore a much easier task.).
- Optimized Nested Loop (ONL): Here we use a nested-loop join optimized in the following manner. We use a state-of-the-art DFT indexing technique to do the search in the inner loop, and we do not count the time needed to build the indexes [8]. We carefully adjust parameters for best performance.

We consider the first 2^{18} data points of the ECG dataset used in [22], with a query length of 256, about one heartbeat; TABLE VII shows the results.

TABLE VII.   Time for a self-join with $M = 256$, varying algorithms

Algorithm	$TSFR_{DAA}$	$HDSJ_{I-SAX}$	ONL	STAMP
Time Required	51.7 hours	19.6 min	28.1 hours	1.17 hours

As these results show, even if we ignore the limitations of the baseline methods, STAMP is still significantly faster.

B. Profile-Based Self-Join

A recent paper notes that many fundamental problems in seismology can be solved by joining seismometer telemetry [28], including the discovery of foreshocks, aftershocks, triggered earthquakes, volcanic activity and induced seismicity. However, the paper notes a join with a query length of 200 on a data stream of length 604,781 requires 9.5 days. Their solution, a clever transformation of the data to allow LSH based techniques, does achieve significant speedup, but at the cost of false negatives and the need for significant parameter tuning. The authors kindly shared their data, and, as we hint at in Figure 6, confirmed that STAMP does not have false negatives.
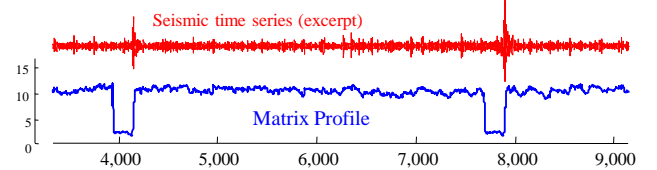

Figure 6. *top*) An excerpt of a seismic time series aligned with its matrix profile (*bottom*). The ground truth provided by the authors of [28] requires that the events occurring at time 4,050 and 7,800 match.

We repeated the $n = 604,781$, $m = 200$ experiment and found it took just 8.9 hours to finish. As impressive as this is, in the next section we show that we can do even better.

C. The Utility of Anytime STAMP

The seismology dataset offers an excellent opportunity to demonstrate the utility of the anytime version of our algorithm. The authors of [28] revealed in their long-term ambition of mining even larger datasets [3]. In Figure 7 we repeated the experiment with the snippet shown in Figure 6, this time reporting the *best-so-far* matrix profile reported by the algorithm at various milestones. Even with just 0.25% of the distances computed (that is to say, 400 times faster) the correct answer has emerged. Thus, we can provide the correct answers to the seismologists in just minutes, rather than the 9.5 days.
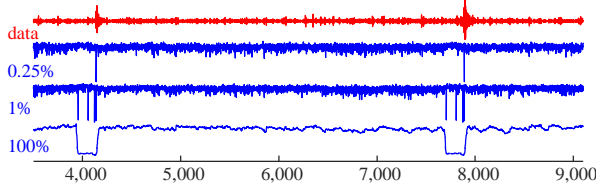
Figure 7. *top*) An excerpt of the seismic data that is also shown in Figure 6. *top-to-bottom*) The approximations of the matrix profile for increasing interrupt times. By the time we have computed just 0.25% of the calculations required, the minimum of the matrix profile points to the ground truth.

To show the generality of this anytime feature of STAMP, we consider a *very* different dataset. As shown in Figure 8.*inset)*, it is possible to convert DNA to a time series [22]. We converted the Y-chromosome of the Chimpanzee this way. The resulting time series is little over one-million in length. We performed a self-join with *m* = 60,000. Figure 8.*bottom* shows the best motif is so well conserved (ignoring the first 20%), that it must correspond to a recent (in evolutionary time) gene duplication event. In fact, in a subsequent analysis we discovered that "*much of the Y (Chimp chromosome) consists of lengthy, highly similar repeat units, or 'amplicons'*" [9].
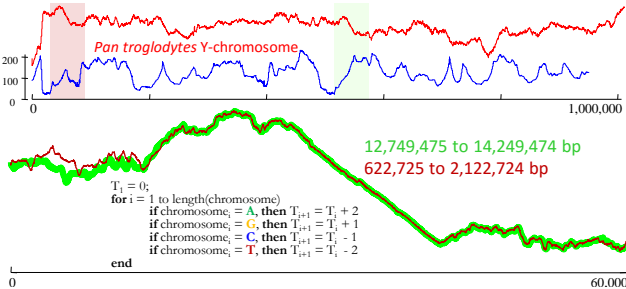

Figure 8. *top*) The Y-chromosome of the Chimp in time series space with its matrix profile. *bottom*) A zoom-in of the top motif discovered using anytime STAMP, we believe it to be an *amplicon* [9].

This demanding join would take just over a day of CPU time (see TABLE V). However, using anytime STAMP we have the result shown above after doing just 0.021% of the computations, in about 18 seconds. At [24] we have videos that show the rapid convergence of the anytime variant of STAMP.

D. Profile-Based Similarity Join Set

In this section we show two uses of similarity join set. The first use is more familiar to APSS users, quantifying what is *similar* between two time series. The second example, quantifying what is *different* between two time series, is novel and can only be supported by threshold-free algorithms that report the nearest neighbor for *all* objects.

Time Series Set Similarity

Given two (or more) time series collected under different conditions or treatments, a data analyst may wish to know what

patterns (if any) are *conserved* between the two time series. To demonstrate the utility of automating this, we consider a simple but intuitive example. Figure 9 shows the raw audio of two popular songs converted to Mel Frequency Cepstral Coefficients (MFCCs). Specifically, the songs used in this example are "Under Pressure" by Queen and David Bowie and "Ice Ice Baby" by the American rapper Vanilla Ice. Normally, there are 13 MFCCs; here, we consider just one for simplicity.
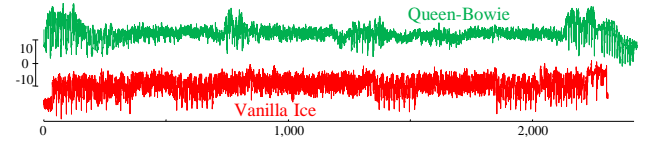

Figure 9. Two songs represented by just the 2nd MFCC at 100Hz. We recognize that it is difficult to see any structure in these times series; however, this difficulty is the *motivation* for this experiment.

Even for these two relatively short time series, visual inspection does not offer immediate answers. The problem here is compounded by the size reproduction, but is not significantly easier with large-scale [24] or interactive graphic tools. We ran J_{AB} (Queen-Bowie, Vanilla Ice) on these datasets with *m* = 500 (five seconds), the best match, corresponding the minimum value of the matrix profile P_{AB}, is shown in Figure 10.
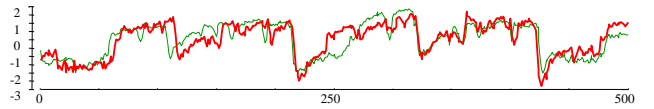

Figure 10. The result of J_{AB} (Queen-Bowie (red/bold), Vanilla-Ice (green/fine)) produces a strongly conserved five second region.

Readers may know the cause for this highly conserved subsequence. It corresponds to the famous baseline of "Under Pressure," which was sampled (plagiarized) by Vanilla Ice in his song. The join took 21.9 seconds. The ability to find conserved structure in apparently disparate time series could open many avenues of research in medicine and industry.

Time Series Difference

We introduce the Time Series Diff (TSD) operator, which informally asks "*What happens in time series T_A, that does not happen in time series T_B?*" Here T_A/T_B could be an ECG before a drug is administered/after a drug is administered, telemetry before a successful launch/before a catastrophic launch etc. The TSD is simply the subsequence referred to by the *maximum* value of the J_{AB} join's profile P_{AB}.

We begin with a simple intuitive example. The UK and US versions of the *Harry Potter* audiobook series are performed by different narrators, and have a handful of differences in the text. For example, the UK version contains:

Harry was passionate about Quidditch. He had played as Seeker on the Gryffindor house Quidditch team ever since his first year at Hogwarts and owned a Firebolt, one of the best racing brooms in the world...

But the corresponding USA version has:

Harry had been on the Gryffindor House Quidditch team ever since his first year at Hogwarts and owned one of the best racing brooms in the world, a Firebolt.

As shown in Figure 11, we can convert the audio corresponding to these snippets into MFCCs and invoke a J_{AB} join set to produce a matrix profile P_{AB} that represents the differences between them. As Figure 11.*left* shows, the low values of this profile correspond to identical spoken phrases (despite having two different narrators). However here we are

interested in the differences, the *maximum* value of the profile. As we can see in Figure 11.*right*, here the profile corresponds to a phrase unique to the USA edition.

Figure 11. The 2nd MFCC of snippets from the USA (pink/bold) and UK (green/fine) *Harry Potter* audiobooks. The J_{AB} join of the two longer sections in the main text produces mostly small values in the profile correspond to the same phrase (*left*), the largest value in the profile corresponds to a phrase unique to the USA edition (*right*).

The time required to do this is just 0.067 seconds, much faster than real time. While this demonstration is trivial, in [24] we show an example applied to ECG telemetry.

E. Profile-Based Motif Discovery

Since their introduction in 2003, time series motifs have become one of the most frequently used primitives in time series data mining, with applications in dozens of domains [2]. There are several proposed definitions for time series motifs, but in [18] it is argued that if you can solve the most basic variant, the closest (non-trivial) *pair* of subsequences, then all other variants only require some minor additional calculations. Note that the locations of the two (tying) minimum values of the matrix profile are *exactly* the locations of the 1st motif pair.

The fastest known *exact* algorithm for computing time series motifs is the MK algorithm [18]. Note, however, that this algorithm's time performance depends on the time series itself. In contrast, the Profile-Based Motif Discovery (PBMD) takes time independent of the data. To see this, we compared the two approaches on an electrocardiogram of length 65,536. In Figure 12.*left* we ask what happens as we search for longer and longer motifs. In Figure 12.*right* we ask what happens if the motif length is fixed to $m = 512$, but the data becomes increasing noisy.
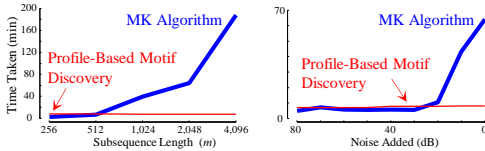

Figure 12. The time required to find the top-motif pairs in a time series of length 2^{16} for increasingly long motif lengths (*left*), and for a length fixed to 512, but in the face of increasing noise levels (*right*).

These results show that even in the best case for MK, PBMD is competitive, but as we have longer queries and/or noisier data, its advantage becomes unassailable. Moreover, PBMD inherits STAMP's anytime and incremental computability, and is easily parallelizable.

F. Profile-Based Discord Discovery

A *time series discord* is the subsequence that has the maximum distance to its nearest neighbor. While this is a simple definition, time series discords are known to be very competitive as novelty/anomaly detectors [5]. Note that as shown in Figure 13, the time series discord is encoded as the *maximum* value in a matrix profile.
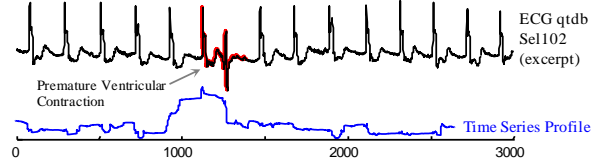

Figure 13. *top*) An excerpt from an ECG incorporating a premature ventricular contraction (red/bold). *bottom*) The time series profile peaks *exactly* at the beginning of the PVC.

The time taken to compute the discord is obviously just the time needed to compute the matrix profile (here, 0.9 seconds). There are a few dozen discord discovery algorithms in the literature. Some of them may be competitive in the *best* case, but just like motif-discovery algorithms they all degenerate to brute force search in the worst case, and none allow the anytime properties that we inherit from using STAMP.

G. Incrementally Maintaining Motifs and Discords

We have demonstrated the ability to detect time series motifs and discords using the matrix profile in the previous two sections. However, we assumed that the entire time series was available beforehand. Here we remove this assumption and show how STAMP*I* allows us to incrementally maintain time series motifs/discords in an online fashion. There are other attempts at one [20][2] or both [25] of these tasks, but they are all approximate and allow false dismissals.

In Section III.E, we introduced the STAMP*I* algorithm. The ability to incrementally maintain the matrix profile implies the ability to *exactly* maintain the time series motif [18] and/or time series discord [5] in streaming data. We simply need to keep track of the extreme values of the incrementally-growing matrix profile, report a new pair of motifs when a new *minimum* value is detected, and report a new discord when we see a new *maximum* value.

We demonstrate the utility of these ideas on the AMPds dataset [13]. Here the kitchen fridge and the heat pump are both plugged into a single metered power supply. For the first week, only the refrigerator is running. At the end of the week, the weather gets cold and the heat pump is turned on. The sampling rate is one sample/minute, and the subsequence length is 100. We apply the STAMP algorithm to the first three days of data, then invoke STAMP*I* to handle newly arriving data, report an *event* when we detect a new extreme value.

Our first event occurs at the 9,864[th] minute (6 day 20 hour 24 minute). As shown in Figure 14, a new minimum value is detected, which indicates a new time series motif. The just-arrived 100-minute-long pattern looks very similar to another pattern that occurred five hours earlier. While there is a lot of regularity in the fridge data in general, the exceptional similarity observed here suggested some underlying physical mechanism that caused such a perfectly-conserved pattern, perhaps a mechanical ice-making "subroutine."
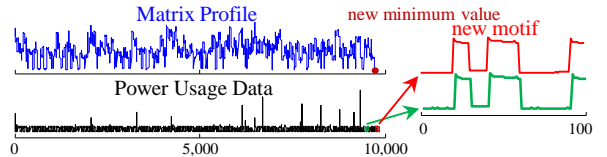

Figure 14. *top*) The matrix profile of the first 9,864 minutes of data. *bottom*) The minimum value of the matrix profile corresponds to a pair of time series motifs in the power usage data. *right*) The time series motif detected.

Our second event occurs at the 10,473[th] minute (7 day 6 hour 33 minute). As shown in Figure 15, a new maximum value

is detected, which indicates a new time series discord. The time series discord corresponds to the first occurrence of a heat pump pattern in the power usage data.
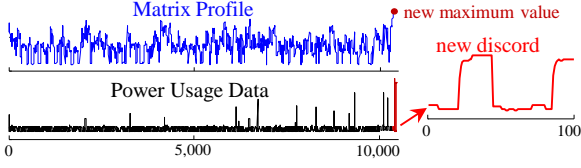

Figure 15. *top*) The matrix profile for the first 10,473 minutes. *bottom*) The maximum value of the matrix profile corresponds to a time series discord. *right*) The time series discord detected is the first heat pump pattern occurrence in the dataset.

The maximum time needed to process a single data point with STAMP*I* in this dataset is 0.005 seconds, which is less than 0.01% of the data sampling rate. Thus, on this dataset we could continue monitoring with the STAMP*I* algorithm for several decades before running out of time or memory.

H. Profile-Based Shapelet Discovery

Shapelets are time series subsequences which are in some sense the maximally representative of a class [23][27]. Shapelets can be used to classify time series (essentially, the nearest *shapelet* algorithm), offering the benefits of speed, intuitiveness and at least on some domains, significantly improved accuracy [23]. However, these advantages come at the cost of a very expensive training phase, with $O(n^2m^4)$ time complexity, where m is the length of the longest time series object in the dataset, and n is the number of objects in the training set. In order to mitigate this high time complexity, researchers have proposed various distance pruning techniques and candidate ranking approaches for both the admissible [27] and approximate [23] shapelet discovery. Nevertheless, scalability remains the bottleneck.

Because shapelets are essentially *supervised motifs*, and we have shown that STAMP can find motifs very quickly, it is natural to ask if STAMP has implications for shapelet discovery. While space limitations prohibit a detailed consideration of this question, we briefly sketch out and test this possibility as follows.

As shown in Figure 16, we can use matrix profile to heuristically "suggest" candidate shapelets. We consider two time series T_A (**green**/bold) and T_B (**pink**/light) with class 1 and class 0 being their corresponding class label, and we take J_{AB}, J_{AA}, J_{BA} and J_{BB}. Our claim is the differences in the heights of P_{AB}, P_{AA} (or P_{BA}, P_{BB}) are strong indicators of good candidate shapelets. The intuition is that if a discriminative pattern is present in say, class 1, but not in class 0, then we expect to see a "bump" in the P_{AB} (the intuition holds if the order is reversed). A *significant* difference (quantified by a threshold shown in dashed line) between the heights of P_{AA} and P_{AB} curves therefore indicates the occurrence of good candidate shapelets, patterns that only occur in *one* of the two classes.
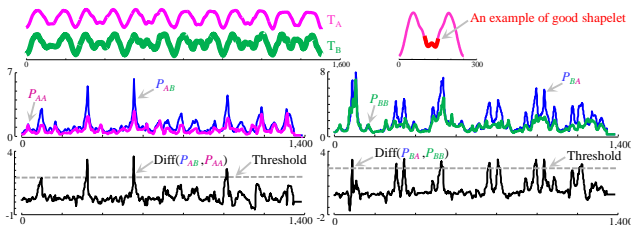

Figure 16. *top.left*) Two time series T_A and T_B formed by concatenating instances of class 1 and 0 respectively of *ArrowHead* [6]. *bottom*) The height

difference between P_{AB} (or P_{BA}) and P_{AA} (or P_{BB}) are suggestive of good shapelets. *top.right*) An example of good shapelet extracted from class 1.

The time taken to compute all four matrix profiles is 1.0 seconds and the time to further evaluate the two twelve candidates selected takes 2.7 seconds. On the same machine, the brute force shapelet classifier takes 4.2 minutes with 2,364 candidates. Note that, in this toy demonstration, the speedup is 68X, however, for larger datasets, the speedup is greater [24].

I. Profile-Based Semantic Segmentation

The goal of time series semantic segmentation is to partition a dataset containing multiple activities/regimes into atomic behaviors or conditions. For example, for human activity data the regimes might later be mapped to {*eating*, *working*, *commuting*,...}[29]. As this example suggests, most work in this area is highly domain-dependent. In contrast, here we show how the matrix profile index can be employed for domain agnostic time series segmentation.

The intuition of our approach is as follows. Within a single regime we might expect that most subsequences will have a nearest neighbor close by (in time). For example, consider the toy problem shown in Figure 17 which shows two obvious regimes. We would expect that the nearest neighbor to the first *run* gait cycle is the second or third or fourth run cycle, but it will almost certainly not be one of the *walk* cycles. In general, this tendency for nearest neighbor pointers *not* to cross the boundaries corresponding to regime changes may be sufficient to discover these boundaries, and of course, this is precisely the information that is encoded in the matrix profile index.

Thus, for each point we count how many "arcs" connecting two nearest neighbors cross it if we connect each subsequence to its nearest neighbor as shown in Figure 17.
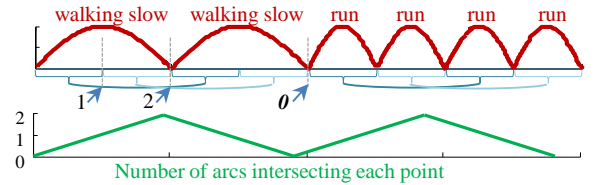

Figure 17. *top*) A (toy) time series (red) and nearest neighbor locations for each subsequence. *bottom*) The number of arcs crossing above each subsequence.

We applied the procedure described above to a heavily studied activity segmentation problem [29], which was derived from the CMU Motion Capture database [16]. The recordings are represented as multi-dimensional time series, and most research efforts carefully select the best subset for the segmentation task. For example, [29] states that "*we only consider the 14 most informative joints out of 29.*" In contrast, we attempt this with a *single* dimension. The only parameter we need to set is sliding window size. In Figure 18 we show the segmenting results obtained using our approach, with annotations taken from [29] for context.

Much of the evaluation in this community lacks formal metrics, preferring instead visual sanity tests like the one in Figure 18. Given this, we can say that our approach is *very* competitive on this dataset, in spite of the fact that we handicapped ourselves to only consider one dimension.

One-dimension of multi-d time series: Subject 86, recording 4, dimension 30

Matrix Profile

Split Points Prediction
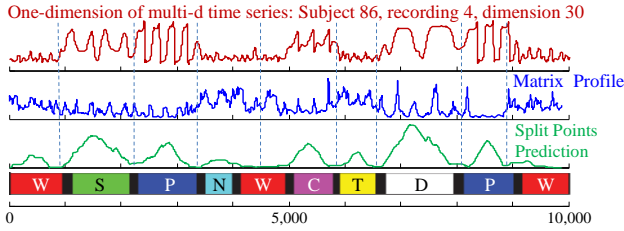
| W | S | P | N | W | C | T | D | P | W |

Figure 18. *top*) A matrix profile (blue) obtained for the time series (red) and number of arcs crossing each point (green). Low values of this green curve coorespond to candidate split points. *bottom*) Human annotations of the activities: w – walk, s – stretch, p – punch, c – chop, t – turn, d – drink.

V. CONCLUSION

We have introduced a scalable algorithm for creating time series subsequences joins. Our algorithm is simple, fast, parallelizable and parameter-free, and can be incrementally updated for moderately fast data arrival rates. We have shown that our algorithm has implications for many existing tasks, such as motif discovery, discord discovery, shapelet discovery and semantic segmentation, and may open up new avenues for research, including computing various definitions of time series set difference. Our code is freely available for the community to confirm, extend and exploit our ideas.

REFERENCES

[1] R. J. Bayardo, Y. Ma and R. Srikant, "Scaling up all pairs similarity search," *WWW 2007*, pp 131-140.

[2] N. Begum and E. Keogh, "Rare time series motif discovery from unbounded streams," *PVLDB* 8(2): 149-160, 2014.

[3] G. Beroza, "Personal Correspondence," Jan 21th, 2016.

[4] T. Bouezmarni and J. Rombouts, "Nonparametric density estimation for positive time series," *CSDA*, 54, 245-261, 2010.

[5] V. Chandola, D. Cheboli and V. Kumar, "Detecting anomalies in a time series database," UMN TR09-004.

[6] T. Chen et al., "The UCR time series classification archive," http://www.cs.ucr.edu/~eamonn/time_series_data/.

[7] "Convolution - Wikipedia, the free encyclopedia," https://en.wikipedia.org/wiki/Convolution, Accessed: 2016-01-19.

[8] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang and E. J. Keogh, "Querying and mining of time series data: experimental comparison of representations and distance measures," *PVLDB* 1(2): 1542-1552. 2008.

[9] J. Hughes et al., "Chimpanzee and human Y chromosomes are remarkably divergent in structure" *Nature* 463, (2010).

[10] H. Lee, R. Ng and K. Shim, "Similarity join size estimation using Locality sensitive hashing," *PVLDB*, 4(6):338–349, 2011.

[11] W. Luo, H. Tan, H. Mao and L. M. Ni, "Efficient similarity joins on massive high-dimensional datasets using mapreduce," In *MDM*'12, IEEE Computer Society, pp. 1-10.

[12] Y. Ma, X. Meng and S. Wang, "Parallel similarity joins on massive high-dimensional data using MapReduce," *Concurrency and Computation*, Volume 28, Issue 1 Jan 2016. Pages 166–183.

[13] S. V. Makonin, "AMPds: a public dataset for load disaggregation and eco-feedback research," *EPEC* 2013, pp 1-6.

[14] MASS: http://www.cs.unm.edu/~mueen/FastestSimilaritySearch.html

[15] G. D. F. Morales and A. Gionis, "streaming similarity self-join," *Proc. VLDB Endow*, 2016.

[16] Motion Capture Database, http://mocap.cs.cmu.edu/

[17] A. Mueen, H. Hamooni and T. Estrada, "Time series join on subsequence correlation," *IEEE ICDM* 2014, pp. 450-459.

[18] A. Mueen, E. Keogh, Q. Zhu, S. Cash and B. Westover, "Exact discovery of time series motif," SDM 2009.

[19] D. Murray et al., "A data management platform for personalised real-time energy feedback," In *EEDAL* 2015.

[20] V. Niennattrakul et al, "Data editing techniques to allow the application of distance-based outlier detection to streams," *ICDM* 2010: 947-952.

[21] D. Patnaik, et al, "Sustainable operation and management of data center chillers using temporal data mining," *KDD* 2009.

[22] T. Rakthanmanon et al., "Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping," In *KDD* 2012, 262-270.

[23] T. Rakthanmanon and E. Keogh, "Fast shapelets: a scalable algorithm for discovering time series shapelets," *SDM*, 2013.

[24] Supporting page. http://www.cs.ucr.edu/~eamonn/MatrixProfile.html

[25] C. D. Truong and D. T. Anh, "An Efficient Method for Motif and Anomaly Detection in Time Series" *IJBIDM*, Vol. 10, No. 4, 2015.

[26] A. Tucker and X. Liu, "A Bayesian Network Approach to Explaining Time Series with Changing Structure," *Intell Data Anal*, 8 (5) (2004).

[27] L. Ye and E. Keogh, "Time series shapelets: a new primitive for data mining," *ACM SIGKDD*, 2009, pp 947-56.

[28] C. Yoon, O. O'Reilly, K. Bergen and G. Beroza, "Earthquake detection through computationally efficient similarity search," *Sci. Adv.* 2015.

[29] F. Zhou, F. Torre and J. Hodgins "Aligned Cluster Analysis for Temporal Segmentation of Human Motion," *IEEE FG*'2008.

[30] S. Zilberstein and S. Russell, "Approximate Reasoning Using Anytime Algorithms," In *Imprecise and Approximate Computation*, Kluwer Academic Publishers, 1995.