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 two time series data mining problems, including motif discovery and novelty discovery.**

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][15][23]. We would like to refer the interested reader to [20] for the longer version of this paper.

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 two different 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

2374-8486/16 $31.00 © 2016 IEEE
DOI 10.1109/ICDM.2016.89

IEEE computer society

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*" [8]. 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 [18]. 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.

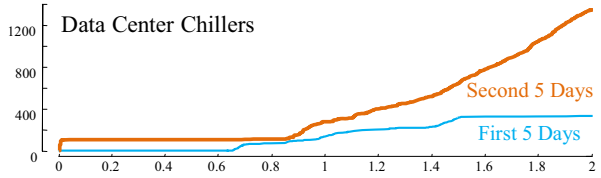


Figure 1. Output size vs. threshold for data center chillers [18]. Values beyond 2.0 are truncated for clarity (but archived at [20]).

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 [9] or SAX [10] 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 [9] 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 [23] 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 [13], 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. $T_{i,m} = t_i, t_{i+1},\ldots, t_{i+m-1}$, 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 [6]. 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.

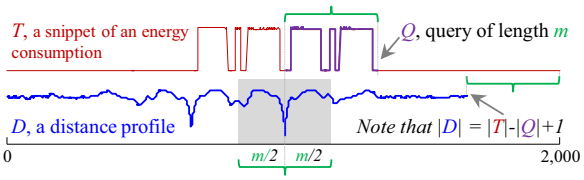


Figure 2. A subsequence Q extracted from a time series T is used as a query to every subsequence in T. The vector of all distances is a distance profile.

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 [15], 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* $\boldsymbol{A}$ 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: $\boldsymbol{A} = \{T_{1,m}, T_{2,m},\ldots, T_{n-m+1,m}\}$, where m is a user-defined subsequence length. We use $\boldsymbol{A}[i]$ 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 $\boldsymbol{A}$ and $\boldsymbol{B}$ and two subsequences $\boldsymbol{A}[i]$ and $\boldsymbol{B}[j]$, a *1NN-join function* θ_{1nn} ($\boldsymbol{A}[i]$, $\boldsymbol{B}[j]$) is a Boolean function which returns "true" only if $\boldsymbol{B}[j]$ is the nearest neighbor of $\boldsymbol{A}[i]$ in the set $\boldsymbol{B}$.

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 $\boldsymbol{A}$ and $\boldsymbol{B}$, a *similarity join set* $\boldsymbol{J_{AB}}$ of $\boldsymbol{A}$ and $\boldsymbol{B}$ is a set containing pairs of each subsequence in $\boldsymbol{A}$ with its nearest neighbor in $\boldsymbol{B}$: $\boldsymbol{J_{AB}}=\{\langle \boldsymbol{A}[i], \boldsymbol{B}[j] \rangle \mid \theta_{1nn}(\boldsymbol{A}[i], \boldsymbol{B}[j])\}$. We denote this formally as $\boldsymbol{J_{AB}} = \boldsymbol{A} \bowtie_{\theta 1nn} \boldsymbol{B}$.

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 $\boldsymbol{J_{AB}}$.

We call this vector the *matrix profile* 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.

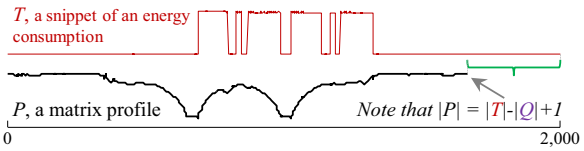


Figure 3. A time series T, and its self-join matrix profile P.

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 [15], 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* $\boldsymbol{J_{AA}}$ is a result of similarity join of the set $\boldsymbol{A}$ with itself. We denote this formally as $\boldsymbol{J_{AA}} = \boldsymbol{A} \bowtie_{\theta 1nn} \boldsymbol{A}$. 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 $\boldsymbol{A}[i]$ and $\boldsymbol{A}[j]$ are subsequences from the same all-subsequences set $\boldsymbol{A}$, $\theta_{1nn}(\boldsymbol{A}[i], \boldsymbol{B}[j])$ is "false" when $\boldsymbol{A}[i]$ and $\boldsymbol{A}[j]$ are a trivially matched pair.

The i^{th} 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 $\boldsymbol{J_{AB}}$ is a vector of integers where $I_{AB}[i] = j$ if $\{\boldsymbol{A}[i], \boldsymbol{B}[j]\} \in \boldsymbol{J_{AB}}$.

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

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!

A. 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 I. 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 with the MASS algorithm [12] using each subsequence $\boldsymbol{B}[idx]$ 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 $\text{length}(D) - 1$.) 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 I 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 $\boldsymbol{B}$ with $\boldsymbol{A}$ in line 4, and ignore trivial match in D when performing *ElementWiseMin* in line 5.

TABLE I. 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, $\boldsymbol{J_{AB}} = \boldsymbol{A} \bowtie_{\theta 1nn} \boldsymbol{B}$
1	$n_B \leftarrow \text{Length}(T_B)$
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**($\boldsymbol{B}[idx]$, T_A) // [12]
5	$P_{AB}, I_{AB} \leftarrow \text{ElementWiseMin}(P_{AB}, I_{AB}, D, idx)$
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}.

B. 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 [24]. To add the anytime nature to the STAMP algorithm, we simply ensure a randomized search order in line 2 of TABLE I.

Zilberstein [24] 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 [24]). 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*. 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 I 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 **Error! Reference source not found.** is very typical, so we could use past convergence curves to predict the quality of solution when interrupted on similar data.

The seismology dataset offers an excellent opportunity to demonstrate the utility of the anytime version of our algorithm. The authors of [23] revealed in their long-term ambition of mining even larger datasets [3]. In Figure 4 we repeated the experiment with the snippet shown in Figure 5, 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.

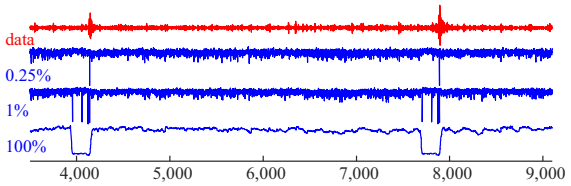


Figure 4. *top*) An excerpt of the seismic data that is also shown in Figure 5. *top-to-bottom*) The approximations of the matrix profile for increasing interrupt times. By the time we have computed just 0.25% of the calculations required, the minimum of the matrix profile points to the ground truth.

C. 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.

D. 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. For simplicity and brevity TABLE II only shows the algorithm to maintain the self-similarity join. The generalizations are obvious.

TABLE II. THE STAMP*I* ALGORITHM

	Procedure STAMPI(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.

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 [20] which contains all datasets and code used in this work. 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 III we show the time required for a self-join with m fixed to 256, for increasingly long time series.

TABLE III. 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 IV, 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 [6][15], the running time of STAMP does not depend on m.

TABLE IV. 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 [20].

B. Profile-Based Self-Join

A recent paper notes that many fundamental problems in seismology can be solved by joining seismometer telemetry [23], 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 5, confirmed that STAMP does not have false negatives.

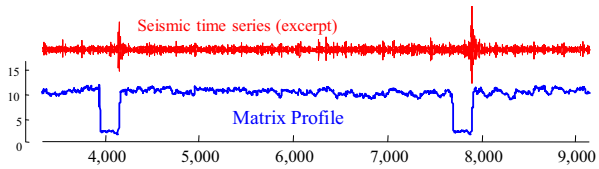


Figure 5. *top*) An excerpt of a seismic time series aligned with its matrix profile (*bottom*). The ground truth provided by the authors of [23] requires that the events occurring at time 4,050 and 7,800 match.

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. 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 [15] 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 [15]. 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 6.*left* we ask what happens as we search for longer and longer motifs. In Figure 6.*right* we ask what happens if the motif length is fixed to $m = 512$, but the data becomes increasing noisy.

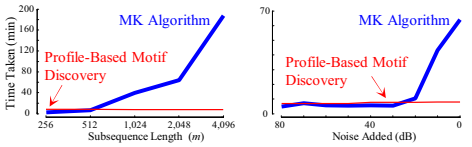


Figure 6. The time required to find the top-motif pairs in a time series of length 2^{16} for increasingly long motif lengths (*left*), and for a length fixed to 512, but in the face of increasing noise levels (*right*).

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.

D. 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 7, the time series discord is encoded as the *maximum* value in a matrix profile.

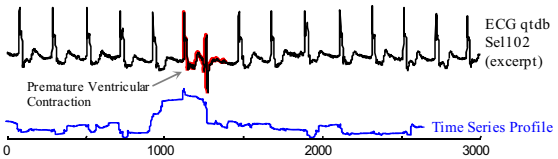


Figure 7. *top*) An excerpt from an ECG incorporating a premature ventricular contraction (red/bold). *bottom*) The time series profile peaks *exactly* at the beginning of the PVC.

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.

E. 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 [17][2] or both [21] of these tasks, but they are all approximate and allow false dismissals.

In Section III.D, we introduced the STAMP*I* algorithm. The ability to incrementally maintain the matrix profile implies the ability to *exactly* maintain the time series motif [15] 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 [11]. 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. As shown in Figure 8, 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.

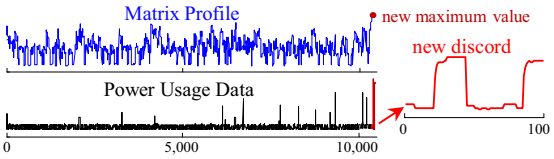


Figure 8. *top*) The matrix profile for the first 10,473 minutes. *bottom*) The maximum value of the matrix profile corresponds to a time series discord. *right*) The time series discord detected is the first heat pump pattern occurrence in the dataset.

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.

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 motif/discord discovery, 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] 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.

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

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

[9] 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.

[10] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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