Matrix Profile Based kNN Search over Large Multi-Dimensional Time Series

Tanmoy Mondal*, Reza Akbarinia* and Florent Masseglia*
*ZENITH Team, INRIA & LIRMM, Univ. Montpellier, France
{tanmoy.mondal, reza.akbarinia, florent.masseglia}@inria.fr

Abstract—Matrix Profile ..

Index Terms—Time series analysis; STAMP; STOMP; All-pairs-similarity search; Motifs and discord discovery; Outliers detection; Anomaly detection; Joins

1 Introduction

The topic of *all pair similarity search* has been highly explored in the last decade. This problem is also known as *similarity join* which can be defined in simple words as follows.....

2 Problem Definition

The following section is rather dense on terminology and definitions which are necessary to concretely define the problem statement and to explain the proposed algorithms/techniques. We begin with the definition of a *"time series"* ($T \in \mathbb{R}^n$), which can be denoted by a sequence of real valued numbers $t_i \in \mathbb{R}$ as: $T = t_1, ..., t_n$, where n denotes the length of the time series. A time series sub-sequence $T_{i,m}$ of a time series T (of length n) is a continuous sub-set of the values/elements from T of length $m; m << n$ starting from position from p, that is $T_{i,m} = T_{i,p}, ..., T_{i,p+m-1}$ for $1 \leq p \leq n-m+1$; where m is a real value which represents the size of sub-sequence. An all sub-sequence set can be defined as an ordered set of all sub-sequences of T which are 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 sub-sequence length.

Definition 1: ***Distance profile :*** The distance between a sub-sequence $T_{i,m}$ with all other sub-sequences of time series T gives a $1D$ vector of distances which is called distance profile of $T_{i,m}$.

The minimum value of this distance vector represents the closest match or $1NN$ and the top k minimum values of this vector represents kNN matches. Considering these $1NN$ or kNN for all the sub-sequences in a time series T are called as either *1NN Matrix Profile* or *kNN Matrix Profile* respectively. If the distance of closest match (1NN) of any sub-sequence T_i is less than a user defined threshold then T_i is called *motif*. In other words, a motif pair can be defined as the unordered pair of sub-sequences $\{T_i, T_j \in T\}$ which is most similar (i.e. *1NN*) and has a distance value less than a user defined threshold (i.e. $Dist(T_i, T_j) \leq \tau$; where τ is an user defined threshold) are called as *motif pair* i.e. $\langle T_i, T_j \rangle$; $|i-j| \geq w$ for $w > 0$. Where w is user defined threshold which tells that two sub-sequences in a pair should be w elements apart. The definition of match, forces the matched sub-sequence couple to be mutually exclusive otherwise the motifs might share the majority of their elements and thus could be essentially the same. This helps to prune out the *trivial* sub-sequence matches.

Definition 2: ***Time Series Discord :*** A sub-sequence $T_i \in T$ is a discord if the distance of it's $1NN$ match is greater than an user defined high threshold; say τ. Whereas, if the distance of T_i with it's k^{th} nearest neighbors is higher than τ then it is called *kNN* discord.

Definition 3: kNN ***Matrix Profile*** : The kNN matrix profile denoted as $P_T[i]$, which can be thought as a k numbers of vertical $1D$ vectors, consists of k number of closest Euclidean distances between any sub-sequence T_i and it's k nearest neighbors, obtained from time series T. Hence, P_T is a *2D* vector or a matrix where the k nearest neighbors of each sub-sequence i.e. T_i are stored vertically or column wise. For the case of *1NN* matrix profile, k is simply taken as 1.

Definition 4: ***kNN Matrix Profile Index*** The *kNN* matrix profile index can be defined as $I_T[i]$ which is a *2D* matrix, where each column of this matrix are consisting of the indexes of k nearest neighbors of any sub-sequence T_i. For the case of *1NN* matrix profile index, k is simply taken as 1.

The primary goal of this article is to find *kNN* matches of each sub-sequence by creating these two meta vectors i.e. the *matrix profile* and the *matrix profile index*.

Until now, we have talked about *1D* time series and calculation of *1NN* & *kNN* matrix profile and matrix profile indexes of *1D* time series. In the following section, we introduce the concept of the calculation of *1NN* & *kNN* matrix profile and matrix profile indexes of *multi-dimensional* time series. A multi-dimensional time series $\hat{T} \in \mathbb{R}^{d \times n}$ which is a d dimensional time series of length n where $\hat{T} = [T_n^1, T_n^2, T_n^3, ..., T_n^d]$ can be imagined as a matrix of n rows and d columns. Whereas, a *multidimensional sub-*

• *M. Tanmoy Mondal is currently with "Signal and Communication" Team, IMT Atlantique, Brest, France.*

sequence $\hat{T}_{i,m} \in \mathbb{R}^{d \times m}$ of a *multi-dimensional* time series $\hat{T}$ can be defined as continuous subset of values from $\hat{T}$ of length m, starting at position/index i, which can be formally defined as $\hat{T}_{i,m} = [T^1_{i,m}, T^2_{i,m}, T^3_{i,m}, ..., T^d_{i,m}]$. It is mentioned in [1] that using all the dimensions of a time series for motifs discovery usually guaranteed to fail. A similar observation was claimed in [2] for time series classifications. Hence, it is recommended to use a sub-set of all the dimensions should be used for multi-dimensional motif discovery.

Definition 5: ***Sub-dimensional sub-sequences:*** A *sub-dimensional* sub-sequence can be defined as $\hat{T}^p_{i,m} \in \mathbb{R}^{p \times m}$, which is a multi-dimensional sub-sequence which is based on some selection criterion (described later), only a sub-set of dimensions are selected, where k is the number of dimensions included.

In this paper, we have mentioned the way to compute the distance between two multi-dimensional time sub-sequences by using only the selected dimensions of each sub-sequences. Hence, the distance function that measures this relation is called *p-dimensional distance*.

Definition 6: ***p-dimensional distance:*** A *p-dimensional* distance function or $dist^p$ can be defined as the distance between two multi-dimensional sub-sequences by automatically choosing the best p dimensions out of existing d dimensions. This can be formally defined as $dist^p(\hat{T}_{i,m}, \hat{T}_{j,m}) = dist(\hat{T}^p_{i,m}, \hat{T}^p_{j,m})$

After mentioning the *p-dimensional distance*, we can now describe the *p-dimensional distance profile*. A *p-dimensional* distance profile of a sub-sequence $\hat{T}_{i,m}$ can be defined as a vector that stores the distance : $dist^p(\hat{T}^p_{i,m}, \hat{T}^p_{j,m}) \forall j \in [1, 2, 3, n - m + 1]$. Like the definition of motif for 1 dimensional time series, a motif for d dimensional time series can be similarly defined as the unordered pair (i.e. $Dist(\hat{T}^p_{i,m}, \hat{T}^p_{j,m}) \leq \tau$; where τ is an user defined threshold) and the starting positions of these two sub-sequences i.e. i and j are w elements apart i.e. $|i - j| \geq w$. Whereas, the top k motifs of any sub-sequence T_i can be defined as $Dist(\hat{T}^p_{i,m}, \hat{T}^p_{j,m}) \leq Dist(\hat{T}^p_{i,m}, \hat{T}^p_{j+1,m}) \leq \tau; j \in [1, 2, 3, k - 1]$. Which means the distance between i^{th} sub-sequence and any other j^{th} sub-sequence is less than $j + 1^{th}$ sub-sequence, where these distance are sorted in ascending order.

Definition 7: ***p-dimensional matrix profile and matrix profile index:*** A *p-dimensional* matrix profile $\hat{P} \in \mathbb{R}^{n-m+1}$ of a multi-dimensional time series $\hat{T}$ can be thought as $1D$ horizontal vector, containing the nearest neighbor distances of each sub-sequence of $\hat{T}$, where the distances are calculated between two sub-sequence by using *p dimensional distance* (see Definition 6). Whereas, the matrix profile index stores the indexes of the best matching (1NN) sub-sequence.

Definition 8: ***top kNN based on p-dimensional distance of a sub-sequence:*** Based on the p dimensional distance profile of any sub-sequence $\hat{T}_{i,m}$, we can sort the distance profile in ascending order and can obtain top k of such distances which can be considered as kNN matches, based on p-dimensional distances of $\hat{T}_{i,m}$.

Definition 9: ***kNN matrix profile by considering p dimensions of a time series sub-sequence:*** A *kNN* matrix profile by considering p dimensions of a time series can be denoted as : $\hat{P}^p_k \in \mathbb{R}^{(n-m+1) \times k}$, which can be thought as k number of vertical vectors or a matrix of k rows and $n - m + 1$ number of columns. This matrix $\hat{P}^p_k$ contains the k nearest neighbor distances of each sub-sequence of $\hat{T}$, where these distances are calculated between two sub-sequence by automatically considering the *p dimensions* (see Definition 6) of the time series sub-sequences.

In the similar manner, the *kNN matrix profile index* matrix will store the indexes of top k matches of every sub-sequences. Based on these two matrices i.e. kNN matrix profile and kNN matrix profile index, we can obtain the answers of many time series data mining tasks such as motifs and discord discovery, similarity search etc. In the follows section, we provide a brief overview of literature related to similarity search of multi dimensional time series.

3 Related Word in Similarity Search

Similarity join can be categorized into two principal categories, *1NN Similarity Join* and *kNN Similarity Join*. The *kNN Similarity Join* can be thought as the immediate and obvious extension of *1NN Similarity Join*. Yeh et. al [3] has proposed *Matrix Profile* based *STAMP/STOMP* algorithm for *1NN Similarity Join*. Matrix profile is designed mainly to perform *1NN Similarity Join* and it can't be directly used for *kNN Similarity Join*. Nevertheless, the technique proposed in [3] doesn't require any parameters to set and it's a fast, robust, *anytime* and *incremental* solution.

In case of all sub-sequence matching of time series, the authors in [4] has proposed an approach to optimize the calculation of *Euclidean Distance* between all the possible combination of sub-sequences exists in the database. They proposed to interleave the early abandoning calculations of *Euclidean Distance* with the concept of online *Z normalization*. By reusing computations of z-normalized distances for overlapping sub-sequences, the authors has highly save the computation time and was able to reduce the search space into quadratic complexity. Mueen et.al proposed MASS algorithm [5] which exploits the consecutive sub-sequence overlapping property to calculate *Z normalized* distance by *Fast Fourier Transform (FFT)* based convolutions, which has a worst case time complexity of $O(n\ logn)$.

In [3], the authors have used the convolution property of *FFT* and *Inverse FFT* for the fast calculation of distances between two sub-sequence pair. Furthermore, an incremental approach is adapted for distance computation of overlapping sequential sub-sequences in which they have used *MASS* algorithm for time series similarity search by computing *z-normalized* euclidean distance between the sub-sequences. In [6], the authors have introduced an anytime algorithm, named as *SCRIMP++* by combining the best features of *STAMP* and *STOMP* for fast convergence. But this technique is also for *1NN* matrix profile and based on the same incremental matching principal as *STOMP*.

4 Related Word in Similarity Search of Multidimensional Time Series

In this article, we are focused to solve *similarity join* problem on multi-dimensional datasets which can be simply defined as :

provided a set of data objects (for our case the sub-sequences), the objective is to retrieve the k nearest neighbors for each object.

A nice attempts were made for multidimensional motif discovery as a means for patient activity monitoring. Motifs discovery are useful to capture repeating patterns across multiple dimensions of the data [7]. This approach is real time for *multidimensional motifs discovery* in time series. These time series are generated by body sensors, used for monitoring the performance of patients during therapy. The authors presented two alternative models for *multidimensional motifs discovery* based on motif co-occurrences and temporal ordering among motifs across multiple dimensions. This method uses an efficient hashing based record to enable speedy update and retrieval of motif sets and the identifications of *multidimensional motifs*. The approach is tested on synthetic and real body sensor data for concurrent processing of multidimensional time series data to find unknown and naturally occurring patterns with minimal delay and tracking similarities among repetitions, mainly during the therapy sessions.

The technique mentioned in [8] achieves the scalability by searching over a piece-wise linear approximation of the original data by transforming it into string which facilitates the finding of recurring motifs. The piece-wise linear approximation techniques has been used in literature over several occasions for $1D$ time series data and has shown interesting results with carefully chosen parameters on relatively smooth data. But it is little unclear that how the authors has used the technique of piece-wise linear approximations for multi-variate data.

Vahdatpour et.al [9] proposed a technique for multi-dimensional time series motifs discovery and apply it in various medical monitoring applications. They calculate time series motifs for each individual dimensions and uses clustering technique to "stitch" together various dimensions. The authors tested their system on the data, gathered from sensors embedded in two different wearable systems, i.e. *SmartCane* and *SmartShoe*. Based on all the domains which were tested by the authors, even in the case of obvious motifs for simple and small problems, the authors could never achieve the accuracy more than 85% (even by considering at most three irrelevant dimensions). Moreover, the main bottleneck of their approach is the necessity to tune 7 parameters which has high influence on their results. But tuning these parameters could be a cumbersome job in the case of new challenging data-sets.

The approach proposed by Tanaka et al.[10] for the motif discovery in multi-dimensional time series data by simply transforming the multi-dimensional time series data into one dimensional data. This method is designed in a manner which requires to consider all dimensions or at least most of the dimensions of the data. Hence, if there are some or even few irrelevant dimensions then it hampers the accuracy of the system. Moreover, the computational speed of the system depends on correct tuning of 5 parameters which is strong bottleneck of this algorithm and due to that it may perform badly on some new data-sets.

The work in [11] addresses the problem of locating sub-dimensional motifs in real-valued, multivariate time series, which is capable to detect sets of recurring patterns along the corresponding relevant dimensions. The classical approaches of motifs discovery are restricted to categorical data i.e. univariate time series and/or multi-variate time series in which it is considered that the temporal patterns spans to all the dimensions. But the technique in [11] generalizes the multi-dimensional pattern discovery in which each motif may span only a subset of the dimensions. Their work show the detrimental effects of irrelevant dimensions on multi-dimensional motif search and they proposed a technique which was somewhat successful and robust for small number of smooth but irrelevant dimensions, or just one noisy irrelevant dimension. But the main issue is that this algorithm is approximate and even in ideal cases this algorithm reports to have 80% accuracy with a six dimensional data, having no noise.

In [12], the authors has proposed a technique for discovering of sub-dimensional motifs of different lengths in multivariate time series. They have introduced an approximate variable-length sub-dimensional motif discovery algorithm called *collaborative hierarchy based motif enumeration* to detect variable length sub-dimensional motifs (even when the motif length is considerably larger than minimum length), given a minimum motif length. A technique known as *DiscMotifs* to discover k most significant motifs from an uni-variate time series is proposed in [12]. First, the algorithm transforms the time series into a SAX representation and then the algorithm divides the SAX representation into sub-sequences. After that these sub-sequences are linearized by projecting them into a one-dimensional space based on their distances from a randomly selected reference point or sub-sequences. By utilizing the linear ordering of sub-sequences, *DiscMotifs* is able to discover k most significant motifs.

A matrix profile based motif discovery technique for multi-dimensional time series known as *mSTAMP* is proposed in [1]. The *mSTAMP* algorithm finds motifs by calculating a cumulative distance through re-ordering of distances, calculated in each dimensions. This method is also capable of only selecting some dimensions as well is capable of ignoring certain user-defined dimensions. Hence, unlike other techniques in the literature, this method is capable to avoid the finding of motifs from all dimensions which many a times ends up in wrong motif discovery.

5 Algorithms

After explaining all the definitions in Section 2 and the relevant state-of-the-art techniques in Section 4, in this section we are ready to describe our proposed algorithm. In the following section, we have proposed an algorithm for finding kNN matches of all the sub-sequences of a multi-dimensional time series based on computed distance profile.

5.1 kNN similarity search of multi-dimensional time series

Finding kNN matches of all the sub-sequences of a time series T^d is performed in the following manner where d is the total number of dimensions of the time series. Let's consider an example where in a time series database $\mathfrak{D}^T$,

there are $\mathfrak{x}$ number of time series of different lengths : $\{T_1^d, T_2^d, T_3^d, ..., T_{\mathfrak{x}}^d\} \in \mathfrak{D}^T$. By concatenating all the time series in $\mathfrak{D}^T$, we can obtain a big time series $\mathcal{T}$. Now the goal is to find the closest match of all the sub-sequences of $\mathcal{T}$. The following Algorithm 4 is proposed which basically computes the distance of each sub-sequence of $\mathcal{T}$ with all the remaining sub-sequences and based on this repetitive process, we find the top k best matches of each query sub-sequences. One such visual example of multi-dimensional time series is shown in Fig. 1b.

The pseudo code of the proposed algorithms is shown in Algorithm 1. All the individual time series from database $\mathfrak{D}^T$ are sequentially concatenated to form a concatenated time series $\mathcal{T}$ (Line 2) and the information e.g. start and end indexes/locations of an individual time series when it is concatenated to generate $\mathcal{T}$ are saved in $Info_{\mathcal{T}}$. Also the file name or index of the individual time series are also saved in $Info_{\mathcal{T}}$. The length of $\mathcal{T}$ and total number of possible sub-sequence of $\mathcal{T}$ are calculated in $n_{\mathcal{T}}$ and $Idx_{\mathcal{T}}$ variable respectively (Line 4). The mean $(\mu_{\mathcal{T}})$ and standard deviation $(\sigma_{\mathcal{T}})$ of all the sub-sequences of $\mathcal{T}$ are calculated by iterating over the dimensions of $\mathcal{T}$ and by using *ComputeMeanStd()* function (Line $5-6$) (to get the details of this function, please see Appendix B of our previous article in [13]). After obtaining the very first sub-sequence of $\mathcal{T}$ in Line 7, the distance between first sub-sequence and all the other sub-sequences are calculated in Line 9 by using the function *MASS()* (to get the details of this function, please see Appendix C of our previous article in [13]). The arguments passed in this function are the first sub-sequence of $\mathcal{T}$ (i.e. $subSeq_1$), mean and standard deviation of $subSeq_1$ i.e. $\mu_{\mathcal{T}}[1]$, $\sigma_{\mathcal{T}}[1]$, complete time series $\mathcal{T}$ and the mean and standard deviations of all the sub-sequence of $\mathcal{T}$ i.e. $\mu_{\mathcal{T}}[1 : Idx_{\mathcal{T}}]$, $\sigma_{\mathcal{T}}[1 : Idx_{\mathcal{T}}]$ respectively (Line $8-9$). The dot product between the $subSeq_1$ and all other sub-sequences of $\mathcal{T}$ is stored in $QT_{initial}$ (Line 10). In Line 11, we perform the remaining operations in *Obtain_kNNMatches()* function.

Now in the function *Obtain_kNNMatches()*, the $3D$ matrices $P_{\mathcal{Q}}$, $I_{\mathcal{Q}}$, $\mathcal{M}_{\mathcal{Q}}$, $\mathfrak{D}_{\mathcal{Q}}$, $\mathfrak{d}_{\mathcal{Q}}$, $\mathfrak{I}_{\mathcal{Q}}$ are initialized with ∞ and zeros respectively (see *Obtain_kNNMatches()* function) in line 1-2. Then we iterate over all the sub-sequences of $\mathcal{T}$ in Line 4 and for each dimension of $\mathcal{T}$, we chop the sub-sequence of size m in Line 6. If the standard deviation of this sub-sequence is not zero (Line 7) and this is the first valid sub-sequence along $iDim^{th}$ dimension then we calculate the distance between i^{th} sub-sequence of $\mathcal{T}$ with all the remaining sub-sequences by using *MASS ()* function . The arguments passed in this function are the i^{th} sub-sequence of $\mathcal{T}$ (i.e. $cutTarget$), mean and standard deviation of $cutTarget$ i.e. $\mu_{\mathcal{T}}[i]$, $\sigma_{\mathcal{T}}[i]$, complete time series $\mathcal{Q}$ and the mean and standard deviations of all the sub-sequences of $\mathcal{Q}$ i.e. $\mu_{\mathcal{Q}}[1 : Idx_{\mathcal{Q}}]$, $\sigma_{\mathcal{Q}}[1 : Idx_{\mathcal{Q}}]$ respectively (Line 9). The dot product between the $cutTarget$ and all other sub-sequences of $\mathcal{Q}$ are stored in QT. Now from 2^{nd} sub-sequence onward in each dimension, the distance between i^{th} sub-sequence and all other sub-sequences are incrementally calculated by using *IndependentSTOMP()* function in Line 12 (to get the details of this function, please see Appendix $E.1$ of our previous article in [13]).

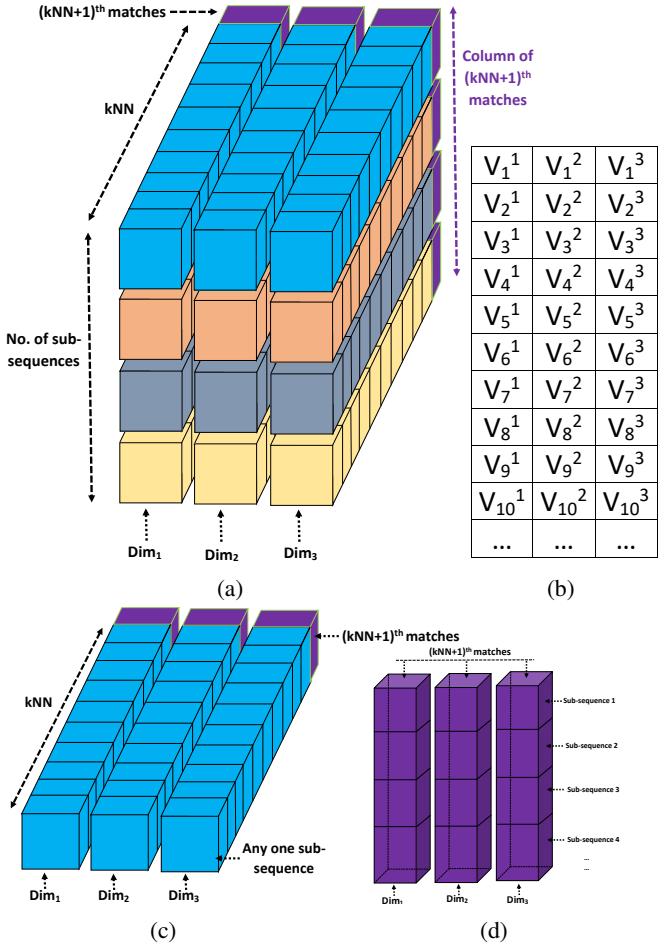

Figure 1: The analysis of computational time of the proposed algorithm on random-walk and seismic data-sets: (a) The proposed technique to manage the matching result comes from two different files in multi core based parallel processing architecture. (b) The proposed technique to handle the matched sub-sequence which belongs at the juncture.

Remember that the objective is to calculate the distance between all the sub-sequence of $\mathcal{T}$ against each sub-sequence (i^{th}) of $\mathcal{T}$ and finally to keep top kNN matches of all the sub-sequences in the variable $P_{\mathcal{T}}$. A visual representation of $P_{\mathcal{T}}$ is shown in Fig. 1a where each row (X axis) represents the sub-sequences of $\mathcal{T}$ and matches along the dimensions of time series (Y axis) are stored column wise. The kNN matches for each query and for each dimensions are stored along Z axis.

Hence, the distances between all the sub-sequence of $\mathcal{T}$ and i^{th} sub-sequence are stored along Z axis of $P_{\mathcal{T}}$ matrix. We continue to store like this until $i \leq kNN$ i.e. k number of sub-sequences and the corresponding indexes i.e. i is stored in $I_{\mathcal{T}}$ matrix (Line $13-14$ in *Obtain_kNNMatches()* function). The distance with all the i^{th} sub-sequences, where $i > kNN$ are stored at $kNN + 1^{th}$ position of $P_{\mathcal{T}}$ matrix (Line $15-17$). After that the remaining operation is performed in function *FunctionForElsePart_1()* in line 18.

In the function *FunctionForElsePart_1()*, it is first checked whether $i = kNN + 1$ or not. That means we perform the following operations just for a single time i.e. when

Algorithm 1: SELFTIMESERIESJOIN($\mathfrak{D}^T$, m, kNN)

Input: The target time series data base ($\mathfrak{D}^T$)
Output: A matrix profile ($P_{T_{conCat}}$) and associated matrix profile index ($I_{T_{conCat}}$)

for $iSeries \leftarrow 1$ ***to*** $length(\mathfrak{D}^T)$ **do**
  $\mathcal{T} \leftarrow [\mathcal{T}, \mathfrak{D}^T[iSeries]]$ ▷ *concatenate individual time series from the data base* $\mathfrak{D}^T$
  $Info_{\mathcal{T}} \leftarrow [startIdx, endIdx, fileName]$ ▷ *store the start, end indexes and the file name*
$n_{\mathcal{T}} \leftarrow length(\mathcal{T})$; $Idx_{\mathcal{T}} \leftarrow n_{\mathcal{T}} - m + 1$
for $iDim \leftarrow 1$ ***to*** d **do**
  $[\mu_{\mathcal{T}}[1 : Idx_{\mathcal{T}}][iDim], \sigma_{\mathcal{T}}[1 : Idx_{\mathcal{T}}][iDim]] \leftarrow ComputeMeanStd(\mathcal{T})$
$subSeq_1 \leftarrow \mathcal{T}[1 : 1 + m - 1][1 : d]$
for $iDim \leftarrow 1$ ***to*** d **do**
  $[\ QT[1 : Idx_{\mathcal{T}}][iDim],\ D^{ignore}[1 : Idx_{\mathcal{T}}][iDim]\ ] \leftarrow MASS(subSeq_1[1 : m][iDim], \mu_{\mathcal{T}}[1][iDim], \sigma_{\mathcal{T}}[1][iDim], \mathcal{T}, \mu_{\mathcal{T}}[1 : Idx_{\mathcal{T}}][iDim], \sigma_{\mathcal{T}}[1 : Idx_{\mathcal{T}}][iDim])$
$QT_{initial} \leftarrow QT$ ▷ *keeping a copy of the very first dot product*
$[P_{\mathcal{T}},\ I_{\mathcal{T}},\ \mathcal{M}_{\mathcal{T}},\ \mathfrak{D}_{\mathcal{T}},\ \mathfrak{d}_{\mathcal{T}},\ \mathfrak{I}_{\mathcal{T}}\ ] \leftarrow Obtain\_kNNMatches(\mathcal{T},\ Idx_{\mathcal{T}}\ , \mathcal{T},\ Idx_{\mathcal{T}}\ , \mu_{\mathcal{T}}, \sigma_{\mathcal{T}}, \mu_{\mathcal{T}}, \sigma_{\mathcal{T}}, d, m, kNN)$ ▷ *call this function, see below*
return $P_{\mathcal{T}}[1 : kNN][1 : Idx_{\mathcal{T}}][1 : d] \leftarrow \mathfrak{D}_{\mathcal{T}}[1 : kNN][1 : Idx_{\mathcal{T}}][1 : d]$
return $I_{\mathcal{T}}[1 : kNN][1 : Idx_{\mathcal{T}}][1 : d] \leftarrow \mathfrak{d}_{\mathcal{T}}[1 : kNN][1 : Idx_{\mathcal{T}}][1 : d]$
return $\mathcal{M}_{\mathcal{T}}[1 : kNN][1 : Idx_{\mathcal{T}}][1 : d] \leftarrow \mathfrak{I}_{\mathcal{T}}[1 : kNN][1 : Idx_{\mathcal{T}}][1 : d]$

Function `Obtain_kNNMatches`($\mathcal{T}$, $Idx_{\mathcal{T}}$, $\mathcal{Q}$, $Idx_{\mathcal{Q}}$ $\mu_{\mathcal{T}}$, $\sigma_{\mathcal{T}}$, $\mu_{\mathcal{Q}}$, $\sigma_{\mathcal{Q}}$, d, m, kNN):
  /* *the following code is executed when* $i > kNN$ *in Algorithm 4* */
  $P_{\mathcal{Q}} \leftarrow ((kNN + 1) \times Idx_{\mathcal{Q}} \times d)\ array$ ▷ *it's a 3D matrix, initialized with infinity*
  $I_{\mathcal{Q}}, \mathcal{M}_{\mathcal{Q}}, \mathfrak{D}_{\mathcal{Q}}, \mathfrak{d}_{\mathcal{Q}}, \mathfrak{I}_{\mathcal{Q}} \leftarrow ((kNN + 1) \times Idx_{\mathcal{Q}} \times d)\ array$ ▷ *these are 3D matrix, initialized with zeros*
  $subSeqFlag \leftarrow (1 \times d)\ vector$ ▷ *it's a 1D horizontal Boolean vector, initialized with* $FALSE$
  for $i \leftarrow 1$ ***to*** $Idx_{\mathcal{Q}}$ **do**
    for $iDim \leftarrow 1$ ***to*** d **do**
      $cutTarget \leftarrow \mathcal{T}[i \text{ to } (i + m - 1)][iDim]$ ▷ *get target sub-sequence by chopping* T_{conCat}
      if $\sigma_{\mathcal{T}}[i][iDim] \neq 0$ **then**
        if $subSeqFlag[1][iDim] == FALSE$ **then**
          $[\ QT[1 : Idx_{\mathcal{Q}}][iDim],\ Dist_{cutTarget}[1 : Idx_{\mathcal{Q}}][iDim]\ ] \leftarrow MASS(cutTarget,\ \mu_{\mathcal{T}}[i][iDim], \sigma_{\mathcal{T}}[i][iDim],\ \mathcal{Q}[1 : Idx_{\mathcal{Q}}][iDim],\ \mu_{\mathcal{Q}}[1 : Idx_{\mathcal{Q}}][iDim],\ \sigma_{\mathcal{Q}}[1 : Idx_{\mathcal{Q}}][iDim])$ ▷ *apply* $MASS$ *algorithm*
          $subSeqFlag[1][iDim] == TRUE$
        else
          $[\ QT[1 : Idx_{\mathcal{Q}}][iDim],\ Dist_{cutTarget}[1 : Idx_{\mathcal{Q}}][iDim]\ ] \leftarrow IndependentSTOMP(\ cutTarget, \mu_{\mathcal{T}}[i][iDim],\ \sigma_{\mathcal{T}}[i][iDim], QT_{initial}[i][iDim],\ \mathcal{Q}[1 : n_{\mathcal{Q}}][iDim], QT[1 : Idx_{\mathcal{Q}}][iDim], \mu_{\mathcal{Q}}[1 : Idx_{\mathcal{Q}}][iDim],\ \sigma_{\mathcal{Q}}[1 : Idx_{\mathcal{Q}}][iDim]\ )$ ▷ *apply* $MASS$ *algorithm*
      if $i <= kNN$ **then**
        $P_{\mathcal{Q}}[i][1 : Idx_{\mathcal{Q}}][1 : d] \leftarrow \sqrt{Dist_{cutTarget}[1 : Idx_{\mathcal{Q}}][1 : d]};\quad I_{\mathcal{Q}}[i][1 : Idx_{\mathcal{Q}}][1 : d] \leftarrow i$
      else
        $P_{\mathcal{Q}}[kNN + 1][1 : Idx_{\mathcal{Q}}][1 : d] \leftarrow \sqrt{Dist_{cutTarget}[1 : Idx_{\mathcal{Q}}][1 : d]}$
        $I_{\mathcal{Q}}[kNN + 1][1 : Idx_{\mathcal{Q}}][1 : d] \leftarrow i$
        $[\ P_{\mathcal{Q}},\ \mathfrak{D}_{\mathcal{Q}},\ \mathfrak{d}_{\mathcal{Q}},\ \mathfrak{I}_{\mathcal{Q}},\ cumDistAll\ ] \leftarrow FunctionForElsePart\_1(i, Idx_{\mathcal{Q}}, P_{\mathcal{Q}},\ \mathfrak{D}_{\mathcal{Q}},\ \mathfrak{d}_{\mathcal{Q}},\ \mathfrak{I}_{\mathcal{Q}},\ kNN)$ ▷ *call this function, see below*
        $[\ \mathfrak{D}_{\mathcal{Q}},\ \mathfrak{d}_{\mathcal{Q}},\ \mathfrak{I}_{\mathcal{Q}}\ ] \leftarrow FunctionForElsePart\_3(Idx_{\mathcal{Q}},\ cumDistAll,\ \mathfrak{D}_{\mathcal{Q}},\ \mathfrak{d}_{\mathcal{Q}},\ \mathfrak{I}_{\mathcal{Q}}, kNN)$ ▷ *call this function, see below*
  return $P_{\mathcal{Q}},\ I_{\mathcal{Q}},\ \mathcal{M}_{\mathcal{Q}},\ \mathfrak{D}_{\mathcal{Q}},\ \mathfrak{d}_{\mathcal{Q}},\ \mathfrak{I}_{\mathcal{Q}}$

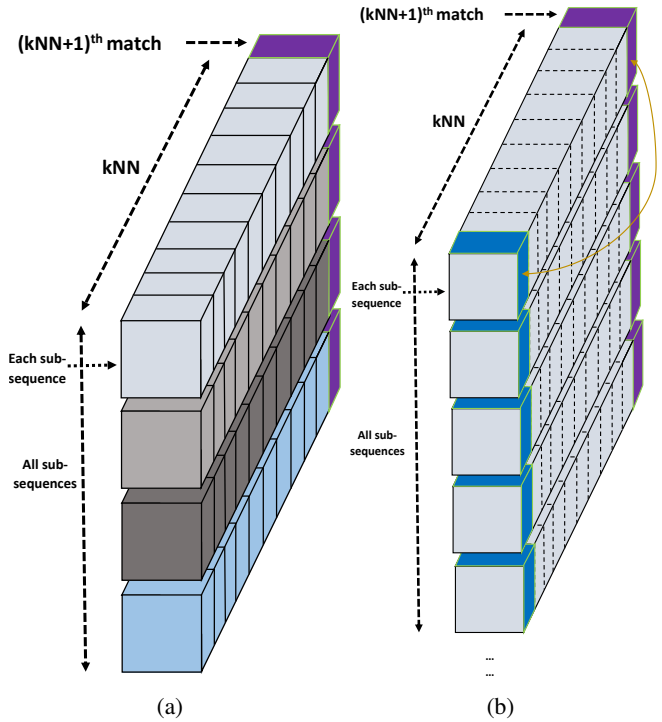

Figure 2: The analysis of computational time of the proposed algorithm on random-walk and seismic data-sets: (a) The proposed technique to manage the matching result comes from two different files in multi core based parallel processing architecture. (b) The proposed technique to handle the matched sub-sequence which belongs at the juncture.

$i = kNN + 1$. We iterate over all the sub-sequence of $\mathcal{T}$ and for each sub-sequence, the $(kNN + 1)$ number of matches for all the dimensions are stored in S_Q variable in Line 3. The $kNN + 1$ number of matches for one such sub-sequence is illustrated in Fig. 1c where the rows of this matrix represents $kNN + 1$ number of matches and these matches are arranged along the columns of each dimensions. Now the matrix S_Q is column wise sorted in descending order (i.e. along the dimensions) and the sorted distances are stored in $distSort_Q$ and the corresponding sorted indexes are stored in $indxSort_Q$ (Line 4). These sorted distances i.e. $distSort_Q$ and the sorted indexes (i.e. reshuffling of dimensions as the sorting was performed along dimensions or columns) i.e. $indxSort_Q$ are stored in $\mathfrak{D}_Q$ and $\mathfrak{d}_Q$ array respectively (line 5-6). Whereas, by using these sorted dimension wise indexes i.e. $indxSort_Q$, we iterate over all the $kNN + 1$ number of matches and the indexes of these matches are rearranged from I_Q array into $\mathfrak{I}_Q$ array (Line 6-9). After obtaining the sorted distances in $\mathfrak{D}_Q$ array, we perform a cumulative sum along the dimensions by using two $3D$ matrices named as $cumDist$ and $cumDistAll$ respectively (Line 10). These matrices can be imagined as the illustration shown in Fig. 2a. Now, we iterate over all the sub-sequences of $\mathcal{T}$ and all the available dimensions (i.e. along d dimensions) in Line 11 and 12 to add the distance values from $\mathfrak{D}_Q$ matrix. Then in $cumDistAll$ matrix, the values are saved by dividing the elements of $cumDist$ matrix by dimension's index i.e. $iDim$. In this way, we are actually iteratively summing up the sorted distances from $\mathfrak{D}_Q$ matrix in $cumDist$ array followed by division (in Line 13) by the dimension's index to actually diminish the contribution (denominator increases in each iteration of $iDim$) of distance values (which are arranged in increasing order along the columns after dimension-wise sorting) in each iterations.

Hence, the $cumDistAll$ matrix (illustration of this matrix is shown in Fig. 2a) will store the single distance value for each sub-sequence which is computed through the contribution of all distances along the dimensions (or columns) of the sub-sequence from $\mathfrak{D}_Q$ matrix. After obtaining the $cumDistAll$ matrix, we again iterate over all the sub-sequences and perform a *priority queue based heap sorting* on kNN number of elements. By using the sorted indexes i.e. $heapSortIdxs$, we rearrange the $cumDistAll$ matrix in Line 16. The property of heap based sorting is that it rearrange the elements in a way so that the maximum value will appear at the 1^{st} position of the array. Based on these sorted indexes, the $\mathfrak{D}_Q$, $\mathfrak{d}_Q$, $\mathfrak{I}_Q$ arrays are updated by using *UpdateIndex()* function (line 17). (the detail of this function is explained later)

The above defined all these operations are performed only once when $i = kNN+1$ otherwise we will enter into the else part in Line 18-19 of the function *FunctionForElsePart_1()*. The operations of the else part are performed by using the function *FunctionForElsePart_2()*. In this function, all the columns at $(kNN+1)^{th}$ location is taken into the M_{kNN+1} variable in Line 1 (see the illustration of all the columns at $(kNN + 1)^{th}$ location is shown in Fig. 1d). After obtaining these columns at $(kNN+1)^{th}$ location, these ones are sorted column-wise (in descending order) and the sorted distances are stored in $distTemp_{kNN+1}$ and sorted indexes are stored in $indxDim_{kNN+1}$ in Line 2. Like before the $\mathfrak{D}_Q$, $\mathfrak{d}_Q$, $\mathfrak{I}_Q$ arrays are updated by using $indxDim_{kNN+1}$. Then like before, we perform a cumulative sum along the dimensions by using two $3D$ matrices named as $cumDistTemp$ and $cumDistAll$ respectively (Line 7). We iterate over all the available dimensions (i.e. along d dimensions) in Line 8-10 and add the distance values from $dimSortDist_Q$ matrix. Then in $cumDistAll$ matrix, the values are saved by dividing the elements in $cumDistTemp$ matrix by dimension index $iDim$. From this function, the updated $\mathfrak{D}_Q$, $\mathfrak{d}_Q$, $\mathfrak{I}_Q$ and $cumDistAll$ matrix are returned.

Hence, we finish executing the *FunctionForElsePart_1()* function and we return to Line 18 of *Obtain_kNNMatches()* function. From there, we call the function *FunctionForElsePart_3()*. The task of this function is to put the newly computed distance values at the $kNN + 1^{th}$ location of $P_{\mathcal{Q}}$ matrix and compare these values with the ones exists at the 1^{st} location of $P_{\mathcal{Q}}$ matrix (such an illustration is shown in Fig. 2b, where the values at $kNN+1^{th}$ column are compared with the ones at 1^{st} column. The values at $kNN+1^{th}$ column are marked by violet color and values at the 1^{st} column are marked in blue). To do that, we compare $Idx_{\mathcal{Q}}$ number of values at the 1^{st} column of $cumDistAll$ array with the ones at $kNN+1^{th}$ column in Line 1 of *FunctionForElsePart_3()* function. This comparison gives the index of elements in $cumDistAll$ array whose value at 1^{st} column are greater than the values at $kNN+1^{th}$ column.

Function `FunctionForElsePart_1`$(i,\ Idx_Q,\ P_Q,\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q,\ kNN)$**:**

/* *the following code is executed when* $i > kNN$ *in Algorithm 4* */

1 **if** $i == kNN + 1$ **then**

2 **for** $p \leftarrow 1$ ***to*** Idx_Q **do**

3 $S_Q[1:kNN+1][1:d] \leftarrow P_Q[1:kNN+1][p][1:d]$ ▷ *get the* $(kNN+1)$ *number of matches of each query sub-sequences*

4 $distSort_Q[1:kNN+1][1:d], indxSort_Q[1:kNN+1][1:d] \leftarrow sortColWise\ (\ S_Q[1:kNN+1][1:d]\ )$ ▷ *sort the distance values column wise i.e. dimension wise and store the sorted distances and reshuffled indexes of the dimensions*

5 $\mathfrak{D}_Q[1:kNN+1][p][1:d] \leftarrow distSort_Q[1:kNN+1][1:d]$
$\mathfrak{d}_Q[1:kNN+1][p][1:d] \leftarrow indxSort_Q[1:kNN+1][1:d]$

6 **for** $iNN \leftarrow 1$ ***to*** $kNN+1$ **do**

7 $tempIndx_D[1:d] \leftarrow indxSort_Q[iNN][1:d]$ ▷ *pick* d *number of sorted indexes of dimensions, stored in each row of* $indxSort_Q$ *array through iterating over* $kNN+1$ *number of rows or stored matches*

8 **for** $iDim \leftarrow 1$ ***to*** d **do**

9 $\mathfrak{I}_Q[iNN][p][iDim] \leftarrow I_Q[iNN][p][tempIndx_D[iDim]]$ ▷ *by using the sorted dimensions in ("*$tempIndx_D[iDim]$*"), get the original sub-sequence index. Then copy that in "*$\mathfrak{I}_Q$*" array to rearrange matching sub-sequence indexes*

10 $cumDist \leftarrow ((kNN+1) \times Idx_Q \times 1)\ array;\quad cumDistAll \leftarrow ((kNN+1) \times Idx_Q \times 1)\ array$ **for** $p \leftarrow 1$ ***to*** Idx_Q **do**

11 **for** $iDim \leftarrow 1$ ***to*** d **do**

12 $cumDist[1:kNN+1][p][1] \leftarrow cumDist[1:kNN+1][p][1] + \mathfrak{D}_Q[1:kNN+1][p][iDim]$ ▷ *for each query sub-sequence, we are iteratively summing up the distances in each dimensions*

13 $cumDistAll[1:kNN+1][p][1] \leftarrow cumDist[1:kNN+1][p][1]/iDim$ ▷ *in each iteration, we divide by the dimension's index i.e. "*$iDim$*"*

14 **for** $p \leftarrow 1$ ***to*** Idx_Q **do**

15 $heapSortIdxs[1:kNN] \leftarrow BuildMaxHeap(cumDistAll[1:kNN][p][1], kNN)$ ▷ *for each query sub-sequences, sort the top* kNN *elements by building a max-heap based structure*

16 $cumDistAll[1:kNN][p][1:d] \leftarrow cumDistAll[\ heapSortIdxs[1:kNN]\ ][p][1:d]$

17 $[\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q\ ] \leftarrow UpdateIndex(1, kNN, heapSortIdxs[1], heapSortIdxs[kNN], p,\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q\ )$

18 **else**

19 $[\ P_Q,\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q,\ cumDistAll\ ] \leftarrow FunctionForElsePart\_2(Idx_Q,\ P_Q,\ cumDistAll,\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q)$ ▷ *call this function when* $i > (kNN+1)$

20 **return** $P_Q,\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q,\ cumDistAll$

After obtaining these indexes (i.e. $idxMax_Q \in 1....Idx_Q$), the values at the 1^{st} column at these indexes/rows in $\mathfrak{D}_Q$, $\mathfrak{d}_Q$, $\mathfrak{I}_Q$ arrays are exchanged with the values at $kNN+1^{th}$ column by using the *UpdateIndex()* function (Line 5, to visualize the scenario, see Fig. 1a and imagine that certain rows from $kNN+1^{th}$ column are exchanged with the same rows in 1^{st} column). The values at $kNN+1^{th}$ column of $cumDistAll$ array are also exchanged with the ones at 1^{st} column at $idxMax_Q$ rows. Then again the heap based sorting is performed on updated $cumDistAll$ array (Line 6) and based on the sorted indexes ($heapSortIdxs$), the top kNN elements of $cumDistAll$ array are rearranged. Then again the $\mathfrak{D}_Q$, $\mathfrak{d}_Q$, $\mathfrak{I}_Q$ arrays are updated by using the sorted indexes $heapSortIdxs$. Finally, the $\mathfrak{D}_Q$, $\mathfrak{d}_Q$, $\mathfrak{I}_Q$ and $cumDistAll$ arrays are returned from this function.

The *UpdateIndex()* function is simply used to update the a range of values in $\mathfrak{D}_Q$, $\mathfrak{d}_Q$, $\mathfrak{I}_Q$ arrays. This function copies the values of an array from location $stRiD1$ to $enRiD1$ into the location $stLeD1$ to $enLeD1$ of the same array at the index, mentioned by $indD2$.

5.2 kNN similarity search for multi-dimensional time series using AAMP algorithm

The AAMP algorithm (explained in detail in) is also capable to perform similarity search by incremental distance calculation and the advantage is that unlike *STOMP* based technique, it uses classical euclidean distance for similarity search. In the following section, we have explained the way to use this algorithm for the similarity search for multi-dimensional data. The pseudo code of the proposed algorithms is shown in Algorithm 2. The length of $\mathcal{T}$ and total number of possible sub-sequence of $\mathcal{T}$ are calculated in $n_{\mathcal{T}}$ and $Idx_{\mathcal{T}}$ variable respectively (Line $1-2$). The $3D$ matrices $P_{\mathcal{Q}}$, $I_{\mathcal{Q}}$, $\mathcal{M}_{\mathcal{Q}}$, $\mathfrak{D}_{\mathcal{Q}}$, $\mathfrak{d}_{\mathcal{Q}}$, $\mathfrak{I}_{\mathcal{Q}}$ are initialized with ∞ and zeros respectively. In

Function `FunctionForElsePart_2` (Idx_Q, P_Q, $cumDistAll$, $\mathfrak{D}_Q$, $\mathfrak{d}_Q$, $\mathfrak{I}_Q$):

/* following code is executed when $i > kNN$ in `FunctionForElsePart_1 ()` */

1 $M_{kNN+1}[1:Idx_Q][1:d] \leftarrow P_Q[1+kNN][1:Idx_Q][1:d]$ ▷ *pick only $(kNN+1)^{th}$ entry of all the query sub-sequences and store it in M_{kNN+1} array*

2 $distTemp_{kNN+1}[1:Idx_Q][1:d], indxDim_{kNN+1}[1:Idx_Q][1:d] \leftarrow sortColWise\ (\ M_{kNN+1}[1:Idx_Q][1:d]\ )$ ▷ *sort the distance values column wise i.e. dimension wise and store the sorted distances and reshuffled indexes of the dimensions; same as Line 3 of "FunctionForElsePart_1"*

3 $\mathfrak{D}_Q[kNN+1][1:Idx_Q][1:d] \leftarrow distTemp_{kNN+1}[1:Idx_Q][1:d]$ ▷ *storing these sorted distances*

4 $\mathfrak{d}_Q[kNN+1][1:Idx_Q][1:d] \leftarrow indxDim_{kNN+1}[1:Idx_Q][1:d]$ ▷ *storing these sorted indexes*

5 **for** $p \leftarrow 1$ ***to*** Idx_Q **do**

6 $\quad \mathfrak{I}_Q[kNN+1][p][1:d] \leftarrow I_Q[kNN+1][p][\ indxDim_{kNN+1}[p][1:d]\ ]$ ▷ *by using the sorted index of dimension in ("$indxDim_{kNN+1}$"), get the original sub-sequence index stored at that location. Then copy that in "$\mathfrak{I}_Q$" array to rearrange matching sub-sequence indexes*

7 $cumDistTemp \leftarrow ((kNN+1) \times Idx_Q \times 1)\ array;\quad cumDistAll \leftarrow ((kNN+1) \times Idx_Q \times 1)\ array$ ▷ *it's a 3D array, initialized with zeros*

8 **for** $iDim \leftarrow 1$ ***to*** d **do**

9 $\quad cumDistTemp[1][1:Idx_Q][1] \leftarrow cumDistTemp[1][1:Idx_Q][1] + dimSortDist_Q[kNN+1][1:Idx_Q][iDim]$ ▷ *the $(kNN+1)^{th}$ entry of all the query sub-sequence are iteratively summed up for all the dimensions*

10 $\quad cumDistAll[kNN+1][1:Idx_Q][1] \leftarrow cumDistTemp[1][1:Idx_Q][1]/iDim$ ▷ *in each iteration, we divide by the dimension's index i.e. "$iDim$"*

11 **return** $P_Q,\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q,\ cumDistAll$

Function `FunctionForElsePart_3` (Idx_Q, $cumDistAll$, $\mathfrak{D}_Q$, $\mathfrak{d}_Q$, $\mathfrak{I}_Q$, kNN):

/* following code is executed from Line 19 of `Obtain_kNNMatches()` */

1 $idxMax_Q \leftarrow findCondition\ (\ cumDistAll[1][1:Idx_Q]][1] > cumDistAll[kNN+1][1:Idx_Q]][1]\ )$ ▷ *find the indexes of query sub-sequences where the distance stored at 1^{st} location is bigger than the distances at $(kNN+1)^{th}$ location*

2 **if** $notEmpty(idxMax_Q)$ **then**

3 $\quad$ **for** $iIndx \leftarrow 1$ ***to*** $length(idxMax_Q)$ **do**

4 $\quad\quad \mathfrak{v} \leftarrow idxMax_Q(iIndx)$ ▷ *get the indexes of such query sub-sequences where the condition in Line 10 is satisfied*

5 $\quad\quad [\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q\ ] \leftarrow UpdateIndex(1, 1, kNN+1, kNN+1, \mathfrak{v},\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q)$ ▷ *call this function to update the $\mathfrak{D}_Q$,$\mathfrak{d}_Q$ and $\mathfrak{I}_Q$ array*

6 $\quad\quad cumDistAll[1][\mathfrak{v}][1:d] \leftarrow cumDistAll[kNN+1][\mathfrak{v}][1:d]$ ▷ *replace the $(kNN+1)^{th}$ distance values by the ones at 1^{st} location for $\mathfrak{v}^{th}$ query sub-sequences*

7 $\quad\quad heapSortIdxs[1:kNN] \leftarrow BuildMaxHeap(\ (cumDistAll[1:kNN][\mathfrak{v}][1]),\ kNN)$ ▷ *for each query sub-sequences, sort $kNN+1$ elements by building a max-heap based structure*

8 $\quad\quad cumDistAll[1:kNN][\mathfrak{v}][1] \leftarrow cumDistAll[heapSortIdxs[1:kNN][\mathfrak{v}][1]$ ▷ *rearrange "cumDistAll" array by using the sorted indexes i.e. "heapSortIdxs"*

9 $\quad\quad [\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q\ ] \leftarrow UpdateIndex(1, kNN, heapSortIdxs[1], heapSortIdxs[kNN], \mathfrak{v},\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q)$ ▷ *call this function to update the $\mathfrak{D}_Q$,$\mathfrak{d}_Q$ and $\mathfrak{I}_Q$ array*

10 **return** $\mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q,\ cumDistAll$

Function `UpdateIndex` ($stLeD1$, $enLeD1$, $stRiD1$, $enRiD1$, $indD2$):

/* this function is used to update set of elements in the array */

1 $\mathfrak{D}_Q[stLeD1:enLeD1][indD2][1:d] \leftarrow \mathfrak{D}_Q[stRiD1:enRiD1]\ [indD2][1:d]$ ▷ *replace values at indexes from $stLeD1$ to $enLeD1$ by the values at indexes from $stRiD1$ to $enRiD1$ for $indD2^{th}$ query sub-sequence*

2 $\mathfrak{d}_Q[stLeD1:enLeD1][indD2][1:d] \leftarrow \mathfrak{d}_Q[stRiD1:enRiD1]\ [indD2][1:d]$

3 $\mathfrak{I}_Q[stLeD1:enLeD1][indD2][1:d] \leftarrow \mathfrak{I}_Q[stRiD1:enRiD1]\ [indD2][1:d]$

4 **return** $\mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q$

Algorithm 2: SELF_AAMP_JOIN($\mathfrak{D}^{\mathcal{T}}$, m, kNN)

Input: The target time series data base ($\mathfrak{D}^{\mathcal{T}}$) and query time series (Q)
Output: A matrix profile (P_Q) and associated matrix profile index (I_Q)

1 $n_{\mathcal{T}} \leftarrow length(\mathcal{T})$; $excZone \leftarrow \frac{m}{2}$ ▷ *get the length of time series $\mathcal{T}$ and Q*
2 $Idxs_{\mathcal{T}} \leftarrow n_{\mathcal{T}} - m + 1$ ▷ *get the total number of sub-sequences in Q*
3 $P_{\mathcal{T}} \leftarrow ((kNN+1) \times Idx_{\mathcal{T}} \times d)\ array$ ▷ *it's a 3D matrix, initialized with infinity*
4 $I_{\mathcal{T}}, \mathcal{M}_{\mathcal{T}}, \mathfrak{D}_{\mathcal{T}}, \mathfrak{d}_{\mathcal{T}}, \mathfrak{I}_{\mathcal{T}} \leftarrow ((kNN+1) \times Idx_{\mathcal{T}} \times d)\ array$ ▷ *these are 3D matrix, initialized with zeros*
5 $iCnt \leftarrow 1$
6 **for** $iJump \leftarrow 1$ ***to*** $(n_{\mathcal{T}} - m)$ **do**
7 **if** $iJump > excZone$ **then**
8 $\mathcal{K}_{\mathcal{T}} \leftarrow (Idx_{\mathcal{T}} \times d)\ array \quad \mathcal{S}_{\mathcal{T}} \leftarrow (Idx_{\mathcal{T}} \times d)\ array$
9 **for** $iDim \leftarrow 1$ ***to*** d **do**
10 $distVal \leftarrow \sum(\mathcal{T}[iJump+1 : (iJump+1)+m-1][iDim] - \mathcal{T}[1:1+m-1][iDim])^2$ ▷ *compute distance with 1^{st} query sub-sequence and target sub-sequence*
11 $distVal \leftarrow |distVal|$
12 $\mathcal{K}_{\mathcal{T}}[1][iDim] \leftarrow distVal; \quad \mathcal{K}_{\mathcal{T}}[iJump+1][iDim] \leftarrow distVal$
13 $\mathcal{S}_{\mathcal{T}}[1][iDim] \leftarrow iJump+1; \quad \mathcal{S}_{\mathcal{T}}[iJump+1][iDim] \leftarrow 1; \quad \mathfrak{E} \leftarrow n_{\mathcal{T}} - m - iJump + 1$
14 **for** $ii \leftarrow 2$ ***to*** $\mathfrak{E}$ **do**
15 $tStart \leftarrow iJump + ii$
16 **for** $iDim \leftarrow 1$ ***to*** d **do**
17 $part_1 \leftarrow (\mathcal{T}[ii-1][iDim] - \mathcal{T}[tStart-1][iDim])^2$
18 $part_2 \leftarrow (\mathcal{T}[ii+m-1][iDim] - Q[tStart+m-1][iDim])^2$
19 $distVal \leftarrow \sum distVal - part_1 + part_2$ ▷ *compute the incremental distance*
20 $distVal \leftarrow |distVal|; \quad \mathcal{K}_{\mathcal{T}}[1][iDim] \leftarrow distVal; \quad \mathcal{K}_{\mathcal{T}}[tStart][iDim] \leftarrow distVal$ ▷ *store the square rooted distance*
21 $\mathcal{S}_{\mathcal{T}}[1][iDim] \leftarrow tStart; \quad \mathcal{S}_{\mathcal{T}}[tStart][iDim] \leftarrow ii$ ▷ *store the index*
22 **if** $iCnt <= kNN$ **then**
23 $P_Q[iCnt][1:Idx_{\mathcal{T}}][1:d] \leftarrow \sqrt{\mathcal{K}_{\mathcal{T}}[1:Idx_{\mathcal{T}}][1:d]}$; $I_Q[iCnt][1:Idx_{\mathcal{T}}][1:d] \leftarrow \mathcal{S}_{\mathcal{T}}[1:Idx_{\mathcal{T}}][1:d]$
24 **else**
25 $P_Q[kNN+1][1:Idx_{\mathcal{T}}][1:d] \leftarrow \sqrt{\mathcal{K}_{\mathcal{T}}[1:Idx_{\mathcal{T}}][1:d]}$ ▷ *store the distance*
26 $I_Q[kNN+1][1:Idx_{\mathcal{T}}][1:d] \leftarrow \mathcal{S}_{\mathcal{T}}[1:Idx_{\mathcal{T}}][1:d]$ ▷ *store the sub-sequence's indexes*
27 $[\ P_{\mathcal{T}},\ \mathfrak{D}_{\mathcal{T}},\ \mathfrak{d}_{\mathcal{T}},\ \mathfrak{I}_{\mathcal{T}},\ cumDistAll\ ] \leftarrow FunctionForElsePart\_1(i, Idx_{\mathcal{T}}, P_{\mathcal{T}},\ \mathfrak{D}_{\mathcal{T}},\ \mathfrak{d}_{\mathcal{T}},\ \mathfrak{I}_{\mathcal{T}},\ kNN)$
28 $[\ \mathfrak{D}_{\mathcal{T}},\ \mathfrak{d}_{\mathcal{T}},\ \mathfrak{I}_{\mathcal{T}}\ ] \leftarrow FunctionForElsePart\_3\ (\ Idx_{\mathcal{T}},\ cumDistAll,\ \mathfrak{D}_{\mathcal{T}},\ \mathfrak{d}_{\mathcal{T}},\ \mathfrak{I}_{\mathcal{T}}, kNN\ )$ ▷ *call this function*
29 $iCnt++$
30 **return** $P_{\mathcal{T}}[1:kNN][1:Idx_{\mathcal{T}}][1:d] \leftarrow \mathfrak{D}_{\mathcal{T}}[1:kNN][1:Idx_{\mathcal{T}}][1:d]$ ▷ *3D array of Matrix Profile*
31 **return** $I_{\mathcal{T}}[1:kNN][1:Idx_{\mathcal{T}}][1:d] \leftarrow \mathfrak{d}_{\mathcal{T}}[1:kNN][1:Idx_{\mathcal{T}}][1:d]$ ▷ *3D array of Index profile*
32 **return** $\mathcal{M}_{\mathcal{T}}[1:kNN][1:Idx_{\mathcal{T}}][1:d] \leftarrow \mathfrak{I}_{\mathcal{T}}[1:kNN][1:Idx_{\mathcal{T}}][1:d]$ ▷ *3D array of dimensions*

line 6, we iteratively jump $n_{\mathcal{T}} - m$ number of times and for each jump, we create two arrays $\mathcal{K}_{\mathcal{T}}$ and $\mathcal{S}_{\mathcal{T}}$ and initialized them with zeros. Then we calculate the classical euclidean distance between the 1^{st} sub-sequence and the $iJump + 1^{th}$ sub-sequence in line 10 by iterating over all the dimensions (in line 9). The calculated distance value ($distVal$) is then stored at 1^{st} and $iJump+1^{th}$ indexes of $\mathcal{K}_{\mathcal{T}}$ matrix and along with that the indexes are also saved in $\mathcal{S}_{\mathcal{T}}$ matrix. After that the distance between other sub-sequences in each dimensions are iteratively and incrementally calculated in Line 14-21. In each iteration of Line 14, we incrementally calculate the distance between ii^{th} and $tStart^{th}$ sub-sequence. In line 17, we operate on the 1^{st} elements of two previous sub-sequences i.e. $ii - 1^{th}$ and $tStart - 1^{th}$ sub-sequences (remember ii starts from 2) whereas in line 18, we operate over the last elements of current sub-sequences i.e. ii^{th} and $tStart^{th}$ sub-sequences. Then the distance value is calculated in Line 19 and it is stored in $\mathcal{K}_{\mathcal{T}}$ variable at the ii^{th} and $tStart^{th}$ indexes (line 20). The corresponding indexes are also stored in $\mathcal{S}_{\mathcal{T}}$ variable in Line 21. Remember that the objective is to calculate the distance between all the sub-sequence of $\mathcal{T}$ against every sub-sequence of $\mathcal{T}$ and finally keep top kNN matches of all the sub-sequences in the variable $P_{\mathcal{T}}$. We can visualize the same pictorial representation of $P_{\mathcal{T}}$, shown in Fig. 1a where each row (X axis) represents the sub-sequences of $\mathcal{T}$ and matches along the dimensions of time series (Y axis) are stored in column wise. The kNN matches for each query is as usual stored along Z axis.

We continue to store the distances like this until $iCnt \leq kNN$ i.e. until k number of diagonal shifts and the corresponding indexes. For sub-sequences index: $iCnt > kNN$ onwards are stored in $kNN + 1^{th}$ positions of $P_{\mathcal{T}}$ matrix (Line 25-26). After that the remaining operation is performed in function *FunctionForElsePart_1()* (which is explained before). After completing the execution of *FunctionForElsePart_1()* function, we call the function *FunctionForElsePart_3()* in Line 28 to perform the remaining operations. At the end, the computed matrices i.e. P_Q, I_Q and $\mathcal{M}_Q$ are returned as the results.

5.3 kNN similarity search for multi-dimensional time series using ACAMP algorithm

The ACAMP algorithm (explained in detail in) is also capable to perform similarity search by incremental distance calculation and like *STOMP* based technique, it uses z-normalized euclidean distance for the similarity search. But this algorithm is computationally faster than the *STOMP* technique. In the following section, we have explained the way to use this algorithm for the similarity search for multi-dimensional data. The pseudo code of the proposed algorithms is shown in Algorithm 3. The length of $\mathcal{T}$ and total number of possible sub-sequence of $\mathcal{T}$ are calculated in $n_{\mathcal{T}}$ and $Idx_{\mathcal{T}}$ variable respectively (Line 1). Then the mean and standard deviation of all the sub-sequences of $\mathcal{T}$ for all the existing dimensions are calculated in Line $2-3$ by using *ComputeMeanStd()* function (to get the details of this function, please see Appendix B of our previous article in [13]). Then the $3D$ matrices i.e. $P_{\mathcal{Q}}$, $I_{\mathcal{Q}}$, $\mathcal{M}_{\mathcal{Q}}$, $\mathfrak{D}_{\mathcal{Q}}$, $\mathfrak{d}_{\mathcal{Q}}$, $\mathfrak{I}_{\mathcal{Q}}$ are initialized with ∞ and zeros respectively.

In line 5, we iteratively jump $n_{\mathcal{T}} - m$ number of times and for each jump, we create three arrays $\mathcal{K}_{\mathcal{T}}$ and $\mathcal{S}_{\mathcal{T}}$ and $\mathcal{F}$ then initialize them with zeros. Then we calculate the z-normalized euclidean distance between the 1^{st} sub-sequence and the $iJump + 1^{th}$ sub-sequence in line 11 by iterating over all the dimensions in line 8. The calculated distance value $(distVal)$ is then stored at 1^{st} and $iJump{+}1^{th}$ indexes of $\mathcal{K}_{\mathcal{T}}$ matrix and along with that the indexes are also saved in $\mathcal{S}_{\mathcal{T}}$ matrix (line 13). The dot product between two sub-sequences i.e. 1^{st} and $iJump + 1^{th}$ sub-sequences are stored in $\mathfrak{P}$ (line 11) which is saved in $\mathcal{F}$ (line 13) for future computations. After that the distance between other sub-sequences in each dimensions are iteratively calculated in line 15-21. For each iteration in line 15, we incrementally calculate the distance between ii^{th} and $tStart^{th}$ sub-sequence. In line 18, we operate on the 1^{st} elements of two previous sub-sequences i.e. $ii - 1^{th}$ and $tStart - 1^{th}$ sub-sequences (remember ii starts from 2) whereas in line 19, we operate over the last elements of current sub-sequences i.e. ii^{th} and $tStart^{th}$ sub-sequences. Then the z-normalized distance value is calculated in line 21 by using *pearson correlation coefficient based z-normalized distance* calculation technique and it is stored in $\mathcal{K}_{\mathcal{T}}$ variable at the ii^{th} and $tStart^{th}$ indexes (line 22). The corresponding indexes are also stored in $\mathcal{S}_{\mathcal{T}}$ variable in line 23.

We continue to store the distances like this until $iCnt \leq kNN$ i.e. until k number of diagonal shifts and the corresponding indexes. For sub-sequences index: $iCnt > kNN$ onwards are stored in $kNN + 1^{th}$ positions of $P_{\mathcal{T}}$ matrix (Line 27-28). After that the remaining operation is performed in function *FunctionForElsePart_1()* (which is explained before). After completing the execution of *FunctionForElsePart_1()* function, we call the function *FunctionForElsePart_3()* in line 30 to perform the remaining operations. At the end, the computed matrices i.e. P_Q, I_Q and $\mathcal{M}_Q$ are returned as the results (line 32-34).

6 Constrained Search

As introduced in [1], that there are two kinds of constraints exists in multi-dimensional motifs searches: *exclusion* and *inclusion*. The *exclusion* puts a constraint to "blacklist" a predefined number of dimensions from the calculation of search which simply means that no motifs can span into the excluded dimensions. Whereas, the term *inclusion* puts a constraint which creates an obligation to include a set of dimensions during the computation of motif, hence all motifs must span these pre-defined dimensions. Like the *mSTAMP* algorithm, the implementation of *exclusion* can be simply achieved by removing the pre-defined dimensions. Whereas, the *inclusion* is achieved by moving the distance computed by using whitelisted dimensions up-to the front then in the same way a column wise-ascending sort is performed.

There are several application of this *constrained search* property in various domains. In [1], the authors has provided a nice example in the medical domain where for *sleep study*, the cardiologist and neurologist would need to exclude certain dimensions of data. Other domain experts may have the similar kinds of requirements (including and/or excluding some dimensions) for the analysis of time series signal in their respective domains.

7 Unconstrained Search

Sometime it is possible that the user knows (even if approximately) the expected motif's dimensionality of patterns in her domain. In that case it is easier and we can simply put a limit on number of dimensionality to be used during the computation of distance for motif search. However, it is also possible that the user has little idea regarding the reasonable dimensions of the motifs in the time series. In such cases, it is needed to perform an unconstrained search where the constrained such as *inclusion* & *exclusion* of dimensions and number of dimensions to be considered as motifs are not applied. Hence, like *mSTAMP* algorithm, our proposed approach is also able to consider multidimensional motif on k dimensions out of the full d dimensional space; where $k << d$ and k is an user given input. Hence, like *mSTAMP* algorithm, here our objective is to search for motifs in all possible dimensions of a given multi-dimensional time series and finally the objective is able to select best motif out of all possible combination of k dimensions.

8 Exploring Anytime Property of Multi-dimensional Motifs Discovery Algorithms

Algorithm 3: SELF_ACAMP_JOIN($\mathfrak{D}^T$, Q, m)

Input: The target time series data base ($\mathfrak{D}^T$)
Output: A matrix profile (P_Q) and associated matrix profile index (I_Q)

$n_{\mathcal{T}} \leftarrow length(\mathcal{T})$ $\quad Idxs_{\mathcal{T}} \leftarrow n_{\mathcal{T}} - m + 1$
for $iDim \leftarrow 1$ ***to*** d **do**
  $[\ \mu_{\mathcal{T}}[iDim], \sigma_{\mathcal{T}}[iDim]\ ] \leftarrow ComputeMeanStd(\mathcal{T})$
$P_{\mathcal{T}} \leftarrow ((kNN+1) \times Idx_{\mathcal{T}} \times d)\ array$; $\quad I_{\mathcal{T}}, \mathcal{M}_{\mathcal{T}}, \mathfrak{D}_{\mathcal{T}}, \mathfrak{d}_{\mathcal{T}}, \mathfrak{I}_{\mathcal{T}} \leftarrow ((kNN+1) \times Idx_{\mathcal{T}} \times d)\ array$
for $iJump \leftarrow 0$ ***to*** $(n_T - m)$ **do**
  if $iJump > excZone$ **then**
    $\mathcal{K}_{\mathcal{T}} \leftarrow (Idx_{\mathcal{T}} \times d)$; $\quad \mathcal{S}_{\mathcal{T}} \leftarrow (Idx_{\mathcal{T}} \times d)$; $\quad \mathcal{F} \leftarrow (1 \times d)\ array$
    for $iDim \leftarrow 1$ ***to*** d **do**
      $qSubSeq \leftarrow \mathcal{T}[1 : 1 + m - 1][iDim]$
      $tSubSeq \leftarrow \mathcal{T}[iJump + 1 : (iJump + 1) + m - 1][iDim]$
      $[distVal, \mathfrak{P}] \leftarrow CalcZnormDist\ (\ qSubSeq, tSubSeq, m, \mu_{\mathcal{T}}[1][iDim],$
       $\sigma_{\mathcal{T}}[1][iDim], \mu_{\mathcal{T}}[1][iDim], \sigma_{\mathcal{T}}[1][iDim]\ )$; $\quad distVal \leftarrow \sqrt{|distVal|}$
      $\mathcal{K}_{\mathcal{T}}[1][iDim] \leftarrow distVal$; $\quad \mathcal{K}_{\mathcal{T}}[iJump + 1][iDim] \leftarrow distVal$
      $\mathcal{S}_{\mathcal{T}}[1][iDim] \leftarrow iJump + 1$; $\quad \mathcal{S}_{\mathcal{T}}[iJump + 1][iDim] \leftarrow 1$; $\quad \mathcal{F}[1][iDim] \leftarrow \mathfrak{P}$
    $\mathfrak{E} \leftarrow n_T - m - iJump + 1$
    for $ii \leftarrow 2$ ***to*** $\mathfrak{E}$ **do**
      $tStart \leftarrow iJump + ii$
      for $iDim \leftarrow 1$ ***to*** d **do**
        $part_1 \leftarrow (\mathcal{T}[ii - 1][iDim] - \mathcal{T}[tStart - 1][iDim])$
        $part_2 \leftarrow (\mathcal{T}[ii + m - 1][iDim] - \mathcal{T}[tStart + m - 1][iDim])$
        $\mathcal{F}[1][iDim] \leftarrow \mathcal{F}[1][iDim] - part_1 + part_2$ ▷ *compute the incremental distance*
        $distVal \leftarrow 2 \times \left[m - \frac{\mathfrak{P} - (m \times \mu_{\mathcal{T}}[ii][iDim] \times \mu_{\mathcal{T}}[tStart][iDim])}{\sigma_{\mathcal{T}}[ii][iDim] \times \sigma_{\mathcal{T}}[tStart][iDim]}\right]$; $\quad distVal \leftarrow |distVal|$
        $\mathcal{K}_{\mathcal{T}}[ii][iDim] \leftarrow distVal$; $\quad \mathcal{K}_{\mathcal{T}}[tStart][iDim] \leftarrow distVal$
        $\mathcal{S}_{\mathcal{T}}[ii][iDim] \leftarrow tStart$; $\quad \mathcal{S}_{\mathcal{T}}[tStart][iDim] \leftarrow ii$
    if $iCnt <= kNN$ **then**
      $P_{\mathcal{T}}[iCnt][1 : Idx_{\mathcal{T}}][1 : d] \leftarrow \sqrt{\mathcal{K}_{\mathcal{T}}[1 : Idx_{\mathcal{T}}][1 : d]}$
      $I_{\mathcal{T}}[iCnt][1 : Idx_{\mathcal{T}}][1 : d] \leftarrow \mathcal{S}_{\mathcal{T}}[1 : Idx_{\mathcal{T}}][1 : d]$
    else
      $P_{\mathcal{T}}[kNN + 1][1 : Idx_{\mathcal{T}}][1 : d] \leftarrow \sqrt{\mathcal{K}_{\mathcal{T}}[1 : Idx_{\mathcal{T}}][1 : d]}$ ▷ *store the distance*
      $I_{\mathcal{T}}[kNN + 1][1 : Idx_{\mathcal{T}}][1 : d] \leftarrow \mathcal{S}_{\mathcal{T}}[1 : Idx_{\mathcal{T}}][1 : d]$ ▷ *store the sub-sequence's indexes*
      $[P_{\mathcal{T}}, \mathfrak{D}_{\mathcal{T}}, \mathfrak{d}_{\mathcal{T}}, \mathfrak{I}_{\mathcal{T}}, dAll] \leftarrow FunctionForElsePart\_1(iCnt, Idx_{\mathcal{T}}, P_{\mathcal{T}}, \mathfrak{D}_{\mathcal{T}}, \mathfrak{d}_{\mathcal{T}}, \mathfrak{I}_{\mathcal{T}}, kNN)$
      $[\ \mathfrak{D}_{\mathcal{T}},\ \mathfrak{d}_{\mathcal{T}},\ \mathfrak{I}_{\mathcal{T}}\ ] \leftarrow FunctionForElsePart\_3(Idx_{\mathcal{T}}, dAll, \mathfrak{D}_{\mathcal{T}}, \mathfrak{d}_{\mathcal{T}}, \mathfrak{I}_{\mathcal{T}}, kNN)$
    $iCnt++$
return $P_{\mathcal{T}}[1 : kNN][1 : Idx_{\mathcal{T}}][1 : d] \leftarrow \mathfrak{D}_{\mathcal{T}}[1 : kNN][1 : Idx_{\mathcal{T}}][1 : d]$ ▷ *3D array of Matrix Profile*
return $I_{\mathcal{T}}[1 : kNN][1 : Idx_{\mathcal{T}}][1 : d] \leftarrow \mathfrak{d}_{\mathcal{T}}[1 : kNN][1 : Idx_{\mathcal{T}}][1 : d]$ ▷ *3D array of Index profile*
return $\mathcal{M}_{\mathcal{T}}[1 : kNN][1 : Idx_{\mathcal{T}}][1 : d] \leftarrow \mathfrak{I}_{\mathcal{T}}[1 : kNN][1 : Idx_{\mathcal{T}}][1 : d]$ ▷ *3D array of dimensions*

Here in this section, we will talk about the ways to perform anytime motifs discovery of the above algorithms.

9 INCREMENTALLY MAINTAINING THE MULTI-DIMENSIONAL MATRIX PROFILE

Here in this section, we will talk about the algorithms or the modification of the above mentioned algorithms for handling streaming data of a time series.

Acknowledgment

We greatly acknowledge the funding from *Safran Data Analytics Lab*. The authors are grateful to Inria Sophia Antipolis - Méditerranée "Nef" computation cluster for providing resources and support.

References

[1] C.-C. M. Yeh, N. Kavantzas, and E. Keogh, "Matrix Profile VI: Meaningful Multidimensional Motif Discovery," *2017 IEEE International Conference on Data Mining (ICDM)*, pp. 565–574, 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8215529/ 2, 3, 10

[2] B. Hu, Y. Chen, J. Zakaria, L. Ulanova, and E. Keogh, "Classification of multi-dimensional streaming time series by weighting each classifier's track record," in *Proceedings - IEEE International Conference on Data Mining, ICDM*, 2013. 2

[3] C. C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, Z. Zimmerman, D. F. Silva, A. Mueen, and E. Keogh, *Time series joins, motifs, discords and shapelets: a unifying view that exploits the matrix profile*. Springer US, 2018, vol. 32, no. 1. 2

[4] B. Campana, T. Rakthanmanon, G. Batista, A. Mueen, Q. Zhu, E. Keogh, B. Westover, and J. Zakaria, "Searching and mining trillions of time series subsequences under dynamic time warping," p. 262, 2012. 2

[5] A. Mueen, H. Hamooni, and T. Estrada, "Time Series Join on Subsequence Correlation," *Proceedings - IEEE International Conference on Data Mining, ICDM*, vol. 2015-Janua, no. January, pp. 450–459, 2014. 2

[6] Y. Zhu, C.-C. M. Yeh, Z. Zimmerman, K. Kamgar, and E. Keogh, "Matrix Profile XI: SCRIMP++: Time Series Motif Discovery at Interactive Speeds," in *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, nov 2018, pp. 837–846. 2

[7] A. Balasubramanian, J. Wang, and B. Prabhakaran, "Discovering Multidimensional Motifs in Physiological Signals for Personalized Healthcare," *IEEE Journal on Selected Topics in Signal Processing*, vol. 10, no. 5, pp. 832–841, 2016. 3

[8] E. Berlin and K. Van Laerhoven, "Detecting leisure activities with dense motif discovery," *UbiComp'12 - Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pp. 250–259, 2012. 3

[9] A. Vahdatpour, N. Amini, and M. Sarrafzadeh, "Toward unsupervised activity discovery using multi-dimensional motif detection in time series," *IJCAI International Joint Conference on Artificial Intelligence*, pp. 1261–1266, 2009. 3

[10] Y. Tanaka, K. Iwamoto, and K. Uehara, "Discovery of time-series motif from multi-dimensional data based on MDL principle," *Machine Learning*, vol. 58, no. 2-3, pp. 269–300, 2005. 3

[11] D. Minnen, C. Isbell, I. Essa, and T. Starner, "Detecting subdimemslomal motifs: An efficient algorithm for generalized multivariate pattern discovery," *Proceedings - IEEE International Conference on Data Mining, ICDM*, no. October, pp. 601–606, 2007. 3

[12] Y. Gao and J. Lin, "Discovering subdimensional motifs of different lengths in large-scale multivariate time series," *Proceedings - IEEE International Conference on Data Mining, ICDM*, vol. 2019-Novem, pp. 220–229, 2019. 3

[13] T. Mondal, R. Akbarinia, and F. Masseglia, "Matrix Profile Based kNN Search over Large Time Series," vol. 00, pp. 1–37, 2020. [Online]. Available: http://arxiv.org/abs/1901.0570812 4, 10, 13

Tanmoy Mondal did his Ph.D from Ecole Polytechnique de l'Université de Tours (EPU), France in 2015. Before his PhD, he worked at several industries and premier R&D centers as a researcher. Currently, he is doing Post-Doc at IMT Atlantique, France. His research interests include time series analysis, pattern recognition, image processing, and computer vision.

Reza Akbarinia is a research scientist at Inria. He received his Ph.D. degree in Computer Science from the University of Nantes in 2007. His research focuses on data management and analysis in large-scale distributed systems (P2P, grid, Cloud) and data privacy. He has authored and co-authored two books and several technical papers in main DB conferences and journals. He has served as PC member in several conferences, such as SIGMOD, VLDB, ICDE, EDBT, CIKM, etc.

Florent Masseglia is a scientific researcher in computer science at Inria since 2002. He works in Montpellier, in the Zenith team of Inria, on the analysis of very large scientific data. These data, derived from observations, experiments and simulation are indeed complex, often very large, and are at the heart of important issues to better understand the studied domains (agronomy, biology, medicine).

SUPPLEMENTARY MATERIALS (SM)

SM: .1 kNN similarity search for independent join

Matching of two independent multi-dimensional time series named as *Query* (Q) and *Target* time series (T) are performed in the following manner. Let's consider an example where in a time series database $\mathfrak{D}^T$, there are $\mathfrak{x}$ number of time series of different lengths : $\{\mathfrak{t}_1, \mathfrak{t}_2, \mathfrak{t}_3, ..., \mathfrak{t}_\mathfrak{x}\} \in \mathfrak{D}^T$. In another time series data base D^Q, let's say there are $\mathfrak{y}$ number of time series exits, where $\mathfrak{x} >>> \mathfrak{y}$. The goal is to find closest match of all the sub-sequences in Q, where, Q is any particular time series from the database D^Q and the matches should come from the time series which exits in database $\mathfrak{D}^T$. By concatenating all the time series in $\mathfrak{D}^T$, we can obtain a big time series $\mathcal{T}$ which is named as *target time series*. The following Algorithm 4 is proposed which basically computes the distance of each sub-sequence of $\mathcal{T}$ with all the sub-sequences of query time series Q and based on this repetitive process, we find the best matches of each query sub-sequences.

The pseudo code of the proposed algorithms is shown in Algorithm 4. All the individual time series from database $\mathfrak{D}^T$ are sequentially concatenated to form a concatenated time series $\mathcal{T}$ (Line 2) and the information e.g. start and end indexes/locations of an individual time series when it is concatenated to generate $\mathcal{T}$ are saved in $Info_\mathcal{T}$. Also the file name or index of the individual time series are also saved in $Info_\mathcal{T}$. The length of $\mathcal{T}$ and total number of possible sub-sequence of $\mathcal{T}$ are calculated in $n_\mathcal{T}$ and $Idxs_\mathcal{T}$ variable respectively (Line 4). Along with that the length of time series Q and number of total sub-sequences in Q are also calculated in n_Q and $Idxs_Q$ variable also computed in line 4. The mean $(\mu_\mathcal{T})$ and standard deviation $(\sigma_\mathcal{T})$ of all the sub-sequences of $\mathcal{T}$ and the mean (μ_Q) and standard deviation (σ_Q) of all the sub-sequences of Q are calculated by iterating over the dimensions of $\mathcal{T}$ (Line $5-6$). After obtaining the very first sub-sequence of Q in Line 7, the distance between first sub-sequence and all the other sub-sequences are calculated in Line 9 by using the function *MASS()* (to get the details of this function, please see Appendix C of our previous article in [13]). The arguments passed in this function are the fist sub-sequence of $\mathcal{T}$ (i.e. $querySubSeq_1$), mean and standard deviation of $querySubSeq_1$ i.e. $\mu_Q[1]$, $\sigma_Q[1]$, complete time series $\mathcal{T}$ and the mean and standard deviations of all the sub-sequence of $\mathcal{T}$ i.e. $\mu_\mathcal{T}[1 : Idx_\mathcal{T}]$, $\sigma_\mathcal{T}[1 : Idx_\mathcal{T}]$ respectively. The dot product between the $querySubSeq_1$ and all other sub-sequences of $\mathcal{T}$ is stored in $QT_{initial}$ (Line 10). In Line 11, we perform the remaining operations in *Obtain_kNNMatches()* function. The *Obtain_kNNMatches()* function is explained in detail in Section 5.1. The arguments passed in this function are; the complete time series $\mathcal{T}$, total number of sub-sequences in $\mathcal{T}$ i.e. $Idx_\mathcal{T}$, the query time series i.e. Q, total number of sub-sequences in Q i.e. Idx_Q, mean and standard deviation of all the sub-sequences in all dimensions $\mathcal{T}$ i.e. $\mu_\mathcal{T}$ and $\sigma_\mathcal{T}$, mean and standard deviation of all the sub-sequences in Q i.e. μ_Q σ_Q and total number of dimensions i.e. d. At the end, the computed matrices i.e. P_Q, I_Q and $\mathcal{M}_Q$ are returned as the results.

SM: I kNN SIMILARITY SEARCH FOR INDEPENDENT JOIN USING AAMP ALGORITHM

In this section, we explain the technique to perform the similarity search between *Query* (Q) and *Target* time series $(\mathcal{T})$ like the one is performed in Appendix SM: .1. The algorithm initiates as usual like the Algorithm 4. Line 1-4 is self explanatory and has been described before. In line 6, we iteratively perform diagonal jump of $Idxs_\mathcal{T} - 1$ number of times and for each jump, we calculate the distance between subsequent query and target sub-sequences. For example, when $iJump = 0$ (i.e. diagonal jump equals to zero), the distance is computed between $QSSq1$ v/s $TSSq1$ followed by distance computation between $QSSq2$ v/s $TSSq2$ etc. (follow the yellow color cells and # symbol in Fig. 3). Then for $iJump = 1$, the distance is computed between $QSSq1$ v/s $TSSq2$ followed by distance computation between $QSSq2$ v/s $TSSq3$ etc. (follow the green color cells and $ symbol in Fig. 3) and so on. In this way, $iJump$ iterate until $n_T - m$ i.e we perform $n_T - m$ numbers of diagonal jumps. But until $n_T - n_Q$ numbers of diagonal jumps (a visual example is shown in Fig. 3 by considering $Idxs_T = 21$ and $Idxs_Q = 8$), we can compute the distance between all the subsequent query sub-sequences and corresponding target sub-sequences (follow maroon color cells and æ symbol in Fig. 3 where distances are calculated between $QSSq1$ v/s $TSSq14$, $QSSq2$ v/s $TSSq15$ etc.). If we take any more jumps after that then we can't compute the distance between all the query sub-sequences and corresponding target sub-sequences. This rationale is implemented in line 7-10 of Algorithm 5. If $iJump \leq (n_T - n_Q)$ then $\mathfrak{E}$ is equal to n_Q that means we can compute distance for all the query sub-sequences otherwise $\mathfrak{E}$ is taken as $Idxs_\mathcal{T} - iJump$ which means we can calculate distance for the number of query sub-sequences, equals to the remaining number of sub-sequences, obtained by subtracting/removing the already taken number of jumps $(iJumps)$ from total number of target sub-sequences $(Idxs_\mathcal{T})$ in each iteration (follow the bottom yellow colored triangular regions in Fig. 3).

In line 11, we check whether $iJump + 1 \leq kNN$ then $pIdx$ is taken as $iJump + 1$ (hence the matches will be saved along Z axis at $iJump + 1^{th}$ row directly) otherwise it is taken as $kNN + 1$ (matches will be saved along Z axis at $kNN + 1^{th}$ row). Now the distance between the 1^{st} query sub-sequence and $iJump + 1^{th}$ sub-sequence of target time series is calculated in line 17 and then the square rooted distance value and $iJump + 1^{th}$ index value are stored at the $pIdx^{th}$ index of $\mathfrak{D}_Q$ and $\mathfrak{I}_Q$ arrays. After that the distance between other sub-sequences in each dimensions are iteratively calculated in Line 20-26. In each iteration of Line 20, we incrementally calculate the distance between ii^{th} sub-sequence of Q and $tStart^{th}$ sub-sequence of $\mathcal{T}$. Like before, in line 22, we operate on the 1^{st} elements of two previous sub-sequences i.e. $ii - 1^{th}$ and $tStart - 1^{th}$ sub-sequences (remember ii starts from 2) whereas in line 23, we operate over the last elements of current sub-sequences i.e. ii^{th} and $tStart^{th}$ sub-sequences of $\mathcal{T}$ and Q respectively. Then the distance value is calculated in Line 24 and the square rooted distance value and $tStart^{th}$ index

Algorithm 4: INDEPENDENTTIMESERIESJOIN($\mathfrak{D}^T$, Q, m)

Input: The target time series data base ($\mathfrak{D}^T$) and query time series (Q)
Output: A matrix profile (P_Q) and associated matrix profile index (I_Q)

1 **for** $iSeries \leftarrow 1$ ***to*** $length(\mathfrak{D}^T)$ **do**
2 $\quad \mathcal{T} \leftarrow [\mathcal{T}, \mathfrak{D}^T[iSeries]]$ ▷ *concatenate individual time series from the data base* $\mathfrak{D}^T$
3 $\quad Info_{\mathcal{T}} \leftarrow [startIdx, endIdx, fileName]$ ▷ *store the start, end indexes and the file name after concatenating an individual time series in* $\mathcal{T}$
4 $n_Q \leftarrow length(Q)$; $Idx_Q \leftarrow (n_Q - m + 1)$; $n_{\mathcal{T}} \leftarrow length(\mathcal{T})$; $Idx_{\mathcal{T}} \leftarrow n_{\mathcal{T}} - m + 1$
5 **for** $iDim \leftarrow 1$ ***to*** d **do**
6 $\quad [\mu_{\mathcal{T}}[iDim], \sigma_{\mathcal{T}}[iDim], \mu_Q[iDim], \sigma_Q[iDim]] \leftarrow ComputeMeanStd(\mathcal{T}, Q)$
7 $querySubSeq_1 \leftarrow Q[1 \text{ to } (1 + m - 1)][1 \text{ to } d]$ ▷ *get the* 1^{st} *sub-sequence from* $\mathcal{T}$
8 **for** $iDim \leftarrow 1$ ***to*** d **do**
9 $\quad [QT[iDim], D^{ignore}[iDim]] \leftarrow MASS(querySubSeq_1, \mu_Q[1][iDim], \sigma_Q[1][iDim], \mathcal{T}, \mu_{\mathcal{T}}[iDim], \sigma_{\mathcal{T}}[iDim])$ ▷ *apply* $MASS$ *to calculate distance between* 1^{st} *query sub-sequence and all the sub-sequence of* $\mathcal{T}$
10 $QT_{initial} \leftarrow QT$ ▷ *keeping a copy of the very first dot product*
11 $[P_{\mathcal{Q}},\ I_{\mathcal{Q}},\ \mathcal{M}_{\mathcal{Q}},\ \mathfrak{D}_{\mathcal{Q}},\ \mathfrak{d}_{\mathcal{Q}},\ \mathfrak{I}_{\mathcal{Q}}\ ] \leftarrow Obtain\_kNNMatches(\mathcal{T},\ Idx_{\mathcal{T}}\ , Q,\ Idx_Q\ , \mu_{\mathcal{T}}, \sigma_{\mathcal{T}}, \mu_Q, \sigma_Q, d)$ ▷ *call this function, see below*
12 **return** $P_Q[1:kNN][1:Idx_Q][1:d] \leftarrow \mathfrak{D}_Q[1:kNN][1:Idx_Q][1:d]$ ▷ *3D array of Matrix Profile*
13 **return** $I_Q[1:kNN][1:Idx_Q][1:d] \leftarrow \mathfrak{d}_Q[1:kNN][1:Idx_Q][1:d]$ ▷ *3D array of Index profile*
14 **return** $\mathcal{M}_Q[1:kNN][1:Idx_Q][1:d] \leftarrow \mathfrak{I}_Q[1:kNN][1:Idx_Q][1:d]$ ▷ *3D array of dimensions*

value are stored at the $pIdx^{th}$ index of $\mathfrak{D}_Q$ and $\mathfrak{I}_Q$ arrays (Line 25-26).

We continue like this until $iJump + 1 \leq kNN$ i.e. we have accumulated initial k number of matches for each sub-sequences of Q otherwise $maxEleFlag$ becomes $TRUE$ in line 15. Hence, in line 27, we check if $maxEleFlag$ is $TRUE$ then call the *FunctionForElsePart_1()* in line 28. After that the remaining operation is performed in line 29 by using the function *FunctionForElsePart_3()*. These two functions are explained before in detail (see Section 5.1 for detailed descriptions). Note the initial two arguments passed in *FunctionForElsePart_1()* function are $iJump + 1$ and $\mathfrak{E}$. The argument $iJump + 1$ is used to verify whether the value of $iJump + 1$ is less than user given kNN value or not whereas the 2^{nd} argument $\mathfrak{E}$ is used to operate only $\mathfrak{E}$ numbers of initial query sub-sequences. After completing the operations within these two functions, we call the *Independent_AAMP_Part_2()* function in line 30.

The calculation of distance between sub-sequences from Q and $\mathcal{T}$ are performed from right to left direction in *Independent_AAMP_Part_2()* function (these operations are visually represented as red colored region/cells at the top of Fig. 3. The operations for these cells or the distance computation between these sub-sequences of Q and $\mathcal{T}$ were not performed before). In line 1, we perform $Idxs_Q - 1$ number of jumps iteratively (notice that $iJump$ starts here from 1 instead of 0 because we want to perform distance computations of $QSSq8\ v/s\ TSSq7$, $QSSq7\ v/s\ TSSq6$ etc. instead of $QSSq8\ v/s\ TSSq8$, $QSSq7\ v/s\ TSSq7$ etc. which are already computed before) and in line 2-8, the distance between $tStart^{th}$ sub-sequence from $\mathcal{T}$ and $qStart^{th}$ sub-sequence from Q is calculated for each dimension and then the square rooted distance is saved in line 9 and the corresponding index

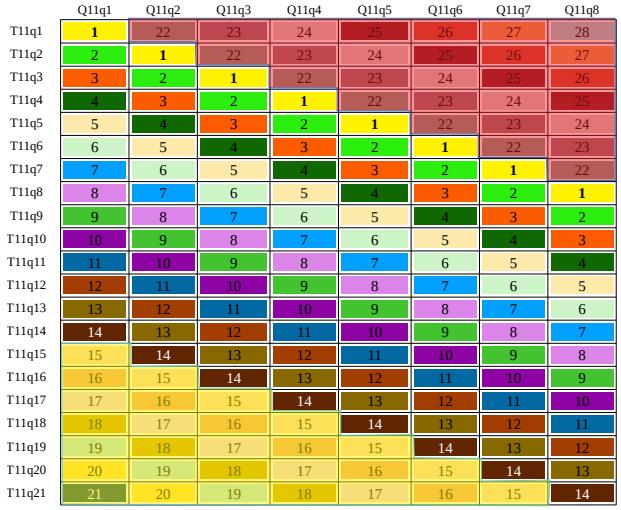

Figure 3: The visual representation of iterative matching between the sub-sequences of query and target time series.

i.e. $tStart$ is saved in line 10. After that the distance between $tStart^{th}$ sub-sequence from $\mathcal{T}$ and $qStart^{th}$ sub-sequence from Q are iteratively calculated in an incremental fashion. To calculate the distance incrementally, in line 14 we operate on the 1^{st} elements of two previous sub-sequences i.e. $tStart^{th}$ and $qStart^{th}$ elements (remember ii starts from 2) whereas in line 15, we operate over the last elements of current sub-sequences i.e. $tStart + m^{th}$ and $qStart + m^{th}$ elements. The distance value is then calculated in Line 16 and then the square rooted distance is saved in $\mathfrak{D}_Q$ variable at $kNN + 1^{th}$ index (line 17). The corresponding index i.e. $tStart$ is saved in $\mathfrak{I}_Q$

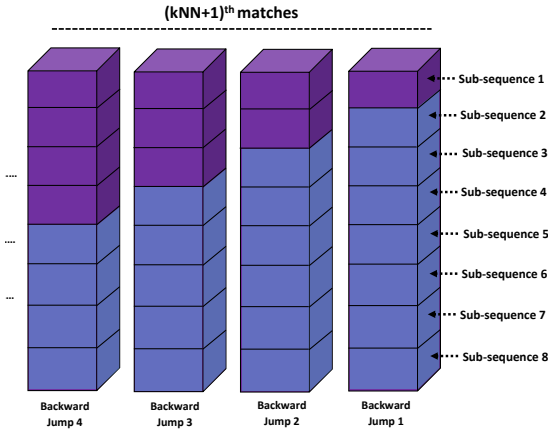

Figure 4: The matches stored at $kNN + 1^{th}$ column of P_Q matrix and considered query sub-sequences in each iteration of backward diagonal jump.

variable at $kNN + 1^{th}$ index in line 18. The indexes of query sub-sequences which are considered in each iteration are saved in $\mathfrak{V}$ which contains $\mathfrak{E}$ number of elements or indexes of query sub-sequences, for which the distances are computed in each iteration (of line 1). The visual representation of the query sub-sequences which are considered in each iteration are shown in Fig. 4, where it can be seen that in 1^{st} iteration, we operate on $QSSq8$-$QSSq2$ (follow the blue colored cells) and in 2^{nd} iteration, we operate on $QSSq8$-$QSSq2$ etc. (notice that the same visual representation is shown in top red colored region or cells in Fig. 3).

Hence by using $\mathfrak{V}$, we can easily choose the distances, stored at $kNN + 1^{th}$ column (along Z axis) of P_Q array and at the rows whose indexes can be obtained from $\mathfrak{V}$. These chosen distances are stored in M_{kNN+1} array in line 20 then this array is sorted column-wise in descending order and the sorted distances are stored in $distTemp_{kNN+1}$ array.

Whereas the sorted indexes are saved in $indxDim_{kNN+1}$ array. After that these $\mathfrak{E}$ number of distances in d dimensions are copied from $distTemp_{kNN+1}$ array into $\mathfrak{D}_Q$ array (line 24) whereas the sorted dimension's indexes from $indxDim_{kNN+1}$ array are copied in $\mathfrak{d}_Q$ array. As mentioned before that $\mathfrak{V}$ contains $\mathfrak{E}$ number of indexes of query sub-sequences whose $kNN + 1^{th}$ neighbors are updated and $indxDim_{kNN+1}$ array contains the sorted dimension's indexes of $kNN + 1^th$ column. Hence, by using these two arrays i.e. $\mathfrak{V}$ and $indxDim_{kNN+1}$, we copy the sorted indexes from I_Q array (corresponding to only those query sub-sequences whose indexes are mentioned in $\mathfrak{V}$) into $\mathfrak{I}_Q$ array. Then in line 29, we iterate over all the query sub-sequences whose indexes were stored in $\mathfrak{V}$ array. For each of such query sub-sequence, we compute the cumulative distance in $cumDistTemp$ array (line 31) by iteratively summing up distance values from $kNN + 1^{th}$ column of $\mathfrak{D}_Q$ array in all the d dimensions. Then this cumulative distance is divided by dimension's index i.e. $iDim$ in line 32 and saved in $dAll$ array at $kNN + 1^{th}$ column. Then the remaining operations are performed in *FunctionForElse_Part_3()* function (which is already explained in detail in Section 5.1) and the updated $\mathfrak{D}_Q$, $\mathfrak{d}_Q$, $\mathfrak{I}_Q$ arrays are obtained in line 33. These updated arrays are returned in line 34 and this way we complete all the operations of *Independent_AAMP_Part_2()* function. Then we return back at the line 30 of Algorithm 5 and finally the top kNN number of elements from the updated $\mathfrak{D}_Q$, $\mathfrak{d}_Q$, $\mathfrak{I}_Q$ arrays are copied into P_Q, I_Q and $\mathcal{M}_Q$ arrays respectively, which are returned in line $31 - 33$ of Algorithm 5.

SM: II kNN SIMILARITY SEARCH FOR INDEPENDENT JOIN USING ACAMP ALGORITHM

In this section, we explain the ACAMP technique to perform the similarity search between *Query* (Q) and *Target* time series $(\mathcal{T})$ like the one is performed in Appendix SM: I. Line 1-2 is self explanatory and are same as in Algorithm 5. In line 3-4, we calculate the mean and standard deviation of all the sub-sequences of $\mathcal{T}$ and Q are computed (similar to line 2-3 of Algorithm 3). Then line 5-17 remains same as the one in line 3-15 of Algorithm 5. After that, the *Z-Normalized euclidean* distance (similar as in line 11 of Algorithm 3) between the 1^{st} query sub-sequence and $iJump + 1^{th}$ sub-sequence of target time series is calculated in line 21 and then the square rooted distance value and $iJump + 1^{th}$ index value are stored at the $pIdx^{th}$ index of $\mathfrak{D}_Q$ and $\mathfrak{I}_Q$ arrays. After that the *Z-normalized euclidean* distance (similar as in line 21 of Algorithm 3) between other sub-sequences in each dimensions are iteratively and incrementally calculated by using *CalcZnormDist()* function in Line 24-29 (in similar way as the incremental distances are computed in line 20-24 of Algorithm 5). The following *CalcZnormDist()* function is used to calculate the $Z-$ normalized distance between sub-sequence $qSubSeq$ and $tSubSeq$. After that the same operation is performed in line 32-38 as the one is done in line 27-33. In this manner, finally we obtain P_Q, I_Q and $\mathcal{M}_Q$ arrays respectively.

Algorithm 5: INDEPENDENT_AAMP_JOIN($\mathfrak{D}^T$, Q, m)

Input: The target time series data base ($\mathfrak{D}^T$) and query time series (Q)
Output: A matrix profile (P_Q) and associated matrix profile index (I_Q)

$n_{\mathcal{T}} \leftarrow length(\mathcal{T})$ $n_Q \leftarrow length(Q)$ ▷ *get the length of time series $\mathcal{T}$ and Q*
$Idxs_Q \leftarrow n_Q - m + 1$; $Idxs_{\mathcal{T}} \leftarrow n_{\mathcal{T}} - m + 1$ ▷ *get the total number of sub-sequences in Q and $\mathcal{T}$*
$P_Q \leftarrow ((kNN + 1) \times Idx_Q \times d)\ array$ ▷ *it's a 3D matrix, initialized with infinity*
$I_Q, \mathcal{M}_Q, \mathfrak{D}_Q, \mathfrak{d}_Q, \mathfrak{I}_Q \leftarrow ((kNN + 1) \times Idx_Q \times d)\ array$ ▷ *these are 3D matrix, initialized with zeros*
$subSeqFlag \leftarrow (1 \times d)\ vector$ ▷ *it's a 1D horizontal Boolean vector, initialized with $FALSE$*
for $iJump \leftarrow 0$ ***to*** $(Idxs_{\mathcal{T}} - 1)$ **do**
  if $iJump \leq (Idxs_{\mathcal{T}} - Idxs_Q)$ **then**
    $\mathfrak{E} \leftarrow Idxs_Q$
  else
    $\mathfrak{E} \leftarrow Idxs_{\mathcal{T}} - iJump$
  if $(iJump + 1) \leq kNN$ **then**
    $pIdx \leftarrow iJump + 1$
  else
    $pIdx \leftarrow kNN + 1$
    $maxEleFlag \leftarrow TRUE$
  for $iDim \leftarrow 1$ ***to*** d **do**
    $distVal \leftarrow \sum(\mathcal{T}[iJump + 1 : iJump + m][iDim] - Q[1 : m][iDim])^2$ ▷ *compute distance with 1^{st} query sub-sequence and target sub-sequence*
    $\mathfrak{D}_Q[pIdx][1][iDim] \leftarrow \sqrt{distVal}$ ▷ *store the square rooted distance*
    $\mathfrak{I}_Q[pIdx][1][iDim] \leftarrow iJump + 1$ ▷ *store the index*
    for $ii \leftarrow 2$ ***to*** $\mathfrak{E}$ **do**
      $tStart \leftarrow iJump + ii$
      $part_1 \leftarrow (\mathcal{T}[tStart - 1][iDim] - Q[ii - 1][iDim])^2$
      $part_2 \leftarrow (\mathcal{T}[tStart + m - 1][iDim] - Q[ii + m - 1][iDim])^2$
      $distVal \leftarrow \sum distVal - part_1 + part_2$ ▷ *compute the incremental distance*
      $\mathfrak{D}_Q[pIdx][ii][iDim] \leftarrow \sqrt{distVal}$ ▷ *store the square rooted distance*
      $\mathfrak{I}_Q[pIdx][ii][iDim] \leftarrow tStart$ ▷ *store the index*
  if $maxEleFlag == TRUE$ **then**
    $[\ P_Q,\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q,\ dAll\ ] \leftarrow FunctionForElsePart\_1(\ iJump + 1,\ \mathfrak{E}, P_Q,\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q,\ kNN)$ ▷ *call this function*
    $[\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q, dAll\ ] \leftarrow FunctionForElsePart\_3(Idx_Q, dAll, \mathfrak{D}_{\mathcal{T}}, \mathfrak{d}_{\mathcal{T}}, \mathfrak{I}_{\mathcal{T}}, kNN)$
  $[\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q\ ] \leftarrow Independent\_AAMP\_Part\_2(\ \mathcal{T}, Q, \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q, n_Q, m, d, dAll)$
return $P_Q[1 : kNN][1 : Idx_Q][1 : d] \leftarrow \mathfrak{D}_Q[1 : kNN][1 : Idx_Q][1 : d]$ ▷ *3D array of Matrix Profile*
return $I_Q[1 : kNN][1 : Idx_Q][1 : d] \leftarrow \mathfrak{d}_Q[1 : kNN][1 : Idx_Q][1 : d]$ ▷ *3D array of Index profile*
return $\mathcal{M}_Q[1 : kNN][1 : Idx_Q][1 : d] \leftarrow \mathfrak{I}_Q[1 : kNN][1 : Idx_Q][1 : d]$ ▷ *3D array of dimensions*

Function `Independent_AAMP_Part_2` ($\mathcal{T}, Q, \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q, n_Q, m, d, dAll$) **:**

/* *the following code is executed when* $i > kNN$ *in Algorithm 4* */

for $iJump \leftarrow 1$ ***to*** $Idxs_Q - 1$ **do**
  $\mathfrak{E} \leftarrow (Idxs_Q) - iJump$
  $\mathfrak{V} \leftarrow (\mathfrak{E} \times 1)\ array$ ▷ *it's a 1D array, initialized with zeros*
  for $iDim \leftarrow 1$ ***to*** d **do**
    $tStart \leftarrow Idxs_Q - iJump$
    $qStart \leftarrow Idxs_Q$
    $\mathfrak{V}[1] \leftarrow qStart$
    $distVal \leftarrow \sum(\mathcal{T}[tStart : tStart + m - 1][iDim] - Q[qStart : qStart + m - 1][iDim])^2$
    $\mathfrak{D}_Q[kNN + 1][qStart][iDim] \leftarrow \sqrt{distVal}$
    $\mathfrak{I}_Q[kNN + 1][qStart][iDim] \leftarrow tStart$
    for $ii \leftarrow 2$ ***to*** $\mathfrak{E}$ **do**
      $tStart \leftarrow (Idxs_Q - (ii - 1)) - iJump$
      $qStart \leftarrow (Idxs_Q - (ii - 1))$
      $part_1 \leftarrow (\mathcal{T}[tStart][iDim] - Q[qStart][iDim])^2$
      $part_2 \leftarrow (\mathcal{T}[tStart + m][iDim] - Q[qStart + m][iDim])^2$
      $distVal \leftarrow \sum distVal + part_1 - part_2$
      $\mathfrak{D}_Q[kNN + 1][qStart][iDim] \leftarrow \sqrt{distVal}$
      $\mathfrak{I}_Q[kNN + 1][qStart][iDim] \leftarrow tStart$
      $\mathfrak{V}[ii] \leftarrow qStart$
  $M_{kNN+1}[1 : Idx_Q][1 : d] \leftarrow P_Q[1 + kNN]\ [\ \mathfrak{V}[1 : \mathfrak{E}]\ ]\ [1 : d]$ ▷ *pick only* $(kNN + 1)^{th}$ *entry of all the query sub-sequences and store it in* M_{kNN+1} *array*
  $distTemp_{kNN+1}[1 : \mathfrak{E}][1 : d], indxDim_{kNN+1}[1 : \mathfrak{E}][1 : d] \leftarrow sortColWise\ (\ M_{kNN+1}[1 : \mathfrak{E}][1 : d]\ )$
  if $length(\ \mathfrak{V}\ ) == 1$ **then**
    $indxDim_{kNN+1} \leftarrow indxDim^T_{kNN+1}$ ▷ *take transpose of* $indxDim_{kNN+1}$ *array*
  $\mathfrak{D}_Q[kNN + 1][1 : \mathfrak{E}][1 : d] \leftarrow distTemp_{kNN+1}[1 : \mathfrak{E}][1 : d]$
  $\mathfrak{d}_Q[kNN + 1][1 : \mathfrak{E}][1 : d] \leftarrow indxDim_{kNN+1}[1 : \mathfrak{E}][1 : d]$
  for $p \leftarrow 1$ ***to*** $\mathfrak{E}$ **do**
    $\mathfrak{I}_Q[kNN + 1]\ [\ \mathfrak{V}[p]\ ]\ [1 : d] \leftarrow I_Q[kNN + 1]\ [\ \mathfrak{V}[p]\ ]\ [\ indxDim_{kNN+1}[p][1 : d]\ ]$
  $cumDistTemp \leftarrow (\mathfrak{E} \times 1)\ array$ ▷ *it's a 2D matrix, initialized with zeros*
  for $iIdx \leftarrow 1$ ***to*** $\mathfrak{E}$ **do**
    for $iDim \leftarrow 1$ ***to*** d **do**
      $cumDistTemp[iIdx][1] \leftarrow cumDistTemp[iIdx][1] + \mathfrak{D}_Q[kNN + 1][iIdx][iDim]$
      $dAll\ [kNN + 1\ ][iIdx][1] \leftarrow cumDistTemp[iIdx][1]/iDim$
  $[\ \mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q, dAll\ ] \leftarrow FunctionForElsePart\_3(Idx_Q, dAll, \mathfrak{D}_\mathcal{T}, \mathfrak{d}_\mathcal{T}, \mathfrak{I}_\mathcal{T}, kNN)$
return $\mathfrak{D}_Q,\ \mathfrak{d}_Q,\ \mathfrak{I}_Q$

Function `CalcZnormDist` ($qSubSeq$, $tSubSeq$, m, μ_Q, σ_Q, μ_T, σ_T) **:**

/* *this function is used to compute Z−normalised distance* */

$prod \leftarrow \sum_{t=1}^{m} qSubSeq_t \times tSubSeq_t$ ▷ *get sum of the products of qSubSeq and tSubSeq*
$dVal \leftarrow 2 \times \left[m \times \left[\frac{prod - (m \times \mu_Q \times \mu_T)}{\sigma_Q \times \sigma_T} \right] \right]$
return $dVal$

Algorithm 6: INDEPENDENT_ACAMP_JOIN($\mathfrak{D}^{\mathcal{T}}, Q, m$)

Input: The target time series data base ($\mathfrak{D}^{T}$) and query time series (Q)
Output: A matrix profile (P_Q) and associated matrix profile index (I_Q)

$n_{\mathcal{T}} \leftarrow length(\mathcal{T})$; $n_Q \leftarrow length(Q)$ ▷ *get the length of time series $\mathcal{T}$ and Q*
$Idxs_Q \leftarrow n_Q - m + 1$; $Idxs_{\mathcal{T}} \leftarrow n_{\mathcal{T}} - m + 1$ ▷ *get the total number of sub-sequences in Q and $\mathcal{T}$*
for $iDim \leftarrow 1$ ***to*** d **do**
  $[\mu_{\mathcal{T}}[iDim], \sigma_{\mathcal{T}}[iDim], \mu_Q[iDim], \sigma_Q[iDim]] \leftarrow ComputeMeanStd(\mathcal{T}, Q)$
$P_Q \leftarrow ((kNN + 1) \times Idx_Q \times d)\ array$ ▷ *it's a 3D matrix, initialized with infinity*
$I_Q, \mathcal{M}_Q, \mathfrak{D}_Q, \mathfrak{d}_Q, \mathfrak{I}_Q \leftarrow ((kNN + 1) \times Idx_Q \times d)\ array$ ▷ *these are 3D matrix, initialized with zeros*
$subSeqFlag \leftarrow (1 \times d)\ vector$ ▷ *it's a 1D horizontal Boolean vector, initialized with $FALSE$*
for $iJump \leftarrow 0$ ***to*** $(Idxs_{\mathcal{T}} - 1)$ **do**
  if $iJump \leq (Idxs_{\mathcal{T}} - Idxs_Q)$ **then**
    $\mathfrak{E} \leftarrow Idxs_Q$
  else
    $\mathfrak{E} \leftarrow Idxs_{\mathcal{T}} - iJump$
  if $(iJump + 1) \leq kNN$ **then**
    $pIdx \leftarrow iJump + 1$
  else
    $pIdx \leftarrow kNN + 1$
    $maxEleFlag \leftarrow TRUE$
  for $iDim \leftarrow 1$ ***to*** d **do**
    $qSubSeq \leftarrow Q[1 : 1 + m - 1][iDim]$
    $tSubSeq \leftarrow \mathcal{T}[iJump + 1 : (iJump + 1) + m - 1][iDim]$
    $[distVal, \mathfrak{P}] \leftarrow$
     $CalcZnormDist\ (\ qSubSeq, tSubSeq, m, \mu_Q[1][iDim], \sigma_Q[1][iDim], \mu_{\mathcal{T}}[1][iDim], \sigma_{\mathcal{T}}[1][iDim]\ )$ ▷
     compute distance with 1^{st} query sub-sequence and target sub-sequence
    $\mathfrak{D}_Q[pIdx][1][iDim] \leftarrow \sqrt{|distVal|}$ ▷ *store the square rooted distance*
    $\mathfrak{I}_Q[pIdx][1][iDim] \leftarrow iJump + 1$ ▷ *store the index*
    for $ii \leftarrow 2$ ***to*** $\mathfrak{E}$ **do**
      $tStart \leftarrow iJump + ii$
      $part_1 \leftarrow (Q[ii - 1][iDim] - \mathcal{T}[ii - 1][iDim])$
      $part_2 \leftarrow (Q[ii + m - 1][iDim] - \mathcal{T}[ii + m - 1][iDim])$
      $\mathfrak{P} \leftarrow \mathfrak{P} - part_1 + part_2$ ▷ *compute the incremental distance*
      $distVal \leftarrow 2 \times \left[m - \frac{\mathfrak{P} - (m \times \mu_Q[ii][iDim] \times \mu_{\mathcal{T}}[tStart][iDim])}{\sigma_Q[ii][iDim] \times \sigma_{\mathcal{T}}[tStart][iDim]}\right]$
      $\mathfrak{D}_Q[pIdx][ii][iDim] \leftarrow \sqrt{|distVal|}$ ▷ *store the square rooted distance*
      $\mathfrak{I}_Q[pIdx][ii][iDim] \leftarrow tStart$ ▷ *store the index*
  if $maxEleFlag == TRUE$ **then**
    ....
  ...
...
return $\mathcal{M}_Q[1 : kNN][1 : Idx_Q][1 : d] \leftarrow \mathfrak{I}_Q[1 : kNN][1 : Idx_Q][1 : d]$