Discord Discovery in Streaming Time Series based on an Improved HOT SAX Algorithm

Pham Minh Chau, Bui Minh Duc, Duong Tuan Anh
Faculty of Computer Science and Engineering
Ho Chi Minh University of Technology
Ho Chi Minh City, Vietnam
{chaupm.cs, ducbk95}@gmail.com, dtanh@hcmut.edu.vn

ABSTRACT

In this paper, we propose an improved variant of HOT SAX algorithm, called HS-Squeezer, for efficient discord detection in static time series. HS-Squeezer employs clustering rather than augmented trie to arrange two ordering heuristics in HOT SAX. Furthermore, we introduce HS-Squeezer-Stream, the application of HS-Squeezer in the framework for detecting local discords in streaming time series. The experimental results reveal that HS-Squeezer can detect the same quality discords as those detected by HOT SAX but with much shorter run time. Furthermore, HS-Squeezer-Stream demonstrates a fast response in handling time series streams with quality local discords detected.

CCS CONCEPTS

• **Information systems → Data Mining**; **Data Stream Mining**

KEYWORDS

Streaming time series, discord discovery, clustering.

ACM Reference format:

Pham Minh Chau, Bui Minh Duc, Duong Tuan Anh. 2018. Discord Discovery in Streaming Time Series based on an Improved HOT SAX Algorithm. *In SoICT '18: Ninth International Symposium on Information and Communication Technology, December 6–7, 2018, Da Nang City, Viet Nam.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287921.3287929

1 INTRODUCTION

The problem of detecting unusual (abnormal, novel, deviant, anomalous, *discord*) subsequences in a time series has recently attracted much attention. Time series anomaly detection brings out the abnormal patterns present in a time series. Application areas that explore such time series anomalies are, for example, fault diagnosis, intrusion detection, fraud detection, financial auditing and data cleansing.

There has been an extensive study on time series anomaly detection in the literature. Various algorithms for time series discord discovery have been proposed, such as brute-force and HOT SAX by Keogh et al. (2005) [7]; WAT by Bu et al. (2007) [1]; a method based on Piecewise Aggregate Approximation (PAA) bit representation and clustering by Li et al. (2013) [11]; a method based on segmentation and cluster-based outlier detection by Kha and Anh (2015) [10]; and a method based on time series segmentation and anomaly scores by Vy and Anh (2016) [16]. Among these above-mentioned algorithms for discord discovery in time series, HOT SAX has been considered as the most popular algorithm. HOT SAX is an unsupervised method of anomaly detection and has been applied in several real applications. However, this algorithm still has one weakness: HOT SAX still suffers from a high computational cost.

Discord detection in streaming time series has emerged as an attractive problem recently. The task of finding the most important discord in streaming time series arises in diverse applications. For example, electrocardiogram time series of a given patient often continuously arrives during his/her treatment. However, discord detection in streaming time series is a non-trivial task. It is challenging to adapt a discord detection algorithm in static time series to a discord detection algorithm for streaming time series.

As for streaming time series, there have been a few researches works of anomaly detection that will be listed as follows. Liu et al. in 2009 proposed a framework for discord detection in streaming time series, called Detection of Continuous Discords (DCD) [13]. DCD can find continuous discord from local segments of a time series stream. One important technique in DCD is that it limits the search space to further enhance the discord detection efficiency. Toshniwal and Yadas (2011) [15] introduced an extension of HOT SAX to find outliers in local segments of a streaming time series. In this work, the criterion to determine time series outliers is the type of deviation from normal behavior, i.e. above or below normal. When there is a newly incoming subsequence, the data distribution of these

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SoICT 2018, December 6–7, 2018, Danang City, Viet Nam
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6539-0/18/12$15.00
https://doi.org/10.1145/3287921.3287929

distances is reevaluated. Due to this reason, the algorithm incurs high computational complexity. Yeh et al. in 2016 [17] proposed a unifying framework that can discover motifs, discords and shapeless from time series. This framework is based on the basic task: finding all pair similarity joins in two time series. The basic algorithm for the basic task, called STAMP, is based on the calculation of sliding dot products between a query (subsequence) and a time series which exploits Fast Fourier Transform (FFT). Based on STAMP, the authors proposed one algorithm for discord discovery in static time series and one algorithm for discord discovery in streaming time series. However, due to FFT, the complexity of STAMP is rather high, $O(n^2 \log n)$. Therefore, all the time series data mining tasks such as discord discovery, which is based on STAMP, still incur high complexity.

Motivated by the framework given by Liu et al., in this work we aim to develop an efficient method for discord detection in streaming time series. This work makes two contributions. First, we propose an improved variant of HOT SAX, called HS-Squeezer, for efficient discord discovery in static time series. In this algorithm, we use a clustering method instead of an augmented trie to support the discord discovery process. Second, we introduce HS-Squeezer-Stream, the application of HS-Squeezer in the framework given by Liu et al. for detecting local discords in streaming time series. The experimental results reveal that HS-Squeezer can detect the same quality discords as those detected by HOT SAX but with much shorter run time. Furthermore, HS-Squeezer-Stream demonstrates a fast response in handling time series streams with quality local discords detected.

The rest of our paper is organized as follows. Some background and related works are provided in Section 2. In Section 3, we introduce the HS-Squeezer algorithm. In Section 4, we introduce our algorithm for discord detection in streaming time series. The extensive experiments of our two proposed algorithms are reported in Section 5. Finally, Section 6 presents some conclusions and remarks for future works.

2 BACKGROUND AND RELATED WORKS

2.1 Definitions

A time series *discord* is a subsequence that is very different from its closest matching subsequence. However, in general, the best matches of a given subsequence (apart from itself) tend to be very close to the subsequence under consideration. For example, given a certain subsequence at position p, its closest match will be the subsequence at the position q where q is far from p just a few points. Such matches are called *trivial matches* and are not interesting. When finding discords, we should exclude trivial matches and keep only *non-self* matches defined as follows.

Definition 1. (Non-self match): Given a time series T containing a subsequence C of length n beginning at position p and a matching subsequence M beginning at position q, we say that M is a non-self match to C if $|p - q| \geq n$, i.e. C does not overlap M.

Definition 2. (Time series discord): Given a time series T, the subsequence D in T is called the most significant discord in T if the distance to its nearest non-self match is largest.

A streaming time series T is a semi-infinite time series sequence of real number $t_1, t_2, \ldots, t_n, \ldots$ where t_n is the most recent data point. Given a streaming time series T, the problem is finding the most significant discord of length l as soon as a newly incoming subsequence C of length l exists. This implies that the discord detection might be repeatedly executed once for every newly incoming data point of T. In storing a streaming time series, to avoid memory overflow, we use larger-size buffer B to contain the local segment of the streaming time series under consideration. In general, at time point t, suppose B contains a time series of length n, $B = t_1, t_2, \ldots, t_n$ and t_{new} is the newly incoming data point. At time point $t + 1$, the time series in the buffer becomes $B = t_2, \ldots, t_n, t_{new}$. That is the data point t_1 is removed out of the buffer at time point $t + 1$ and it is called the *obsolete* data point. Notice that the length of the buffer is specified in advance.

2.2 HOT SAX Algorithm

The HOT SAX algorithm was proposed by Keogh et al. in 2005 [7]. To find discord of length n in a time series T, HOT SAX extracts subsequences of length n by shifting a sliding window of length n across the time series T. Next, these subsequences are converted into PAA representation [12] and then to SAX representation [12] that has the SAX word length w. HOT SAX puts these SAX words in an array where the index refers back to the original subsequence. Once HOT SAX has this ordered list of SAX words, the algorithm can place them in a tree structure (called augmented trie) where the leaf nodes contain a linked list index of all word occurrences that map there. The parameters we have to supply to HOT SAX are the length of discord n, the SAX alphabet size a, and the SAX word length w. To realize an early exit of the for loops, HOT SAX applies the two following order heuristics: (a) *Outer loop heuristic*: It first visits subsequences corresponding to the SAX words that have the smallest word count, and then it visits the rest of the subsequences in random order; (b) *Inner loop heuristic*: for each candidate in outer loop, it first searches its nearest non-self match in the leaf node of the tree that has the same SAX word, and then it visits the rest of the subsequences in random order.

2.3 Squeezer Algorithm for Clustering Categorical Data

Squeezer [5] is a modified version of the incremental clustering method Leader [4] for clustering categorical data. The main ideas of Squeezer will be explained as follows.

Let $A_1, \ldots, A_m$ be a set of categorical attributes with domains $D_1, \ldots, D_m$. respectively. Let the dataset D be the set of patterns where each pattern t: $t \in D_1 \times \ldots \times D_m$. Let TID be the set of unique identifiers of every pattern. For each $tid \in TID$, the attribute value for A_i of corresponding pattern is represented as $tid.A_i$.

Definition 3: (*Cluster*) *Cluster* $\subseteq TID$ is a subset of TID.

Definition 4: Given a cluster C, the set of different attribute

values on A_i with respect to C is defined as: $VAL_i(C) = \{tid\ A_i \mid tid \in C\}$ where $1 \leq i \leq m$.

Definition 5: Given a cluster C, let $a_i \in D_i$, the support of a_i in C with respect to A_i is defined as: $Sup(a_i) = |\{tid \mid tid.A_i = a_i, tid \in C\}|$.

Definition 6: (*Similarity*) Given a cluster C and a pattern t with $tid \in TID$, the similarity between C and tid is defined as:

$$Sim(C, tid) = \sum_{i=1}^{m} \frac{Sup(a_i)}{\sum_{a_j \in VAL_i(C)} Sup(a_j)} \quad (1)$$

where $a_i = tid.\ A_i$

The Squeezer algorithm has n patterns as input, *similarity threshold s* as a parameter supplied by user and produces clusters as final results. Initially, the first pattern in the database is read in and the first cluster is constructed to contain the first pattern, i.e. C = {1}. Then, the consequent patterns are read iteratively. For each pattern, by the similarity function, its similarities with all existing clusters are computed. The largest value of similarity is chosen. If it is larger than the given threshold, s, the pattern will be assigned to the cluster that has the largest value of similarity. If the above condition does not hold, a new cluster must be created with this pattern. The algorithm continues until it has scanned all the patterns in the dataset.

The outline of the Squeezer algorithm is described as Figure 1.

```
Algorithm Squeezer (D, s)
  while (D has unread pattern)
  begin while
    pattern = getCurrentPattern(D);
    if (pattern.tid = 1)
      then addNewCluster(pattern.tid)
    else {
      for each existed cluster C do
          simComputation(C, pattern);
      get the max value of similarity: sim_max;
      get the corresponding Cluster Index: index;
      if sim_max ≥ s then
          addPatternToCluster (pattern, index)
      else
          addNewCluster(pattern.tid) }
  end while
  outputClusteringResult ();
```

Figure 1: The Squeezer algorithm.

The line *simComputation(C, pattern)* is computed by Eq. 1.

2.3 A Framework for Discord Detection in Time Series Stream

Liu et al. in 2009 proposed a framework, called DCD (Detection of Continuous Discords), for discord detection in streaming time series [13]. One important technique in DCD is that it limits the search space to further enhance the detection efficiency. The pseudo-code of the framework DCD is described as in Figure 2.

In the framework DCD, we use two data structures: the buffer B and array V. In explaining the use of array V, we need the definition of *nearest non-self neighbor distance* of a subsequence as follows.

Procedure DCD

Input: A given time series stream and the discord length of n

Output: The local discords (discord in the buffer)

```
1.  Initialize ();
2.  Read the time series data points into B // B is the buffer
3.  [prev_loc, dist]=FindDiscord(B, n);
4.  Output the local discord LD = [prev_loc, t]; //t is the current time
5.  Add dist into array V;
6.  while (the time series stream is not stopped) do
7.     Read the next data point into B;
8.     [loc, dist] = FindDiscord(B, n);
9.     if loc ≠ prev_loc - 1 && valid(dist, V) then
10.       Output the local discord LD = [loc, t]; //t is the current time
11.    end if
12.    prev_loc = loc;
13.    Add dist into array V;
14. endwhile
```

Figure 2: The Framework DCD for discord detection in streaming time series.

Definition 7: (*Nearest non-self neighbor distance*): Given a time series T, for any subsequence P, Q is the nearest non-self match of P, the distance from P to Q is the *nearest non-self neighbor distance* of P.

The new incoming data points will arrive to the buffer continuously. With the assumption that we start to find the discord when the buffer is full. At that moment, the *FindDiscord* procedure is invoked to detect the discord in the time series segment in the buffer B. The nearest non-self neighbor distance of the current discord is stored in the array V. Next, we read the new incoming data point from the stream, we put it in the right end of the buffer B and remove the left most data point in the buffer out of the buffer. At this moment, we invoke again the *FindDiscord* procedure to find the new discord. Whenever the discord is found, we output the discord if its location is different from that of the previous discord. The loop continues until the stream terminates.

To find the interesting discords from the given time series stream, we need to use the *valid* function which is defined as follows.

valid(*dist*, *V*) is true if *dist* > *mean*(*V*)**threshold*

where *mean*(*V*) is the mean value of all elements in the array *V*, *threshold* and the size of the array *V* are specified in advance by user.

A simple solution for *FindDiscord* procedure is to use the existing window-based algorithm such as HOT SAX to find the discord from the current buffer. In that case, we called the online discord detection algorithm as Brute-force HOT SAX (BFHS) algorithm. BFHS is not efficient since it has to search the whole buffer in each time of discord detection.

Liu et al. proposed a new *FindDiscord* procedure that can limit the search space to further enhance the discord detection efficiency. Before describing the *FindDiscord* procedure, we need one more related definition.

Definition 8 (*Small match*): Given a time series *T*, for any

subsequence P of T, Q is a non-self match of P, if $Dist(P, Q) < dist$, where $dist$ is the nearest non-self neighbor distance of the current local discord of the time series, then Q is a small match of P and P is a small match of Q.

The pseudo-code of Procedure *FindDiscord* is described as in Figure 3.

Let loc be the position of local discord at time t, $dist$ be the nearest non-self neighbor distance of local discord at time t and $currDist$ be the distance between the local discord at the time point t and the new arriving subsequence. Procedure FindDiscord considers two possible cases:

- Case a: If $currDist < dist$ or $loc = 1$, we have to search all possible subsequences in the buffer to find the new local discord at time point $t+1$. For this case, the search space is the Candidates set which consists of all the subsequences in the buffer from location 1 to location $|B| - n+1$.
- Case b: Otherwise, there may be some subsequences whose nearest non-self neighbor distances become larger than that of the local discord at time point t. For this case, Liu et al. suggested that the search space can be reduced to the Candidates set that consists of: (i) the small match of the first subsequence in the buffer, (ii) the local discord at time point t, and (iii) the new arriving subsequence.

Procedure FindDiscord

Input: B: the time series buffer at time $t+1$; n: the length of discord;

Output: The position and non-similar distance of local discord at time $t+1$.

1. read the next data point t_{new};
2. currDist = Dist($t_{loc},\ldots,t_{loc+n-1}$; $t_{m-n+2},\ldots,t_m$, t_{new});
   // *loc*: the position of local discord at time t;
   // *dist*: the nearest non-self neighbor distance of local discord at time t.
3. **if** *currDist* < *dist* // The case (a1)
4. *Candiates* = {the subsequence 1: $|B|-n+1$};
5. **else if** $loc = 1$ // The case (a2)
6. *Candidates* = {the subsequence 1: $|B|-n+1$};
7. **else** // The case (b)
8. *Candiates* = {the small match of subsequence $(1,n)(t)$∪
   {The local discord at time t}∪
   {The subsequence$(m-n+1, n)(t+1)$}};
9. [*loc*, *dist*]= Search(*Candidates*, *n*, *B*);

Figure 3: Procedure FindDiscord.

Liu et al. suggested that the *Search* function in *FindDiscord* procedure can be some discord detection function such as HOT SAX algorithm.

3 HS-SQUEEZER – AN IMPROVED VARIANT OF HOT SAX

In this section, we introduce our proposed algorithm, called HS-Squeezer, for discord discovery in static time series. HS-Squeezer is based on clustering. The rationale behind our algorithm is as follows. Clustering can allocate SAX words with high similarity into the same cluster. So we can use clustering instead of trie tree in arranging the two ordering heuristics for the HOT SAX algorithm.

The algorithm HS-Squeezer consists of the four following steps.

Step 1: Using PAA and SAX, transform methods to convert extracted subsequences from the time series under the sliding window into *SAX words*.

Step 2: Apply Squeezer algorithm to clusters the SAX words.

Step 3: From the clustering results, create the two ordering heuristics as follows.

Outer-loop order: The cluster with the smallest number of subsequences will be considered first. Unusual subsequences are very likely to belong to the clusters with small number of subsequences. After the outer loop has exhausted this set of candidates, the subsequences from the other clusters are visited.

Inner-loop order: the subsequences in the same cluster with the subsequence under consideration should be examined first. Subsequences in the same cluster are very likely to be highly similar. By that way, the inner loop can terminate early.

Step 4: Apply discord detection process as in HOT SAX with the two nested loops using the two created ordering heuristics.

Note that after using approximate approach to have the two ordering heuristics, in Step 4 we calculate the exact distance between each pair of subsequences for discord detection. Hence, the result would not be affected by the approximate approach.

To estimate the suitable length of PAA-frame for each dataset (i.e. the PAA transformation in Step 1), we apply PLA segmentation with bottom up algorithm [6] to segment the time series into several linear segments. The average of the lengths of all these segments will be used to estimate the length of the PAA-frame. Furthermore, from the discord length n and the size of the PAA-frame, we can easily determine the SAX word length w for HS-Squeezer by the following formula:

$$w = n/(\text{size of PAA-frame})$$

4 HS-SQUEEZER-STREAM FOR DISCORD DETECTION IN STREAMIG TIME SERIES

Our proposed algorithm for online discord detection, called HS-Squeezer-Stream, applies the framework DCD given by Liu et al. (2011). HS-Squeezer-Stream still uses the *Candidates* set proposed in DCD to limit the search space in finding discord subsequence. The important point in our proposed algorithm is that while in DCD Liu et al. suggest the original HOT SAX can be used in the role of the *Search* function in the *FindDiscord* procedure, our proposed algorithm uses HS-Squeezer as the *Search* function in the *FindDiscord* procedure. Due to that, we name our proposed algorithm as HS-Squeezer-Stream.

HS-Squeezer-Stream has to update the supporting data structure, i.e. the cluster structure of the SAX words, during the discord detection process in a streaming time series. At every new incoming data point, the subsequence which contains the new data point, after discretization, will be assigned to the suitable cluster and the subsequence which contains the out-of-date data point will be removed from its current cluster. Due to the

incrementality of the Squeezer clustering algorithm, this update work can be achieved efficiently.

4.1 Time Series Normalization

HOT SAX requires normalization of the time series data. Therefore, at each new incoming data point, the time series segment in the buffer is updated and we have to normalize again the data in the buffer before applying discord detection in it. We apply Z-score normalization [3] for each normalization of the data in the current buffer. Our technique of normalization in this particular context applies a *delay policy* as follows. When a new data point arrives, if the new mean and standard deviation of the current buffer become greater than the thresholds *threshold_mean* and *threshold_std* (given by the user) then we have to normalize all the data points in the buffer. Otherwise, only the new data point is normalized based on the current mean and current standard deviation.

4.2 How to Determine the Buffer Size

The size of the buffer can affect on the efficiency of the discord detection in streaming time series. If the buffer size is small, the detected discord results will vary so frequently. Otherwise, if the buffer size is too large, the computational cost will increase remarkably. In this work, we estimate the buffer size based on the *period* of the time series under consideration. The buffer size should be a multiple of the period of the time series in order that for cyclic time series, the new incoming data point and the obsolete data point will have some similarity. There exist a few methods for periodicity detection in time series. In this work, we employ Autocorrelation Function to estimate the period of the time series [2]. The main idea of this method is that if a time series has a period, a significant autocorrelation coefficient will occur at the time lag equal to the period or multiples of the period. Therefore, in this work we calculate the autocorrelation coefficients between two subsequences $C(1, n - k)$ and $C(1+ k, n - k)$ in the same time series T at the same time lag k varying from 1 to $n/2$. The estimated period is the value of the lag k which brings out the largest autocorrelation coefficient.

The autocorrelation coefficient between two subsequences $C(1, n - k)$ and $C(1+ k, n - k)$ can be calculated by the formula:

$$C = \frac{Covariance(C(1,n-k), C(1+k,n-k))}{Std(C(1,n-k)).Std(C(1+k,n-k))} = \frac{\frac{1}{n-k}\sum_{1}^{n-k}(t_i - \overline{t_1})(t_{i+k} - \overline{t_{1+k}})}{Std(C(1,n-k)).Std(C(1+k,n-k))}$$

where $\overline{t_1}, \overline{t_{1+k}}$ are the two mean values of the subsequences $C(1, n - k)$ and $C(1+ k, n - k)$, respectively and $Std(C(1, n - k))$, $Std(C(1+ k, n - k))$ are the standard deviations of the subsequences $C(1, n - k)$ and $C(1+ k, n - k)$, respectively.

5 EXPERIMENTAL EVALUATION

In this section, we describe the results of the two main experiments: Experiment 1 for discord detection in static time series (*offline discord detection*) and Experiment 2 for discord detection in streaming time series (*online discord detection*).

5.1 Experiment 1: Offline Discord Detection

For this experiment, we implemented two algorithms for discord detection in static time series: HOT SAX and HS-Squeezer. The experiment aims to compare HS-Squeezer with the original HOT SAX algorithm in terms of time efficiency and discord detection accuracy.

This experiment was conducted on the datasets from the UCR Time Series Data Mining archive ([8], [9]). There are seven datasets used in this experiment. The datasets are from different areas such as medicine, industry and science. The names and lengths of the seven datasets are shown in Table I. For each dataset in Table I, we also give the discord length n in the fourth column.

During experiment with Squeezer clustering, we observe that the number of clusters created by Squeezer depends on the parameters *threshold* and w. Through experiment, we found out that the number of clusters should be in the range between 10 and 30, and select the two parameters (*threshold*, w) = (0.85, 5) for most of the datasets.

Table I: Length and discord length for each dataset

Dataset Description	Dataset Name	Length of Time series	Discord Length (n)
Space Shuttle Marotta Valve	(TEL 16, TEK 17)	4993, 5000	128
Electrocardiogram	(ECG)	21600	40
Power Data	(Power_ Data)	35040	200
Patient's respiration data	(nprs43, nprs44)	18020, 24125	160
Earth Rotation Parameters	ERP	198400	64

So the parameters for the two comparative algorithms are selected as follows. For HOT SAX: $w = 3$, $a = 3$; for HS-Squeezer: *threshold* = 0.85 and $w = 5$ where w is the SAX word length, a is the size of the alphabet used in SAX transform and threshold is the similarity threshold used in Squeezer clustering algorithm.

5.1.1 Effectiveness. Following the tradition established in previous works, such as [1], [7], [11], the accuracy of a given discord discovery algorithm is basically based on human inspection of the discords detected by that algorithm. Notice that in most of the seven datasets in Experiment 1, the discords have been annotated by experts; therefore, we can spot the discords by eye with not much effort.

We compare HS-Squeezer and HOT SAX in discord detection accuracy over the 7 datasets. For each dataset, we found out that the discord detected by HS-Squeezer is exactly the same as the discord detected by HOT SAX.

Table II: Execution times of HOT SAX and HS-Squeezer

Dataset	HOT SAX	HS-Squeezer
TEK 16	2.116	0.692
TEK 17	2.554	0.37
ECG	14.979	5.91
Power_data	93.189	11.921
nqrs43	16.599	6.097
nqrs44	25.955	14.349
ERP	2056.595	309.45

5.1.2 Effeciency. Table II shows the runtimes (in seconds) of the two algorithms: HOT SAX, and HS-Squeezer in discord detection over the 7 datasets. From the experimental results in Table II, we can see that HS-Squeezer performs much faster than HOT SAX in all the datasets. On average, HS-Squeezer runs about 4.41 times faster than HOT SAX.

5.2 Experiment 2: Online Discord Detection

For this experiment, we implemented two algorithms for discord detection in streaming time series: HS-Squeezer-Stream and Brute Force HOT SAX (BFHS). BFHS is a concrete version of the framework DCD (Figure 2) in which HOT SAX plays the role of the *FindDiscord* precedure for detecting discord in the time series segment currently in the buffer. That means BFHS has to search the whole buffer in each time of discord detection.

We implemented all the algorithms with Visual C# 2013 (Window Form), and conducted the experiment on a HP, Intel(R) Core(TM) i5 CPU M430 @ 2.27GHz (4 CPUs), 4GB RAM, Windows 8.1 Pro 64-bit.

Table III: Length and discord length for each dataset

Dataset Description	Dataset Name	Time series length	Length of streaming time series	Discord length
Space Shuttle Marotta Valve	TEL 16, TEK 17	2992	2000	128
Electro-cardiogram	ECG	16600	5000	40
Power Data	Power_ Data	30040	5000	200
Patient's respiration data	nprs43	13020	5000	160
Earth Rotation Parameters	ERP	193400	5000	64

This experiment uses the datasets from the UCR Time Series Data Mining archive ([8], [9]). There are 5 datasets used in this experiment. The names and lengths of the five datasets are shown in Table III. For each dataset in Table III, we also give the discord length n in the fifth column.

For HS-Squeezer-Stream we have to estimate seven parameters: the period of the time series, the buffer length of the time series, the SAX word length w, the size of the alphabet a, the mean threshold (*Mean*), the threshold of standard deviation (*Std*) for data normalization and the similarity threshold (*Sim*) for Squeezer algorithm. The values of the parameters in the two algorithms for each dataset are shown in Table IV.

Table IV: Parameters for each dataset

Dataset	Period	Buffer	w	a	Mean	Std	Sim
TEK 16	1007	1007*2	5	3	0.001	0.01	0.85
ECG	371	371*10	5	3	0.001	0.01	0.95
Power_data	672	672*5	4	3	0.05	0.15	0.85
nqrs43	40	40*75	3	3	0.05	0.15	0.85
ERP	1280	1280*2	4	3	0.001	0.01	0.95

5.2.1 Effectiveness. We compare HS-Squeezer-Stream to BFHS in discord detection accuracy over the 5 streaming time series. For each dataset, we found out that the discord detected by HS-Squeezer-Stream is almost the same as that detected by BFHS. The reason is that both of the two algorithms use exact distances to find discord. The slight difference mainly comes from the orders in the two loops when more than one pair of subsequences have the same distance.

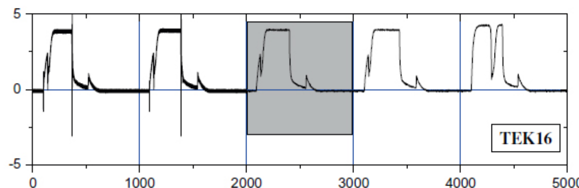

Figure 4: TEK 16 time series with five periods (period length ≈ 1000).

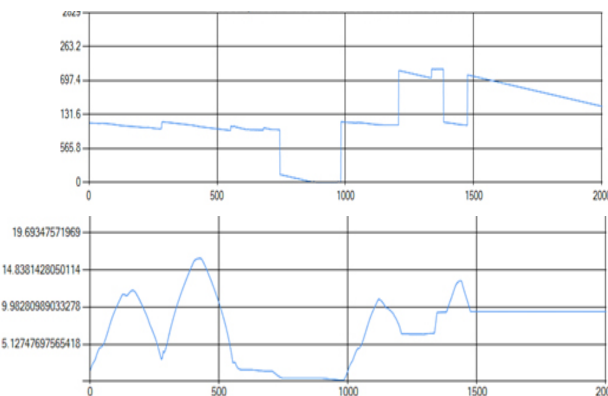

Figure 5: Online discord detection on TEK 16 dataset by HS-Squeezer-Stream.

Figure 4 shows the TEK 16 dataset with 5 periods (period length is about 1000). In Figure 5 we evaluate our HS-Squeezer-Stream algorithm in a real case (with dataset TEK 16). The top graph shows the locations of the found discords varying at each time point. The bottom graph shows the nearest non-self neighbor distance of each input subsequence at each time point.

5.2.2 Efficiency. Table V shows the runtimes (in seconds) of the two algorithms: BFHS and HS-Squeezer-Stream in discord discovery over the 5 streaming datasets. From the experimental results in Table V, we can see that HS-Squeezer-Stream performs much faster than BFHS in all the datasets. On average, HS-Squeezer-Stream runs about 3.32 times faster than BFHS.

Table V: Execution times of BFHS and HS-Squeezer-Stream

Dataset	BFHS	HS-Squeezer-Stream
TEK 16	1226	832
ECG	8940	1449
Power_data	11400	2811
nqrs43	7740	4080
ERP	3618	1129

6 CONCLUSIONS

We proposed an improved variant of HOT SAX, called HS-Squeezer, for efficient time series discord detection. Given a raw time series, we represent data by SAX discretization. Based on SAX representation, we apply a categorical data clustering algorithm for detecting discord in the time series. We employ the clustering result to arrange two ordering heuristics for discord detection process. In addition, we extend HS-Squeezer to a new algorithm, called HS-Squeezer-Stream for online discord detection. We experimentally showed that this online detection algorithm is remarkably faster than BFHS, the brute force search approach which is based on the original HOT SAX.

In the future, we plan to develop a new algorithm for discord discovery in streaming time series which is based on bounding boxes and it should be more efficient than the method given by Sanchez et al. in 2014 [14].

ACKNOWLEDGEMENT

We are grateful to Prof. Eamonn J. Keogh for kindly providing necessary datasets for the research work.

REFERENCES

[1] Y. Bu, T.W. Leung, A.W.C. Fu, E. Keogh, J. Pei, and S. Meshkin. 2007. WAT: Finding top-k discords in time series database. In *Proc. of 2007 SIAM International Conference on Data Mining*, Minneapolis, Minnesota, USA, pp. 449-454.

[2] F. X. Diebold. 2007. *Elements of Forecasting*, Fourth Edition. Thomson South-Western.

[3] J. Han and M. Kamber. 2011. *Data Mining: Concepts and Techniques*, 3rd Edition. Morgan Kaufmann Publishing.

[4] J. A. Hartigan. 1975. *Clustering Algorithms*, John Wiley & Sons.

[5] Z. He, X. Xu and S. Deng. 2002. Squeezer: An Efficient Algorithm for Clustering Categorical Data. *J. Computer Science and Technology*, Vol. 17, No. 5, 611-624.

[6] E. Keogh, S. Chu, D. Hart, M. Pazzani. 2002. An Online Algorithm for Clustering Categorical Data. *J. Computer Science and Technology*, Vol. 17, no. 5, 611-624.

[7] E. Keogh, J. Lin and A. Fu. 2005. HOT SAX: efficiently finding the most unusual time series subsequence. In *Proc. of 5th IEE Int. Conf. on Data Mining*, (ICDM), Houston, Texas, pp. 226-233.

[8] E. Keogh, J. Lin, and A. Fu, [online] http://www.cs.ucr.edu/~eamonn/discords/. Accessed in 2017.

[9] E. Keogh and T. Folias. The UCR Time Series Data Mining Archive. [http://www.cs.ucr.edu/~eamonn/TSDMA/index.html].

[10] N.H. Kha, D.T. Anh. 2015. From cluster-based outlier detection to time series discord discovery. *Trends and Applications in Knowledge Discovery and Data Mining-PAKDD 2015 Workshops*, Ho Chi Minh City, Vietnam, May, X..L.Li et al. (Eds.), LNAI 9441, Springer, 16-28.

[11] G. Li, O. Braysy, L. Jiang, Z. Wu, Y. Wang. 2013. Finding time series discord based on bit representation clustering. *Knowledge-Based Systems*, vol.52, pp. 243-254.

[12] J. Lin, E. Keogh, S. Lonardi, B. Chiu. 2003. Symbolic representation of time series, with implications for streaming algorithms. In *Proc. of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, San Diego, CA, June 13.

[13] Y. Liu, X. Chen, F. Wang, J. Yin. 2009. Efficient Detection of Discords for Time Series Stream. Q.Li et al.(eds*), Advances in Data and Web Management*, vol. 5446, pp. 629-634.

[14] H. Sanchez, B. Bustos. 2014. Anomaly Detection in Streaming Time Series Based on Bounding Boxes. In Traina A.J.M., Traina C., Cordeiro R.L.F. (eds) *Similarity Search and Applications*. SISAP 2014. LNCS 882,. Springer.

[15] D. Toshniwal and S. Yadav. 2011. 'Adaptive outlier detection in streaming time series. In *Proc. of International Conference on Asia Agriculture and Animal.* ICAAA 2011, Hong Kong, China, pp. 186-191, 2011

[16] N. D. K. Vy, D. T. Anh. 2016. Detecting Variable Length Anomaly Patterns in Time Series Data. In *Proc. of Int. Conf. on Data Mining and Big Data* (DMBD 2016), Bali, Indonesia, June 25-30, Y. Tan, Y. Shi (Eds.), LNCS 9714, Springer, pp. 279-287.

[17] C.C.M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H.A. Dau, D.F. Silva, A. Mueen, E. Keogh. 2016. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets. In 2016 IEEE 16th International Conference on Data Mining (ICDM), Barcelona, Spain, pp. 1317-1322.