Knowledge-Based Systems 54 (2013) 243–254

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Finding time series discord based on bit representation clustering

Guiling Li [a], Olli Bräysy [b,c], Liangxiao Jiang [a], Zongda Wu [d,*], Yuanzhen Wang [e]

[a] School of Computer Science, China University of Geosciences, Wuhan 430074, China
[b] VU University Amsterdam, De Boelelaan 1081, 1081 HV Amsterdam, Netherlands
[c] Procomp Solutions Oy, Kiviharjuntie 11, FI-90220 Oulu, Finland
[d] Oujiang College, Wenzhou University, Wenzhou 325035, Zhejiang, China
[e] School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China

ARTICLE INFO

Article history:
Received 31 January 2013
Received in revised form 9 September 2013
Accepted 12 September 2013
Available online 28 September 2013

Keywords:
Time series data mining
Discord discovery
Clustering
Pruning
Dimensionality reduction

ABSTRACT

The problem of finding time series discord has attracted much attention recently due to its numerous applications and several algorithms have been suggested. However, most of them suffer from high computation cost and cannot satisfy the requirement of real applications. In this paper, we propose a novel discord discovery algorithm *BitClusterDiscord* which is based on bit representation clustering. Firstly, we use PAA (Piecewise Aggregate Approximation) bit serialization to segment time series, so as to capture the main variation characteristic of time series and avoid the influence of noise. Secondly, we present an improved K-Medoids clustering algorithm to merge several patterns with similar variation behaviors into a common cluster. Finally, based on bit representation clustering, we design two pruning strategies and propose an effective algorithm for time series discord discovery. Extensive experiments have demonstrated that the proposed approach can not only effectively find discord of time series, but also greatly improve the computational efficiency.

1. Introduction

Time series discord is the subsequence of a time series, which has the biggest difference in all the subsequences of the time series [1]. Recently, finding time series discord has attracted much attention due to its numerous applications [1–7]. Let us take a specific application example from the health care sector. Both Electrocardiogram (ECG) and Electroencephalogram (EEG) can be deemed as time series [8–14]. It is an important routine for doctors to analyze whether there is a discord in a patient's ECG or EEG so as to determine the patient's health status. There are also numerous other applications. Time series discord discovery is e.g. very valuable in data mining tasks such as anomaly detection, improving quality of clustering and data cleansing [1].

Several algorithms on time series discord discovery have already been published. Naïve method [1] considers each subsequence as discord candidate, and uses two layer loops to detect whether the candidate is discord. The complexity of naïve method is $O(n^2)$. Simple pruning method [1] adopts pruning strategy based on naïve method, providing some improvements but far less than enough. The efficiency of these methods is not good enough for real applications. For example, in the health care sector it is very important for the doctor to rapidly diagnose the patient's symptoms according to ECG checklists and start the immediate treatment when necessary.

As time series often has the characteristic of high dimensionality, we choose to reduce the dimensionality of raw time series through approximate bit representation. Clustering is another strategy to accelerate discord discovery through filtering the candidates effectively, we apply clustering based on the bit representation. According to the obtained clustering results, we notice that a pruning process is also required to reduce the required computing time. Thus, in this paper we propose a novel pruning method for clustering.

The main contributions of this paper are highlighted as follows:

- *Bit representation clustering*: We propose an improved K-Medoids clustering algorithm based on bit representation for time series, called as *BitCluster*. It provides a primary filtering and clearly accelerates the discord discovery.
- *Pruning based discord discovery*: We propose a discord discovery algorithm after the bit representation clustering, called as *BitClusterDiscord*. In order to improve efficiency, we design two pruning strategies. The first is a heuristic pruning whereas the second is based on the cluster center distance.
- *Experimental study*: We conduct extensive experiments on diverse datasets to test the proposed algorithm from multiple aspects. The results show that our algorithm finds discords effectively, scales well and is more effective than the previous methods.

* Corresponding author.
E-mail addresses: guiling@cug.edu.cn (G. Li), olli.braysy@pp.inet.fi (O. Bräysy), ljiang@cug.edu.cn (L. Jiang), zongda1983@163.com (Z. Wu), wangyz2005@163.com (Y. Wang).

0950-7051/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.knosys.2013.09.015

The rest of the paper is organized as follows. In Section 2 we discuss the previous work on time series discord discovery. In Section 3 we give the problem statement, and in Section 4 we describe the bit representation for time series. In Section 5 we present the improved clustering algorithm based on bit representation. We propose the novel discord discovery algorithm in Section 6 and in Section 7 we show the experimental results and evaluations. Finally, in Section 8 we conclude our work and point out the future work.

2. Related work

Existing time series discord discovery algorithms can be categorized into three types: naïve method, simple pruning method and method based on dimensionality reduction techniques.

The main of idea of naïve method is to define the subsequence with the largest distance to its nearest neighbor as discord. This is done by considering each subsequence as candidate and finding first out the nearest non-self match. The method can be implemented by a two layer nested loop. The outer loop considers each possible candidate subsequence and the inner loop is a linear scan to identify the nearest non-self match of the candidate. This method is simple, easy to implement, and can produce the exact solution. The main problem is that the complexity is $O(n^2)$.

Simple pruning method [1] makes some improvement over the naïve method by applying a random sampling method to prune the candidates. More precisely, in the simple pruning method the above mentioned inner loop is checked through in a random order and the loop may be terminated before the end is reached if certain conditions are met. Here the key idea is that if the distance of certain subsequence to its non-self match is smaller than the smallest distance found until now, even if the non-self match is not the nearest, it can be confirmed that the subsequence is not the discord.

The above two methods are mainly designed for raw time series. By applying time series dimensionality reduction techniques, discord can be found also based on the approximate representation. Keogh et al. [1] presented the first study on the discord problem and proposed a heuristic method called HOT SAX. Their approach is based on SAX (Symbolic Aggregate approXimation), applying the heuristic pruning in two layer loops. In the outer loop, the subsequence with larger distance to its nearest neighbor has a priority to be selected for comparison. Correspondingly, in the inner loop, the subsequence with smaller distance to the current candidate has a priority to be compared. Thus, the method can be terminated sooner. However, SAX should meet the requirement of Gaussian distribution in data and the similarity measure of SAX is not accurate when comparing the adjacent symbols in the search table.

Li et al. [4] applied SAX in image domain and studied shape discord discovery. They map shape sketch to time series, apply SAX representation and then utilize heuristic strategy to find discords. Since the same shape can be converted to different time series with different rotation directions so as to different SAX representations, they handle this problem with rotation invariant Euclidean distance.

Fu et al. [2] found discords by Harr transform. They first make Harr transform for subsequences, and then utilize breadth first split algorithm to order the outer and inner loop. This method relies on the basis function of Harr transform. Bu et al. [3] studied top-k discord discovery in time series databases and proposed WAT algorithm based on Harr wavelet and augmented trie. They first employ Harr wavelet transform to approximate time series. Then, they discretize transformed sequences by symbols and reorder the candidate subsequences using heuristic method with augmented trie.

Most discord discovery algorithms assume that data resides in the memory. To solve disk aware discord discovery in huge date sets, Yankov et al. [5] proposed a two-phase discord detection algorithm. The first phase selects the candidates and the second phase identifies discords. The algorithm requires two linear scans over the disk.

Both symbolic and bit representation methods are used to discretize time series. Symbolic methods, such as SAX and *a*SAX, segment time series and discretize the subsequences into symbols. Bit methods such as PAA (Piecewise Aggregate Approximation) bit serialization [15,16], RLE (Run Length Encoding) [17] and BCM (Bit Coding Mode) [18] segment time series and discretize the subsequences into bit string. The merit of discretization is its simplicity and low storage cost. The effect of feature preserving depends on the concrete discretization method. Bit representation of time series is simple and has low storage cost, and bit operations are simple and efficient. So, in this paper, we study discord discovery problem on the basis of bit representation for time series.

3. Problem statement

In order to describe the problem of discord discovery, we give the related definitions as follows [1].

Definition 1 (*Time Series*). Time Series is a data sequence, where data element arrives with time order, denoted by $T = t_1, \ldots, t_n$, here n is the length of T.

Definition 2 (*Subsequence*). Given a time series T with length n, the subsequence of T is the consecutive position sampling with length m(with $m \ll n$) in T, denoted by $C = t_p, \ldots, t_{p+m-1}$, where $1 \leqslant p \leqslant n - m + 1$.

Definition 3 (*Distance*). *Dist* is a function as:

$$Dist(C, M) \to R \tag{1}$$

wherein the input parameters are two subsequences C and M, the function returns a non-negative number R, called as the distance from M to C. *Dist* function must hold symmetry, i.e., $Dist(C, M) = Dist(M, C)$.

Definition 4 (*Non-self Match*). Given a time series T, including a subsequence C with length m starting from position p and a matching subsequence M starting from position q. If $|p - q| \geqslant m$, M is called as a non-self match to C at the distance $Dist(M, C)$.

Definition 5 (*Time Series Discord*). Given a time series T, containing a subsequence D with length m starting from position l. If D has the largest distance to its nearest non-self match, D is called as the discord of T. That is to say, $\forall$ subsequence C of T, non-self match M_C of C, non-self match M_D of D, the inequality $\min(Dist(D, M_D)) > \min(Dist(C, M_C))$ is satisfied.

Our work is based on the above definitions, focusing on finding discord in time series.

4. Bit representation

Our discord discovery algorithm is based on bit representation via dimensionality reduction. In this section, we introduce PAA bit serialization and related bit distances.

4.1. PAA bit serialization

As PAA bit serialization is based on PAA, we first briefly review Piecewise Aggregate Approximation (PAA) representation of time series introduced in [16].

We denote a time series T with length n as $T = t_1, \ldots, t_n$. Let w be the dimensionality of space to be transformed ($1 \leqslant w \leqslant n$). For convenience, we assume that w is a factor of n. A time series T with length n can be represented in w space by a vector $\overline{T} = \{\overline{t_1}, \overline{t_2}, \ldots, \overline{t_w}\}$, where the ith element of $\overline{T}$ is calculated by:

$$\overline{t_i} = \frac{w}{n} \sum_{j=\frac{n}{w}(i-1)+1}^{\frac{n}{w}i} t_j \tag{2}$$

The intuition of Eq. (2) is that, in order to reduce the dimension from n to w, first the data in n dimension space is divided into w equal-sized segments. Then the mean value of the data in the same segment is calculated and the mean values arranged by time order are formed as a vector. The vector is used as PAA approximate representation of time series T. Fig. 1 is the illustration of PAA representation.

Though PAA can extract the mean feature of each segment, it is unable to show the trend between segments. The pioneer work of bit representation is Clipped data [19]. Based on PAA technique, Li [15] proposed the PAA bit serialization method. Given a time series T, first the data reduce dimensionality through PAA technique to obtain PAA representation $\overline{T}$. Then the mean values of adjacent segments are compared, according to the comparison, a bit of 1 or 0 is used to describe the trend of adjacent segments. These bits arranged by time constitute a bit string, denoted by $\overrightarrow{T} = \{\overrightarrow{t}_1, \overrightarrow{t_2}, \ldots, \overrightarrow{t_{w-1}}\}$, which is the representation of PAA bit serialization of T. Let $\overline{t_i}$ and $\overline{t_{i+1}}$ be the adjacent elements in $\overline{T}$, $\overrightarrow{t_i}$ is defined as:

$$\overrightarrow{t_i} = \begin{cases} 1 & \text{if } (\overline{t_{i+1}} > \overline{t_i}) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

We continue to use the time series T in Fig. 1 as an example to go further. The illustration of PAA bit serialization is shown in Fig. 2. The PAA bit serialization supports dimensionality reduction. It can not only avoid the affect of noise, but also show the main trend of time series.

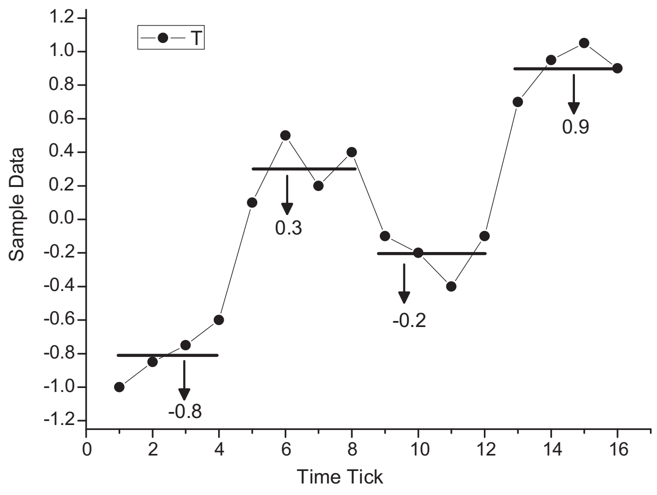

Fig. 1. An illustration of PAA. $T = \{-1.0, -0.85, -0.75, -0.6, 0.1, 0.5, 0.2, 0.4, -0.1, -0.2, -0.4, -0.1, 0.7, 0.95, 1.05, 0.9\}$, $n = |T| = 16$. $\overline{T} = \{-0.8, 0.3, 0.2, 0.9\}$, $w = |\overline{T}| = 4$.

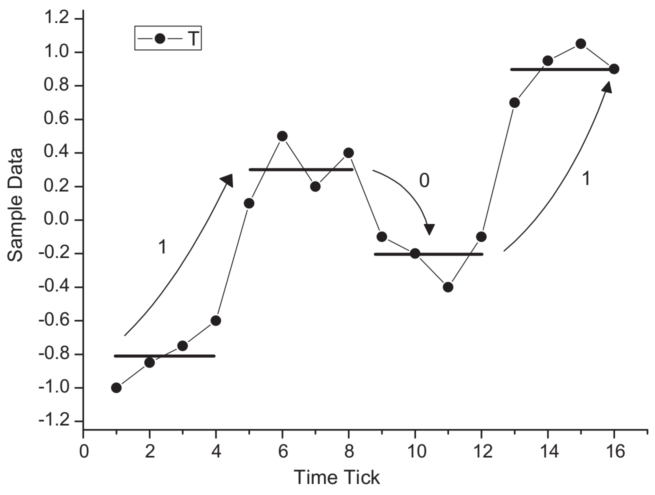

Fig. 2. An illustration of PAA bit serialization: $\overrightarrow{T} = \{1, 0, 1\}$, $|\overrightarrow{T}| = w - 1 = 3$.

4.2. Bit distance

Let S and T denote two time series, $\overrightarrow{S} = \{\overrightarrow{s}_1, \overrightarrow{s_2}, \ldots, \overrightarrow{s_{w-1}}\}$ and $\overrightarrow{T} = \{\overrightarrow{t}_1, \overrightarrow{t_2}, \ldots, \overrightarrow{t_{w-1}}\}$ denote their corresponding PAA bit serialization. We measure the similarity of $\overrightarrow{S}$ and $\overrightarrow{T}$ using bit distance *BitSeries_Dist* as:

$$BitSeries\_Dist(\overrightarrow{S}, \overrightarrow{T}) = \frac{1}{w-1} \sum_{i=1}^{w-1} BDist(\overrightarrow{s_i}, \overrightarrow{t_i}) \tag{4}$$

wherein

$$BDist(\overrightarrow{s_i}, \overrightarrow{t_i}) = \begin{cases} 1 & \text{if } (\overrightarrow{s_i} != \overrightarrow{t_i}) \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Eq. (5) considers the bit values $\overrightarrow{s_i}$ and $\overrightarrow{t_i}$ in the corresponding positions of two bit strings and calculates their distance in a way similar to XOR operations of binaries. If $\overrightarrow{s_i}$ and $\overrightarrow{t_i}$ are unequal, $BDist(\overrightarrow{s_i}, \overrightarrow{t_i})$ is 1, otherwise 0. Eq. (4) sums *BDist* for all i and divides the length $w - 1$ of $\overrightarrow{S}$ and $\overrightarrow{T}$. *BitSeries_Dist* holds non-negativity, symmetry, and triangle inequality. It has the advantages of simple calculation and high efficiency, which provides good support for bit based clustering.

5. Bit representation clustering

Clustering can allocate patterns with high similarity into the same cluster. We can use clustering to exclude some unqualified subsequences and improve the speed of discord discovery. Clustering can be an effective method of primary filtering for subsequences in time series.

Li [15] proposed BK-means clustering algorithm based on PAA bit serialization. The time complexity of BK-means is linear. However, the returned k cluster centers are the means of distances in each cluster, not the real subsequences in the raw time series. Therefore, it is unable to find the corresponding relationship between the cluster center and raw subsequence from the clustering result. In order to find discord of time series effectively, we need to find this relationship from the clustering result. This goal can be achieved through K-Medoids method [20]. In Section 5 we present the improved K-Medoids clustering algorithm.

Based on the PAA bit serialization mentioned in Section 4, we put forward the improved K-Medoids clustering algorithm, called as *BitCluster*. We first obtain bit series dataset by the PAA bit serialization method, denoted by *BD* with size of |*BD*|. We need to define at start also the number of clusters, *k*.*BitCluster* first randomly allocates *k* representative bit series as the initial cluster centers. Then according to the bit distance from non-representative bit series to *k* initial cluster centers, it assigns the remained bit series to the nearest clusters. The pseudo code of *BitCluster* is described in Algorithm 1.

In Algorithm 1, *bCluster* is a *struct* array, which stores the set of *k* clusters. Each array element in *bCluster* includes the cluster center, radius, number of members, and the actual members of the cluster.

Lines 1–4 perform the initialization, selecting *k* random objects as initial cluster centers. Lines 5–15 scan the *BD*, clustering bit series and storing the result. Function *BelongtoCenter*() judges whether the current bit series *j* belongs to some of the *k* representatives. If not, it calculates the distance between bit series *j* and *k* cluster centers to seek cluster *p* with the smallest distance, and assigns bit series *j* to cluster *p*, then revises variables *bCluster* and *PatternCTag*, as shown in lines 12–14.

Algorithm 1. Clustering algorithm *BitCluster* based on PAA bit serialization

```
INPUT: Bit series dataset BD, clustering parameter k
OUTPUT: set of k clusters bCluster, cluster identification tags
  PatternCTag
1     for i=1 to k do
2       RandInitial(bCluster[i].Center);
3       bCluster[i].NumMembers=1;
4     end of for
5     for j=1 to |BD| do
6       if (!BelongtoCenter(bCluster, j, k))
7         p=1;
8         for i=1 to k do
9           if (BitSeries_Dist(BD[j],
      BD[i])<BitSeries_Dist(BD[j],BD[bCluster[p].Center]))
10            p=i;
11          end of for
12          PatternCTag[j]=p;
13          bCluster[p].Member[bCluster[p].NumMembers]=j;
14          bCluster[p].NumMembers++;
15    end of for
16    for i=1 to k do
17      bCluster[i].Radius=CalClusterRadius(bCluster[i]);
18    end of for
```

Lines 16–18 call function *CalClusterRadius*() to calculate cluster radius of each cluster. Cluster radius is defined as the largest distance from each object in the same cluster to cluster center. In order to facilitate the successive discord discovery effectively, we use Euclidean distance to calculate cluster radius. The biggest Euclidean distance from the raw subsequences corresponding to each bit series in the cluster to the raw subsequence corresponding to cluster center is deemed as cluster radius of current cluster. *BitCluster* stores both the set of *k* clusters and cluster identification tags of each subsequence. We can find pattern features and the distribution in time series by clustering.

The computational complexity of Algorithm 1 is $O(k * |BD|)$. We must choose parameter *k*. As noted above, the goal of clustering is to serve successive discord discovery. The choice of *k* depends on the dataset and the experiments. The number of clusters is related to the size of the dataset and the changing characteristics of the dataset. We set an interval for *k* according to the dataset beforehand and verify the value by experiments. We will introduce the value of *k* in our experiments in Section 7.

6. Discord discovery by clustering pruning

Bit representation clustering is a critical step for accelerating the discord discovery. In this section, we describe two pruning strategies that are applied as part of our algorithm as well as the algorithm itself. The first pruning is a heuristic pruning whereas the second pruning is based on cluster center distance.

6.1. Pruning strategies

6.1.1. Heuristic pruning

Inspired by the HOT SAX [1] heuristic, we also apply pruning in both outer and inner loop. Considering that we know the number of members in each cluster by applying Algorithm 1, in the outer loop we set high priority to select the subsequences from the smaller-sized clusters. This is done to obtain larger distance to nearest neighbor and to terminate inner loop sooner. In the inner loop, as the members in the same cluster have high similarity, we set high priority to compute the distance between subsequences in the same cluster, to achieve early termination of the loop.

6.1.2. Pruning by cluster center distance

Considering the relation between different clusters, we can prune according to the distance between cluster centers. As we know the cluster center and the cluster radius of each cluster, we can analyze the relation between two arbitrary clusters to prune. By observing the distance between two cluster centers, the radius of two clusters and current compared threshold, we can rule out some computations, as shown in Observation 1.

Observation 1. Let $C1$ and $C2$ denote two clusters and let $O1$ and $O2$ refer to their corresponding cluster centers and $r1$ and $r2$ be the radius of the clusters and *best_so_far_dist* define the current threshold. If $Dist(O1, O2) - r1 - r2 > best\_so\_far\_dist$, there will be no nearest neighbors between the two clusters.

The above equation subtracts two cluster radiuses from the distance between two cluster centers. We define it here as the distance between two clusters. If the distance between two clusters is larger than *best_so_far_dist*, it means that there cannot be nearest neighbors for members in the two clusters. Fig. 3 illustrates the relation between two clusters.

6.2. Discord discovery algorithm

Based on the clustering algorithm described in Section 5 and two pruning strategies depicted in Section 6.1, we summarize here our time series discord discovery algorithm, called as *BitClusterDiscord*. The pseudo code of *BitClusterDiscord* is given in Algorithm 2.

Algorithm 2 contains three major steps. First, it uses PAA bit serialization for raw time series, obtaining bit series dataset *BD*. Second, it performs clustering in the bit series dataset. As a third step, two pruning strategies based on clustering results are applied to find the discords.

In Algorithm 2, *m* denotes the length of discord. Lines 1–2 initialize two variables: the starting position of discord

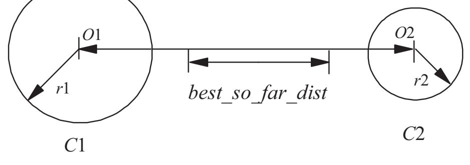

Fig. 3. The relation between two clusters.

best_so_far_loc, and the distance from discord to its nearest neighbor *best_so_far_dist*. Lines 3–5 employ dimensionality reduction technique as described in Section 4.1 and make PAA bit serialization for all subsequences with length *m*. The bit strings are stored in the bit series dataset *BD*.Line 6 employs Algorithm 1 *BitCluster* for clustering the data in the bit series dataset *BD*. The clustering algorithm returns the set of *k* clusters and cluster identification tags. The purpose of returning the cluster identification tags is to facilitate the successive pruning by cluster center distance.

Lines 7–8 generate the order of two layer loops for discord discovery. We design two functions Pr *oduceOuterOrder*() and Pr *oduceInnerOrder*() to generate the change order of outer loop and inner loop. The loop order is in accordance with heuristic pruning strategy described in Section 6.1.1.

The two layer loop structure for discord discovery is shown in lines 9–23. In line 9, the outer loop variable *p* changes according to the order by Pr *oduceOuterOrder*(). In line 11, the inner loop variable *q* changes according to the order by Pr *oduceInnerOrder*(). Line 12 judges whether *p* and *q* are non-self match. Lines 13–14 make pruning by cluster center distance described in Section 6.1.2. After clustering by *BitCluster*, the cluster tags which all subsequences belong to are known. The cluster information is used to search the clusters which *p* and *q* belong to. Furthermore, we can judge whether the pruning condition is satisfied based on Observation 1.

Algorithm 2. Bit representation clustering based discord discovery *BitClusterDiscord*

```
INPUT: Time series $T$ with length $n$, discord length $m$,
  compression ratio by PAA $cr$, clustering parameter $k$
OUTPUT: Discord starting position best_so_far_loc, the largest
  distance from discord to nearest neighbor best_so_far_dist
1   best_so_far_loc = -1;
2   best_so_far_dist = 0;
3   for $i$=1 to $n-m+1$
4     for subsequence $t_i...t_{i+m-1}$, use PAA bit serialization;
5   end of for
6   BitCluster(BD, k);
7   Pr oduceOuterOrder();
8   Pr oduceInnerOrder();
9   for p in T by outer loop order
10    nearest_neighbor_dist = inf inity;
11    for q in T by inner loop order
12      if ($|p-q| \geqslant m$)
13        if (distance between clusters which p and q belong
  to>=best_so_far_dist)
14          break;
15        if
  ($Dist(t_p...t_{p+m-1}, t_q...t_{q+m-1}) <$ nearest_neighbor_dist)
16          nearest_neighbor_dist = $Dist(t_p...t_{p+m-1}, t_q...t_{q+m-1})$;
17        if ($Dist(t_p...t_{p+m-1}, t_q...t_{q+m-1}) <$ best_so_far_dist)
18          break;
19      end of for
20      if (nearest_neighbor_dist > best_so_far_dist)
21        best_so_far_dist = nearest_neighbor_dist;
22        best_so_far_loc = p;
23  end of for
```

Lines 15–18 update the two distance values in the inner loop, *best_so_far_dist* and *nearest_neighbor_dist*, respectively. If the distance from current subsequence to its non-self match is smaller than current *best_so_far_dist*, even if this non-self match is not the nearest neighbor, we can verify that the current subsequence cannot be discord. So the inner loop can be terminated. Lines 20–22 update the current discord position and distance in the outer loop.

When finishing the outer and inner loop, we find discord in time series *T*. Variable *best_so_far_loc* records the starting position of discord in *T* and *best_so_far_dist* records the largest distance from discord to its nearest neighbor.

As Algorithm 2 is based on Algorithm 1, Algorithm 1 randomly produces *k* initial cluster centers at first, in order to avoid the fluctuation and influence by randomization, Algorithm 2 should be executed twenty times to obtain reliable results.

7. Experimental evaluation

In this section, we describe the experiments on extensive test datasets and evaluate our proposed discord discovery algorithm. For the computations, we used Pentium Dual 1.73 GHz CPU, 1G memory, 120G hard disk and Windows XP operating system. The programming language is C++. We evaluate the algorithm based on four performance metrics: effectiveness, efficiency, parameter analysis and scalability.

7.1. Effectiveness

In this section we show the utility of *BitClusterDiscord* to find discords in different application domains.

7.1.1. Discord discovery in space telemetry

We consider first finding discords in sensor time series. Fig. 4 gives an example, representing normal Space Shuttle Marotta Valve Time Series annotated by NASA engineer [1,21].

In Fig. 4, the zoom-in effect of the sub-graph in the top graph is shown in the bottom, which denotes a normal cycle with annotation. A normal cycle contains two phases, Energizing Phase and De-Energizing Phase, respectively.

Dataset TEK16 shown in Fig. 5 represents similar Marotta Valve Time Series [1,21]. This series has five energize/de-energize cycles. The bottom graph in Fig. 5 is zoom-in for the 5th cycle in the top graph. NASA engineer annotates the 5th cycle as Poppet pulled significantly out of the solenoid before energizing, so obviously this segment is discord. We apply Algorithm 2 on the test dataset TEK16 in Fig. 5, and we set the discord length to 128, the compression ratio to 4 and the number of clusters to 28. The position of discord with length 128 found by our algorithm *BitClusterDiscord* is identical with the position annotated by experts.

Dataset TEK17 shown in Fig. 6 illustrates another Marotta Valve Time Series [1,21]. The series has again five energize/de-energize cycles. The bottom graph in Fig. 6 is the zoom-in for the 3rd cycle in the top. NASA engineer annotates the 3rd cycle as Poppet pulled out of the solenoid before energizing, which is obviously discord. We apply Algorithm 2 on the test dataset TEK17 in Fig. 6, and we set the discord length to 128, the compression ratio to 4 and the number of clusters to 28.The discord with length 128 found by *BitClusterDiscord* has the identical starting position as that annotated by experts.

7.1.2. Discord discovery in electrocardiograms

Electrocardiograms (ECGs) are time series transited by electrical detection device and caused by heart beats. They have been validated as an important time series [8–10]. In Fig. 7 there is an example representing a normal ECGs data series annotated by cardiologist [1,22]. The bottom curve of Fig. 7 shows the zoom-in for a subsequence in the upper part. A normal ECG cycle includes five critical positions named as $\langle P, Q, R, S, T\rangle$ that show four special regular waves.

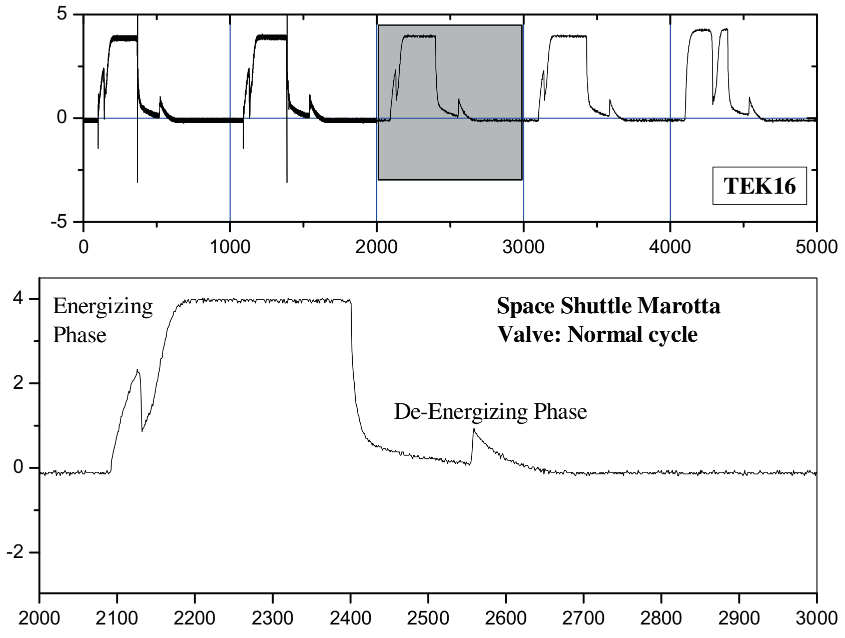

Fig. 4. Space Shuttle Marotta Valve time series (top) five cycles (bottom) a normal cycle.

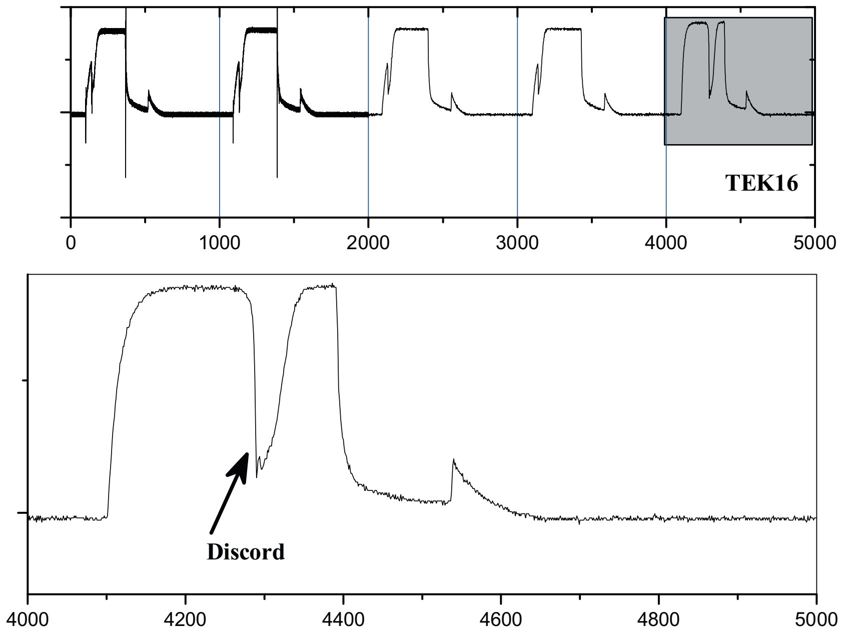

Fig. 5. Dataset TEK16 (top) five cycles (bottom) zoom-in of the 5th cycle, discord with length 128.

The subset of test data qtdbsele0606 is shown in Fig. 8 [1,22], where the bottom part is a zoom-in for the discord section annotated by experts. According to the medical suggestion by cardiologist, the discord length in ECGs is approximately 1/4 the length of a single normal heartbeat. We apply Algorithm 2 on the test dataset qtdbsele0606 in Fig. 8, and we set the discord length to 40, the compression ratio to 4 and the number of clusters to 28. The position of discord with length 40 found by *BitClusterDiscord* coincides with the discord position annotated by experts. We can see the ST wave in the below of Fig. 8 is obviously different with the normal wave in Fig. 7.

7.2. Efficiency

In this section we evaluate the efficiency of *BitClusterDiscord*. We choose three candidates to compare with *BitClusterDiscord*, i.e., naïve method denoted by *NaiveDiscord*, simple pruning method denoted by *Simple* Pr *uneDiscord* and HOT SAX, respectively.

The main idea of the simple pruning method is to achieve pruning by random sampling. The order of outer loop is same as naïve method, while the order of inner loop is generated randomly, and the early abandon condition is designed in the inner loop, if the distance from the current subsequence to its non-self match is

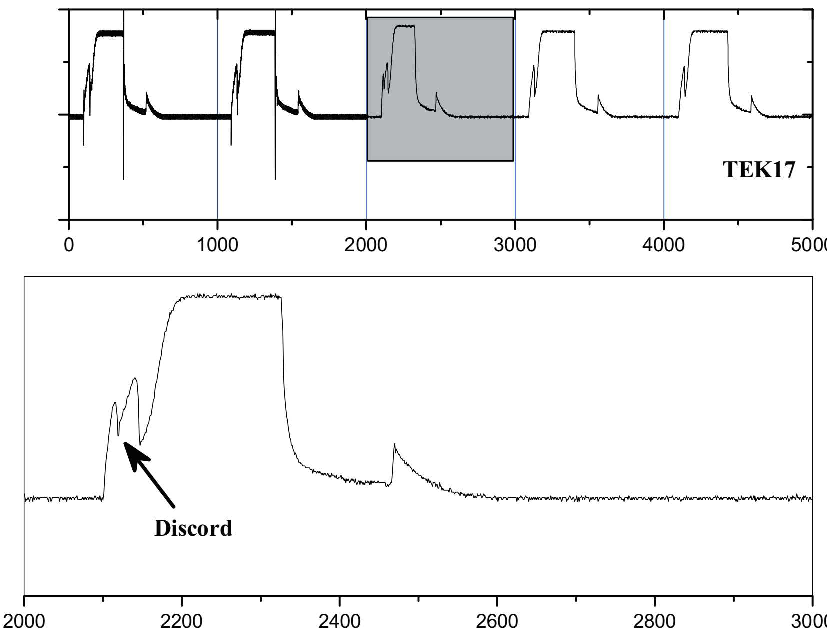

Fig. 6. Dataset TEK17 (top) five cycles (bottom) zoom-in of the 3rd cycle, discord with length 128.

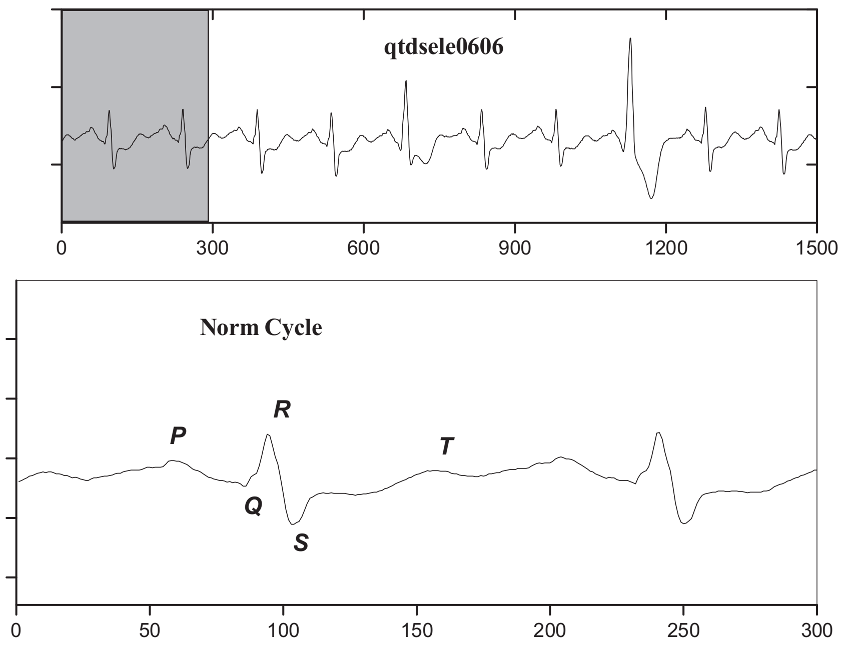

Fig. 7. ECG (top) normal ECGs data (bottom) a normal ECG cycle annotated by cardiologist.

smaller than the current *best_so_far_dist*, it performs the break sentence and achieves early abandon.

HOT SAX is a heuristic method based on SAX technique [1]. First it adopts SAX to reduce dimensionality, then designs the heuristic strategies for the outer and inner loop based on two data structures, i.e., array and trie tree. In the outer loop, the element with the smallest count in the array has the high priority to be chosen; in the inner loop, trie tree is searched to find the elements in the linked list index at the leaf nodes according to the current candidate *i* in the outer loop, so as to achieve early abandon.

In this section, we not only compare the efficiency of *BitClusterDiscord* with that of candidate algorithms, but also analyze the efficiency of three major steps in *BitClusterDiscord*.

7.2.1. Efficiency comparisons with candidate algorithms

We measure the running time and number of calls to Euclidean distance (ED) as efficiency metrics, comparing *BitClusterDiscord* with three candidate algorithms, i.e., *NaiveDiscord*, *Simple* Pr *uneDiscord* and HOT SAX, respectively. The experiments are conducted on ten datasets. The experimental results are obtained by

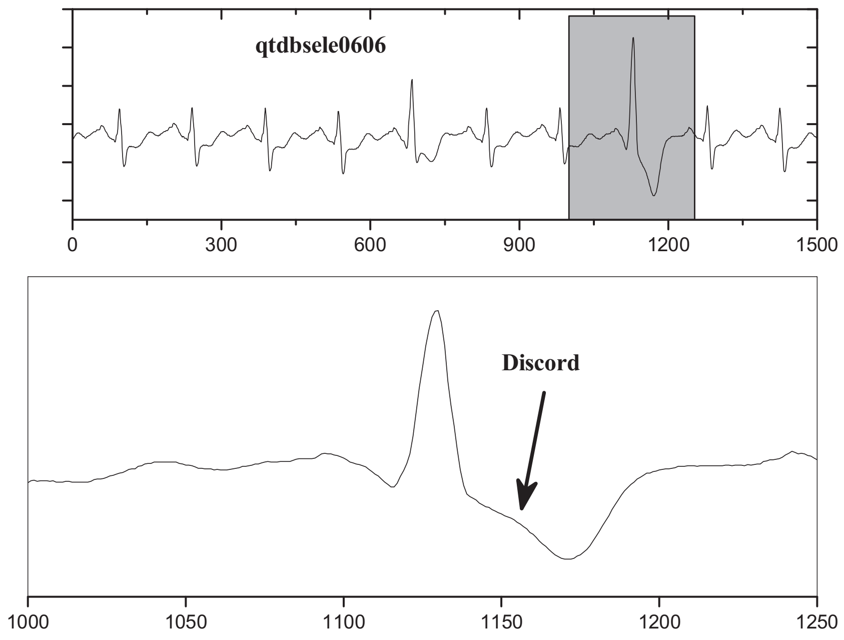

Fig. 8. Dataset qtdbsele0606 (top) whole series (bottom) zoom-in, discord with length 40.

Table 1
Time comparison of four algorithms (unit: second).

Dataset	Len	N	S	H	B
TEK16	128	83.5259 ± 0.301	10.732 ± 0.245	2.209 ± 0.049	1.446 ± 0.219
TEK17	128	83.0274 ± 0.234	9.5749 ± 0.222	4.613 ± 0.108	6.485 ± 0.518
chfdb_chf01_275	40	17.25 ± 0.0308	1.901 ± 0.047	0.326 ± 0.017	1.432 ± 0.251
chfdb_chf13_45590	40	17.2335 ± 0.066	1.838 ± 0.085	0.366 ± 0.009	0.194 ± 0.055
qtdbsele0606	40	295.3013 ± 10.894	15.427 ± 2.486	1.381 ± 0.072	1.461 ± 0.163
xmitdb_x108_0	120	102.783 ± 7.1427	3.138 ± 0.413	4.828 ± 0.122	1.238 ± 0.268
mitdb_100_180	120	92.3787 ± 0.213	9.072 ± 0.201	6.71 ± 0.165	6.367 ± 0.314
mitdbx_mitdbx_108	120	1671.029 ± 105.822	44.902 ± 3.46	17.608 ± 0.426	4.588 ± 1.03
nprs43	160	1770.591 ± 153.257	31.302 ± 3.765	14.49 ± 5.85	5.884 ± 0.697
nprs44	160	2739.1688 ± 178.462	46.492 ± 4.759	11.173 ± 1.203	6.594 ± 1.113

executing the algorithms twenty times. The results of CPU time on ten datasets are shown in Table 1.

In Table 1, the first column stands for ten datasets [21,22], the second column stands for the discord length, and the other four columns stand for running time in unit second corresponding to four algorithms. We use N, S, H and B for abbreviations to represent *NaiveDiscord*, *Simple* Pr *uneDiscord*, HOT SAX and *BitClusterDiscord*, and we report the means and standard deviations of running time in Table 1.

Here we set the discord length depending on the dataset. The ten datasets are classified into three data types. TEK16 and TEK17 belong to Space Shuttle data, nprs43 and nprs44 belong to Patient Monitoring data, and the other six datasets belong to ECGs data. For Space Shuttle data, we chose the length of discord according to the two phases in the normal cycle: Energizing Phase and De-Energizing Phase. For Patient Monitoring data, we chose the length of discord according to breath cycle. For ECGs, we chose the length of discord according to the heartbeat cycle, the six datasets in ECGs did not have the same totally sampling rate, so the discord lengths were not the same.

We set the compression ratio to 4, the cluster number to 28, the alphabet size of SAX to 4 in our experiments. From Table 1 we can observe a common regularity that the CPU time of *BitClusterDiscord* is smaller than with the competing methods on most datasets.

The four algorithms need to compute ED between subsequences. The number of times that ED function is called occupies a large proportion in the total time of algorithm, so we compare number of calls to ED in four algorithms. As we focus on the orders of magnitude of ED calls, we evaluate the *log* of number of ED calls, here the base of the *log* is 10. The results on ten datasets are shown in Table 2.

In Table 2, the first column stands for ten datasets, the second column stands for the discord length, and the other four columns stand for the *log* of number of calls to ED function corresponding to four algorithms, noted by N, S, H, B as Table 1. We can observe that number of ED calls in *BitClusterDiscord* is much smaller than in competing methods.

7.2.2. Efficiency comparisons of three major steps in Algorithm 2

Algorithm 2 *BitClusterDiscord* contains three major steps. They are PAA bit serialization, bit representation clustering and discord discovery based on clustering, denoted by *BitRes*, *BitCluster* and *BitDiscord*, respectively. The relative computational requirements of three steps on ten datasets are illustrated in Table 3.

In Table 3, the first column denotes the ten datasets, the second column stands for the discord length, and the other three columns denote the time proportions of three steps in Algorithm 2, in the form of means and standard deviations. The results are obtained by twenty times executions. According to Table 3, clearly largest part of the computing time is consumed by the third discord discovery step.

Table 2
log(Number of ED calls) comparison of four algorithms.

Dataset	Len	N	S	H	B
TEK16	128	7.3525	6.3959 ± 0.0056	5.7508 ± 0.00998	3.6878
TEK17	128	7.3525	6.3395 ± 0.0056	6.074 ± 0.0096	3.6878
chfdb_chf01_275	40	7.1298	5.9758 ± 0.0109	5.3302 ± 0.0261	3.5696
chfdb_chf13_45590	40	7.1296	5.9544 ± 0.0092	5.39 ± 0.0086	3.5695
qtdbsele0606	40	8.3476	6.4062 ± 0.0071	5.965 ± 0.0145	4.175
xmitdb_x108_0	120	7.4255	5.5299 ± 0.0062	6.1099 ± 0.0081	3.7223
mitdb_100_180	120	7.4256	6.3335 ± 0.0098	6.2692 ± 0.0111	3.7228
mitdbx_mitdbx_108	120	8.6592	6.4957 ± 0.018	6.6797 ± 0.0111	4.332
nprs43	160	8.4975	6.191 ± 0.0195	6.1923 ± 0.0223	4.2527
nprs44	160	8.7533	6.3182 ± 0.0167	6.3139 ± 0.012	4.3796

Table 3
Time proportion of three steps in total time.

Dataset	Len	BitRes	BitCluster	BitDiscord
TEK16	128	3.32% ± 0.51%	18.11% ± 3.03%	78.57% ± 3.53%
TEK17	128	0.77% ± 0.10%	4.02% ± 0.38%	95.21% ± 0.42%
chfdb_chf01_275	40	1.35% ± 0.55%	4.93% ± 1.18%	93.72% ± 1.41%
chfdb_chf13_45590	40	9.58% ± 3.86%	40.12% ± 11.73%	50.29% ± 14.10%
qtdbsele0606	40	5.29% ± 1.80%	19.83% ± 3.25%	74.54% ± 4.50%
xmitdb_x108_0	120	4.48% ± 1.35%	23.68% ± 5.55%	71.83% ± 6.71%
mitdb_100_180	120	0.81% ± 0.11%	4.35% ± 0.25%	94.84% ± 0.25%
mitdbx_mitdbx_108	120	4.75% ± 1.09%	25.13% ± 5.52%	70.12% ± 6.60%
nprs43	160	3.69% ± 0.43%	18.36% ± 2.22%	77.95% ± 2.60%
nprs44	160	4.48% ± 0.78%	22.13% ± 3.46%	73.39% ± 4.23%

7.3. Parameter analysis

Algorithm 2 *BitClusterDiscord* has two parameters, the compression ratio by PAA dimensionality reduction and number of clusters. These two parameters only affect the efficiency of the algorithm *BitClusterDiscord*, not the final result of the discord location, which depends only on the discord length. So we observe and analyze how they affect the running time of *BitClusterDiscord*.

7.3.1. Compression ratio

As mentioned above, *BitClusterDiscord* includes three major steps. We note the computing time related to the three steps as *BitResTime*, *BitClusterTime* and *BitDiscordTime*, respectively. The total consumed time of *BitClusterDiscord* is noted by *TotalTime*. We first analyze the change regularity of the computing time when compression ratio changes.

The experimental results on two datasets TEK16 and TEK17 are shown in Figs. 9 and 10.

From Figs. 9 and 10 we can observe that with the gradual increasing of compression ratio, *BitResTime* gradually decreases and *BitClusterTime* also decreases, while *BitDiscordTime* gradually increases and *TotalTime* also increases. When the compression ratio becomes larger, the following regularity can be inferred:

– The number of segments obtained by PAA decreases, thus the trend about adjacent PAA segments decreases. Consequently, *BitResTime* should decrease.
– The length of bit representation decreases, thus clustering will cost less time, that is, *BitClusterTime* will decrease.
– The resolution level of trend reflected by bit representation for raw subsequence is not high enough, which causes that the pruning effect in clustering-based discord discovery is not as expected, consequently, *BitDiscordTime* increases, causing *TotalTime* to increase.

The experimental results coincide with the theoretical inference, which validates the theoretical regularity.

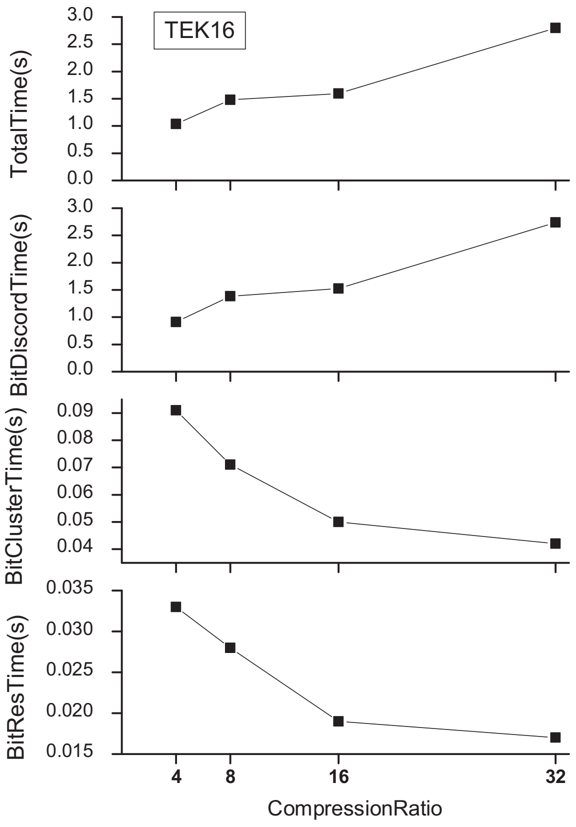

Fig. 9. Compression ratio vs. time on TEK16.

7.3.2. Number of clusters

Here we analyze how *BitClusterTime*, *BitDiscordTime* and *TotalTime* change with the number of clusters. Experimental results on datasets TEK16 and TEK17 are shown in Figs. 11 and 12.

From Figs. 11 and 12 we can observe the common regularity on both TEK16 and TEK17, that is, when number of clusters becomes larger, *BitClusterTime* increases. With the increase of number of clusters, *BitDiscordTime* and *TotalTime* on TEK16 gradually decrease, although both kinds of time on TEK17 have fluctuation, in the overall view, it shows the gradual falling trend. Let us analyze this trend, when number of clusters increases, Algorithm 1 *BitCluster* will spend more computation cost for assigning non-representative objects to their nearest clusters, causing *BitClusterTime* increases. As the objects in the same cluster will have high similarity, when number of clusters increases, bringing higher discrimination among different clusters and achieving better pruning power, so *BitDiscordTime* and *TotalTime* will gradually decrease.

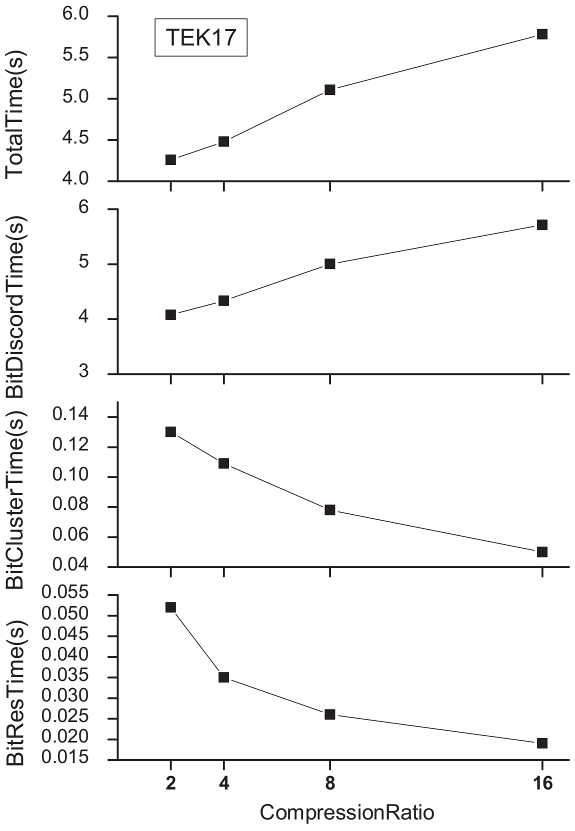

Fig. 10. Compression ratio vs. time on TEK17.

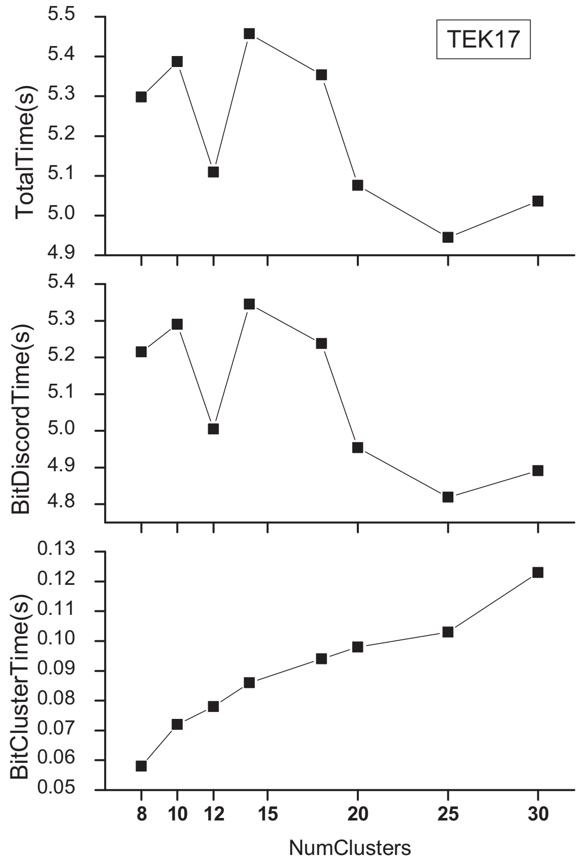

Fig. 12. Cluster number vs. time on TEK17.

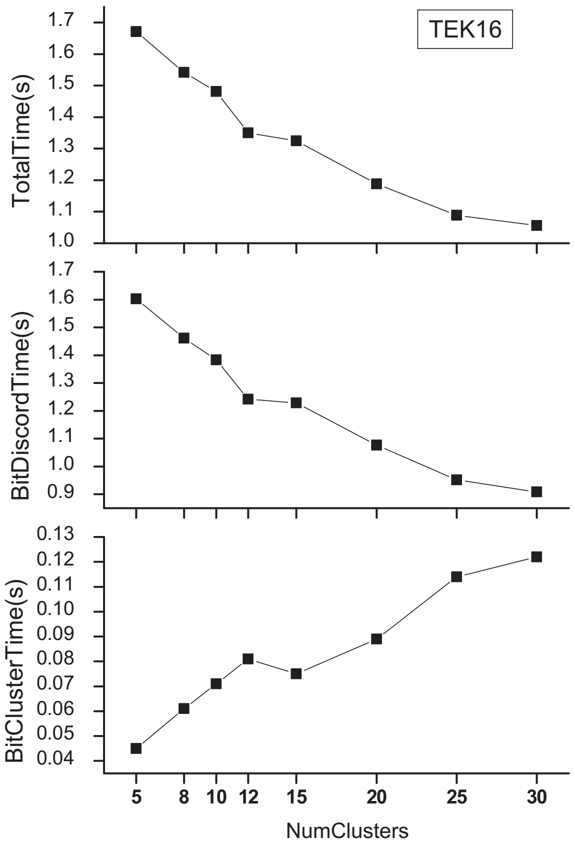

Fig. 11. Cluster number vs. time on TEK16.

Table 4
Scalability of *log*(number of ED calls).

Subset size	Random_walk		Power_data	
	NaiveDiscord	*BitClusterDiscord*	*NaiveDiscord*	*BitClusterDiscord*
1k	5.85	2.97	5.8	2.96
2k	6.54	3.29	6.52	3.29
4k	7.18	3.6	7.17	3.6
8k	7.81	3.91	7.8	3.9
16k	8.42	4.21	8.42	4.21
32k	9.03	4.52	9.03	4.51

7.4. Scalability

Scalability analysis mainly investigates how the efficiency of Algorithm 2 *BitClusterDiscord* changes with the increase of data size. We measure the scalability of the number of ED calls and the running time speedup.

7.4.1. Number of ED calls

We choose Random_walk and Power_data [1] as test datasets and compare *NaiveDiscord* with *BitClusterDiscord*. As the purpose of scalability is to measure the orders of magnitude of a speedup in terms of ED calls, we chose to report the *log* of the number of ED calls, here the base of the *log* is 10. For the results we run the algorithms with different settings for twenty times. The experimental results are shown in Table 4.

In Table 4 the 1st column stands for the size of subset of data, the 2nd and 3rd columns stand for the *log* of number of ED calls

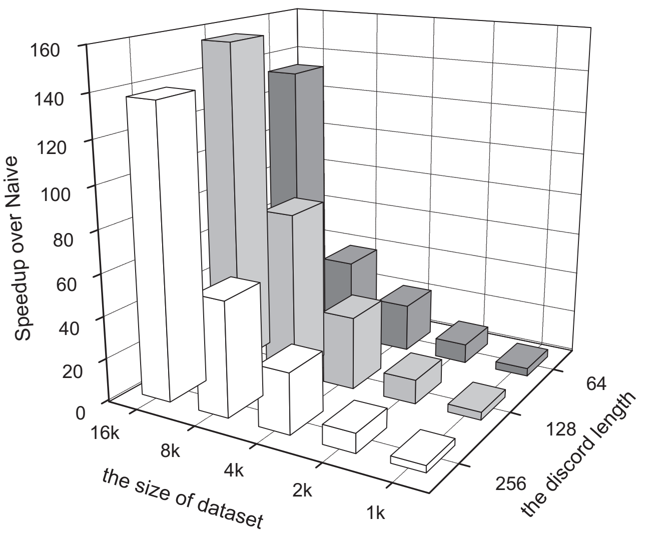

Fig. 13. Scalability of running time on Random_walk.

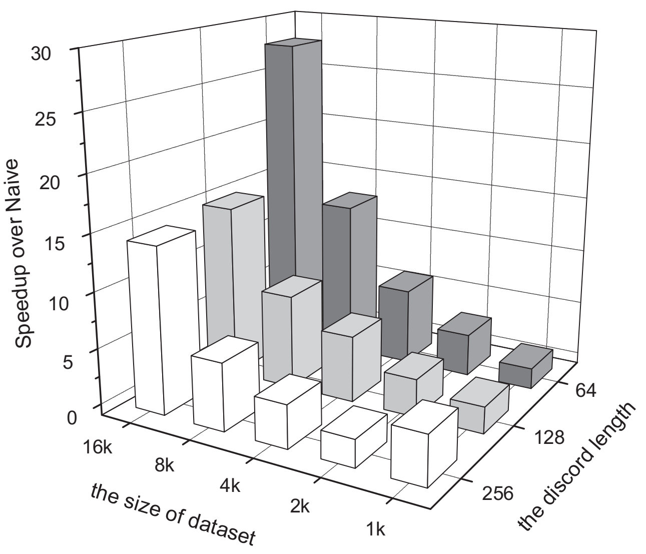

Fig. 14. Scalability of running time on Power_data.

of two algorithms on Random_walk, and the 4th and 5th columns stand for the *log* of number of ED calls of two algorithms on Power_data.

From Table 4 we can observe, as the size of subset of data increases, number of ED calls in *NaiveDiscord* increases by order of magnitude, while number of ED calls in *BitClusterDiscord* scales clearly better.

7.4.2. Running time speedup

We evaluate the running time speedup of *BitClusterDiscord* over *NaiveDiscord*. The test datasets are Random_walk and Power_data. The results are shown in the three-dimension graphs (Figs. 13 and 14), wherein axis *X* stands for the size of dataset, axis *Y* stands for the length of discord, and axis *Z* stands for speedup of *BitClusterDiscord* over *NaiveDiscord*.

From Figs. 13 and 14 we can observe that, by increasing the size of dataset and changing the length of discord, the speedup shows an ascendant trend. Experimental evaluation from the two aspects, i.e., number of ED calls and running time speedup, shows that Algorithm 2 *BitClusterDiscord* has good scalability.

8. Conclusion

This paper focuses on studying the problem of finding time series discord. Given a raw time series, we segment and represent data by the PAA bit serialization method. Based on bit representation, we propose a new clustering algorithm *BitCluster*. *BitCluster* allocates subsequences with similar change patterns into the same cluster, assuring subsequences in the same cluster have high similarity, so as to accelerate discord discovery and realize meaningful filtering. Based on the presented clustering algorithm, we propose the discord discovery algorithm *BitClusterDiscord* which utilizes two pruning strategies. We conduct extensive experiments on datasets to evaluate the proposed algorithm from four aspects. The experimental results show that the proposed algorithm can find discord effectively and efficiently. In the future we will apply the proposed algorithm to streaming time series.

Acknowledgements

We thank anonymous reviewers and Prof. S. Tang for their valuable comments and suggestions. This research was supported in part by the National Natural Science Foundation of China (No. 61202171), the Program for New Century Excellent Talents in University (No. NCET-12-0953) and the Fundamental Research Funds for the Central Universities, China University of Geosciences (Wuhan) (No. CUGL120281).

References

[1] E. Keogh, J. Lin, A. Fu, HOT SAX: efficiently finding the most unusual time series subsequence, in: J. Han, B.W. Wah, V. Raghavan, et al. (Eds.), ICDM, Houston, Texas, 2005, IEEE Computer Society, Washington, DC, 2005, pp. 226–233.
[2] A. Fu, T. Leung, E. Keogh, et al., Finding time series discords based on Harr transform, in: X. Li, O. Zaiane, Z. Li (Eds.), ADMA, Xi'an, China, Springer, 2006, pp. 31–41.
[3] Y. Bu, O. Leung, A.W. Fu, et al., WAT: finding top-K discords in time series database, in: Proceedings of 7th SIAM International Conference on Data Mining, SIAM, Minneapolis, Minnesota, USA, 2007.
[4] W. Li, E. Keogh, X. Xi, SAXually explicit images: finding unusual shapes, in: P. Perner (Ed.), Proc. of the 6th Int'l Conf. on Data Mining. Hong Kong, China, IEEE Computer Society, Washington, DC, 2006, pp. 711–720.
[5] D. Yankov, E. Keogh, U. Rebbapragada, Disk aware discord discovery: finding unusual time series in terabyte sized datasets, in: Y.W.S. Stanley (Ed.), Proc. of the 23rd Int'l Conf. on Data Engineering, Istanbul, Turkey, 2007, IEEE Computer Society, Washington DC, 2007, pp. 381–390.
[6] J. Shieh, E. Keogh, ISAX: indexing and mining terabyte sized time series, in: Y. Li, B. Liu, S. Sarawagi (Eds.), Proc. of ACM SIGKDD, Las Vegas, Nevada, USA, 2008, ACM Press, New York, 2008, pp. 623–631.
[7] J. Wang, F.-L. Chung, Z. Deng, et al., Weighted spherical 1-mean with phase shift and its application in electrocardiogram discord detection, Artificial Intelligence in Medicine 57 (1) (2013) 59–71.
[8] A. Koski, M. Juhola, M. Meriste, Syntactic recognition of ECG signals by attributed finite automata, Pattern Recognition 28 (12) (1995) 1927–1940.
[9] S. Kim, P. Smyth, Segmental hidden markov models with random effects for waveform modeling, Journal of Machine Learning Research 7 (2006) 945–969.
[10] A. Kampouraki, G. Manis, C. Nikou, Heartbeat time series classification with support vector machines, IEEE Transactions on Information Technology in Biomedicine 13 (4) (2009) 512–518.
[11] D. Fisch, T. Gruber, B. Sick, SwiftRule: mining comprehensible classification rules for time series analysis, IEEE Transactions on Knowledge and Data Engineering 23 (5) (2011) 774–787.
[12] Z. Syed, J. Guttag, Unsupervised similarity-based risk stratification for cardiovascular events using long-term time-series data, Journal of Machine Learning Research 12 (2011) 999–1024.
[13] X. Wang, A. Mueen, H. Ding, et al., Experimental comparison of representation methods and distance measures for time series data, Data Mining and Knowledge Discovery 26 (2) (2013) 275–309.
[14] H. Li, C. Guo, Piecewise cloud approximation for time series mining, Knowledge-Based Systems 24 (4) (2011) 492–500.
[15] X. Li, Similarity Search on Time Series Based on Clustering Bit Sequences with Changing Mode. Master Thesis, Huazhong University of Science and Technology, 2007.

[16] E. Keogh, K. Chakrabarti, M. Pazzani, et al., Dimensionality reduction for fast similarity search in large time series databases, Knowledge and Information Systems 3 (3) (2000) 263–286.
[17] A. Bagnall, C.A. Ratanamahatana, E. Keogh, et al., A bit level representation for time series data mining with shape based similarity, Data Mining and Knowledge Discovery 13 (1) (2006) 11–40.
[18] K. Lu, S. Lin, J. Qiao, et al., FSMBO: fast time series similarity matching based on bit operation, in: G. Wang, J. Chen, M.R. Fellow (Eds.), Proc. of the 9th Int'l Conf. for Young Computer Scientists, Zhang Jia Jie, Hunan, China, IEEE Computer Society, Washington, DC, 2008, pp. 1866–1871.
[19] C.A. Ratanamahatana, E. Keogh, A.J. Bagnall, et al., A novel bit level time series representation with implications for similarity search and clustering, in: T.B. Ho, D.W. Cheung, Huan Liu (Eds.), PAKDD, Hanoi, Vietnam, Springer, 2005, pp. 771–777.
[20] L. Kaufman, P.J. Rousseeuw, Finding Groups in Data: An Introduction to Cluster Analysis, John Wiley & Sons, 1990.
[21] http://www.cs.ucr.edu/~eamonn/.
[22] http://physionet.org/physiobank/.