Discord Monitoring for Streaming Time-series

Shinya Kato, Daichi Amagata, Shunya Nishio, and Takahiro Hara

Osaka University, Osaka, Japan
{kato.shinya, amagata.daichi, nishio.syunya, hara}@ist.osaka-u.ac.jp

Abstract. Many applications generate time-series and analyze it. One of the most important time-series analysis tools is anomaly detection, and discord discovery aims at finding an anomaly subsequence in a time-series. Time-series is essentially dynamic, so monitoring the discord of a streaming time-series is an important problem. This paper addresses this problem and proposes SDM (Streaming Discord Monitoring), an algorithm that efficiently updates the discord of a streaming time-series over a sliding window. We show that SDM is approximation-friendly, i.e., the computational efficiency is accelerated by monitoring an approximate discord with theoretical bound. Our experiments on real datasets demonstrate the efficiency of SDM and its approximate version.

Keywords: Time-series · Discord · Sliding window

1 Introduction

Motivation. Many real-world applications generate time-series and want to utilize it for obtaining useful knowledge [6]. Anomaly or outlier detection supports this, because it can find unusual observations and helps data cleaning, thereby time-series anomaly (or outlier) detection has been extensively studied [19, 23]. One of the most effective time-series anomaly detection is discord discovery [10, 11, 21]. Given a time-series, the discord of the time-series is the subsequence with the largest distance to its nearest neighbor among all subsequences (the formal definition is introduced in Section 2). Fig. 1 illustrates an example, and the red subsequence is the discord of an ECG time-series.

It has been shown that discord discovery can be employed in industry [3], medical care [20], and Web [21]. Note that time-series generated in the above applications is essentially dynamic [12]. Therefore, the discord of a time-series is updated over time. This fact renders an important problem of discord monitoring of a streaming time-series. We address this problem with a count-based sliding window setting in this paper.

Technical challenge. When the window slides, we have a new subsequence and the oldest subsequence expires. Due to them, the nearest neighbor of each subsequence may change, i.e., the discord may also change. Applications, which employ discord monitoring, require real-time update of the discord, hence we need to evaluate whether the discord is updated or not.

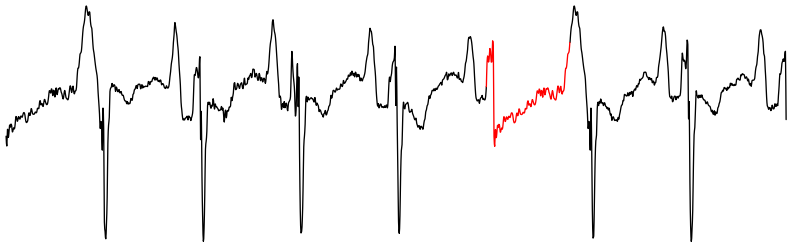


Fig. 1. A streaming ECG time-series and its discord (the red subsequence)

A straightforward approach to achieve this is to re-evaluate the nearest neighbor of each subsequence whenever the window slides. However, this approach obviously incurs an expensive cost. Although an existing discord detection algorithm for *static* time-series [10] can also be utilized for discord monitoring, it cannot incrementally update the discord. Therefore, when the discord changes, it has to re-evaluate the discord from scratch, which is not suitable for real-time monitoring, as shown in our experiments.

From the above discussion, we see that an efficient solution needs to have the following properties: it can (i) evaluate whether the discord changes or not quickly and (ii) discover the discord when it is updated by the window slide as soon as possible.

Overview of our solution and contribution. We devise SDM (Streaming Discord Monitoring) to achieve such a solution. SDM exploits a nearest neighbor search based on *sequential scan*, which is fast for high-dimensional data [18]. This approach is efficient for identifying the subsequences whose nearest neighbors change, thus satisfies the property (i). Besides, SDM maintains two kinds of nearest neighbor for each subsequence, to prune unnecessary computation when the discord changes, which enables result update without computing the discord from scratch, i.e., SDM satisfies the property (ii). SDM furthermore reduces the worst update time by using an approximation with bound guarantee.

To summarize, our contributions are as follows:

- We address the problem of time-series discord monitoring over a sliding window. To the best our knowledge, this paper is the first to address this problem.
- We propose SDM to efficiently solve the problem.
- We propose an approximate version of SDM, namely A-SDM, which reduces the update time of the worst case and provides a theoretical guarantee w.r.t. the monitoring result.
- We empirically evaluate SDM and A-SDM on real datasets, and the results show that they quickly update the discord and the worst update time is significantly faster than those of competitors.

Road-map. Section 2 defines the problem of this paper and Section 3 introduces related works. In Section 4, we present our algorithm SDM, and in Section 5, we show our experimental results. Finally, in Section 6, we conclude this paper.

2 Problem definition

A streaming time-series t is an unbound sequence of real values and represented as $t = (t[1], t[2], ...)$. First of all we define subsequence of t.

Definition 1 (Subsequence). *Given a time-series t and a subsequence size l, the subsequence s_p, which starts at $t[p]$, is defined as follows.*

$$s_p = (t[p], t[p+1], ..., t[p+l-1])$$

Let $s_p[x]$ be the x-th value of s_p, then $s_p = (s_p[1], s_p[2], ..., s_p[l])$. Next, we use z-normalized Euclidean distance between two subsequences. Note that z-normalized Euclidean distance is often utilized to measure the similarity between time-series [15, 22].

Definition 2 (z-normalized Euclidean distance). *Given two subsequences s_p and s_q with length l, their z-normalized Euclidean distance, $dist(s_p, s_q)$, is.*

$$dist(s_p, s_q) = \sqrt{\sum_{i=1}^{l} (\frac{s_p[i] - \mu(s_p)}{\sigma(s_p)} - \frac{s_q[i] - \mu(s_q)}{\sigma(s_q)})^2},$$

where $\mu(s)$ and $\sigma(s)$ are respectively the mean and standard deviation of $\{s[1], ..., s[l]\}$.

It is obvious that $dist(s_p, s_{p+1})$ is small but this observation is not interesting and interrupts obtaining a meaningful result [2, 14]. We therefore ignore trivial matched subsequences, which are defined below.

Definition 3 (Trivial match) [5]. *The set S_p of subsequences that have trivial match relationships with s_p is*

$$S_p = \{s_q \mid p - l + 1 \leq q \leq p + l - 1\}.$$

Here, the applications introduced in Section 1 are interesting only in recent values of a streaming time-series [13]. We hence employ a count-based sliding window to consider only the most recent w values of the time-series t, as with existing works [9, 12]. That is, we can represent t on the sliding window as $t = (t[i], ..., t[i + w - 1])$, and $t[i + w - 1]$ is the latest value of t. When the window slides, a new subsequence, which has $t[i + w - 1]$, is generated and the subsequence, which has $t[i-1]$, expires. Note that, when we refer to a subsequence s of t, s is on the window hereafter. Note furthermore that all subsequences on the window are memory-resident.

Our problem is discord monitoring, and the discord is obtained from the nearest neighbor of each subsequence. We therefore define the nearest neighbor of a given subsequence s.

Definition 4 (Nearest neighbor). *Given a subsequence s_p and a set S of all the subsequences of a streaming time-series t, the nearest neighbor of s_p is the subsequence that satisfies* $\mathrm{argmin}_{s_q \in S \setminus S_p} dist(s_p, s_q)$.

Let $s_p.dist_{NN}$ be the distance between s_p and its the nearest neighbor, then our problem is formally defined as follows.

PROBLEM DEFINITION. *Given a streaming time-series t, a windows size w, and a subsequence size l, we monitor the discord s^* of t that satisfies*

$$s^* = \underset{s_p \in S}{\operatorname{argmax}}\, s_p.dist_{NN}.$$

Applications that employ discord monitoring requires real-time update of the discord. We therefore aim at minimizing computation time to update s^*.

3 Related work

Although there exist many works that mine useful information from time-series [6] and temporal data [1, 7], we here focus on existing studies that have addressed discord detection/monitoring.

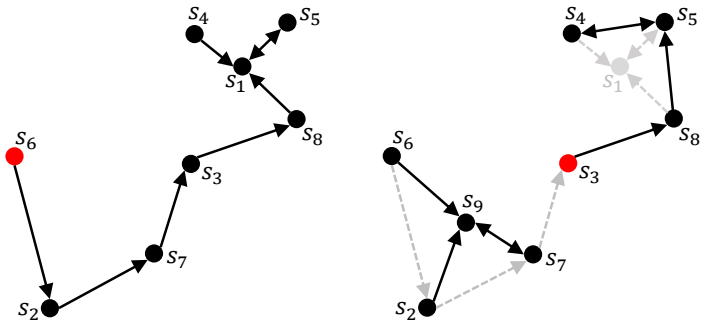
Discord discovery of a static time-series. Several works have proposed efficient discord discovery algorithms for a *static* time-series. Literature [10] has proposed HOT SAX to find the discord of a time-series that residents in-memory. HOT SAX basically computes the nearest neighbor for each subsequence s_p, but terminates the computation when we know that an upper-bound of $s_p.dist_{NN}$ is less than $s^*.dist_{NN}$. To enable this early termination efficiently, HOT SAX transforms subsequences to symbols (or strings), and uses an idea that the distance between two subsequences with similar symbols tends to be small. Hence, HOT SAX evaluates the distance of subsequences with similar symbols in an early iteration. We can employ HOT SAX for our environment but incurs significant cost when s^* expires, i.e., the update time in the worst case is too long. We show this in Section 5.

Literature [21] assumes that a given time-series is disk-resident and has proposed an algorithm based on linear scan, which deals with a totally different setting to ours. Literature [8] has proposed a parallel algorithm for discord detection. We note that parallel computation is beyond the scope of this paper.

Discord monitoring of a dynamic time-series. Literatures [17, 22] have proposed discord monitoring algorithms for a streaming time-series. Literature [17] utilizes an R-tree to quickly evaluate whether a new subsequence can become the discord or not. On the other hand, literature [22] employs a data structure Matrix Profile, which has been originally proposed for discord discovery of a static time-series. Unfortunately, they consider append-only case, and dealing with deletions of subsequences is not trivial for them.

4 SDM: Streaming Discord Monitoring

Recall that, when the window slides, a new subsequence s_n is generated and the oldest subsequence s_e expires. They may change the discord, because



(a) Before s_9 is generated (b) After s_9 is generated

Fig. 2. A streaming time-series on a sliding window with $w = 8$. The arrows indicate the nearest subsequences, and an arrow length represents the distance to the nearest subsequence. The red subsequence is the discord.

- s_n may become the discord,
- the subsequences, whose nearest neighbors were s_e, may become the discord,
- if s_e is the discord, s^* certainly changes, and
- the nearest neighbors of some subsequences may become s_n, which also may derive the discord update.

Example 1. *Fig. 2 illustrates a streaming time-series t on a sliding window with $w = 8$. Assume $l = 2$, thus each subsequence s of t is represented by a two-dimensional point. Note that the arrows indicate the nearest subsequences (for example, the nearest subsequence of s_1 is s_5), and arrow length shows the distance to the nearest subsequence. In Fig. 2(a), the discord is s_6, and after the window slides, s_1 expires and s_9 is generated as shown in Fig. 2(b). We see that the nearest neighbors of s_4, s_5, s_6, s_7, and s_8 change, due to the expiration of s_1 and the generation of s_9. This results in the discord update (from s_6 to s_3).*

4.1 Main idea

Our algorithm SDM achieves an efficient discord update by using simple data structures and by exploiting sequential scan. More specifically, we solve the problem with the following ideas:

1. If we maintain the nearest neighbor of each subsequence and the distance, we can easily identify the discord.
2. By scanning all subsequences, we can compute the nearest neighbor of s_n, update the nearest neighbors of the other subsequences incrementally, and identify the subsequences whose nearest neighbors were s_e before the window slides.

The scan-based approach (i.e., the second idea) seems simple, but it is effective for efficient discord monitoring. One may consider an index-based approach,

e.g., HOT SAX, to compute the nearest neighbor of s_n fast by pruning some subsequences. Such an approach unfortunately loses chances of updating the nearest neighbors of pruned subsequences, although they may need to be updated. This consequently incurs an expensive cost to verify the nearest neighbors of many subsequences. In Section 5, we show that such an approach takes a significant update cost in the worst case, suggesting that it is not suitable for real-time monitoring.

4.2 Data structure

SDM maintains *NN_{older}-tuple*, *$NN_{younger}$-tuple*, and *NN-tuple* for each subsequence.

DEFINITION 5 (NN_{older}-TUPLE). *NN_{older}-tuple of a subsequence s_p is a pair of*

- *$s_p.id_{NN_{older}}$: the identifier of the subsequence which is nearest to s_p among the set of subsequences which have been generated before s_p is generated, and*
- *$s_p.dist_{NN_{older}}$: the distance to the above nearest subsequence.*

DEFINITION 6 ($NN_{younger}$-TUPLE). *$NN_{younger}$-tuple of a subsequence s_p is a pair of*

- *$s_p.id_{NN_{younger}}$: the identifier of the subsequence which is nearest to s_p among the set of subsequences which have been generated after s_p is generated, and*
- *$s_p.dist_{NN_{younger}}$: the distance to the above nearest subsequence.*

DEFINITION 7 (NN-TUPLE). *NN-tuple of a subsequence s_p is a pair of the identifier of the nearest neighbor of s_p and $s_p.dist_{NN}$.*

Note that, if $s_p.dist_{NN_{older}} < s_p.dist_{NN_{younger}}$, NN-tuple of s_p is its NN_{older}-tuple. Otherwise, NN-tuple of s_p is its $NN_{younger}$-tuple. Furthermore, if NN-tuple of s_p is its $NN_{younger}$-tuple, i.e., $s_p.id_{NN_{older}} \geq s_p.id_{NN_{younger}}$, we do not need to consider its NN_{older}-tuple, because its NN_{older}-tuple never becomes its NN-tuple. This is an important observation to avoid unnecessary distance computation.

EXAMPLE 2. *Fig. 3 summarizes NN_{older}-tuple, $NN_{younger}$-tuple, and NN-tuple of each subsequence in Fig. 2. In Fig. 3(a), for example, NN_{older}-tuple of s_2 is empty, because $s_2.dist_{NN_{older}} > s_2.dist_{NN_{younger}}$. In Fig. 3(b), highlighted parts are updated parts from Fig. 3(a), which are derived from the expiration of s_1 and the generation of s_9.*

We can see that the discord can be obtained from NN-tuple of each subsequence, e.g., s_6 and s_3 in Figs. 3(a) and 3(b), respectively. That is, if we can efficiently update NN-tuples, we can efficiently monitor the discord. We use sequential scan to do this, as discussed in Section 4.1.

Why we use older and younger nearest neighbors. Let s_p be the older nearest neighbor of s_q, i.e., $s_q.id_{NN_{older}} = p$. Also, s_r be the younger nearest neighbor of s_q. It is trivial that s_p expires before s_q does. When s_p expires,

Subsequence	NN_{older}-tuple	$NN_{younger}$-tuple	NN-tuple
s_1	-	⟨5,1.0⟩	⟨5,1.0⟩
s_2	-	⟨7,2.7⟩	⟨7,2.7⟩
s_3	-	⟨8,2.5⟩	⟨8,2.5⟩
s_4	⟨1,0.8⟩	⟨5,1.5⟩	⟨1,0.8⟩
s_5	⟨1,1.0⟩	⟨8,1.8⟩	⟨1,1.0⟩
s_6	-	⟨7,2.8⟩	⟨2,3.0⟩
s_7	⟨3,2.2⟩	⟨8,3.2⟩	⟨3,2.2⟩
s_8	⟨1,1.4⟩	-	⟨1,1.4⟩

(a) Before the window slides

Subsequence	NN_{older}-tuple	$NN_{younger}$-tuple	NN-tuple
s_2	-	⟨9,1.5⟩	⟨9,1.5⟩
s_3	-	⟨8,2.5⟩	⟨8,2.5⟩
s_4	-	⟨5,1.5⟩	⟨5,1.5⟩
s_5	⟨4,1.3⟩	⟨8,1.8⟩	⟨4,1.3⟩
s_6	-	⟨9,2.4⟩	⟨9,2.4⟩
s_7	-	⟨9,1.2⟩	⟨9,1.2⟩
s_8	⟨5,1.6⟩	⟨9,4.2⟩	⟨5,1.6⟩
s_9	⟨7,1.2⟩	-	⟨7,1.2⟩

(b) After the window slided

Fig. 3. NN_{older}-tuple, $NN_{younger}$-tuple, and NN-tuple of each subsequence in Fig. 2

if $dist(s_p, s_q) < dist(s_q, s_r)$, we do not know the nearest neighbor of s_q. (If $dist(s_p, s_q) \geq dist(s_q, s_r)$, s_r is the nearest neighbor of s_q, so we do nothing in this case.) However, if $s_q.dist_{NN_{younger}} = dist(s_q, s_r)$ is smaller than $s^*.dist_{NN}$, we can guarantee that s_q is not the discord, because $s_q.dist_{NN} \leq s_q.dist_{NN_{younger}}$. That is, we do not have to search its nearest neighbor, which reduces the update time. Furthermore, even if $dist(s_q, s_r)$ is not smaller than $s^*.dist_{NN}$, we just need to search the *older* nearest neighbor of s_q. The search space is only a set of subsequences s_p such that $p < q$, which is much smaller than the set of all subsequences.

We can see that if we employ only NN-tuple, we need a large update cost when the nearest neighbor of s_q expires. This is because we cannot prune its nearest neighbor search and its search space is large.

4.3 Algorithm description

Rationale of utilizing sequential scan. When the window slides, we need to confirm the discord update. This is to check the nearest neighbors of the other subsequences s_p. Recall that the nearest neighbor of s_p may become the new subsequence s_n. We see that for all s_p, s_n is younger, thereby the check (normally) corresponds to update the $NN_{younger}$-tuple of each subsequence. For fixed l, this check needs $O(w)$ time, because we need $O(1)$ time to check $\min\{s_p.dist_{NN_{younger}}, dist(s_p, s_n)\}$ for each s_p. This is exactly the same cost of sequential scan of all subsequences on the window.

Algorithm. SDM is designed based on the above analysis and the motivation of our data structure (Section 4.2), and is described in Algorithm 1. We first maintain a temporal discord s^*_{temp}, which is the previous discord (i.e., the one before the window slided) at initialization. Recall that S is the set of all subsequences. We second remove the expired subsequence s_e from S and insert the new subsequence s_n into S. Next, we initialize NN_{older}-tuple, $NN_{younger}$-tuple, and NN-tuple of s_n. We then execute Nearest-Neighbor-Search. In a nutshell, this function computes the nearest neighbor of s_n while updating $NN_{younger}$-tuples of the other subsequences.

Algorithm 1: SDM

Input: s_e: the expired subsequence, s_n: the new subsequence
Output: s^*: the discord

1 $s^*_{temp} \leftarrow$ the discord before the window slided
2 $S \leftarrow S \backslash \{s_e\}$, $S \leftarrow S \cup \{s_n\}$ ▷ S is the set of all subsequences on the window
3 $s_n.\langle \cdot, \cdot \rangle_{NN} \leftarrow \varnothing$, $s_n.\langle \cdot, \cdot \rangle_{NN_{older}} \leftarrow \varnothing$, $s_n.\langle \cdot, \cdot \rangle_{NN_{younger}} \leftarrow \varnothing$
4 $s^* \leftarrow$ Nearest-Neighbor-Search

Algorithm 2: Nearest-Neighbor-Search

Input: S: the set of all subsequences, s^*_{temp}: a temporal discord
Output: s^*: the discord

1 **for** $\forall s_p \in S \backslash S_n$ **do**
2 $d \leftarrow dist(s_p, s_n)$
3 **if** $s_n.dist_{NN_{older}} > d$ **then**
4 $s_n.\langle \cdot, \cdot \rangle_{NN_{older}} \leftarrow \langle p, d \rangle$
5 **if** $s_p.dist_{NN_{younger}} > d$ **then**
6 $s_p.\langle \cdot, \cdot \rangle_{NN_{younger}} \leftarrow \langle n, d \rangle$
7 **if** $s_p.\langle \cdot, \cdot \rangle \neq \varnothing \wedge s_p.dist_{NN} > s_p.dist_{NN_{younger}}$ **then**
8 $s_p.\langle \cdot, \cdot \rangle_{NN} \leftarrow s_p.\langle \cdot, \cdot \rangle_{NN_{younger}}$
9 **if** $s_p = s^*_{temp}$ **then**
10 $s^*_{temp} \leftarrow$ Discord-Update
11 **if** $s_p.id_{NN} = e$ **then**
12 $s_p.\langle \cdot, \cdot \rangle_{NN_{older}} \leftarrow \varnothing$, $s_p.\langle \cdot, \cdot \rangle_{NN} \leftarrow \varnothing$
13 **if** $s_p = s^*_{temp}$ **then**
14 $s^*_{temp} \leftarrow$ Discord-Update
15 **if** $s_p.dist_{NN_{younger}} > s^*_{temp}.dist_{NN}$ **then**
16 $s^*_{temp} \leftarrow$ Older-Nearest-Neighbor-Search(s_p)
17 $s_n.\langle \cdot, \cdot \rangle_{NN} \leftarrow s_n.\langle \cdot, \cdot \rangle_{NN_{older}}$
18 $s^* \leftarrow s^*_{temp}$

Nearest-Neighbor-Search. Algorithm 2 details this function. Given a subsequence $s_p \in S \backslash \{s_n\}$, we do the following. First, we compute $dist(s_p, s_n)$. Then we update NN_{older}-tuple of s_n (lines 3–4) and $\mathrm{NN}_{younger}$-tuple of s_p (lines 5–7) if necessary. Note that if $\mathrm{NN}_{younger}$-tuple of s_p is updated, we update NN-tuple of s_p if necessary (lines 8–10). Besides, if NN-tuple of s_p is updated by the above operation and s_p is s^*_{temp}, the distance to the nearest neighbor of s^*_{temp} becomes smaller. This means that the discord may change, so we execute Discord-Update, which is introduced later, to obtain a temporal discord.

Next, we check whether the nearest neighbor of s_p expires. If so, we initialize NN_{older}-tuple and NN-tuple of s_p (line 12), and if s_p is s^*_{temp}, we execute Discord-Update. In addition, if $s_p.dist_{NN_{younger}} \leq s^*_{temp}.dist_{NN}$, we see that s_p is not s^*, thus we prune its nearest neighbor search, as mentioned in Section

Algorithm 3: Discord-Update

Output: s^*_{temp}: a temporal discord
1 $s^*_{temp}.\langle\cdot,\cdot\rangle_{NN} \leftarrow \langle -1, 0\rangle$
2 **for** $\forall s_p \in S$ *such that* $s_p.\langle\cdot,\cdot\rangle_{NN} \neq \varnothing$ **do**
3 **if** $s^*_{temp}.dist_{NN} < s_p.dist_{NN}$ **then**
4 $s^*_{temp} \leftarrow s_p$
5 **for** $\forall s_p \in S$ *such that* $s_p.\langle\cdot,\cdot\rangle_{NN} = \varnothing$ **do**
6 **if** $s_p.dist_{NN_{younger}} > s^*_{temp}.dist_{NN}$ **then**
7 Up-Max-Heap(H, s_p) ▷ s_p is inserted into a heap H
8 **while** $H \neq \varnothing$ **do**
9 $s_p \leftarrow$ Down-Max-Heap(H) ▷ s_p is poped from H
10 **if** $s_p.dist_{NN_{younger}} < s^*_{temp}.dist_{NN}$ **then**
11 **break**
12 **else**
13 $s^*_{temp} \leftarrow$ Older-Nearest-Neighbor-Search(s_p)
14 **return** s^*_{temp}

4.2. Otherwise, we execute Older-Nearest-Neighbor-Search(s_p), which computes NN_{older}-tuple of s_p by sequential scan and updates s^*_{temp} if necessary.

After we do the above operations for all $s_p \in S \backslash \{s_n\}$, we obtain NN-tuple of s_n (line 17) and the correct s^* (line 18).

Discord-Update. We describe this function in Algorithm 3. When we need to update a temporal discord s^*_{temp}, we first initialize its NN-tuple (line 1). Then, by scanning a set of subsequence s_p such that their NN-tuple is not empty, we obtain s^*_{temp} (lines 2-4). In addition, we scan a set of subsequences s_p such that their NN-tuple is empty and satisfies $s_p.dist_{NN_{younger}} < s^*_{temp}.dist_{NN}$, to maintain them in a heap H (lines 5–7). Note that, the subsequences in H are sorted in descending order of $s_p.dist_{NN_{younger}}$.

Next, if H is not empty, we pop the front subsequence s_p of H and check whether s_p satisfies $s_p.dist_{NN_{younger}} < s^*_{temp}.dist_{NN}$ (lines 8–9). If so, we terminate Discord-Update (line 11). Otherwise, we execute Older-Nearest-Neighbor-Search(s_p) (line 13). It is important to note that, thanks to H and $\mathrm{NN}_{younger}$-tuple, we can avoid unnecessary executions of Older-Nearest-Neighbor-Search$(\cdot)$.

We here discuss the space and time complicities of SDM.

Theorem 1 (Space complexity). *The space complexity of SDM is $O(w)$.*

Proof. We maintain NN_{older}-tuple, NN_{older}-tuple, and NN-tuple for each subsequence, which requires $O(1)$ space. Because the number of subsequences on the window is $O(w)$, SDM requires only $O(w)$ space. □

Theorem 2 (Time complexity). *The time complexity of SDM is $O((1+c)wl + c'(w + h\log h))$, where c is the number of executions of* Older-Nearest-Neighbor-

Search$(\cdot)$, *c' is the number of executions of* Discord-Update, *and h is the heap size at line 8 of Algorithm 3.*

Proof. The main cost of SDM is incurred by Nearest-Neighbor-Search. Because Nearest-Neighbor-Search scans S and computes distances between s_p and s_n, which requires at least $O(wl)$. In addition, during the scan, it may execute (some) Older-Nearest-Neighbor-Search$(\cdot)$, which requires $O(wl)$, and Discord-Update, which requires $O(w+h\log h)$. Therefore Nearest-Neighbor-Search requires $O(wl + cwl + c'(w + h\log h))$, which concludes the proof. □

4.4 Approximation of SDM

From Theorem 2, we can see that the cost of SDM is dependent on the execution numbers of Older-Nearest-Neighbor-Search$(\cdot)$ and Discord-Update. In the experiments, we observe that we usually have $c = c' = 0$ for each window slide in practice. However, if the nearest neighbors of many subsequences expire at a time, SDM incurs a large cost to update the discord, since c and c' become large. If applications do not require the correct result but are tolerant of an approximate (but highly accurate) result, we can alleviate the update time by decreasing c and c'.

We propose A-SDM (Approximate-SDM), which monitors an approximate discord with a theoretical guarantee. Surprisingly, A-SDM needs very slight extension to SDM. Given an approximation factor ϵ (> 1), which is specified by an application, we replace line 15 of Algorithm 2 and line 10 of Algorithm 3 with

$$s_p.dist_{NN_{younger}} > \epsilon \cdot s^*_{temp}.dist_{NN},$$

and replace line 6 of Algorithm 3 with

$$s_p.dist_{NN_{younger}} < \epsilon \cdot s^*_{temp}.dist_{NN}.$$

Then we have the following theorem.

Theorem 3 (Accuracy guarantee). *Let s_{out} be the subsequence monitored by A-SDM. We have*

$$s_{out}.dist_{NN} \geq \frac{s^*.dist_{NN}}{\epsilon}. \tag{1}$$

Proof. We first demonstrate that Discord-Update, i.e., Algorithm 3, holds Inequality (1). Recall that, if $s_p.dist_{NN_{younger}} \leq \epsilon \cdot s^*_{temp}.dist_{NN}$, Discord-Update does not insert s_p into H. Even if s_p is s^*, Inequality (1) holds. This is because $s_p.dist_{NN} \leq s_p.dist_{NN_{younger}} \leq \epsilon \cdot s^*_{temp}.dist_{NN}$. The same discussion is applied to line 10.

We turn our attention to line 15 of Algorithm 2. It is important to notice that at line 15, s^* is s_p or s^*_{temp}. If $s^*_{temp} = s^*$, A-SDM monitors the correct answer, so Inequality (1) holds. On the other hand, if $s_p = s^*$ and $s_p.dist_{NN_{younger}} \leq \epsilon \cdot s^*_{temp}.dist_{NN}$, we monitor $s^*_{temp} = s_{out}$. This does not violate Inequality (1) by using the above discussion at Discord-Update. We therefore conclude that Theorem 3 is true. □

5 Experiment

All experiments were conducted on a PC with Intel Xeon Gold 6154 (3.0GHz) and 512GB RAM.

5.1 Setting

Datasets. We used the following three real datasets in this paper.

- Bitcoin[1]: This is a streaming time-series of bitcoin transactions. Its length is 100,000.
- ECG [4]: This is a streaming time-series of electrocardiogram. Its length is 100,000.
- Google-cpu [16]: This is a streaming time-series of CPU usage rate generated by Google data center. Its length is 133,902.

Algorithms. We evaluated the following algorithms, and all of them were implemented in C++.

- HOT SAX [10]: This is a discord detection algorithm for a static time-series. We extended the original algorithm to deal with the discord update based on window sliding. This is a competitor of SDM.
- N-SDM: This is a variant of SDM, and employs only NN-tuple but does not employ NN_{older}-tuple and $\mathrm{NN}_{younger}$-tuple. This algorithm is employed to investigate how efficiently NN_{older}-tuple and $\mathrm{NN}_{younger}$-tuple function.
- SDM: This is the proposed algorithm in this paper.
- A-SDM: The approximation version of SDM.

We do not consider the other existing discord detection/monitoring algorithms, because all of them cannot deal with subsequence deletions and discord expirations. The original HOT SAX also cannot do it, but is a state-of-the-art algorithm that computes the discord from scratch.

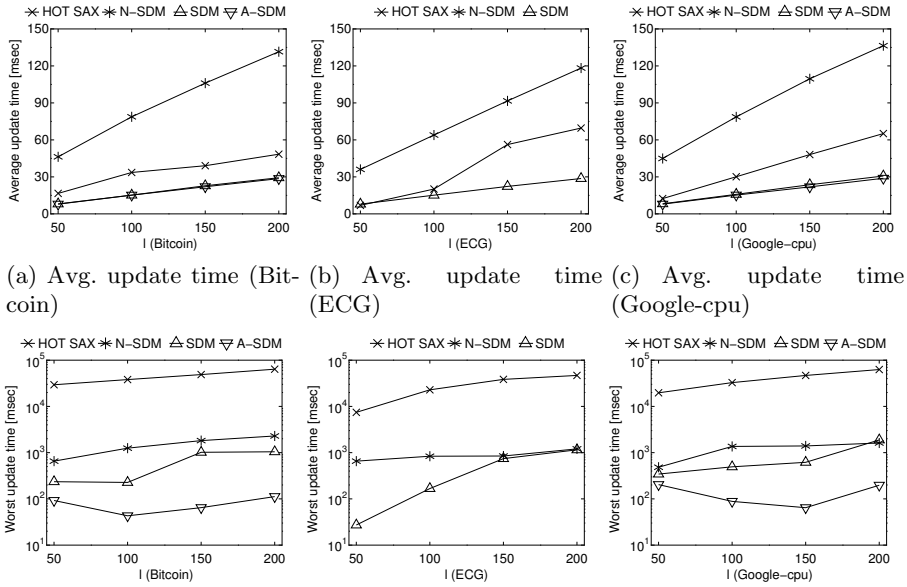
Criteria. We measured the average update time per window sliding, the worst update time, and the practical approximation rate ($= s^*.dist_{NN}/s_{out}.dist_{NN}$).

5.2 Results

We here show our experimental results. Note that the default values of l, w, and ϵ are 100, 10000, and 1.2. When we investigate the impact of a given parameter, the other parameters are fixed.

Impact of l. We investigate how subsequence size affects the performance of each algorithm, and Fig. 4 illustrates the result. We here focus on average update time, which is shown in Figs. 4(a)–4(c). The first observation is that SDM has a linear scalability w.r.t. l. This is reasonable, as Theorem 2 verifies. Second, SDM

[1] `http://api.bitcoincharts.com/v1/csv/`



(a) Avg. update time (Bitcoin) (b) Avg. update time (ECG) (c) Avg. update time (Google-cpu)

(d) Worst update time (Bitcoin) (e) Worst update time (ECG) (f) Worst update time (Google-cpu)

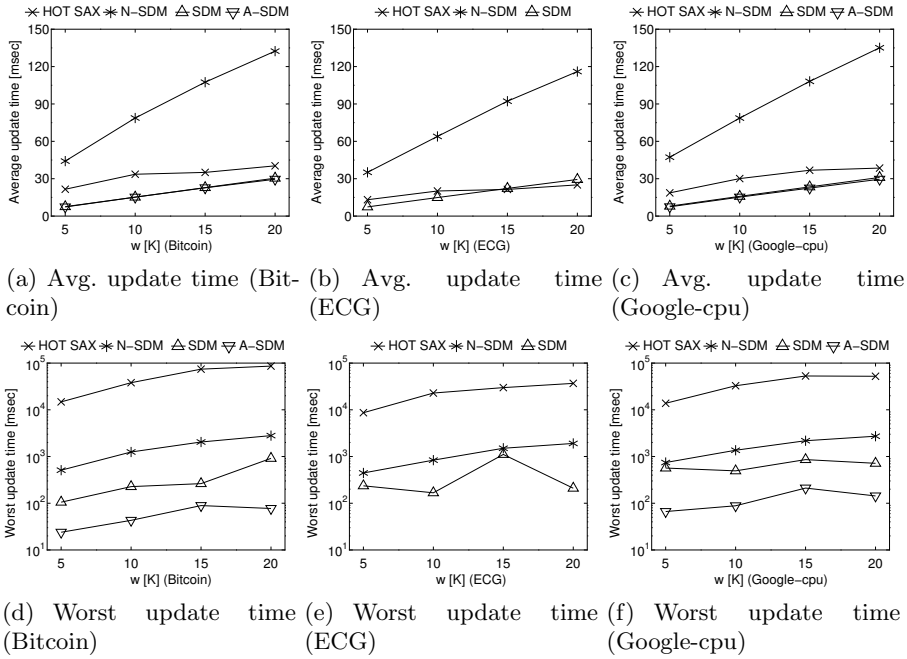
Fig. 4. Impact of l (subsequence size)

(and A-SDM) is (are) faster than the competitors. For each window slide, the average numbers of executions of Older-Nearest-Neighbor-Search$(\cdot)$ and Discord-Update are small, so SDM and A-SDM show similar update time. Also, this result suggests the effectiveness of the sequential scan based approach, as HOT SAX, which is an index-based approach, incurs more update time. Furthermore, compared with N-SDM, SDM is much faster. This is due to NN_{older}-tuple and $\mathrm{NN}_{younger}$-tuple, as discussed in Section 4.2.

Next, we focus on Figs. 4(d)–4(f), which depict the worst update time. It can be seen that SDM has a competitive (or better) performance with (than) N-SDM, which always holds the exact nearest neighbor for each subsequence, and is significantly faster than HOT SAX. (We omit the result of A-SDM on ECG, because it shows a similar performance to SDM.) This result demonstrates that approaches for static time-series are not suitable for streaming time-series.

Impact of w. We next test the scalability of each algorithm w.r.t. the window size. Focus on average update time, and we can see that SDM and A-SDM are basically faster than the other algorithms, as shown in Figs. 5(a)–5(c). Besides, SDM and A-SDM scale linearly w.r.t. the window size, which is also validated by Theorem 2.

Figs. 5(d)–5(f) illustrate that the worst update time of SDM is always faster than those of HOT SAX and N-SDM. This result is derived from the fact that



(a) Avg. update time (Bitcoin) (b) Avg. update time (ECG) (c) Avg. update time (Google-cpu)

(d) Worst update time (Bitcoin) (e) Worst update time (ECG) (f) Worst update time (Google-cpu)

Fig. 5. Impact of w (window size)

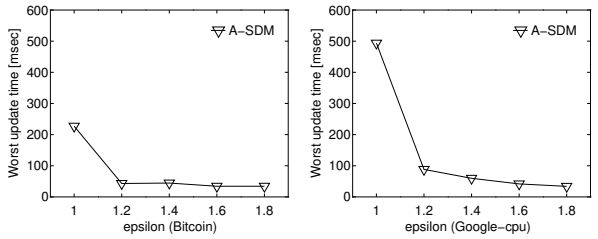
SDM reduces unnecessary distance computation (i.e., nearest neighbor search) by its data structure. A-SDM furthermore reduces the worst update time, and its worst update time is much faster than those of HOT SAX and N-SDM.

Impact of ϵ. Finally, we study the impact of the approximation factor ϵ of A-SDM. Theoretically, a large ϵ provides small update time but inaccurate result. Fig. 6 illustrates the practical relationship. From Figs. 6(a)–6(b), as ϵ becomes large, the worst update time becomes shorter. This is a quite intuitive result. (We confirmed that the average update time is not affected by ϵ, so the result is omitted.) On the other hand, Figs. 6(c)–6(d) show that the practical approximation rate is almost 1 (less than 1.03 actually), even if ϵ becomes large. That is, A-SDM monitors a highly accurate result continuously.

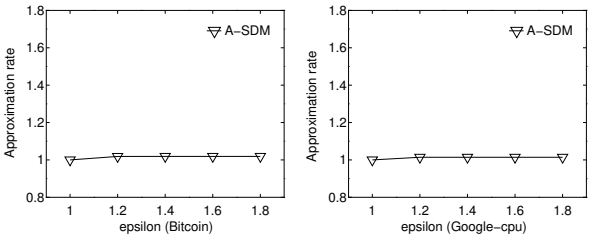
6 Conclusion

Recent applications have been generating streaming time-series, and monitoring outlier from the time-series is an important operator for anomaly detection and data cleaning. Motivated by this observation, in this paper, we addressed a novel problem of monitoring time-series discord over a sliding window.

As an efficient solution for this problem, we proposed SDM (Streaming Discord Monitoring). SDM exploits nearest neighbor search based on sequential



(a) Worst update time (Bitcoin) (b) Worst update time (Google-cpu)



(c) Rate (Bitcoin) (d) Rate (Google-cpu)

Fig. 6. Impact of ϵ (approximation factor)

scan, to obtain the nearest neighbor of a new subsequence and identify the subsequences which need to update their nearest neighbor. We showed that SDM is simple, efficient, and easy to approximate the answer for further accelerating its efficiency. Our experiments using real datasets demonstrate that SDM can monitor the discord efficiently and A-SDM reduces the worst time update while keeping high accuracy.

Acknowledgment. This research is partially supported by JSPS Grant-in-Aid for Scientific Research (A) Grant Number 18H04095, JSPS Grant-in-Aid for Scientific Research (B) Grant Number JP17KT0082, and JSPS Grant-in-Aid for Young Scientists (B) Grant Number JP16K16056.

References

1. Amagata, D., Hara, T.: Mining top-k co-occurrence patterns across multiple streams. TKDE **29**(10), 2249–2262 (2017)
2. Begum, N., Keogh, E.: Rare time series motif discovery from unbounded streams. PVLDB **8**(2), 149–160 (2014)
3. Bu, Y., Leung, T.W., Fu, A.W.C., Keogh, E., Pei, J., Meshkin, S.: Wat: Finding top-k discords in time series database. In: SDM. pp. 449–454 (2007)
4. Chen, Y., Keogh, E., Hu, B., Begum, N., Bagnall, A., Mueen, A., Batista, G.: The ucr time series classification archive (2015), `www.cs.ucr.edu/~eamonn/time_series_data/`

5. Chiu, B., Keogh, E., Lonardi, S.: Probabilistic discovery of time series motifs. In: KDD. pp. 493–498 (2003)
6. Esling, P., Agon, C.: Time-series data mining. ACM Computing Surveys **45**(1), 12 (2012)
7. Gupta, M., Gao, J., Aggarwal, C.C., Han, J.: Outlier detection for temporal data: A survey. TKDE **26**(9), 2250–2267 (2014)
8. Huang, T., Zhu, Y., Mao, Y., Li, X., Liu, M., Wu, Y., Ha, Y., Dobbie, G.: Parallel discord discovery. In: PAKDD. pp. 233–244 (2016)
9. Kato, S., Amagata, D., Nishio, S., Hara, T.: Monitoring range motif on streaming time-series. In: DEXA. pp. 251–266 (2018)
10. Keogh, E., Lin, J., Fu, A.: Hot sax: Efficiently finding the most unusual time series subsequence. In: ICDM. pp. 226–233 (2005)
11. Keogh, E., Lin, J., Lee, S.H., Van Herle, H.: Finding the most unusual time series subsequence: algorithms and applications. Knowledge and Information Systems **11**(1), 1–27 (2007)
12. Lam, H.T., Pham, N.D., Calders, T.: Online discovery of top-k similar motifs in time series data. In: SDM. pp. 1004–1015 (2011)
13. Li, Y., Zou, L., Zhang, H., Zhao, D.: Computing longest increasing subsequences over sequential data streams. PVLDB **10**(3), 181–192 (2016)
14. Li, Y., Yiu, M.L., Gong, Z., et al.: Quick-motif: An efficient and scalable framework for exact motif discovery. In: ICDE. pp. 579–590 (2015)
15. Linardi, M., Zhu, Y., Palpanas, T., Keogh, E.: Matrix profile x: Valmod-scalable discovery of variable-length motifs in data series. In: SIGMOD. pp. 1053–1066 (2018)
16. Reiss, C., Wilkes, J., Hellerstein, J.L.: Google cluster-usage traces: format+ schema. Google Inc., White Paper pp. 1–14 (2011)
17. Sanchez, H., Bustos, B.: Anomaly detection in streaming time series based on bounding boxes. In: International Conference on Similarity Search and Applications. pp. 201–213 (2014)
18. Teflioudi, C., Gemulla, R., Mykytiuk, O.: Lemp: Fast retrieval of large entries in a matrix product. In: SIGMOD. pp. 107–122 (2015)
19. Wang, X., Lin, J., Patel, N., Braun, M.: A self-learning and online algorithm for time series anomaly detection, with application in cpu manufacturing. In: CIKM. pp. 1823–1832 (2016)
20. Wei, L., Keogh, E., Xi, X.: Saxually explicit images: Finding unusual shapes. In: ICDM. pp. 711–720 (2006)
21. Yankov, D., Keogh, E., Rebbapragada, U.: Disk aware discord discovery: Finding unusual time series in terabyte sized datasets. Knowledge and Information Systems **17**(2), 241–262 (2008)
22. Yeh, C.C.M., Zhu, Y., Ulanova, L., Begum, N., Ding, Y., Dau, H.A., Silva, D.F., Mueen, A., Keogh, E.: Matrix profile i: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In: ICDM. pp. 1317–1322 (2016)
23. Zhang, A., Song, S., Wang, J., Yu, P.S.: Time series data cleaning: From anomaly detection to anomaly repairing. PVLDB **10**(10), 1046–1057 (2017)