Time Series Discord Discovery Based on iSAX Symbolic Representation

Huynh Tran Quoc Buu and Duong Tuan Anh
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology
dtanh@cse.hcmut.edu.vn

Abstract

Among several algorithms have been proposed to solve the problem of time series discord discovery, HOT SAX is one of the widely used algorithms. In this work, we employ state-of-the-art iSAX representation in time series discord discovery. We propose a new time series discord discovery algorithm, called HOTiSAX, by employing iSAX rather than SAX representation in discord discovery algorithm. The incorporation requires two new auxiliary functions to handle approximate non-self match search and exact non-self match search in the discord discovery algorithm. Besides, we devise a new heuristic to offer a better ordering for examining subsequences in the outer loop of HOTiSAX algorithm. We evaluate our algorithm with a set of experiments. Experimental results show that the new algorithm HOTiSAX outperforms the previous HOT SAX.

1. Introduction

Finding unusual patterns or *discords* in large time series has recently attracted a lot of attentions in research community and has vast applications in diverse domains such as medicine, finance, biology, engineering and industry ([3],[4],[8],[9]). The concept of time series discord, which has been first introduced by Keogh et al., 2005 ([5]), grasps the sense of the most unusual subsequence within a time series. Time series discords are subsequences of a longer time series that are maximally different to all the rest of the time series subsequences. The Bruce-force Discord Discovery (BFDD) algorithm, suggested by Keogh et al., ([5]) is an exhaustive search algorithm that requires the time complexity $O(n^2)$ to find discords. To reduce the time complexity of BFDD algorithm, Keogh et al. ([5]) also proposed a generic framework, called Heuristic Discord Discovery (HDD) algorithm, with two heuristics suggested to impose the two subsequence orderings in the outer loop and the inner loop, respectively in the BFDD algorithm. To improve the efficiency of HDD algorithm by removing unnecessary computations, the input time series should be first discretized by Symbolic Aggregate Approximation (SAX) technique ([7]) into a symbolic string before the two aforementioned heuristics can be applied in the inner and outer loops of the HDD algorithm. This algorithm is named as HOT SAX by Keogh et al. ([5]).

Shieh and Keogh, 2008 [10] proposed a new symbolic representation, iSAX, which is an extension of SAX. The iSAX representation offers three important advantages. First, it is a multiresolution representation, similar in spirit to wavelet transform. Second, it has an appropriate index structure which allows us to index time series with zero overlap at leaf nodes. Finally, thank to its bit level representation and its suitable index structure, it allows very fast approximate search and exact search on terabyte sized time series datasets. In [10], Shieh and Keogh showed that both kinds of search can be subroutines in several data mining algorithms. Although iSAX is a very promising symbolic representation for time series, so far, surprisingly, there have been so few applications of iSAX in time series data mining tasks. Recently, only one research work is reported in literature: Castro and Azevedo, 2010, ([2]) proposed a method to apply iSAX in time series motif discovery.

In this work, we employing state-of-the-art iSAX representation in time series discord discovery. We propose a new time series discord discovery algorithm, called HOTiSAX, by incorporating iSAX rather than SAX representation into HOT SAX algorithm. The incorporation requires two new auxiliary functions to handle approximate non-self match search and exact non-self match search in the discord discovery algorithm. Besides, we devise a new heuristic to offer a better ordering for visiting subsequences in the outer loop of the proposed algorithm. We evaluate our

algorithm with a set of experiments. Experimental results show that the new algorithm HOTiSAX outperforms the previous HOT SAX.

2. Background

2.1 Time Series Discord

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. Such matches are called *trivial matches* and are not interesting. When finding discords, we should exclude trivial matches; otherwise they spoil our effort to obtain true discords since the true discord may also be similar to its closest trivial match [5].

Definition 2.1. *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 the position q, we say that M is a non-self match to C if $|p - q| \geq n$.

Definition 2.2. *Time series discord*: Given a time series T, the subsequence C of length n beginning at position p is said to be a top-one discord of T if C has the largest distance to its nearest non-self match.

2.2 Symbolic Aggregate Approximation (SAX)

A time series $C = c_1...c_n$ of length n can be represented in a reduced w-dimensional space as another time series $D = d_1...d_w$ by segmenting C into w equally-sized segments and then replacing each segment by its mean value d_i. This dimensionality reduction technique is called Piecewise Aggregate Approximation (PAA). After this step, the time series D is transformed by SAX into a symbolic sequence $A = a_1...a_w$ in which each real value d_i is mapped to a symbol a_i from an alphabet of size a. SAX encodes the reduced time series based on Gauss distribution function.

Given a time series reduced by PAA, we determine a sorted list of breakpoints $B = \beta_1,...,\beta_{a-1}$ by partitioning the area under the N(0,1) Gauss curve into a sections such that each section from β_i to β_{i+1} has the same area $1/a$ ($\beta_0 = -\infty$ and $\beta_a = +\infty$). The breakpoints can be obtained by using a statistical table (see Table 1). Symbols are derived from the sections. The sections below the smallest breakpoint are assigned the first symbol. The section between the first and the second breakpoints the second symbol, and so on. Figure 1 illustrates the idea.

Table 1. Breakpoints for cardinalities from 3 to 6.

	a = 3	a = 4	a = 5	a = 6
β_1	-0.43	-0.67	-0.84	-0.97
β_2	0.43	0	-0.25	-0.43
β_3		0.67	0.25	0
β_4			0.84	0.43
β_5				0.97

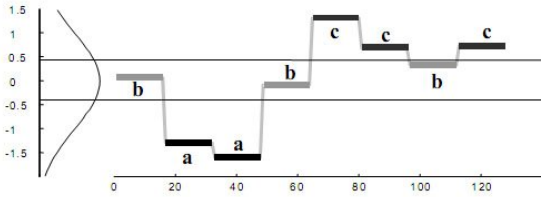


Figure 1. A time series is discretized by first obtaining a PAA approximation and then using predetermined breakpoints to map the PAA coefficients into SAX symbols. In the example above, with n = 128, w = 8 and a =3, the time series mapped to the word **baabccbc**.

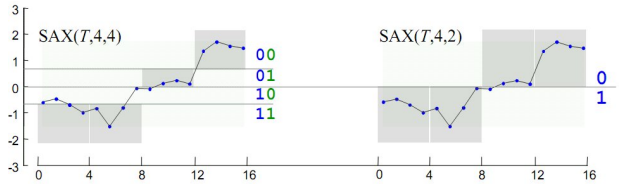
Note that in the example in Figure 1 the three symbols "a", "b", "c" are approximately equiprobable. The concatenation of symbols that represent a subsequence is called a *word*.

2.3 The iSAX Representation

The iSAX representation extends classic SAX by allowing different resolutions for the same SAX word. While in SAX, we represent each SAX symbols as a letter or integer, here however in iSAX we use binary numbers such as *"00", "01", "10",…* (see Figure 2). A time series T can be represented in *i*SAX as follows

$$\text{SAX}(T, w, a) = T^a = \{t_1, t_2, \ldots, t_{w-1}, t_w\}$$

where w is the word length and a is the size of the alphabet. So, there are b levels of resolutions ($b = log_2 a$).



(a) $S(T,4,4) = T^4 = \{11,11,01,00\}$

(b) $S(T,4,2) = T^2 = \{1,1,0,0\}$

Figure 2. A time series T converted into iSAX ([10])

Converting from a higher resolution to lower resolution is simple: one just needs to ignore one trailing bit as we reduce the resolution by half. For example if we convert T to SAX with a cardinality of 8, we have SAX(T, 4, 8) = T^8 = {110, 110, 011, 001}. From this, we can convert to any lower resolution that differs by a power of two. Some following examples make this clear.

SAX(T, 4, 8) = T^8 = {110, 110, 011, 000}
SAX(T, 4, 4) = T^4 = {11, 11, 01, 00}
SAX(T, 4, 2) = T^2 = {1, 1, 0, 0}

Converting from a lower to a higher resolution is a bit quite complex since one can have several possibilities for the higher resolution. Details on how this kind of conversion is performed can be referred to [10].

iSAX Index Structure. Given an example time series T that maps to SAX(T, 4, 4) = T^4= {10,11,10,10} = {2^4, 3^4, 2^4,2^4}. We inserted T into a file that has the SAX word encoded in the file name, such as 2.4_3.4_2.4_2.4.txt. If we have a very large number of time series in the dataset, we will find a huge skew in the distribution of the time series. When we partition the time series in the dataset to the files whose names are given in the above-mentioned manner, more than half of the files will be empty and the largest file contains perhaps 20% of the entire dataset. Therefore, we need to have a user defined threshold *th*, which is the maximum number of time series in a file, and a mapping technique that ensures each file has at least one and at most *th* time series in it. If the number of time series in a file exceeds *th*, we will split the file. For example, file {10^4,11^4,10^4,10^4} = {2^4, 3^4, 2^4,2^4} splits into two child files{100^8,11^4,10^4,10^4} = {4^8, 3^4, 2^4,2^4} and {101^8,11^4,10^4,10^4} = {5^8, 3^4, 2^4,2^4}.

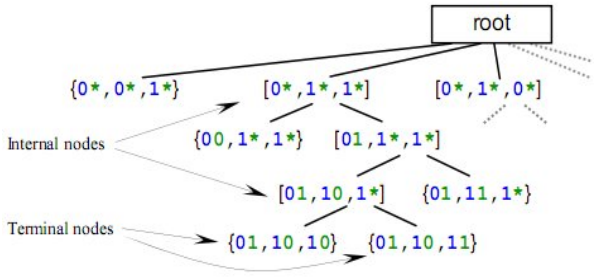


Figure 3. An illustration of iSAX index structure ([10])

The use of iSAX enable the creation of index structures that are hierarchical, containing no overlapping regions unlike R-tree, R*-tree, etc., and a controlled fan-out rate. Figure 3 describes a simple tree-based index structure which reveals the effectiveness and scalability of indexing using iSAX.

The index structure hierarchically subdivides the SAX space, resulting in differentiation between time series entries until the number of entries in each subspace falls below *th*. Such a construct is implemented using a tree, where each node represents a subset of the SAX space such that this space is a superset of the SAX space formed by the union of its descendents. A node's representative SAX space is associated with an iSAX word and evaluation between nodes or time series is done through comparisons of iSAX words. There are three classes of nodes in a tree: terminal node, internal node and root node. A *terminal node* is a leaf node which contains a pointer to an index file on disk with raw time series entries. An *internal node* indicates a split in SAX space and is created when a number of time series contained by a terminal node exceeds *th*. The *root node* is representative of the complex SAX space and works like a nonterminal node. A hash from iSAX words to nodes is maintained to distinguish between entries.

To work on the iSAX index structure, Shieh and Keogh [10] also defined three important operations: iSAX index insert, approximate search, and exact 1-nearest-neighbor search. The *index insert* is used in building the index structure. The *approximate search* is used to find a terminal node in the index with the same iSAX representation as the query. The *exact search* is used to find the exact nearest neighbor to a query. The exact search is based on the approximate search and lower bounding distance function to reduce the search space. Both types of search will be used as important subroutines in time series discord discovery algorithm. Details of these two operations can be referred in [10].

3. Time Series Discord Discovery Based on iSAX

In this section we describe the proposed algorithm HOTiSAX. We start by describing the generic algorithm for discord discovery, the data structures, the new outer loop heuristic and the two auxiliary functions that speed up the working of the inner loop in HOTiSAX.

3.1 Generic Algorithm for Time Series Discord Discovery

Keogh et al. [5] proposed a generic framework, called HDD algorithm, for finding discords in time series. HDD augments two heuristics that impose two subsequence orderings in the outer loop and the inner

loop, respectively in the original bruce-force algorithm (BFDD).

Based on the main ideas of HDD algorithm, in this work we propose a simpler generic algorithm for time series discord discovery in which we do not elaborate the inner loop but exploit an existing subroutine (called *non-self-nearest-neighbor*()) that can find the nearest non-self match for a given subsequence. The pseudocode of the new generic algorithm is given in Figure 4. In this algorithm, we extract all the possible candidate subsequences in the outer loop, then we find the distance to the nearest non-self-match for each candidate subsequence. The candidate subsequence with the largest distance to its nearest non-self match is the discord. The parameters we have to supply to the generic algorithm consists of the time series *T*, the length of discords *n* and the outer loop ordering *Outer*.

```
Function Heuristic_Discord_Search (T, n, Outer)
best_so_far_dist = 0
best_so_far_loc = NaN
for each subsequence p in T ordered by heuristic Outer do
  nearest_neighbor_dist = infinity
  Let q be the non-self-nearest-neighbor(p) and
      nearest_neighbor_dist be the distance between p and q.
  if nearest_neighbor_dist > best_so_far_dist then
    best_so_far_dist = nearest_neighbor_dist
    best_so_far_loc = p
  endif
endfor
return [best_so_far_dist, best_so_far_loc ]
```

Figure 4. The generic algorithm for discord discovery

3.2 From HOT SAX to HOTiSAX

HOT SAX algorithm, proposed by Keogh et al., 2005 [5], is the application of SAX representation into the framework of HDD algorithm. To find discords of length *n* in a time series *T*, HOT SAX begins by creating a SAX representation of the entire time series, by sliding a window of length *n* across the time series *T*, extracting subsequences, converting them to SAX words and placing them in an array where the index refers back to the original subsequence. Once we have this ordered list of SAX words, we can place them into an *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 the HOT SAX algorithm consists of the length of discords *n*, the cardinality of the SAX alphabet size *a*, and the SAX word size *w*.

Two heuristics. HOT SAX uses the two following heuristics:

- *Outer Loop Heuristic*. After constructing the augmented trie, we could reorder the candidate subsequences using following heuristic. The leaf nodes are visited in ascending order according to their word count (how many words in the node). Thus, subsequences in nodes with smallest word count are considered first in the outer loop, and then search in random order for the rest of the subsequences.
- *Inner Loop Heuristic*. When the subsequence *p* is considered in the outer loop, we reorder the inner loop search order by inner loop heuristic as follows. We find a node containing *p* in the trie, all the subsequences in this node are searched first. After exhausting this set of subsequences, the unsearched subsequences are visited in a random order.

In this work, we propose another algorithm for time series discord discovery, called HOTiSAX. HOTiSAX is just the application of iSAX representation into the framework of the generic algorithm we outline in Figure 4. In HOTiSAX, we exploit the existing subroutine that can find the non-self nearest match for a given subsequence and refine the outer loop heuristics.

3.3 The Data Structures for HOTiSAX

We begin by creating two data structures to support our new algorithm, HOTiSAX for time series discord discovery. The first is the iSAX index structure which is a hierarchical tree described in subsection 2.3, however, leaf nodes here do not keep pointers to the files containing the associated subsequences, instead, they keep pointers to the linked lists that contain the positions of the associated subsequences in the original time series. The second data structure is an array where the first column contains the positions of the subsequences in the original time series, the second and third columns contain the iSAX words (at base resolution and finest resolution) and the last column contains a count of how often each word occurs in the array (see Figure 5.). The outer loop in HOTiSAX works in the same way as in HOT SAX, scanning the iSAX words at the finest resolution to select the one with the smallest count in the rightmost column of the array. However, we reorder the scan order of the subsequences in the outer loop by applying the technique described in the next subsection.

3.4 Refining the Outer Loop Heuristic

To enable HOT SAX to work effectively on iSAX representation, we need to refine the outer loop heuristic further. Here we take the order of the subsequences that map to the same current iSAX word

into account. We sort the subsequences of the same iSAX word in such a way that the subsequences with the high chance of being a discord will be examined first in the outer loop.

1	2.3_0.3_0.3	2.6_1.6_0.3	3
2	2.3_0.3_1.3	5.6_0.3_7.12	1
3	2.3_0.3_0.3	2.3_0.3_0.3	3
::	::	::	
::	::	::	
(m-n-1)	2.3_1.3_1.3	4.6_2.6_3.6	2
(m-n)	0.3_2.3_1.3	0.6_8.1_1.3	1
(m-n+1)	1.3_2.3_0.3	3.6_2.3_0.3	2

Figure 5. The array structure used in HOTiSAX algorithm

After selecting the iSAX word (at the base resolution) with the smallest count, we call the sort function *DiscordCandidates_Sort* to reorder the subsequences associated with such the iSAX word. The outline of the sort function *DiscordCandidates_Sort* used for generating outer loop heuristic ordering is described as follows. Given an iSAX word, we search the node on the index structure that associates with the word and apply the sort function on the node. If the node is a terminal node, that means the resolution is at the finest, the function returns all the subsequences in the node without sorting. If the node is an internal node, we fetch all its children. If it has only one child, apply the sort function on this child. If it has two children, we sort the subsequences in each child recursively, then compare the number of subsequences in each child node. The child node with smaller number of subsequences will be arranged with higher priority in the visiting order. Eventually, the function returns a list of sorted subsequences.

3.5 Two Auxiliary Functions

In the generic algorithm described in subsection 3.1, inside the loop we have to exclude trivial matches. However, the approximate search and exact 1-nearest-neighbor search defined by Shieh and Keogh in [10] do not tackle the problem of trivial matches. To overcome this gap, we define the two new approximate search and exact 1-nearest-neighbor search in such a way that we can exclude trivial matches and hence speed up the working of our discord discovery algorithm HOTiSAX.

- Pseudo code of the *approximate non-self match search* function is shown in Figure 6. Given an iSAX word, the function finds the terminal node (in the index) associated with the same iSAX word, then scans all the subsequences pointed to by the node to get one subsequence among them that has the smallest distance to the iSAX word. This result subsequence is viewed as an approximate non-self match to the iSAX word and the distance from this subsequence to the word is returned by the function.

```
1.  Input = {iSAX_word_terminal, p, n, best_so_far_dist}
2.  Output = {approximate_dist, break_soon}
3.   Function [Output] = NonSelfApproximateSearch(Input)
4.      node = Hash.ReturnNode(iSAX_word_terminal)
5.      dist = infinity
6.      break_soon = false
7.      found = false
8.       while (!found &&  node is not root)
9.          foreach q in node.Indices
10.           if  |p − q| ≥ n
11.             found = true
12.             D = Dist(t_p…,t_{p+n-1},  t_q…,t_{q+n-1})
13.             if    D <  best_so_far_dist
14.                   remove p out of outer Loop
15.                   break_soon = true
16.                   break
17.             endif
18.             if  D  <  dist
19.                    dist = D
20.             endif
21.           endif
22.         end                                    //end for loop
23.         if (!found)
24.             node = node.parent
25.         endif
26.    end                                         //end while loop
27.  Return [dist,  break_soon]
```

Figure 6. Approximate non-self match search function

We pass to the function the threshold value *best_so_far_dist*. Assume that the subsequence in question *ts* maps to an iSAX word *iSAX_word_terminal* (we use the third column in the array to obtain the iSAX word associated with *ts*). Then we find the node in the index that has the same iSAX representation with *iSAX_word_terminal.* Next, we scan all the subsequences contained at the node which are non-self matches with *ts* and calculate the distances between each of them to *ts* in order to find the nearest neighbor among these non-self-matches (lines 9-22). If there exists no such a subsequence, we move up to the parent of the current node (lines 23-24). If the parent is not the root, we can scan all the subsequences at the parent node which are non-self matches with *ts*. The algorithm repeats the process until the parent node is a root or we can find a subsequence that is a non-self match to *ts*. In calculating the distances between *ts* to

each of the subsequences at the search node, if there exists one subsequence which distance to *ts* is less than *best_so_far_dist* then we mark an *early termination* by returning the flag *break_soon* to the calling function (lines 13-17).

• Pseudo code of the *exact non-self 1-nearest neighbor search* function used in HOTiSAX is shown in Figure 7.

```
1.  Input = {iSAX_word_terminal, p, n, best_so_far_dist}
2.  Output = {dist, break_soon}
3.  Function [Output] = NonSelfExactSearch(Input)
4.    [nearest_neighbor_dist, break_soon] =
                          NonSelfApproximateSearch (Input)
5.    if break_soon
6.      return nearest_neighbor_dist
                  //the returned value is of no use
7.    endif
8.    PriorityQueue pq
9.    pq.Add(root)
10.   while !pq.IsEmpty
11.       min = pq.ExtractMin()
12.       if min.dist ≥ nearest_neighbor_dist
13.         break
14.       endif
15.       if min is terminal
16.         [tmp, break_soon] =
            IndexFileDist(p, min.Indices, best_so_far_dist)
17.         if break_soon
18.           return nearest_neighbor_dist
                    //the returned value is of no use
19.         endif
20.         if nearest_neighbor_dist > tmp
21.           nearest_neighbor_dist = tmp
22.         endif
23.       elseif min is internal or root
24.         foreach node in min.children
25.           node.dist =
              MINDIST_PAA_iSAX(t_p...,t_p+n-1, node.iSAX)
26.           pq.Add(node)
27.         end
28.       endif
29.   end                          // end while
30. Return [nearest_neighbor_dist, break_soon]
```

Figure 7. Exact non-self match search function

Given an iSAX word, the function tries to find the subsequence that is its nearest non-self match and returns the distance from the iSAX word to this nearest neighbor. The main idea of this function is that by quickly using *NonSelfApproximateSearch* function to obtain the distance from the word to its approximate non-self match. Once a baseline *nearest_neighbor_dist* is obtained, a priority queue is created to examine nodes whose distance is potentially less than the *nearest_neighbor_dist*. This priority queue is first initialized with the root node. The function *NonSelfExactSearch* repeatedly extracts the node with the smallest distance value from the priority queue, terminating when either the priority queue is empty or an early termination is met. Early termination occurs when the lower bound distance we compute equals or exceeds the distance of *best_so_far_dist*. This implies that the remaining entries in the queue cannot quality as the nearest neighbor and can be discarded.

Assume that the outer loop of HOTiSAX is now visiting the subsequence at position *p*. It has to call the exact non-self match function to find the exact nearest-match for the subsequence *p*. It has to pass to the exact non-self-match function four parameters: the iSAX word at the current terminal node *iSAX_word_terminal*, the position *p* of the current subsequence in the outer loop, the length of the discord *n* and the current value of the *best_so_far_dist* variable. The value of *best_so_far_dist* is passed to the approximate non-self match search function (line 4) or the function that computes the distances from the current subsequence *p* to each of the subsequences at a certain node (function *IndexFileDist* at line 16). These two functions can indicate an early termination if any returned smallest distance is less than *best_so_far_dist* (line 5 and line 17).

Notice that when introducing iSAX, a new symbolic representation of time series, Shieh and Keogh ([10]) also defined a distance measure on it by defining a MINDIST function that returns the minimum distance between the original time series of two iSAX words. The MINDIST function is used in the line 25 of the *exact non-self match search* function. Details of the MINDIST function can be referred in [10].

The two auxiliary functions can work very fast due to the effectiveness of the iSAX index structure.

3.6 The HOTiSAX algorithm

After defining the two auxiliary functions, we can describe our HOTiSAX algorithm. Pseudo-code of the HOTiSAX algorithm is shown in Figure 8.

After selecting the iSAX word (at base resolution) with the smallest count, we access to the node associated with that word in the iSAX index structure, and call the *DiscordCandidates_Sort* to arrange the search order on the subsequences of each such iSAX word. After establishing outer loop heuristic ordering, we pass this ordering to the HOTiSAX algorithm.

The algorithm starts by scanning the subsequences according to the outer loop heuristic ordering. For each subsequence at position *p*, we obtain the iSAX word (at the finest resolution) in the array structure (line 5).

Next, we calculate the distance between the subsequence *p* to its nearest neighbor by calling the exact non-self match function *NonSelfExactSearch* (line 7). If an early termination does not incur in this non-self match search, we check if the current *nearest_neighbor_dist* is larger than *best_so_far_dist* or not; if it is, we update the value of *best_so_far_dist* (lines 9-11). Otherwise, if an *early termination* happens in the exact non-self match search (indicated by the flag *break_soon*), the algorithm goes back to examine the next subsequence in the outer loop since the subsequence in question *p* could not be a discord.

```
1.  Function [dist, loc] = HOTiSAX(T, n, Outer)
2.  best_so_far_dist = 0
3.  best_so_far_loc = NaN
4.  for each p in T ordered by heuristic Outer
5.      iSAX_word_terminal = iSAXArray[p][2]
                //Get second column of entry at p
6.      Input = {iSAX_word_terminal, p, n, best_so_far_dist}
7.      [nearest_neighbor_dist, break_soon] =
                        NonSelfExactSearch(Input)
8.      if !break_soon
9.          if nearest_neighbor_dist > best_so_far_dist
10.             best_so_far_dist = nearest_neighbor_dist
11.             best_so_far_loc = p
12.         endif
13.     endif
14. end                         //End Loop
15. Return [best_so_far_dist, best_so_far_loc]
```

Figure 8. The HOTiSAX algorithm

4. Experimental Results

For experiments, the five datasets ERP, EEG, KOSKI, POWER and STOCK are selected. The test datasets are from "The UCR Time Series Data Mining Archive" [6]. For each datasets, we create time series of lengths 5000, 10000, 15000, 20000. We conducted all the experiments on a Core2Duo P8400 2.26GHz 2GB RAM PC. For performance comparison, we adopt two following evaluation metrics: the number of times the distance function is called and CPU time. Notice that the distance function is the function *DIST* that is invoked at line 12 of the *approximate non-self match search* function (see Figure 6) For the first evaluation metric, we apply the guidelines from [5].. Besides, we use the real CPU time as a supplement for the first metric.

We compare the efficiency of HOT SAX and HOTiSAX in terms of the number of distance function calls and CPU times, on the five datasets. HOT SAX and HOTiSAX are implemented with Microsoft C#. After some preliminary experiments for parameter tuning, we found out that the working of our HOT SAX and HOTiSAX are at their best with the parameter setting: the word length $w = 4$, the size of the alphabet $a = 4$, and the length of the discord $n = 128$. Here, due to space limit, we report the experimental results on only three datasets (ERP, KOSKI and POWER) in Figures 9, 10 and 11. The experimental results on five datasets show that HOTiSAX outperforms HOT SAX in terms of running time with the same accuracy.

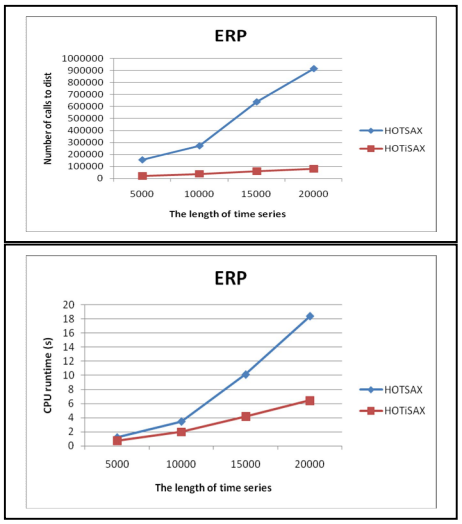


Figure 9. (a) The number of calls to the distance function required by HOT SAX and HOTiSAX on dataset ERP (b) The running time of HOT SAX and HOTiSAX on dataset ERP

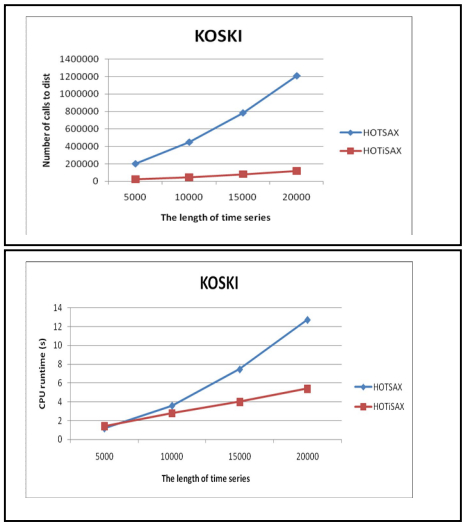


Figure 10. (a) The number of calls to the distance function required by HOT SAX and HOTiSAX on dataset KOSKI (b) The running time of HOT SAX and HOTiSAX on dataset KOSKI

Note that as the lengths of time series increases, the differences between two algorithms get larger. For the time series of length 2000, on POWER dataset, the HOTiSAX is almost four times faster than HOT SAX. In general, these results strongly suggest that we can reasonably expect at least 2 orders of magnitude of a speedup for most problems.

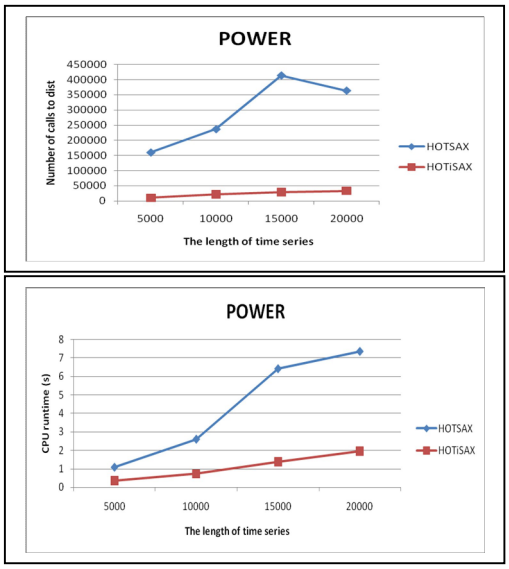


Figure 11. (a) The number of calls to the distance function required by HOT SAX and HOTiSAX on dataset POWER (b) The running time of HOT SAX and HOTiSAX on dataset POWER

We attribute the remarkable performance improvement of HOTiSAX to two factors: (i) the refinement of our new outer loop heuristic which exploits the multi-resolution feature of iSAX representation and (ii) the effectiveness of iSAX index structure which allows very fast approximate non-self match search and exact non-self match search.

5. Conclusions

Although iSAX is a very promising symbolic representation for time series, so far, surprisingly, there have been so few applications of iSAX in time series data mining tasks. In this paper, we propose a new time series discord discovery algorithm, called HOTiSAX, by incorporating iSAX representation into HOT SAX algorithm. The incorporation requires two new auxiliary functions to handle approximate non-self match search and exact non-self match search in the discord discovery algorithm. Besides, we devise a new heuristic to offer a better ordering for examining subsequences in the outer loop of HOTiSAX algorithm. Experimental results show that the new algorithm HOTiSAX outperforms the previous HOT SAX.

For future work, we plan to investigate: (i) extending our experiments to larger datasets and with time series of longer lengths and (ii) applying iSAX representation in the time series discord discovery algorithm WAT, proposed by Bu et al., 2007 [1], which requires fewer input parameters than HOT SAX.

References

[1] Bu, Y., Leung, T.W., Fu, A., Keogh, E., Pei, J. and Meshkin, S: WAT: Finding Top-KC Discords in Time Series Database, In: Proc. of the 2007 SIAM International Conference on Data Mining (SDM'07), Minneapolis, MN, USA, April 26-28, 2007.

[2] Castro, N. and Azevedo, P.: Multiresolution Motif Discovery in Time Series. In: Proc. of the 2010 SIAM International Conference on Data Mining (SDM'10), April 29 – May 1, Columbus, Ohio, 2010, pp. 665-676.

[3] Dasgupta, F. and Forrest, S.: Novelty Detection in Time Series Data Using Ideas from Immunology. In: Proc. of the 5th International Conference on Intelligent Systems, 1996.

[4] Keogh, E., Lonardi, S. and Chiu, B.: Finding Surprising Patterns in a Time Series Database in Linear Time and Space. In: KDD '02: Proc. of 8th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, New York, NY, USA, 2002, pp. 550-556.

[5] Keogh,E., Lin, J. and Fu, A.: HOT SAX: Efficiently Finding the Most Unusual Time Series Subsequence. In: Proc. of 5th IEEE Int. Conf. on Data Mining (ICDM), 2005, pp. 226-233.

[6] Keogh, E. and Folias, T.: The UCR Time Series Data Mining Archive (2002). [http://www.cs.ucr.edu/~eamonn/TSDMA/index.html].

[7] Lin, J., Keogh, E., Lonardi, S., Chiu, B. A 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, 2003

[8] Ma, J. and Perkins, S.: Online Novelty Detection on Temporal Sequences. In: Proc. of the 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, NY, USA, ACM Press, pp. 614-618 (2003)

[9] Salvador, S., Chan, P. and Brodie, J.: Learning States and Rules for Time Series Anomaly Detection. In: Proc. of 17th International FLAIRS Conference, 2004, pp. 300-305.

[10] Shieh, J. and Keogh, E.: iSAX: Indexing and Mining Terabyte Sized Time Series. In: Proc. of 14th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, 2008, pp. 623-631.