# WAT: Finding Top-$K$ Discords in Time Series Database*

Yingyi Bu[1], Tat-Wing Leung[1], Ada Wai-Chee Fu[1], Eamonn Keogh[2], Jian Pei[3] and Sam Meshkin[2]
[1]The Chinese University of Hong Kong, {yybu,twleung,adafu}@cse.cuhk.edu.hk
[2]University of California–Riverside, eamonn@cs.ucr.edu
[3]Simon Fraser University, jpei@cs.sfu.edu

**Abstract**

Finding discords in time series database is an important problem in a great variety of applications, such as space shuttle telemetry, mechanical industry, biomedicine, and financial data analysis. However, most previous methods for this problem suffer from too many parameter settings which are difficult for users. The best known approach to our knowledge that has comparatively fewer parameters still requires users to choose a word size for the compression of subsequences. In this paper, we propose a Haar wavelet and augmented trie based algorithm to mine the top-$K$ discords from a time series database, which can dynamically determine the word size for compression. Due to the characteristics of Haar wavelet transform, our algorithm has greater pruning power than previous approaches. Through experiments with some annotated datasets, the effectiveness and efficiency of our algorithm are both attested.

## 1 Introduction

Mining unusual patterns or discords in large scale time series has vast applications in domains as diverse as medicine, space shuttle telemetry, surveillance, biology and industry. However, finding top-$K$ discords in time series in many occasions is more important than only discovering the most unusual pattern, since the result would be a more informative and complete subsequence set, rather than the most unusual single subsequence, which might be sensitive to parameter settings of algorithms or noises in data.

In the past decade, many real worthy time series problems such as dimension reduction [1], segmentation [2], indexing [3, 4] have been well addressed. However, time series discords, which are first suggested by Keogh et al. [5], have not been paid enough attention to. Although many algorithms have been proposed for anomaly detection [6], most of them are not designed for the specific characteristics of time series data. P. Chan et,al. [7] propose a well designed box model for finding anomaly points in multiple time series, yet we study here a different problem which is to find unusual subsequences in a given time series sequence. The anchor algorithm named **SAX**(**s**ymbol **a**ggregate **a**pproximation) for finding the most unusual time series subsequence is proposed in [5], but it still requires an unintuitive parameter–word size for compression, which we try to remove in our work.

In this paper, we consider the mining of top-$K$ discords in time series database. We propose our algorithm **WAT** (**w**avelet and **a**ugmented **t**rie), which employs Haar wavelet transform and symbol word mapping orderly on raw time series, and then uses a breadth first search on a prefix tree built on the transformed sequences to approximate the perfect search order of all candidate subsequences. Due to the characteristics of Haar wavelet transform, better searching order could be obtained, since the first few elements of transformed subsequence would have better pruning power than the simple symbol aggregate approximation [5]. To remove the unintuitive parameter of word size, we use a dynamic adaptive approach to get suitable word size for different datasets. We also consider an efficient solution of computing the top-$K$ discords for multiple discord lengths. As a result, our method is well adaptive to different applications with no unintuitive parameter setting tasks from users. From the experiments on various large scale datasets, we find that our algorithm is both effective and efficient. Some other works [1, 8] also utilize wavelet transform in time series algorithms, yet their goal is to reduce dimensionality for efficient similarity search. Differently, our use of wavelet transform in **WAT** is to gain approximate perfect searching orders for candidate subsequences rather than reduce data.

The rest of the paper is organized as follows. The problem definition is provided in Section 2. In Section 3, we introduce our algorithm of finding top-$K$ discords. We perform an empirical evaluation in Section 4 to demonstrate both the effectiveness and the efficiency of our algorithm for finding discords. Finally, Section 5 presents some conclusions and expectations for future work.

## 2 Problem Definition

Intuitively a discord is a subsequence that looks very different from its closest matching subsequence. However, in general, the best matches of a given subsequence (apart from

*This research is supported by the RGC Earmarked Research Grant of HKSAR CUHK 4120/05E.

itself) tend to be very close to the subsequence in question. Such matches are called trivial matches and are not interesting. When finding discords, we should exclude trivial matches; otherwise, we may fail to obtain the true discord since the true discord may also be similar to its closest trivial match. Therefore, we need to formally define a non-self match.

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

We now can define time series discord by using the definition of non-self matches:

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

The problem of locating top one discord can obviously be solved by a brute force sequential scan which compares the distance from each subsequence to its nearest non-self match. The subsequence which has the greatest such value is the discord. A base algorithm is given in [5], as shown in Algorithm 1. 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 in inner loop. The candidate subsequence with the largest distance to its nearest non-self match is the discord. We shall refer to this algorithm as the base Algorithm.

**Algorithm 1** Base Algorithm

```
1:  discord distance = 0
2:  discord location = 0
3:  //Begin Outer Loop
4:  for Each p in T ordered by heuristic Outer do
5:     nearest non-self match distance = infinity
6:     //Begin Inner Loop
7:     for Each q in T ordered by heuristic Inner do
8:        if |p − q| ≥ n then
9:           Dist = Dist (t_p, ...t_{p+n−1}, t_q, ...t_{q+n−1})
10:          if Dist < nearest non-self match distance then
11:             nearest non-self match distance = Dist
12:          end if
13:          if Dist < discord distance then
14:             break;
15:          end if
16:       end if
17:    end for
18:    if nearest non-self match distance > discord distance
       then
19:       discord distance = nearest non-self match distance
20:       discord location = p
21:    end if
22: end for
23: Return (discord distance, discord location)
```

In general there can be more than one unusual patterns in a given time series. We are therefore interested to find the top-$K$ discords.

DEFINITION 2.3. *Top-K Time Series Discord: Given a time series $T$, let us impose a total order on the subsequences of length $n$ in descending order of their distances to their nearest non-self match. Ties are also given an arbitrary order. Next scan the list in the given order and remove any subsequence that has some overlapping region with any of the preceding subsequences in the current sorted list. Let the nearest match distance of the $K^{th}$ subsequence in the resulting sorted list $L$ be $d$. A subsequence $D$ in $L$ is among the top-$K$ discords of $T$ if $D$ has a nearest match distance greater than or equal to $d$.*

It is quite intuitive why no overlapping is allowed for the discords, since we assume that discords are signals to alert our users to some unusual or problematic behavior in the pattern, we expect users to examine the discords for further actions. There is not much added value to alert a user twice to the same region of a time series. Algorithm 1 finds the $1^{st}$ discord. For the $2^{nd}$, $3^{rd}$ discord, etc., it is possible to re-apply a similar process, except that each time after finding the top-$k'$ discord, we can remove those discords overlapping with $k'$ from the candidates' list ($p$ in the algorithm). According to the definition, there is no overlapping region between the top-$K$ discords. That means that if a subsequence $D$ of length $n$ beginning at position $p$ is the $1^{st}$ discord, then subsequences beginning at any positions between $p-n+1$ and $p+n-1$ cannot be the $2^{nd}$, $3^{rd}$ discord, etc.

## 3 The Proposed Algorithm: WAT

If a sequential searching order is used in the base algorithm, the worst case time complexity will be $O(km^2)$, where $m$ is the length of time series. However, if other than the first iteration of the double "for" loop, all the subsequence loops can break at Line 13, 14, the time complexity can become $O(km)$. In order to archive such a perfect searching order, we apply Haar wavelet transform on candidate subsequences, build an augmented trie, and then employ outer loop and inner loop heuristics. Besides, we also try to prune across top-$k$ discords and different discord length.

**3.1 Haar Wavelet Transform** Haar wavelet Transform is widely used in different applications such as computer graphics, image, signal processing and time series querying [1, 8]. We propose to apply this technique first to approximate the time series subsequences, as the resulting wavelet can represent the general shape of a time sequence. The time complexity of Haar wavelet transform is linear in the size of the time series. Detailed transform algorithm could be referred to [1].

**3.2 Discretization** After Haar wavelet transform, we shall impose the heuristic Outer and Inner orders based on the transformed subsequences. The wavelet transformed subsequences are then normalized by a standard normalization procedure(to deduct mean, and then be divided by standard deviation). In order to reduce the complexity of time series comparison, we would further transform each of the normalized sequences into a sequence (word) of finite symbols. The alphabet mapping is decided by discretizing the value range for each Haar wavelet coefficient. The distribution of the Haar wavelet coefficients may affect the performance of our algorithm, but it will not affect the correctness. We assume the coefficients are in Gaussian distribution and from our empirical study it can give us pretty good results. Now we can determine the "cutpoints" by using a Gaussian curve. The cutpoints define the discretization of the $i^{th}$ coefficient.

DEFINITION 3.1. ***Cutpoints**: Cutpoints are a sorted list of numbers $B = \beta_1, \beta_2, ..., \beta_{a-1}$, where $a$ is the number of symbols in the alphabet, such that area under a N(0,1) Gaussian curve from $\beta_i$ to $\beta_{i+1} = 1/a$, $\beta_0$ and $\beta_a$ are defined as $-\infty$ and $\infty$, respectively.*

The following table lists the cutpoints for number of symbols from 2 to 7. For example, the area under a N(0,1) Gaussian curve from $-\infty$ to -0.84 is equal to 1/5.

| | Number of symbols $a$ | | | | | |
|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 |
| $\beta_1$ | 0 | -0.43 | -0.67 | -0.84 | -0.97 | -1.07 |
| $\beta_2$ | | 0.43 | 0 | -0.25 | -0.43 | -0.57 |
| $\beta_3$ | | | 0.67 | 0.25 | 0 | -0.18 |
| $\beta_4$ | | | | 0.84 | 0.43 | 0.18 |
| $\beta_5$ | | | | | 0.97 | 0.57 |
| $\beta_6$ | | | | | | 1.07 |

We then can make use of the cutpoints to map all Haar coefficients into different symbols. For example, if the $i^{th}$ coefficient from a Haar wavelet is in between $\beta_0$ and $\beta_1$, it is mapped to the first symbol 'a'. If the $i^{th}$ coefficient is between $\beta_{j-1}$ and $\beta_j$, it will be mapped to the $j^{th}$ symbol, etc. In this way we form a word for each subsequence.

DEFINITION 3.2. ***Word mapping**: A word is a string of alphabet. A subsequence $C$ of length $n$ can be mapped to a word $\hat{C} = \hat{c_1}, \hat{c_2}, ..., \hat{c_n}$. Suppose that $C$ is transformed to a Haar wavelet $\bar{C} = \{\bar{c_1}, \bar{c_2}...., \bar{c_n}\}$. Let $\alpha_j$ denote the $j^{th}$ element of the alphabet, e.g., $\alpha_1 = a$ and $\alpha_2 = b$, .... Let $B = \beta_0, ...\beta_a$ be the Cutpoints for the $i$-th coefficient of the Haar transform. Then the mapping from to a word $\hat{C}$ is obtained as follows:*

$$\hat{c_i} = \alpha_j \iff \beta_{j-1} \leq \bar{c_i} < \beta_j \tag{3.1}$$

**3.3 Augmented Trie** When each subsequence is mapped to a word, all the words are placed in an **array** with a pointer referring back to the original subsequences.

Next, we make use of the array to build an augmented **trie** by an iterative method. At first, there is only a root node which contains a linked list index of all words in the trie. In each iteration all the leaf nodes split. In order to increase the tree height from h to $h + 1$, where $h$ is the tree height before splitting, the $(h+1)^{th}$ symbol of each word under the splitting node is considered. If we consider all the symbols in the word, then the word length is equal to the subsequence length.

Here we make use of the property of Haar wavelets to dynamically adjust the effective word length according to the data characteristics. The word length is determined by the following heuristic.

**Word length heuristic**: *Repeating the above splitting process in a breadth first manner in the construction of the trie until (i) there is only one word in some(one or more) current leaf nodes, or (ii) the $n^{th}$ symbol has been considered.*

The Haar coefficient can help us to view a subsequence in different resolutions, so the first symbol of each word gives us the lowest resolution for each subsequence. In our algorithm, more symbols are to be considered when the trie grow taller, which means that higher resolution is needed for discovering the discord. The reason why we choose to stop at the height where some leaf node contains only one word (or subsequence) is that the single word is much more likely to be the discord, because it cannot be placed into the same node with any other subsequence, so its distance to its nearest match would be far. This is based on the property that Haar transform can preserve the data nature even after a tail sequence is truncated. Words appearing in the same node are similar at the resolution at that level, hence they are closer to their nearest match. Therefore the height at that point implies that we can find an obvious winner to be a candidate discord, and the trie height at that point stands for a good choice for the effective length of the words that can be used in the algorithm.

**3.4 Outer Loop Heuristic** After constructing the tree, we could reorder the candidate subsequences using following heuristic.

**Heuristic:** *the leaf nodes are visited in ascending order according to their word count(how many word in the node). Thus, subsequences in nodes with smallest word count are considered first in the outer loop.*

We search all the subsequences in nodes with the smallest count first, and then search in random order for the rest of the subsequences. The rationale behind our Outer heuristic is the following. When we are building an augmented trie, we are recursively splitting all the leaf nodes until there is only one subsequence in some one leaf node. A trie node with only one subsequence is more likely to be a discord since there are no similar subsequences grouped with it in the same node. This will increase the chance that we get the true discord as the subsequence early considered in the outer loop, which can then prune away other subsequences

quickly. Conversely, the rest of the subsequences are less likely to be the discord. As there should be at least two subsequences map to same tree node, the distance to their nearest non-self match must be very small. More or less, the trie height can reflect the smoothness of the datasets. For smooth dataset, the trie height is usually small, as we can locate the discord at low resolution. On the other hand, the trie height is usually large for a more complex data set.

**3.5 Inner Loop Heuristic** When the $i^{th}$ subsequence $p$ is considered in the outer loop, we reorder the inner loop search order by inner loop heuristic.

**Heuristic**: *We find a node which gives us the longest matching path to the 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.*

The rationale behind our Inner heuristic is the following. In order to break the inner loop, we need to find a subsequence that has a distance to the $p$ less than the best_so_far discord distance, hence the smallest distance to $p$ will be the best to be used. As subsequences in a node with a path close to $p$ are very likely to be similar, by visiting them first, the chance for terminating the loop is increased.

**3.6 Pruning Across The $K$ Discords** In the algorithm in order to break the inner loop, we will try to find out the nearest neighbor of each candidate subsequence. Although we do not actually find the nearest neighbor, the nearest so far neighbor may help us to break the inner loop efficiently.

In order to find the top-$K$ discords, we repeat similar steps to find the first discord, $2^{nd}$ discord, ..., when we find the $i^{th}$ discord, for $1 \leq i \leq K-1$, we record the nearest_so_far neighbor of each candidate subsequence. The nearest_so_far_neigbour may help us to break the inner loop early when we are looking for the next top discord. It finally will converge to the nearest_neigbour. Furthermore, the inner loop search order of each subsequence is always the same, we can remember all the examined subsequences and every time we only compute the distance with the unexamined subsequences. In this way, lots of computation could be efficiently pruned.

**3.7 More on Discord Length** Although the Haar wavelet is defined for sequences that has a length equal to $2^i$, for some positive integer $i$, it is easy to accommodate discord lengths that are not equal to $2^i$, by padding zeros to the sequence to make up a length of $2^i$, for a smallest $i$. Obviously, the length of the subsequences are increased after transformation. It means the computational time of Euclidean distance between the transformed subsequences must be longer than original subsequences. Therefore, the transformed subsequences are only used for heuristic outer loop and heuristic inner loop. For the distance calculation, we will use the original subsequences.

In general we have a problem of how to set the discord length, especially for data without any obvious periodical cycles. In such a case, one may try a range of discord lengths and a trivial solution is to execute the algorithm repeatedly for the different discord lengths.

It turns out that we can have a better solution based on some properties of Euclidean distance and Haar transform. The first strategy is to store the intermediate components for the Euclidean distance computation for a shorter subsequences and reuse for longer lengths which covers the shorter subsequences. Hence there is no need to recompute the Euclidean distance from scratch. However extra storage is required.

For Haar wavelet transform, assume the normalization factor of 1/2[1] Assume we have series, one is $f(5)$ = (7 3 5 1 8), another one is $f(6)$ = (7 3 5 1 8 8). This is an example where we consider length $i$ and then length $i+1$. Consider the transformation to Haar wavelets, the process is shown on Figure 1 and Figure 2. Note that each sequence is padded with zeros to attain a length of $2^3$.

In this example, it is not difficult to discover that most of the intermediate averages and differences of different resolution levels are the same. However, there are some differences between the two conversion processes. We have located the differences in both Figure 1 and Figure 2 with circles. All the differences are related to the new added element 8. In fact, for any $f(i)$ and $H(f(i))$, if we change one of the padding zero elements in $f(i)$, the new $H(f(i))$ is the same as the old $H(f(i))$ except for the coefficients that are related to the changed element.

With this property, if we need to find discords with lengths from $a$ to $b$, where $a < b \leq$ (total length of the time series) and $a$ and $b$ are integers . First we can use a sliding window to extract all the subsequences with length $a$ and convert them into Haar wavelets for our algorithm. After finding the discord of length $a$, we can make use of the previous data to prepare the Haar wavelets for all the subsequences with length $a+1, a+2, ....$ This can help us to save part of the computation time, but we need extra space for storing the intermediate averages and differences for each subsequence.

## 4 Empirical Evaluation

We investigate both the effectiveness and efficiency of **WAT** through extensive experiments. The test datasets, which represent the time series from different domains, are obtained from "The UCR Time Series Data Mining Archive" [9].

[1] In our experiments, we find that normalization factors of 1/2 and $1/\sqrt{2}$ are similar in performance.

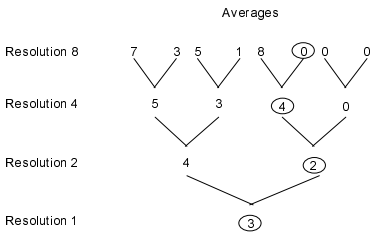


Figure 1: After transformed $f(5)$, it becomes (3 1 1 2 2 2 4 0)

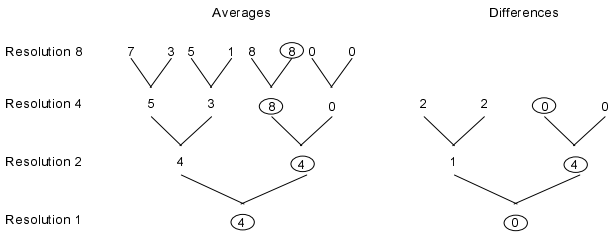


Figure 2: After transformed $f(6)$, it becomes (4 0 1 4 2 2 0 0)

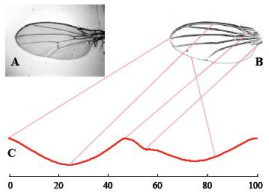


Figure 3: Data Conversion

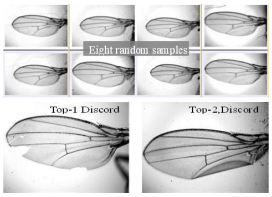


Figure 4: Result Discords

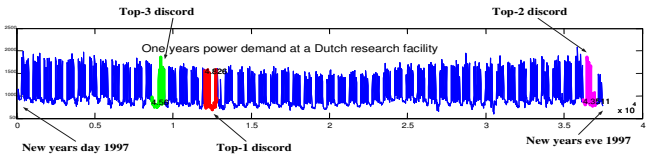


Figure 5: Top-3 discords in power consumption history of a Dutch research facility in year 1997

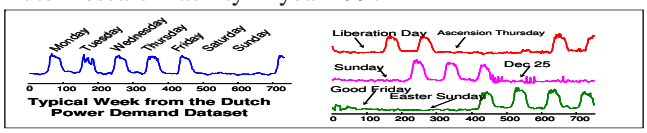


Figure 6: Details of power consumption in normal weeks and discords

**4.1 Effectiveness** We set the alphabet size to 3 and distance measure to Euclidean distance to conduct experiments on two annotated data sets. We compare the resulting discords with the expected results from the annotations based on the domain meaning of the dataset.

**4.1.1 On The Wing Shape Dataset** Our algorithm can be successfully used to find unusual shapes, after the shapes have undergone suitable preprocessing. For brevity we omit the details of necessary image processing steps which is illustrated in [10]. The visual intuition of the conversion process on a fruit flies wing is shown in Figure 3, where A is a raw image of a fruit flies wing, one of 1,000 such images considered for this experiment, and after image processing to increase contrast and remove noise, the contour of the wing B is mapped to a one-dimensional "time series". We are able to ignore the potential tricky problem of rotation invariance by always choosing the leftmost point of the wing as our starting point. The entire wing image was analyzed up to, but not including, the articulation (1 mm from wing attachment to thorax). The top part of Figure 4 shows Eight random samples from our database of 1,000 fruit flies wing, while the bottom part of Figure 4 shows the top two discords discovered by our algorithm. Note the discords have plausible visual intuition. The top-1's wing is damaged, possibly (and ironically) by another insect. The top-2 discord has part of the wing folded back on itself, most likely due to a technician's mishandling of the sample.

**4.1.2 On The Power Consumption Dataset** We try to find top-3 discords from a dataset[2] that records the power consumption for a Dutch research facility for the entire year of 1997. Our objective is to find the 3 most unusual weeks. We do not require the week should be start from Monday to Sunday, so the week could be any 7 days. Note that in this experiment, we set the length of discord to be 750 which is longer than the normal week, as we need to ensure that the discord length must long enough to cover the whole week. The top-3 discords in power consumption history are shown in Figure 5.

It was not easy to find out the differences between the top 3 discords and the rest of the subsequences. However, we could find out some interesting patterns, if we compared the top 3 discords with normal subsequences in detail. The left part of Figure 6 shows the power demands for a normal week, we found that the power consumption from Monday to Friday was relatively high. The right part of Figure 6 shows the top 3 unusual weeks. From the result, we could locate the top 3 unusual weeks that all contained two holidays.

**4.2 Efficiency** For comparing the efficiency of **WAT** with **SAX**, 10 data sequences from 3 datasets(ECG, ERP, and random walk) are picked. For each dataset, time series of lengths 512, 1024, 2048, 4096 and 8192 are randomly extracted, forming 4 derived datasets of varying dimensions. In Figure 7, we compared the pruning power of **WAT** with **SAX** in terms of the number of times the distance function is called. In this experiment, we set the length of the

[2]It is from http://www.cs.ucr.edu/ eamonn/discords/

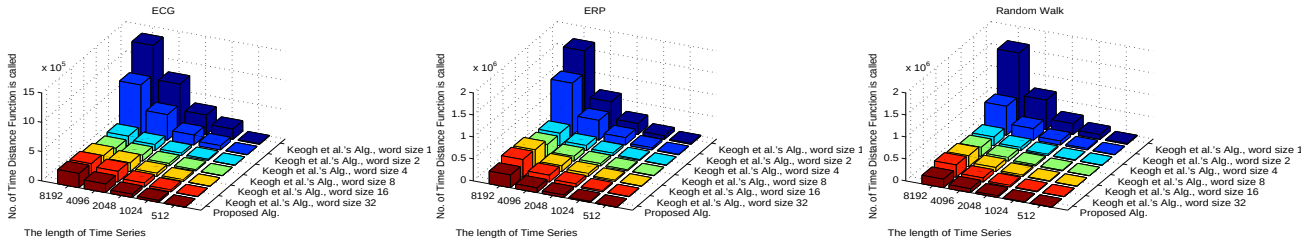


Figure 7: Pruning Power Comparison ( WAT and SAX)

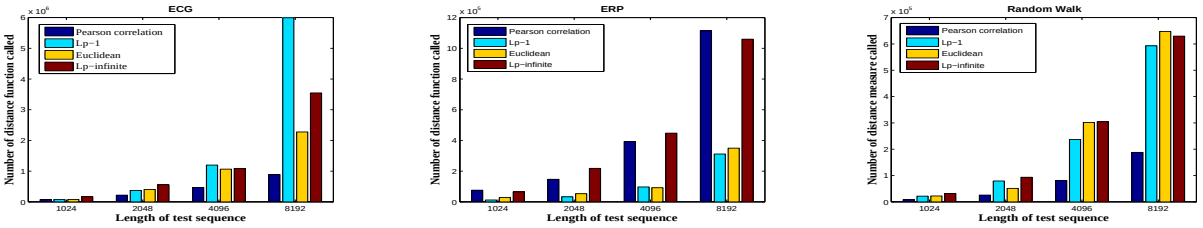


Figure 8: Pruning Power Comparison (WAT under Different Distance Measures)

discord to 128 and find the top-1 discord on all the created subsequences. Each of the experiments is repeated 10 times and the average value is taken. From Figure 7, we could find that **WAT** successfully beats **SAX** on pruning power. We then compare the total running time, and still find that **WAT** is better.

We also study the pruning power of **WAT** under different distance measures, including Pearson correlation, Lp-1, Euclidean and Lp-$\infty$ distances, for which the datasets and test time series are the same as those used in comparison with **SAX**. Like the previous experiment, we also find top-1 discord with discord length 128. We compare the pruning power of distance measures in Figure 8. From the results, we could see none of the distances could beat others on all datasets. We also conduct some running time comparisons on different distance measures, but the best distance measure is also varied on different discord length and different dataset.

## 5 Conclusion and Future Work

To effectively and efficiently find top-$K$ discords has lots of applications in various domains, since top-$K$ result set is more stable than the most unusual one pattern. This paper proposes a novel algorithm **WAT** to efficiently find top-$K$ discords in large scale time series. Our algorithm makes use of some valuable properties of Haar transform and is close to the ideal of parameter free data mining. However, in this work, we focus on finding unusual time series with one dimension only. In the near future, we want to extend our algorithm for handling multidimensional time series. Moreover, we are working towards an extension of our algorithm for discovering discords on data streams, where real time anomaly detection is required.

## References

[1] Kin pong Chan and Ada Wai-Chee Fu. Efficient time series matching by wavelets. In *ICDE*, pages 126–133, 1999.

[2] Aris Anagnostopoulos, Michail Vlachos, Marios Hadjieleftheriou, Eamonn J. Keogh, and Philip S. Yu. Global distance-based segmentation of trajectories. In *KDD*, pages 34–43, 2006.

[3] Eamonn J. Keogh. Exact indexing of dynamic time warping. In *VLDB*, pages 406–417, 2002.

[4] Yunyue Zhu and Dennis Shasha. Warping indexes with envelope transforms for query by humming. In *SIGMOD Conference*, pages 181–192, 2003.

[5] Eamonn J. Keogh, Jessica Lin, and Ada Wai-Chee Fu. HOT SAX: Efficiently finding the most unusual time series subsequence. In *ICDM*, pages 226–233, 2005.

[6] Victoria J. Hodge and Jim Austin. A survey of outlier detection methodologies. *Artif. Intell. Rev.*, 22(2):85–126, 2004.

[7] Philip K. Chan and Matthew V. Mahoney. Modeling multiple time series for anomaly detection. In *ICDM*, pages 90–97, 2005.

[8] Ivan Popivanov and Renée J. Miller. Similarity search over time-series data using wavelets. In *ICDE*, pages 212–, 2002.

[9] Eamonn J. Keogh and T. Folias. The UCR time series data mining archive.

[10] Eamonn J. Keogh, Li Wei, Xiaopeng Xi, Sang-Hee Lee, and Michail Vlachos. LB_Keogh supports exact indexing of shapes under rotation invariance with arbitrary representations and distance measures. In *VLDB*, pages 882–893, 2006.