Finding Time Series Discords Based on Haar Transform

Ada Wai-chee Fu[1], Oscar Tat-Wing Leung[1], Eamonn Keogh[2], and Jessica Lin[3]

[1] Department of Computer Science and Engineering,
The Chinese University of Hong Kong
{adafu, twleung}@cse.cuhk.edu.hk
[2] Department of Computer Science and Engineering,
University of California, River, CA 92521
eamonn@cs.ucr.edu
[3] Department of Information and Software Engineering,
George Mason University
jessica@ise.gmu.edu

Abstract. The problem of finding anomaly has received much attention recently. However, most of the anomaly detection algorithms depend on an explicit definition of anomaly, which may be impossible to elicit from a domain expert. Using discords as anomaly detectors is useful since less parameter setting is required. Keogh et al proposed an efficient method for solving this problem. However, their algorithm requires users to choose the word size for the compression of subsequences. In this paper, we propose an algorithm which can dynamically determine the word size for compression. Our method is based on some properties of the Haar wavelet transformation. Our experiments show that this method is highly effective.

1 Introduction

In many applications, time series has been found to be a natural and useful form of data representation. Some of the important applications include financial data that is changing over time, electrocardiograms (ECG) and other medical records, weather changes, power consumption over time, etc. As large amounts of time series data are accumulated over time, it is of interest to uncover interesting patterns on top of the large data sets. Such data mining target is often the common features that frequently occur. However, to look for the unusual pattern is found to be useful in some cases. For example, an unusual pattern in a ECG can point to some disease, unusual pattern in weather records may help to locate some critical changes in the environments.

Algorithms for finding the most unusual time series subsequences are proposed by Keogh et al in [6]. Such a subsequence is also called a time series *discord*, which is essentially a subsequence that is the least similar to all other subsequences. Time series discords have many uses in data mining, including improving the quality of clustering [8,2], data cleaning and anomaly detection [9,1,3]. By a comprehensive set of experiments, these previous works demonstrated the utility of discords on different domains such as medicine, surveillance and industry.

Many algorithms have been proposed for detecting anomaly in a time series database. However, most of them require many un-intuitive parameters. Time series discords, which were first suggested by Keogh et al, are particular attractive as anomaly detectors because they only require three parameters. The efficiency of the algorithm in [5] is based on an early pruning step and a reordering of the search order to speed up the search. Each time series is first compressed into lower dimensions by a piecewise linear transformation, so that the result is a shorter string (word) of alphabets, where each alphabet corresponds to a range of measured values that has been replaced by the mean value. Hence users are required to choose two parameters, the cardinality of the alphabet size$a$, and the word size w. For the parameter a, previous works reported that a value of either three or four is the best for any task on any dataset that have been tested.

However, for the parameter w, there is no single suitable value for any task on any dataset. It has been observed that relatively smooth and slowly changing datasets favor a smaller value of w; otherwise a larger value for w is more suitable. Unfortunately, we still have questions on how to determine a time series is smooth or not and what is the meaning of a larger value of w.

In this paper we propose a word size free algorithm by first converting subsequences into Haar wavelets, then we use a breadth first search to approximate the perfect search order for outer loop and inner loop. In this way, it is possible to dynamically select a suitable word size.

2 Background

We first review some background material on time series discords, which is the main concern of our proposed algorithm. Then Haar transform will be introduced, which provides an efficiency way to estimate the discord in a given time series and it plays an important role in our algorithm.

2.1 Time Series Discords

In general, the best matches of a given subsequence (apart from itself) tend to be very close to the subsequence in question. Such matches are called trivial matches. When finding discords, we should exclude trivial matches; otherwise, we may fail to obtain true patterns. Therefore, we need to formally define a non-self match.

Definition 1. *Non-self Match: Given a time series T, containing a subsequence C of length n beginning at position p and a matching subsequence M beginning at 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. *Time Series Discord: Given a time series T, the subsequence D of length n beginning at position l is said to be the discord of T if D has the* *largest distance to its nearest non-self match. That is, all subsequence C of T, non-self match M_D of D, and non-self match M_C of C, minimum Euclidean Distance of D to M_D > minimum Euclidean Distance of C to M_C.*

The problem to find discords can obviously be solved by a brute force algorithm which considers all the possible subsequences and finds the distance to its nearest non-self match. The subsequence which has the greatest such value is the discord. However, the time complexity of this algorithm is $O(m^2)$, where m is the length of time series. Obviously, this algorithm is not suitable for large dataset.

Keogh et al introduced a heuristic discord discovery algorithm based on the brute force algorithm and some observations [5]. They found that actually we do not need to find the nearest non-self match for each possible candidate subsequence. According to the definition of time series discord, a candidate cannot be a discord, if we can find any subsequence that is closer to the current candidate than the current smallest nearest non-self match distance. This basic idea successfully prunes away a lot of unnecessary searches and reduces a lot of computational time.

2.2 Haar Transform

The Haar wavelet Transform is widely used in different applications such as computer graphics, image, signal processing and time series querying [7]. We propose to apply this technique to approximate the time series discord, as the resulting wavelet can represent the general shape of a time sequence. Haar transform can be seen as a series of averaging and differencing operations on a discrete time function. We compute the average and difference between every two adjacent values of $f(x)$. The procedure to find the Haar transform of a discrete function $f(x) = (9\ 7\ 3\ 5)$ is shown below.

Example

Resolution	**Averages**	**Coefficients**
4	(9 7 3 5)	
2	(8 4)	(1 -1)
1	(6)	(2)

Resolution 4 is the full resolution of the discrete function $f(x)$. I n resolution 2, (8 4) are obtained by taking average of (9 7) and (3 5) at resolution 4 respectively. (1 -1) are the differences of (9 7) and (3 5) divided by two respectively. This process is continued until a resolution of 1 is reached. The Haar transform $H(f(x)) = (c\ d_0^0\ d_0^1\ d_1^1) =$ (6 2 1 -1) is obtained which is composed of the last average value 6 and the coefficients found on the right most column, 2, 1 and -1. It should be pointed out that c is the *overall average value* of the whole time sequence, which is equal to $(9+7+3+5)/4 = 6$. Different resolutions can be obtained by adding difference values back to or subtract difference from an average. For instance, (8 4) = (6+2 6-2) where 6 and 2 are the first and second coefficient respectively.

Haar transform can be realized by a series of matrix multiplications as illustrated in Equation (1). Envisioning the example input signal $\boldsymbol{x}$ as a column vector with length $n = 4$, an intermediate transform vector $\boldsymbol{w}$ as another column vector and Haar transform matrix $\mathbf{H}$

$$\begin{bmatrix} x'_0 \\ d^1_0 \\ x'_1 \\ d^1_1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{1}$$

The factor 1/2 associated with the Haar transform matrix can be varied according to different *normalization* conditions. After the first multiplication of $\boldsymbol{x}$ and $\mathbf{H}$, half of the Haar transform coefficients can be found which are d^1_0 and d^1_1 in $\boldsymbol{w}$ interleaving with some intermediate coefficients x'_0 and x'_1. Actually, d^1_0 and d^1_1 are the last two coefficients of the Haar transform. x'_0 and x'_1 are then extracted from $\boldsymbol{w}$ and put into a new column vector $\boldsymbol{x'} = [x'_0\ x'_1\ 0\ 0]^T$. $\boldsymbol{x'}$ is treated as the new input vector for transformation. This process is done recursively until one element is left in $\boldsymbol{x'}$. In this particular case, c and d^0_0 can be found in the second iteration.

Hence can convert a time sequence into Haar wavelet by computing the average and difference values between the adjacent values in the time series recursively. It can be also varied according to different normalization conditions. The algorithm shown in Algorithm 1 is using the orthonormal condition. This transformation can preserve the Euclidean distance between two time series, and is therefore useful in our algorithm. If we only consider a prefix of the transformed sequences, the Euclidean distance between two such prefixes will be a lower bounding estimation for the actual Euclidean distance, the longer the prefix the more precise the estimation. Also note that the transformation can be computed quickly, requiring linear time in the size of the time series.

Algorithm 1. Haar Transform

```
1: // Initializtion
2: w = size of input vector
3: output vector = all zero
4: dummy vector = all zero
5:
6: //start the conversion
7: while $w > 1$ do
8:    $w = w/2$
9:    for $i = 0; i < w; i{+}{+}$ do
10:       $dummy\ vector[i] = \frac{input\ vector[2*i]+input\ vector[2*i+1]}{\sqrt{2}}$
11:       $dummy\ vector[i+w] = \frac{input\ vector[2*i]-input\ vector[2*i+1]}{\sqrt{2}}$
12:    end for
13:    for $i = 0; i < (w*2); i{+}{+}$ do
14:       $output\ vector[i] = dummy\ vector[i]$
15:    end for
16: end while
```

3 The Proposed Algorithm

We follow the framework of the algorithm in [6]. In this algorithm, we extract all the possible candidate subsequences in 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 2. Base Algorithm

```
1: //Initialization
2: discord distance = 0
3: discord location = 0
4:
5: // Begin Outer Loop
6: for Each p in T ordered by heuristic Outer do
7:    nearest non-self match distance = infinity
8:    //Begin Inner Loop
9:    for Each $q$ in T ordered by heuristic Inner do
10:       if $|p - q| \geq n$ then
11:          Dist = Euclidean Distance $(t_p, t_{p+1}, ...t_{p+n-1},\ t_q, t_{q+1}, ...t_{q+n-1})$
12:          if Dist < discord distance then
13:             break;
14:          end if
15:          if Dist < nearest non-self match distance then
16:             nearest non-self match distance = Dist
17:          end if
18:       end if
19:    end for
20:    //End For Inner Loop
21:    if nearest non-self match distance > discord distance then
22:       discord distance = nearest non-self match distance
23:       discord location = p
24:    end if
25: end for
26: //End for Outer Loop
27:
28: //Return Solution
29: Return (discord distance, discord location)
```

In the above algorithm, we the heuristic search order for both outer and inner can affect the performance. In fact, if a sequential search order is used, this algorithm will become a brute force algorithm. Note that the discord D is the one that maximizes the minimum distance between D and any other non-self subsequence E

$$\max_D(\min_E(Dist(D, E))$$

The Outer heuristic should order the true discord first since it will get the maximum value for discord distance which has the best chance to prune other candidates at Line 12. Given the subsequence p, the Inner heuristic order should pick the subsequence q closest to p first, since it will give the smallest $Dist$ value, and which will have the best chance to break the loop at Line 12. In this section, we will discuss our suggested heuristic search order, so that the inner loop can often be broken in the first few iterations saving a lot of running time.

3.1 Discretization

We shall impose the heuristic Outer and Inner orders based on the Haar transformation of subsequences.We first transform all of the incoming sequences by the Haar wavelet transform. In order to reduce the complexity of time series comparison, we would further transform each of the transformed sequences into a sequence (word) of finite symbols. The alphabet mapping is decided by discretizing the value range for each Haar wavelet coefficient. We assume that for all i, the i^{th} coefficient of all Haar wavelets in the same database tends to be evenly distributed between its minimum and maximum value, so we can determine the "cutpoints" by partitioning this specify region into several equal segments. The cutpoints define the discretization of the $i-th$ coefficient.

Definition 3. ***Cutpoints:*** *For the i^{th} coefficient, cutpoints are a sorted list of numbers $B_i = \beta_{i,1}, \beta_{i,2}, ..., \beta_{i,m}$, where m is the number of symbols in the alphabet, and*

$$\beta_{i,j} - \beta_{i,j+1} = \frac{\beta_{i,a} - \beta_{i,0}}{a} \tag{2}$$

$\beta_{i,0}$ and $\beta_{i,a}$ are defined as the smallest and the largest possible value of the i^{th} coefficient, respectively.

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 $\beta_{i,0}$ and $\beta_{i,1}$, it is mapped to the first symbol 'a'. If the i^{th} coefficient is between $\beta_{i,j-1}$ and $\beta_{i,j}$, it will be mapped to the j^{th} symbol, etc. In this way we form a word for each subsequence.

Definition 4. ***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 α_j denote the j^{th} element of the alphabet, e.g., $\alpha_1 = a$ and $\alpha_2 = b$, ... Let $B_i = \beta_{i,1}, ...\beta_{i,m}$ 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:*

$$\hat{c_i} = \alpha_j \iff \beta_{i,j-1} \leq \bar{c_i} < \beta_{i,j} \tag{3}$$

3.2 Outer Loop Heuristic

First, we transform all the subsequences, which are extracted by sliding a window with length n across time series T, by means of the Haar transform. The transformed subsequences are transformed into words by using our proposed discretizing algorithm. Finally, all the words are placed in an **array** with a pointer referring back to the original sequences. Figure 1 illustrates this idea.

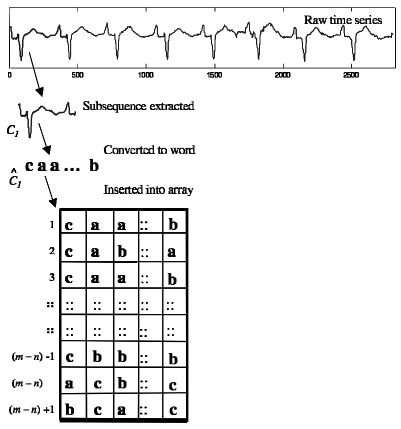

Fig. 1. An array of words for building an augmented trie

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 are 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. In previous work [6] a shorter word length is used by using a piecewise linear mechanism to compress the subsequence, which means that user need to determine the word length beforehand. 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 any current leaf node 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 compared to any other word which appears with other words in the same node, since such words in the same node are similar at the resolution at that level. Hence the height at that point implies a good choice for the length of the word that can be used in the algorithm.

We found that the performance of this breadth first search on the trie approach is pretty good, since it can efficiently group all the similar subsequences under the same tree node and the distance between any two subsequences under the same node are very small.

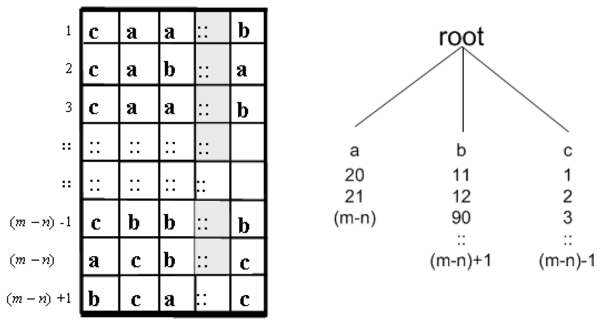

Fig. 2. 1^{st} symbol is considered for splitting the root node. All leaf nodes will be split, since no leaf node contains only 1 word.

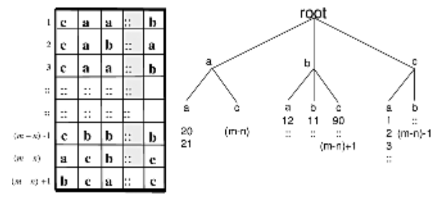

Fig. 3. 2^{nd} symbol is considered. No tree node is split in next iteration, since there is only 1 word mapped to 'ac'.

Heuristic: *the leaf nodes are visited in ascending order according to the word count.*

We search all the subsequences in the nodes with the smallest count first, and we search in random order for the rest of the subsequences. The intuition 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 a leaf node. A trie node with only one subsequence is more likely to be a discord since there are no similar nodes that are grouped with it in the same node. This will increase the chance that we get the true discord as the subsequence used in the outer loop, which can then prune away other subsequences quickly.

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 tire height is usually large for a more complex data set.

From this observation, it is obvious that the first subsequence that map to a unique word is very likely to be an unusual pattern. On the contrary, 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.

3.3 Inner Loop Heuristic

When the i^{th} subsequence P is considered in the outer loop, we look up words in the i^{th} level of the trie.

Heuristic: *We find a node which gives us the longest matching path to 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 intuition 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 i^{th} word in the outer loop 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 search is increased.

4 Empirical Evaluation

We first show the utility of time series discords, and then we show that our algorithm is very efficient for finding discords. The test datasets, which represent the time series from different domains, were obtained from "The UCR Time Series Data Mining Archive"[4].

4.1 Anomaly Detection

Anomaly Detection in a time series database has received much attention [9,1,3]. However, most anomaly detection algorithms require many parameters. In contrast our algorithm only requires two simple parameters, one is the length of the discord, another is the alphabet size, and for the alphabet size, it is known that either 3 or 4 is best for different tasks on many datasets from previous studies.

To show the utility of discords for anomaly detection, we investigated electrocardiograms (ECGs) which are time series of the electrical potential between two points on the surface of the body caused by a beating heart. In the experiment, we set the length of the discord to 256, which is approximately one full heartbeat and set the alphabet size to be three.

Our algorithm has successfully uncovered the discord in figure 4. In this example, it may seem that we can discover the anomaly by eye. However, we typically have a massive amount of ECGs, and it is impossible to examine all of them manually.

4.2 The Efficiency of Our Algorithm

Next we study the efficiency of the algorithm. From 4 datasets, 10 data sequences were picked. For each data sequence, prefixes of lengths 512, 1024, 2048, 4096 and 8192 were truncated, forming 4 derived datasets of varying dimensions. In figure 5, we compared the base Algorithm with our proposed algorithm in terms of the number of times the Euclidean distance function is called. In this experiment, we set the length of the discord to 128 and found the discord on all

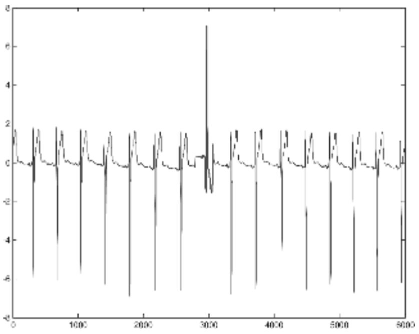

Fig. 4. A time series discord (marked in bold line) was found at position 2830

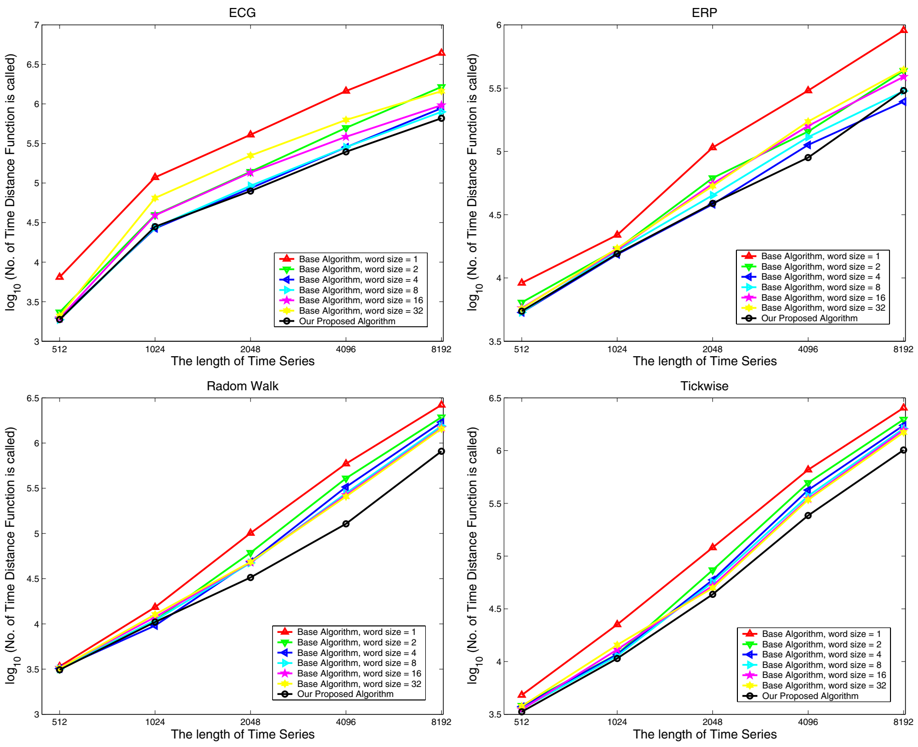

Fig. 5. Number of times distance function is called by the base Algorithm and Our Proposed Algorithm

the created subsequences. Each of the experiments was repeated 10 times and the average value was taken.

In this experiment, we did not measure the CPU time directly in order to ensure that there was no implementation bias. In fact, it has been discovered that the distance function accounts for more than 99% of the running tine. By measuring the number of distance computation, we have a relatively fair measure for the running time.

From the experimental results, we found that there was no special value for word size which was suitable for any task on any dataset. The results suggested that relatively smooth and slowly changing datasets favor a smaller value of word size, whereas more complex time series favor a larger value of word size. However, it is difficult to determine whether a dataset is smooth or otherwise. In most cases our proposed algorithm gave a better performance without the consideration of the word size comparing with the base Algorithm.

5 Conclusion and Future Work

We introduce a novel algorithm to efficiently find discords. Our algorithm only requires one intuitive parameter (the length of the subsequence). In this work, we focused on finding the most unusual time series subsequence. We plan to extent our algorithm to K time series discord which refers to finding K discords with the largest distance to its nearest non-self match.

Acknowledgements. This research was supported by the RGC Research Direct Grant 03/04, and the RGC Earmarked Research Grant of HKSAR CUHK 4120/05E.

References

1. D. Dasgupta and S. Forrest. Novelty detection in time series data using ideas from immunology, 1996.
2. E. Keogh. Exact indexing of dynamic time warping, 2002.
3. E. Keogh, S. Lonardi, and B. Chiu. Finding surprising patterns in a time series database in linear time and space. In *Proceedings of The Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '02)*, Edmonton, Alberta, Canada, July 2002.
4. E. Keogh and T.Folias. The ucr time series data mining archive.
5. E. J. Keogh, S. Lonardi, and C. A. Ratanamahatana. Towards parameter-free data mining. In *KDD*, pages 206–215, 2004.
6. J. Lin, E. J. Keogh, A. W.-C. Fu, and H. V. Herle. Approximations to magic: Finding unusual medical time series. In *CBMS*, pages 329–334, 2005.
7. K. pong Chan and A. W.-C. Fu. Efficient time series matching by wavelets. In *ICDE*, pages 126–133, 1999.
8. C. A. Ratanamahatana and E. J. Keogh. Making time-series classification more accurate using learned constraints. In *SDM*, 2004.
9. C. Shahabi, X. Tian, and W. Zhao. TSA-tree: A wavelet-based approach to improve the efficiency of multi-level surprise and trend queries on time-series data. In *Statistical and Scientific Database Management*, pages 55–68, 2000.