A Study of Time Series: Anomaly Detection and Trend Prediction

LEUNG Tat Wing

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

August 2006

Thesis / Assessment Committee

Professor WONG Man Hon (Chair)

Professor FU Wai Chee (Thesis Supervisor)

Professor LEUNG Ho Fung (Committee Member)

Professor X. Sean Wang (External Examiner)

Abstract of thesis entitled:

A Study of Time Series: Anomaly Detection and Trend Prediction

Submitted by LEUNG Tat Wing
for the degree of Master of Philosophy
at The Chinese University of Hong Kong in July 2006

This paper discusses two important problems in time series.

Anomaly Detection: The problem of finding anomalous subsequences has received much attention. However, most of the anomaly detection algorithms need an explicit definition of unusual pattern, which may be impossible to elicit from a domain expert. Using discords as anomaly definition is useful, since discords only require one intuitive parameter (the length of the subsequence). In this thesis, we introduce a new algorithm for finding discords. By converting the sequences into Haar wavelets, using the properties of Haar wavelets, we are able to locate the discord by searching the wavelets from low resolution to high resolution.

Trend Prediction: Time series trend prediction is not a new topic. People in different areas use different approaches to solve this problem. In spite of the many previous works, it is still a very difficult problem, especially for financial data such as stock price, bond price and index. It may be because of their high volatility. We believe that analyzing the basic patterns of a time series can provide us with a lot of information, since data points in time series must reflect all the underlying generating principle and correlation that exist between data points. In this thesis,

we propose an algorithm that applies the previous knowledge about the movement of the stock price on moving average. And we will show that this algorithm can develop effective investment strategies.

摘要

這篇論文將會討論兩個在時間序列上很重要的問題。

異常偵測: 尋找異常子序列這個問題已經受到關注。但是，大部份的異常偵測演算法都需要明確地定義甚麼是不正常的模式，這對於領域專家來說是不太可能。利用「不一致」來做一個異常探測器是十分有用的，因爲它們只需要一個靠直覺得知的參數 (子序列的長度) 。在本論文，我們會介紹一個新的演算法來尋找不一致的子序列。我們建議轉化序列爲哈爾小波。以後再利用哈爾小波的特性,這樣我們便能從低解像度到高像度去尋找不一致。

預計趨勢: 時間序列的趨勢預計已經不是一個新課題。人們在不同的領域用不同的方法去決解這個問題。雖然有很多以前的著作，但是這個問題仍然是十分困難。特別是金融的數據，例如股票的價格、債券的價格和一些指數。這可能是因爲它們的高揮發度。我們相信分析時間序列的基本模式可以提供我們很多有用的資料，因爲在時間序列的數據點一定會反映出所有潛在的生產原則及數據點之間的相互關係。在本論文，我們提議的演算法將過往對於股票價格的資料應用在移動平均線上。我們會展示這個演算法能夠發展出一個有效的投資策略。

Acknowledgement

I would like to express my sincere thanks to my supervisor Professor Ada Wai-Chee Fu for her kindly guidance, support and help throughout this dissertation. I would like to express my special thanks and appreciation to Professor Man-Hon Wong for his valuable comments.

I would also like to give my thanks to my fellow colleagues, Albert Au-Yeung, Royce Ching, Tilen Ma, Shirley Ng, Yuk-Man Wong, Hei-Tat Lam, Pik-Wah Chan, Chi-Wing Wong and Cheuk-Han Ngai. They have given me a joyful and wonderful time in my research. These two years should not have been that fruitful without them.

This work is dedicated to my family for the support and patience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, data mining is very important for different people from different positions. Because of the advanced technology, we can obtain a large amount of data easily. Unfortunately, it is not easy to extract some useful information from the raw data. And it is also well known there is no generic method to get meaningful information from different type of data. Therefore, many computer scientists in data mining community suggested many approaches to mine different knowledge for different datasets.

We are especially interested in time series. As in many applications, time series have been found to be a natural and useful form of data representation. The definition is shown in definition 1 and two time series examples are shown in figure 1.1 and 1.2.

Definition 1 ***Time Series****: A time series $T = t_1, ..., t_n$ is an ordered set of n real values.*

Many important applications around us involve time series such as financial data, electrocardiograms(ECG) and other medical records, weather changes, power consumption etc. As large amounts of time series data are accumulated over time, it is interesting to uncover interesting patterns on top of the large datasets. Some researchers focus on motifs discovery [6, 21], as finding approximately repeated subsequences is a core task for

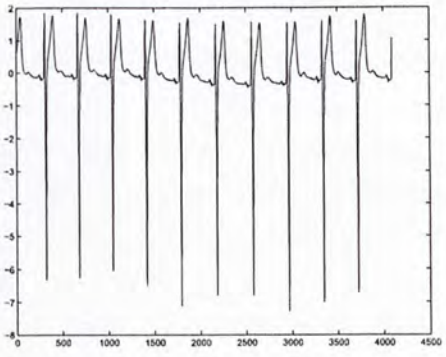


Figure 1.1: An electrocardiograms(ECG)of a patient

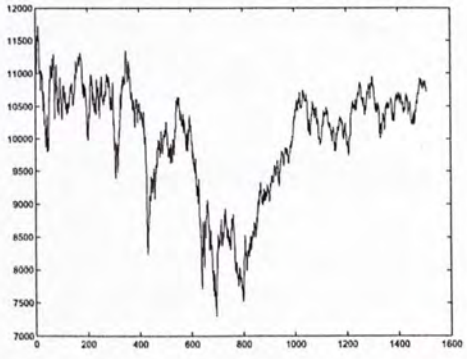


Figure 1.2: DOW Index from January 2000 to December 2005

various data mining problems such as mining association rules in time series. Some researchers focus on time series classification problem [28, 34, 31]. Although classification, association rules and frequent patterns are old problems in data mining community, they become very challenging problems in time series.

In fact, there are many problems in time series such as clustering, query by context, time series segmentation etc. In this thesis, we focus on two problems. One is unusual pattern discovery, and the other one is time series trend prediction.

We focus on these two problems, as they are very important. When we are making decisions, most of time we need to ask two

questions. One is 'Is there any unusual event in current process'. The other one is 'What is the future movement'. Hence, there is a strong motivation to solve these two problems.

And we also find that these two problems can be solved by using the same techniques. The first technique is discretization, in fact many researchers will map time series into a sequence of words. After this mapping, we can handle the original time series by using existing string operation algorithms. For example, subsequence matching problem can convert into substring matching problem. The second technique is tree structure, it can provide a fast way to locate any subsequence. The tree structure can also help us to group the similar subsequences together and locate the unusual subsequences.

1.1 Unusual Pattern Discovery

Most of the data mining algorithms target common features that frequently occur. However, looking for the unusual pattern is found to be useful in many cases. For example, an unusual pattern in an ECG can point to some diseases, unusual pattern in weather records may help to locate some critical changes in environments.

Unfortunately, this problem is not easy. The major difficulty is what is the definition of anomaly. Some existing algorithms need an explicit definition of anomalies. It is not an easy task even for domain experts. Some algorithms need users to provide a collection of previously observed data, which is considered normal. Then any newly observed pattern will be compared with the normal dataset, so no specific model for normal behavior is needed. However, we may not have enough data to define normal behavior. And some anomalies may be contained in normal dataset, we may need another algorithm to remove anomalies from the normal dataset. This opens the possibility of a chicken

and egg paradox.

Lin et al. [24] and Keogh et al. [19] suggested a new definition of anomaly in [24, 19]. According to these authors, we can define the most unusual time series subsequence by a single parameter that is the length of the interesting subsequence. We are very interested in this definition, as it is a new problem first suggested in 2005. We find that there is still room for improvement to existing algorithms.

In Chapter 2 of this thesis, we first have a brief review on related work and background. Then we will discuss our proposed algorithm in details. We will investigate this problem in different aspects that were not considered in previous works. At the end of Chapter 2, we will discuss the experimental results.

1.2 Trend Prediction

Comparing with unusual pattern discovery, time series trend prediction is a very old topic. Although people from different areas have put many efforts and much time on this problem, there is no approach for solving this problem well. However, it is very important in many decision making processes. For example, an investor in a financial market may want to know whether the DOW index will move in an up trend or a down trend for the coming few months. It may affect his decision in buying or selling a stock. As another example, government may want to know the birth rate and the population size for the coming years, as they may affect the government policies.

In fact, this problem is very challenging. As we have mentioned before, no one could solve this problem well especially for financial data such as stock price, bond price and index. This may be because of their high volatility. Some people suggested the movement of stock price is a random process, that is the chance for moving upward or moving downward is always 50%.

However, some people suggested that past information can provide knowledge for trend prediction. Because of the difficulty and the importance of this problem, there is a strong motivation for us to solve this problem.

In Chapter of 3 this thesis, we first review some commonly used prediction techniques. Then, we discuss different types of moving averages that are the widely used prediction tools in financial community. We also demonstrate how a moving curve can help us to predict the trend. Next, we will mention our proposed algorithm for improving the performance of a well known tool, moving average in details. At the end of Chapter 3, we will discuss the experimental results.

1.3 Thesis Organization

In this thesis, we first describe the problem unusual pattern discovery and the proposed algorithm in Chapter 2. Then, we describe time series trend prediction and the proposed solution in Chapter 3. Finally, we conclude our thesis in Chapter 4.

□ **End of chapter.**

Chapter 2

Unusual Pattern Discovery

2.1 Introduction

Algorithm for finding the most unusual time series subsequence [19] was firstly proposed by Keogh et al.. Such a subsequence is also called as time series discord, which is the least similar to all other subsequences. Time series discords have many uses in data mining, including improving the quality of clustering [28, 13], data cleaning and anomaly detection [30, 7, 14, 35, 38]. With a comprehensive set of experiments, Keogh et al. demonstrated the utilities of discords in different domains such as medicine, surveillance and industry.

Keogh et al. also proposed an algorithm [19] based on early pruning and reordering the search order to speed up the search.

Algorithm proposed by Keogh et al. needs users choose two parameters, the cardinality of the SAX [23] alphabet size a, and the SAX word size w. For the parameter a, extensive experiments were carried out by many researchers. Results suggest that a value of either three or four is the best for any task on any dataset.

However, for parameter w, there is no suitable value for any task on any dataset. Keogh et al. suggested that relatively smooth and slowly changing datasets favor a smaller value of w; otherwise a larger value 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 larger value of w [20].

We propose a word size free algorithm by first converting subsequences into Haar wavelets [27], then using a breadth first search to approximate the perfect search order for outer loop and inner loop. We will discuss the perfect search later in section 2.3.

2.2 Related Work

2.2.1 Time Series Discords

Many algorithms have been proposed for detecting anomaly in a time series database. However, most of them require many unintuitive parameters. Time series discords [24, 19], which were first suggested by Keogh et al., are particular attractive as anomaly detectors because they only require three parameters.

Now, we will define the discord step by step. First, in order to distinguish an unusual pattern from a given time series, we must need a distance function to measure the distance between all pairs of subsequences. Therefore, we need to formally define a distance measure Distance Measure(C, M).

Definition 2 ***Distance Measure****: It is a function that has C and M as inputs and returns a nonnegative value R, which is said to be the distance from M to C. For subsequent definitions to work we require that the Distance Measure be symmetric, that is, Distance Measure(C,M)=Distance Measure(M,C).*

Euclidean distance measure is the most common one in the literature that can fulfill the above requirement. For the rest of this paper, we will use Euclidean distance as the distance function.

Definition 3 ***Euclidean Distance****: Given two series C and M of length n, the Euclidean Distance between them is defined*

as:

$$Distance\ Measure(C, M) = \sqrt{\sum_{i=1}^{k}(c_i - m_i)^2} \qquad (2.1)$$

Before calculating the distance between C and M, we must ensure they are both normalized to have zero mean and standard deviation of one. As it is well known [18] that it is meaningless to compare time series with different offsets and amplitudes. In this thesis, we assume that all the subsequences are normalized.

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 [24, 19, 6].

Definition 4 ***Non-self Match*** *(By Keogh et al.): 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 [24, 19] by using the definition of non-self matches:

Definition 5 ***Time Series Discord*** *(By Keogh et al.): 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, for all subsequence C of T, non-self match* M_D *of D, and non-self match* M_C *of C, minimum Distance Measure of D to* $M_D >$ *minimum Distance Measure of C to* M_C*.*

2.2.2 Brute Force Algorithm

The brute force algorithm is the first simple and obvious algorithm for finding discords. It simply considers all the possi-

ble subsequences and finds the distance to its nearest non-self match. The subsequence which has the greatest such value is the discord. Algorithm 1 illustrates the idea of the brute force approach. However, time complexity of this algorithm is $O(m^2)$, where m is the length of time series. Obviously, this algorithm is not suitable for long time series.

Algorithm 1 Brute Force Algorithm

```
1: //Initialization
2: discord distance = 0
3: discord location = NaN
4:
5: //Begin Outer Loop
6: for p = 1 to |T| − n + 1 do
7:    nearest non-self match distance = infinity
8:    //Begin Inner Loop
9:    for q = 1 to |T| − n + 1 do
10:      if |p − q| ≥ n then
11:         Dist = Distance Measure(t_p, ...t_{p+n−1}, t_q, ...t_{q+n−1})
12:         if Dist < nearest non-self match distance then
13:            nearest non-self match distance = Dist
14:         end if
15:      end if
16:   end for
17:   //End For Inner Loop
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: //End for Outer Loop
24:
25: //Return Solution
26: Return (discord distance, discord location)
```

2.2.3 Keogh et al.'s Algorithm

Keogh et al. introduced a heuristic discord discovery algorithm [24] based on the brute force algorithm and some observations. 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 such that the distance between this subsequence to the current candidate is smaller than the current smallest nearest non-self match distance. This basis idea successfully prunes away a lot of unnecessary searches and reduces a lot of computational time.

So Koegh et al. suggested two heuristics, one to determine the order in which the outer loop visits the possible candidate subsequences, and the other one to determine the order in which the inner loop visits the subsequences for a given current candidate. The algorithm is shown in Algorithm 2.

SAX Representation

The full name of SAX is Symbolic Aggregate Approximation, it can convert a time series into a sequence of symbols. It was first suggested by Lin et al. [23].

In order to reduce the dimensions of a time series to save computational time, the time series is first be symbolized [11]. A time series is divided into w segments, the average value of the data falling within a same segment is calculated. These average values are the new data points in the dimensionality-reduced representation. This representation is known as Piecewise Aggregate Approximation (PAA). After transforming a time series into PAA representation, we further transform it into a sequence of finite symbols. Since time series subsequences that are normalized tend to have a highly Gaussian distribution, we can determine the 'breakpoints' that produce same areas under

Algorithm 2 Keogh et al.'s algorithm

```
1: //Initialization
2: discord distance = 0
3: discord location = NaN
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| ≥ n then
11:      Dist = Distance Measure(t_p, ...t_{p+n−1}, t_q, ...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)
```

Gaussian curve.

Definition 6 ***Breakpoints*** *(by Lin et al.): Breakpoints are a sorted list of numbers* $B = \beta_1, \beta_2, ..., \beta_{a-1}$ *where a is the number of symbols such that area under a N(0,1) Gaussian curve from* β_i *to* $\beta_{i+1} = 1/a$ *(*β_0 *and* β_a *are defined as* $-\infty$ *and* ∞*, respectively).*

These breakpoints can be determined by looking them up in a statistical table. Table 2.2 gives the breakpoints for values of a from 3 to 7.

	Number of symbols a				
	3	4	5	6	7
β_1	-0.43	-0.67	-0.84	-0.97	-1.07
β_2	-0.43	0	-0.25	-0.43	-0.57
β_3		0.67	0.25	0	-0.18
β_4			0.84	0.43	0.18
β_5				0.97	0.57
β_6					1.07

Table 2.1: All breakpoints for number of symbols from 3 to 7. For example, area under a N(0,1) Gaussian curve from $-\infty$ to -0.67 is equal to 0.25.

We can assign a symbol to each region, a PAA value is then mapped to the symbol for the region that it falls in. Figure 2.1 illustrates the idea.

Approximating the Magic Outer Loop

We begin by sliding a window with length n across time series T, extracting the subsequences, then converting all the normalized subsequences into SAX words. All the words are placed in an array where the index refers back to the original sequence.

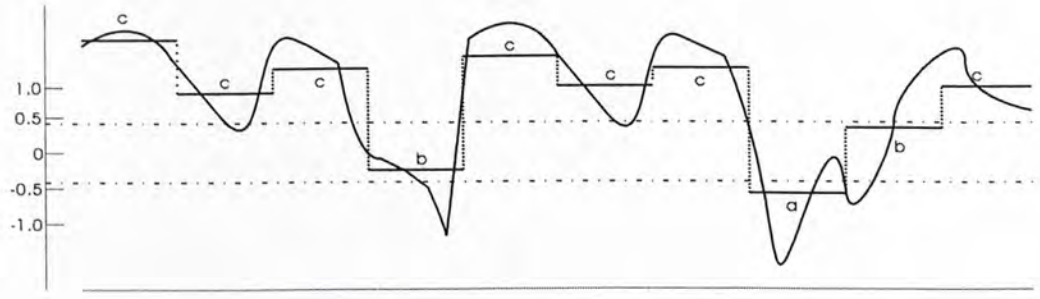


Figure 2.1: A time series is transformed into PAA representation and then convert into a sequence of symbols by using predetermined breakpoints

Now all the SAX words can be embedded into an augmented trie where the leaf nodes contain a linked list of all words that map there. The count of the number of occurrences of each word is stored to the rightmost column of the array. Figure 2.2 illustrates the idea.

Now we can prepare the search order of the outer loop. First, we find the leaf node with smallest number of occurrences. All the subsequences mapped to this node will be examined first. For the rest of the subsequences, they are visited in random. This heuristic can help us to approximate the location of discord, as intuitively these subsequences are less similar to the rest of the subsequences.

Approximating the Magic Inner Loop

When the i^{th} word is considered in the outer loop, we look up the word that it maps to, by examining the i^{th} word in the array. We then find a node which gives us a longest matching path in the trie, all the subsequences in this node are searched first. After exhausted this set of subsequences, the unsearched subsequences are visited in a random order. According to the experimental results from Keogh et al., it is very efficient in grouping similar

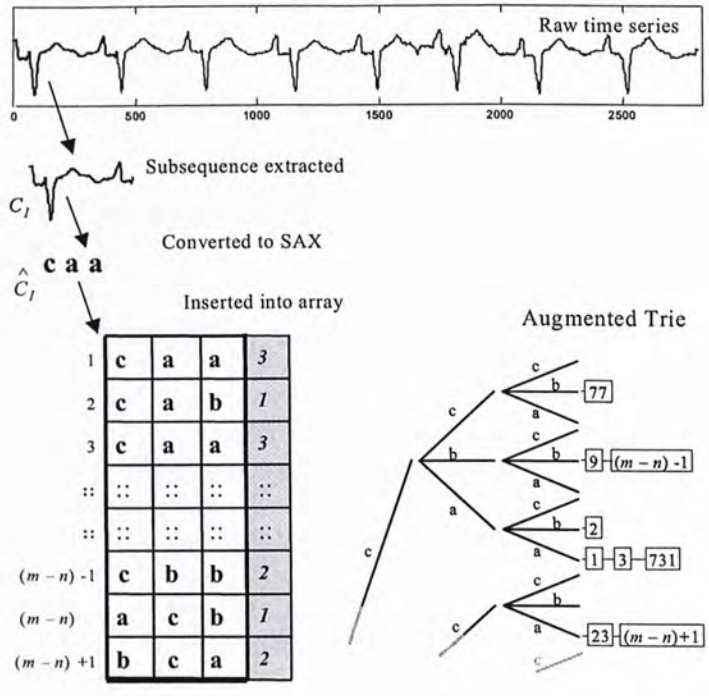


Figure 2.2: Keogh et al.'s idea is illustrated in this diagram

subsequences together. Then it can gives us a greater chance to break the inner loop.

2.2.4 Performance Analysis

In the previous sections, we have introduce brute force algorithm and Keogh et al.'s algorithm. Actually, they are nearly the same except for the search order of the inner loop and outer loop. In this section, we want to show the importance of the search order and discuss more about the magic case.

Algorithm 3 shows us a generic framework for solving discord discovery problem. By changing the heuristics for finding the search order of the outer loop and inner loop, the performance of this algorithm will be completely different. In fact, if we apply sequential search order for both outer loop and inner loop, it will

Algorithm 3 Generic framework

```
1: //Initialization
2: discord distance = 0
3: discord location = NaN
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| ≥ n then
11:         Dist = Distance Measure(t_p, ...t_{p+n−1}, t_q, ...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)
```

become a smarter brute force algorithm. If we apply Keogh et al.'s suggested heuristic into this algorithm, it will become the one that we have mentioned in section 2.2.3.

In order to have a better understanding of the importance of the search order, we will consider 3 cases:

- **Best case**: We call it the magic case. In this case, we can obtain prefect orderings. For outer loop, the subsequences are sorted by descending order of the non-self distance to their nearest neighbors, so the discord is placed at the first position of the outer loop. For inner loop, the subsequences are sorted by ascending order of the distance to the current candidate in the outer loop.

  In this case, for the first time running the inner loop, we must complete the whole loop. However, after that we can break the inner loop during the first iteration. It is because we can obtain the true discord distance after first time running the inner loop and we can obtain the nearest neighbor of the second candidate in the outer loop list at first iteration of the inner loop. Furthermore, it is trivial that the distance between the second candidate and its nearest neighbor must be smaller than the true discord, otherwise the second candidate will be the discord. Thus, the inner loop can be broken during the first iteration.

  In order to make it clear, we use m to represent the length of the discord and use n to represent the total length of the given sequence. Obvious, the time complexity is equal to 1 occurrence of $(m - n + 1)$ steps which comes from the first inner loop plus $(m - n)$ occurrences of single step of the remaining inner loop, so it equals to $O(m)$.

- **Average case**: We call it the random case, as for both inner loop and outer loop we just randomly order the subsequences.

It is difficult to analyze the performance of the random case, since it greatly depends on the data. However, we can ensure that the performance must be bounded from below $O(m)$ and from above $O(m^2)$.

- **Worst case**: We call it the perverse case. It is exactly the reverse orderings of the magic case. For outer loop, the subsequences are sorted by ascending order of the non-self distance to their nearest neighbors, so the true discord is placed at last position of the outer loop. For inner loop, the subsequences are sorted by descending order of the distance to the current candidate in the inner loop.

  Unlike the best case, this time we can not break the inner loop for the first few iterations. It is because now the true discord is placed at the end of the outer loop list. It means we can not find the true discord until we examine the last subsequence in the outer loop list. Actually, when we complete the first inner loop, we can obtain a best_so_far_distance. However, this value can not help us to break the inner loop in the future time. Based on the features of the outer loop orderings and inner loop orderings, the distance between the second candidate and its nearest neighbor must be greater than the current best_so_far_distance and we can only locate the nearest neighbor at the last iteration of the inner loop. Thus, it is impossible the break the inner loop.

  It is obvious that the time complexity is $O(m^2)$, as we must go through $(m-n)$ occurrences of the $(m-n)$ steps of inner loop calculation.

After this analysis, we know that the best solution of this problem can not be better than $O(m)$ in time complexity. And we also know that if we can approximate the magic orderings, our solution will be very close to the best solution. More about

this analysis can found in this paper [24]. Keogh et al. suggested a solution for approximating the best case. We found that there is still room for improvement. Later, we will show that our suggested solution is also close to magic case, which is more close to parameterless.

2.3 Proposed Approach

We follow the framework of the algorithm in [24]. 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.

In the above algorithm, we discover that 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 in algorithm 4. Given the subsequence p, the Inner heuristic order should pick the subsequence q closest to p first, since it gives the smallest $Dist$ value, and which will have the best chance to break the loop at Line 12 in algorithm 4. 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.

Algorithm 4 Base Algorithm

```
1: //Initialization
2: discord distance = 0
3: discord location = NaN
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| ≥ n then
11:         Dist = Distance Measure(t_p, ...t_{p+n−1}, t_q, ...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)
```

2.3.1 Haar Transform

The Haar wavelet Transform is widely used in different applications such as computer graphics, image, signal processing and time series querying [27]. 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 $f(x) =$ (7 5 3 5) is shown below.

Example

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

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

Haar transform can be realized by a series of matrix multiplications as illustrated in Equation (2.2). Envisioning the example

input signal $\vec{x}$ as a column vector with length $n = 4$, an intermediate transform vector $\vec{w}$ as another column vector and Haar transform matrix **H**

$$\begin{bmatrix} x'_0 \\ d^1_0 \\ x'_1 \\ d^1_1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{2.2}$$

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

Hence we 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 5 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 transforation can be computed quickly, requiring linear time in the size of the time series. In our experiments, we find that normalization

factor of $1/\sqrt{2}$ can give us a better performance comparing with $1/2$, it may be because factor of $1/\sqrt{2}$ can preserve the distance of two sequences.

Algorithm 5 Haar transform

```
1: //Initialization
2: n = length of input vector
3: output vector = input vector
4: dummy vector = all zeros
5:
6: //state the conversion
7: while n >= 2 do
8:    n=n/2
9:    for (i = 0; i < n; i++) do
10:      dummy vector[i]=(output vector[2*i]+output vector[2*i+1])/√2
11:      dummy vector[i+n]=(output vector[2*i]-output vector[2*i+1])/√2
12:   end for
13:   for (i = 0; i < (n*2); i++) do
14:      output vector[i]=dummy vector[i]
15:   end for
16: end while
17:
18: //Return Solution
19: Return(output vector)
```

2.3.2 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. 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 in fact it can give us a petty good result. Now we can determine the 'cutpoints'[23] by using a Gaussian curve. The cutpoints define the discretization of the $i-th$ coefficient.

Definition 7 ***Cutpoints*** *(By Lin et al.): 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* β_i *to* $\beta_{i+1} = 1/a$
β_0 *and* β_a *are defined as* $-\infty$ *and* ∞, *respectively.*

These breakpoints are determined by looking them up in a statistical table. Table 2.2 gives the breakpoints for values of a from 2 to 7.

	Number of symbols a					
	2	3	4	5	6	7
β_1	0	-0.43	-0.67	-0.84	-0.97	-1.07
β_2		0.43	0	-0.25	-0.43	-0.57
β_3			0.67	0.25	0	-0.18
β_4				0.84	0.43	0.18
β_5					0.97	0.57
β_6						1.07

Table 2.2: All breakpoints for number of symbols from 2 to 7. For example, area under a N(0,1) Gaussian curve from $-\infty$ to -0.84 is equal to 0.2.

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 β_0 and β_1, it is mapped to the first symbol 'a'. If the i^{th} coefficient is between β_{j-1} and β_j, it will be mapped to the j^{th} symbol, etc. In this way we form a word [23] for each subsequence.

Definition 8 ***Word mapping*** *(By Lin et al.): 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* $B_i = \beta_0, ...\beta_a$ *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:*

$$\hat{c_i} = \alpha_j \iff \beta_{j-1} \leq \bar{c_i} < \beta_j \tag{2.3}$$

2.3.3 Augmented Trie

First, we transform all the normalized 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 discretization algorithm. Finally, all the words are placed in an **array** with a pointer referring back to the original sequences. Figure 2.3 illustrates this idea.

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 array. 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 [24] a pre-selected word length is required which is used by using a piecewise linear mechanism to compress the subsequences, and that also determine the trie height. This 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 pro-

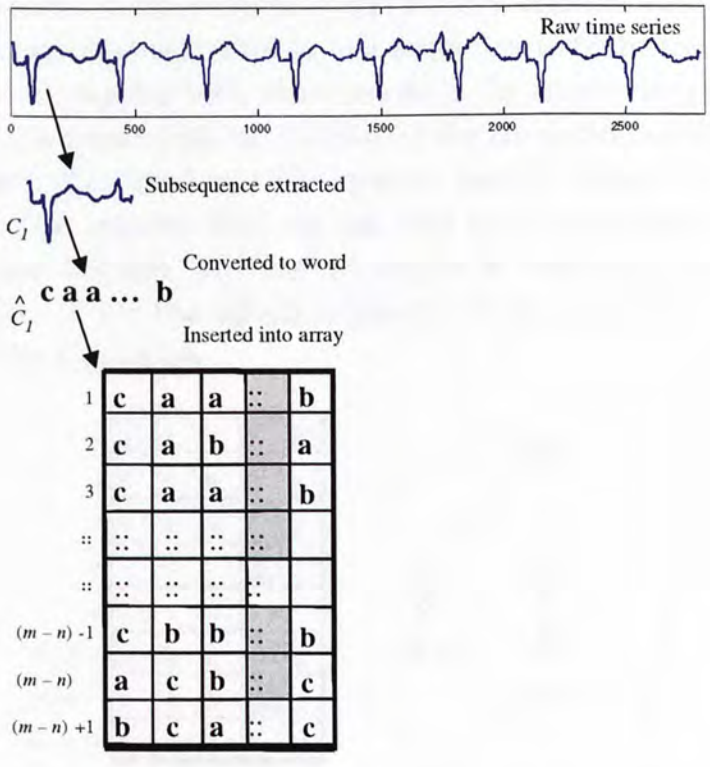


Figure 2.3: An array of words for building an augmented trie

cess 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, 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. For a word which appears with other words in the same trie node, such words in the same node are similar at the resolution at that level, hence they are closer to their nearest match. Hence 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.

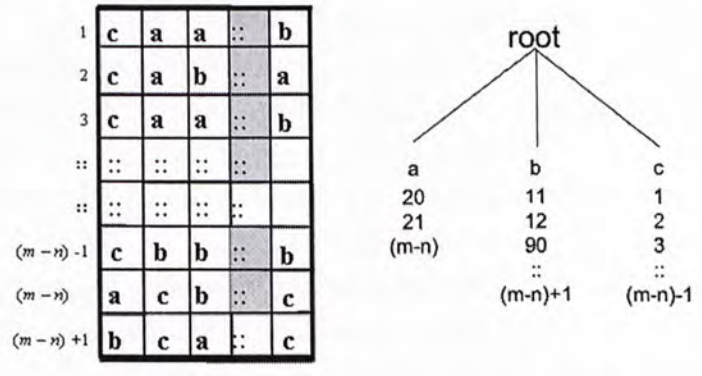


Figure 2.4: 1^{st} symbol is considered for splitting the root node. All leaf nodes will be split, since no leaf node contains only 1 word.

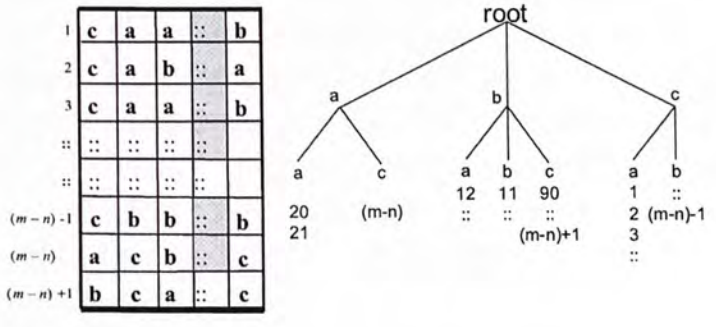


Figure 2.5: 2^{nd} symbol is considered. No tree node is split in next iteration, since there is only 1 word mapped to 'ac'.

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

2.3.4 Approximating the Magic Outer Loop

Heuristic: the leaf nodes 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.

2.3.5 Approximating the Magic Inner Loop

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

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.

2.3.6 Experimental Result

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 time series from different domains, were obtained from 'The UCR Time Series Data Mining Archive' [15].

Anomaly Detection

Anomaly Detection in a time series database has received much attention [30, 7, 14]. However, most of the anomaly detection algorithms require many parameters. The beauty of using discords as anomaly detectors is that our algorithm only require two simple parameters, one is the length of the discord, another is the alphabet size that according to our experience 3 is the best for any dataset. To show the utility of discords for anomaly detection, we investigated electrocardiograms (ECGs) which are a 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 as 256 that is approximately one full heartbeat and set the alphabet size to be three.

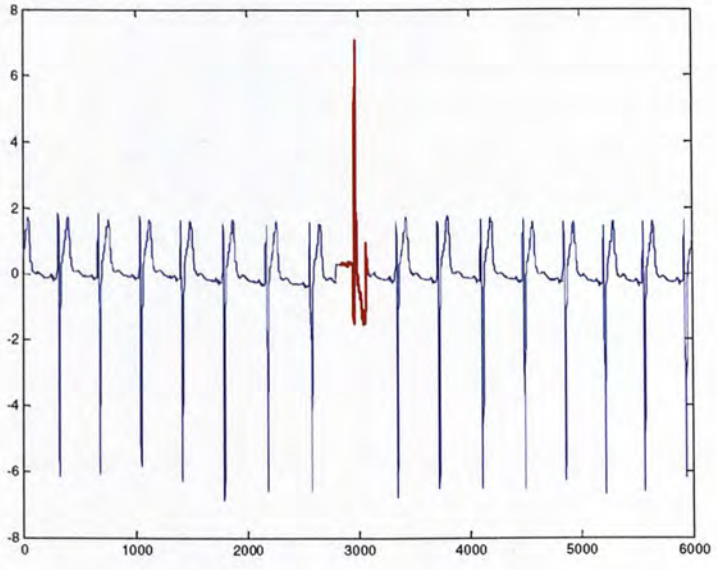


Figure 2.6: A time series discord (marked in bold line)was found at position 2830

In figure 2.6, it is easy to discover the anomaly by eye. However, we may have many of ECGs in our database, it is impossible to examine all of them manually.

The Performance of our Algorithm

From 5 datasets, 10 data sequences were picked. For each data sequence, subsequences of lengths 512, 1024, 2048, 4096 and 8192 were randomly extracted, forming 5 derived datasets of varying dimensions. In below figures, we compared the Keogh et al.'s algorithm with our proposed algorithm in terms of number of times the Euclidean distance function is called. In this experiment, we set the length of the discord as 128 and found

the discord on all the created subsequences. Each of the experiments was repeated 10 trials and the average value was taken.

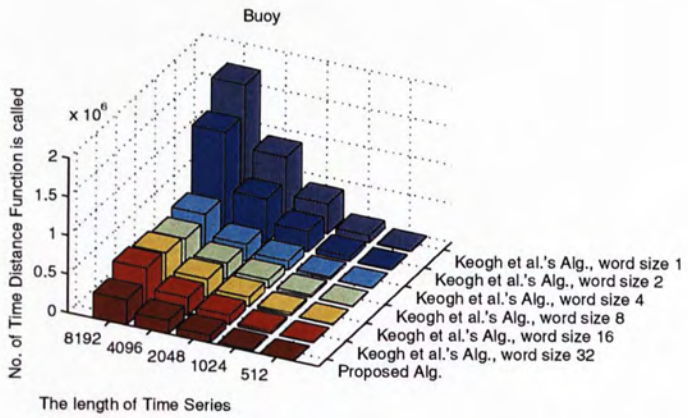
In this experiment, we did not measure the CPU time directly, in order to ensure that there was no implementation bias. In fact, Keogh et al. discovered that the distance function accounts for more than 99% of the running time. Due to the fairness, it provides us another way to measure the running time.

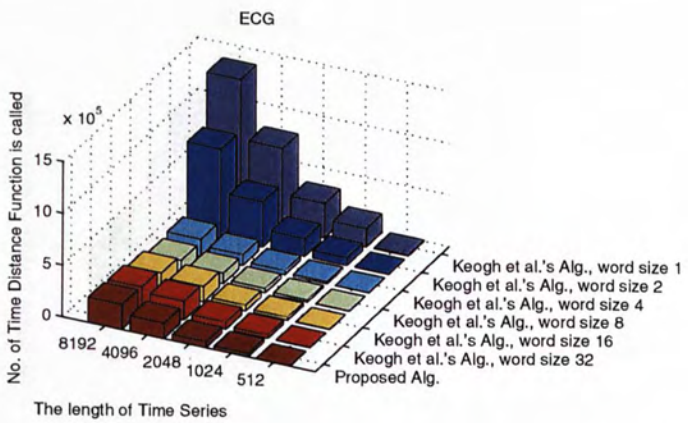
From the figure 2.7, we found that there was no any special value for word size which was suitable for any task on any dataset. Keogh et al. suggested relatively smooth and slowly changing datasets favor a smaller value of word size, whereas more complex time series favor a larger value of word size. Besides this factor, the length of the discord will also affect the value of the optimal word size. The details will be given in the next section.

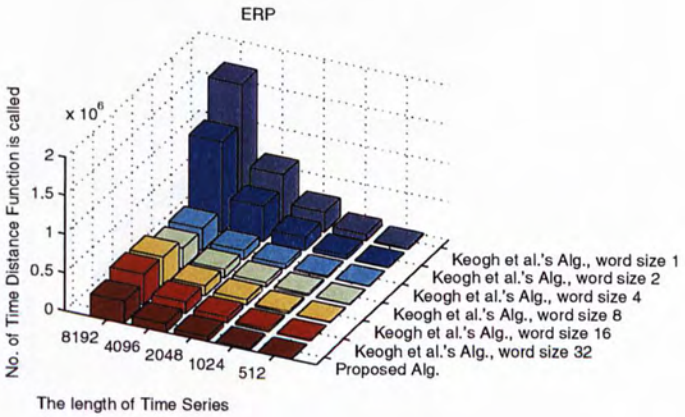
Although we claimed that the distance function accounts for more than 90% of the running time, people may be still interested in total running. In this experiment, we measured the CPU time directly. Both algorithms were implemented by ANSI C. And the experiment was run in Dell Optiplex 280 Intel P4 3.2Ghz with 2GB RAM. Figure 2.8 showed that distance function is a good measure for comparing the real running time.

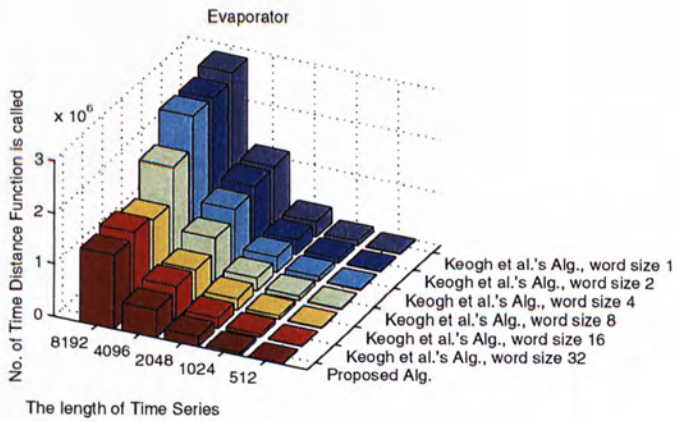
Experiments on Alphabet Size

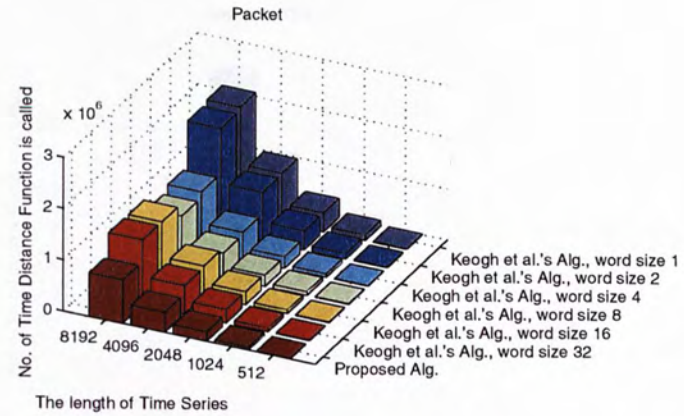
Our proposed algorithm requires users to choose the value of one parameter, the alphabet size. In fact, this value does not affect the correctness of our algorithm, it only affects the performance. In our algorithm, we want to map the discord to a unique or rare word. On the other hand we try to group the other similar subsequences together and map them into the same word. If the alphabet size is too small, we do not have enough information to distinguish a discord from the rest of other subsequences. If the alphabet size is too large, almost all subsequences will

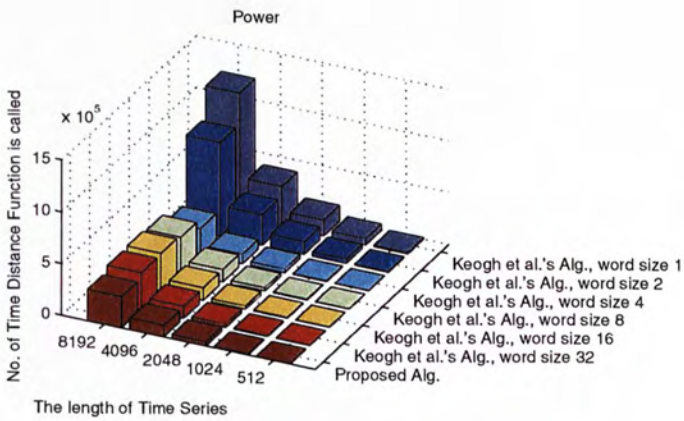

Buoy
x 10^6
2
1.5
1
0.5
0
No. of Time Distance Function is called
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.

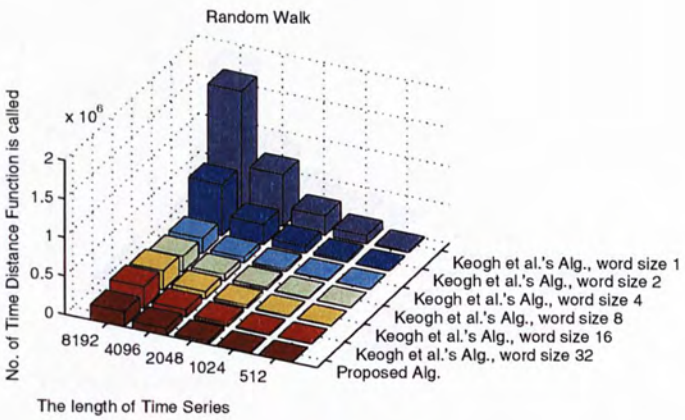

ECG
x 10^5
15
10
5
0
No. of Time Distance Function is called
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.

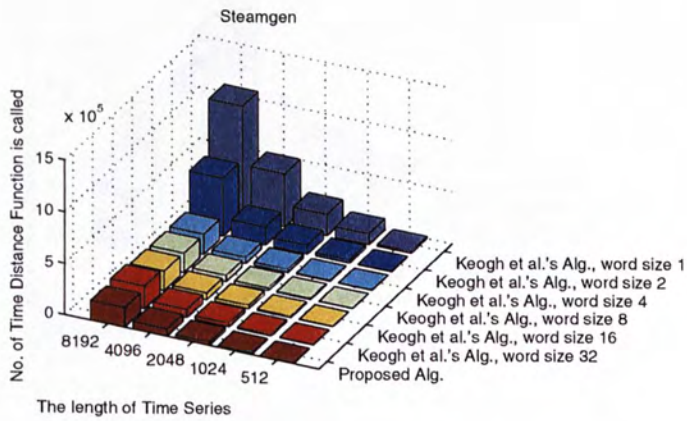

ERP
x 10^6
No. of Time Distance Function is called
2
1.5
1
0.5
0
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.


Evaporator
x 10^6
No. of Time Distance Function is called
3
2
1
0
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.


Packet
x 10^6
3
2
1
0
No. of Time Distance Function is called
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.


Power
x 10^5
15
10
5
0
No. of Time Distance Function is called
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.


Random Walk
x 10^6
2
1.5
1
0.5
0
No. of Time Distance Function is called
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.


Steamgen
x 10^5
15
10
5
0
No. of Time Distance Function is called
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.

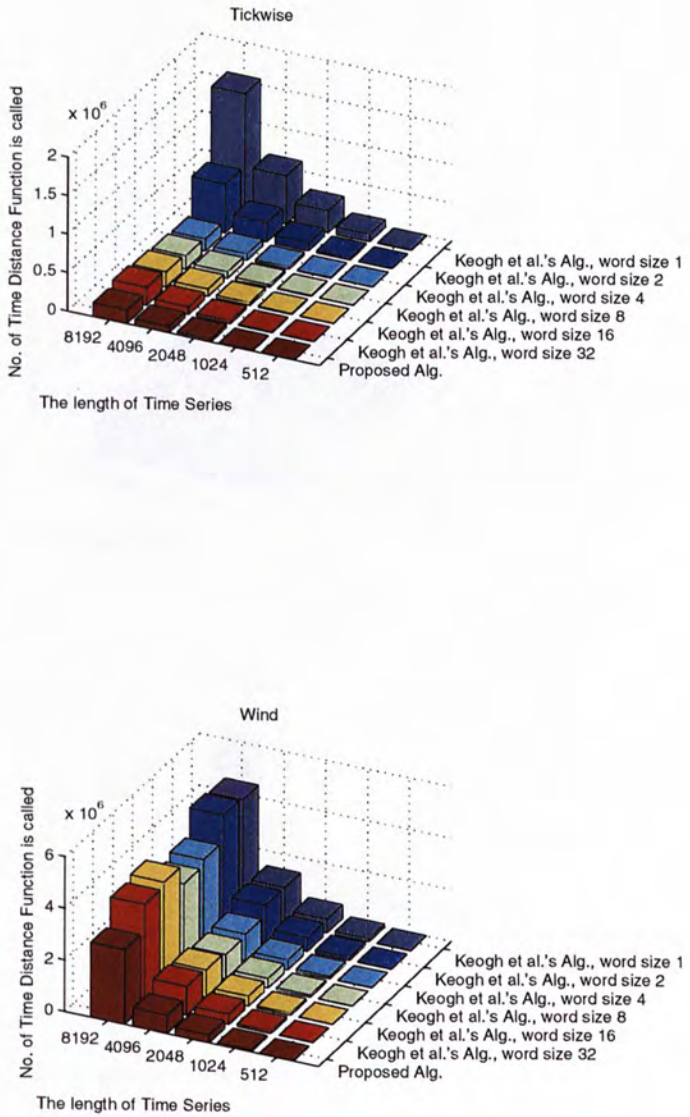
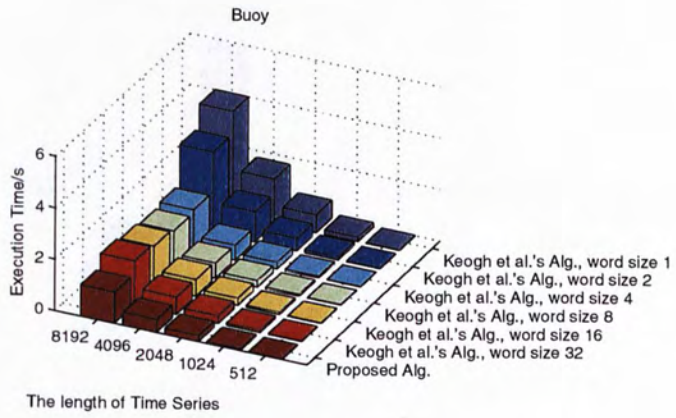
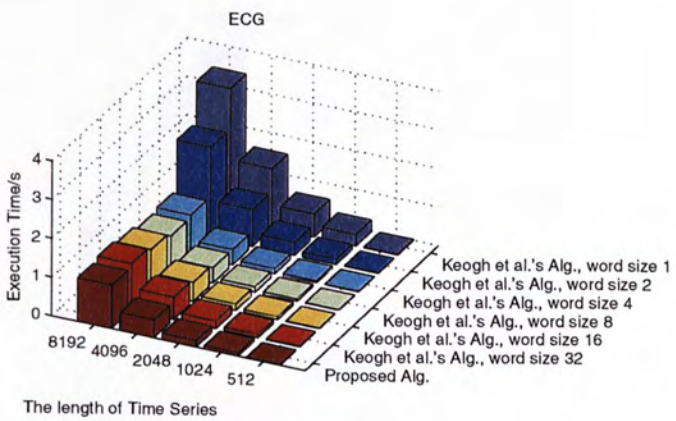
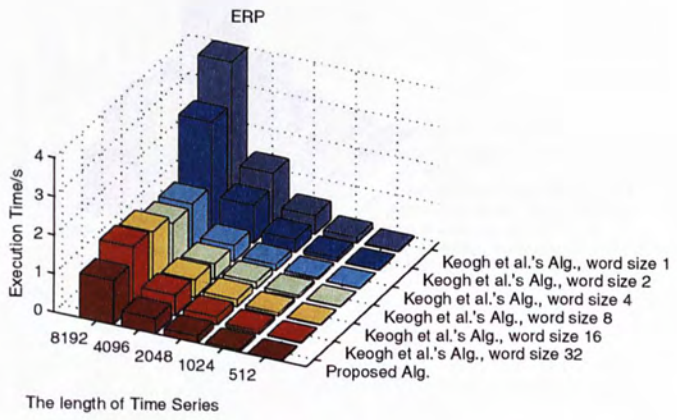
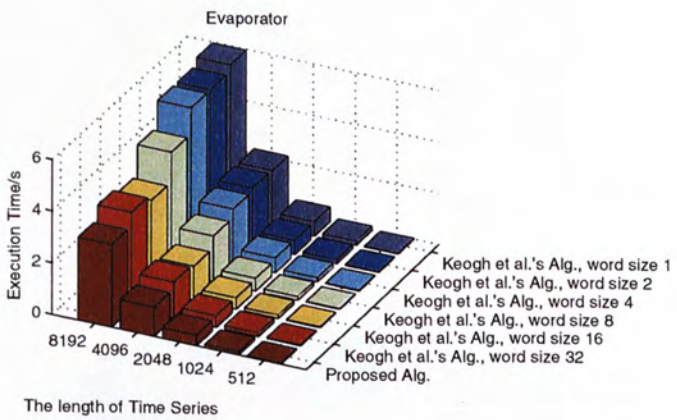


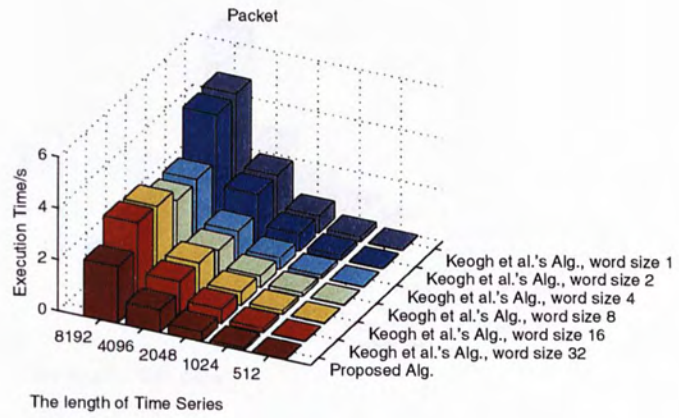
Figure 2.7: Number of times distance function is called by Keogh et al.'s Algorithm and Our Proposed Algorithm

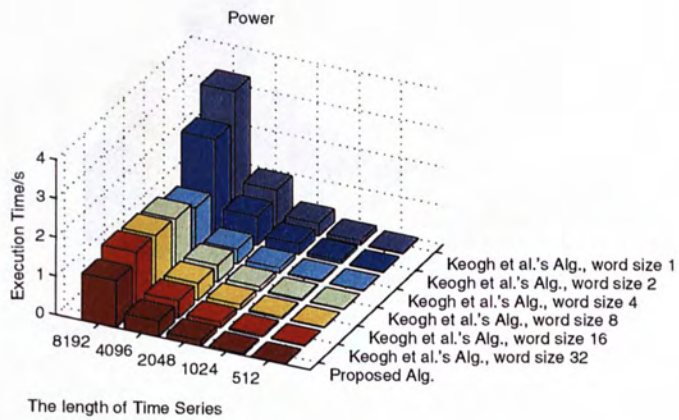

Buoy
Execution Time/s
0
2
4
6
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.


ECG
Execution Time/s
0
1
2
3
4
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.


ERP
Execution Time/s
0
1
2
3
4
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.


Evaporator
Execution Time/s
0
2
4
6
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.


Packet
Execution Time/s
0
2
4
6
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.


Power
Execution Time/s
0
1
2
3
4
8192
4096
2048
1024
512
The length of Time Series
Keogh et al.'s Alg., word size 1
Keogh et al.'s Alg., word size 2
Keogh et al.'s Alg., word size 4
Keogh et al.'s Alg., word size 8
Keogh et al.'s Alg., word size 16
Keogh et al.'s Alg., word size 32
Proposed Alg.

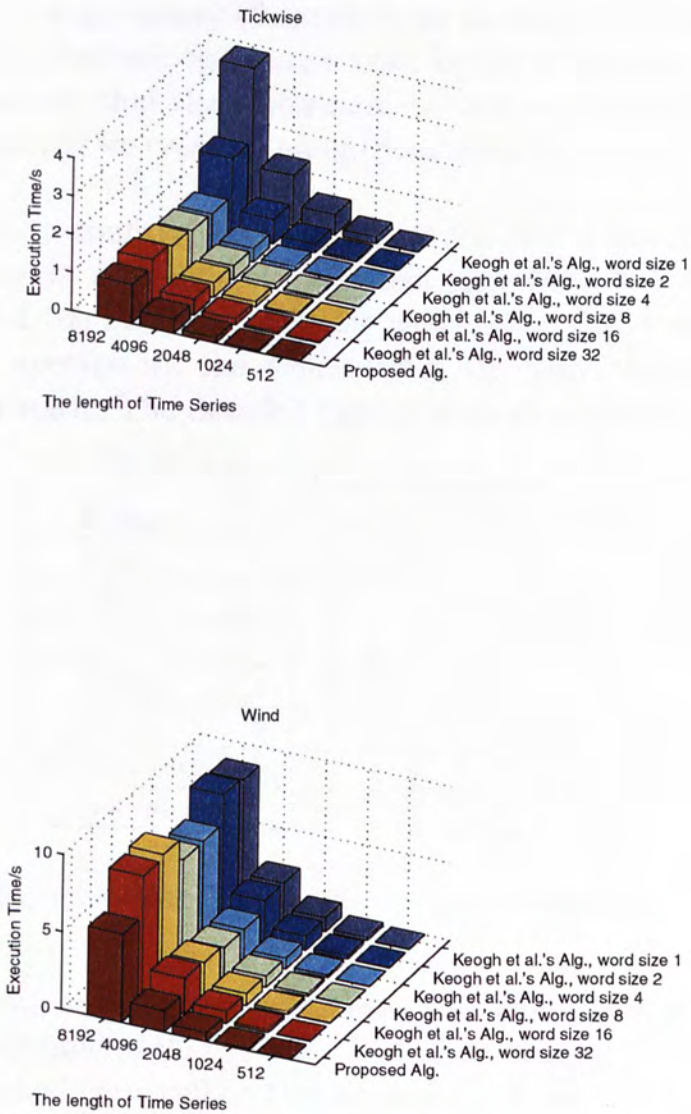


Figure 2.8: The running time of Keogh et al.'s Algorithm and Our Proposed Algorithm

be mapped to unique words. This suggests that the value of alphabet size will affect the performance of our algorithm.

According to our experiments, a value of 3 is a petty good choice for a large variety of datasets, so we simply hardcode the value of alphabet size to 3 in our work for the default case. However, we believe that the performance of our proposed algorithm will be better if we can find an optimal alphabet size for a given dataset.

In order to confirm this, we tried to find the discords on different datasets with different alphabet sizes from 2 to 9. Then we recorded the best alphabet size which gives the best performance on average for the sequences in the same dataset with the same length. The detailed figures were shown below.

	Length of sequences				
Dataset	512	1024	2048	4096	8192
Burstin	3	3	4	3	3
ECG	2	3	3	3	4
ERP	3	2	2	2	2
Packet	3	3	3	3	3
Power	3	6	3	3	2
Random Walk	2	3	2	2	3
Tickwise	3	3	2	2	3

Table 2.3: The best alphabet size for different length of sequences in different datasets

In general, sequences coming from the same dataset shared the same optimal alphabet size. Note that this optimal alphabet size worked very well on the entire dataset, although it could not obtain the best performance on some sequences. This result suggests that a simple preprocessing for finding an optimal alphabet size can further improve the performance of our suggested method.

Also, 3 is good choice for the alphabet size. Although it could

not obtain the best result for some of the sequences, on average it performed very well in different datasets.

2.4 More on discord length

In previous section, we have discussed our proposed algorithm. However, we only focus on the discord length that is equal to 2^i, where i is a positive integer. Fortunately, by slightly modifying the haar transform process, this problem can be solved easily. In this section, we will discuss this problem.

2.4.1 Modified Haar Transform

It is true that original haar transform only considers the sequences with length equal to 2^i, where i is a positive integer. However, by adding some tricks on the transformation, the original process can apply on all cases. For example, the procedure to find the Haar transform of a discrete function $f(x) = (7\ 3\ 5\ 1\ 8)$ is shown below.

Example

Resolution	Averages	Coefficients
8	(7 3 5 1 8 0 0 0)	
4	(5 3 4 0)	(2 2 4 0)
2	(4 2)	(1 2)
1	(3)	(1)

As now there are only 5 elements in $f(x)$, we must first add some elements into $f(x)$, after that we can use the original process for transformation. The simplest way is adding zeros. If the length of the sequence is n, we should add $2^{\lceil \frac{\lg(n)}{\lg(2)} \rceil} - n$ zeros at the end of the original sequence. Resolution 8 is the full resolution of discrete function $f'(x)$ after adding 3 zeros at the end of $f(x)$.

Then we can treat it as a normal sequence for conversion. In resolution 4, (5 3 4 0) are obtained by taking average of (7 3), (5 1), (8 0) and (0 0) at resolution 8 respectively. (2 2 4 0) are the differences of (7 3), (5 1), (8 0) and (0 0) divided by two respectively. This process is continued until resolution of 1 is reached. The Haar transform $H(f(x)) = (3\ 1\ 1\ 2\ 2\ 2\ 4\ 0)$ is obtained which is composed of the last average value 3 and the coefficients found on the right most column, 1, 1, 2, 2, 2, 4 and 0.

Although we can use this Haar wavelet to calculate the distance between subsequences, we will not do it. As we have mentioned before, the distance function accounts for more than 99% of the running time, we do not want to add any overhead on distance function. 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. In this case, the transformed subsequences are only used for heuristic outer loop and heuristic inner loop. For the distance calculation, we will use the original subsequences.

2.4.2 Fast Haar Transform Algorithm

Sometimes, we may need to find the discord of a range of discord length. It seems that there is no smart method to solve this problem, we can only run the algorithm for several times for different discord lengths. However, we discovered that we can have a minor optimization on Haar transform. Assume we have 2 functions, one is $f(x) = (7\ 3\ 5\ 1\ 8)$, another one is $f'(x) = (7\ 3\ 5\ 1\ 8\ 8)$. Obviously, if we add 8 into $f(x)$, it will become $f'(x)$. Now, we convert both functions into haar wavelets, the process is shown on figure 2.9 and figure 2.10.

In this example, it is not difficult to discover that most of

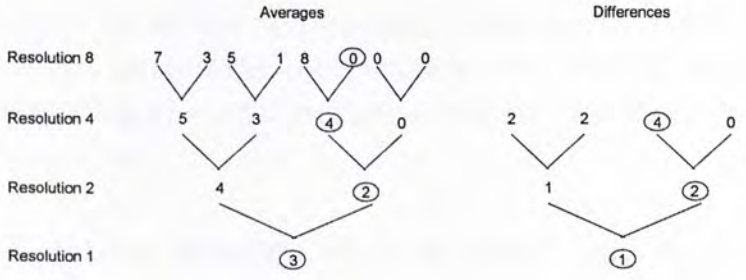


Figure 2.9: After transformed $f(x)$, it becomes (3 1 1 2 2 2 4 0)

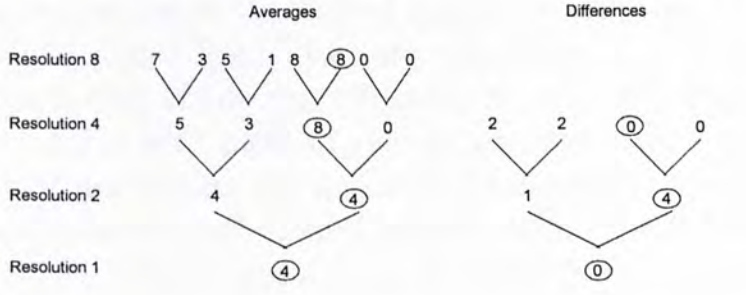


Figure 2.10: After transformed $f'(x)$, it becomes (4 0 1 4 2 2 0 0)

the intermediate averages and differences of different resolution levels are the same. However, there are still some differences between the two conversion processes. We have located the differences in both figure 2.9 and figure 2.10 with circles. We can see all the differences are related to the new added element 8. In fact, for any $f(x)$ and $H(f(x))$, if we change one of the element in $f(x)$, the new $H(f(x))$ is the same as the old $H(f(x))$ except the coefficients that are related to the changed element.

By using this property, if we need to find the discord of 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 $a+1$. It 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.

2.4.3 Relation between discord length and discord location

In this section, we want to show that discord location is always the same regardless of the discord length. In other words, if we find the data points from a to b are very different to other data points, no matter of the discord length the result discord must at least overlap with most of the data points from a to b. If it is true, it also means the definition of discord is meaningful. Otherwise, it means the discord location highly depends on the discord length. It may not help us to locate the true discord of a given time series.

In order to show that the definition of discord is meaningful, we tried to find the discord of lengths 600, 700, 800, 900 and 1000 on a ECG. The result is shown in table 2.4.

	Discord location	
Length of discord	Start	End
600	10871	11470
700	10698	11397
800	10695	11494
900	10697	11596
1000	10699	11698

Table 2.4: Discord location for different length of discord

We could discover all the discords overlap from data point 10871 to 11397. Hence, we plotted the original sequence and the overlapping area on figure 2.11.

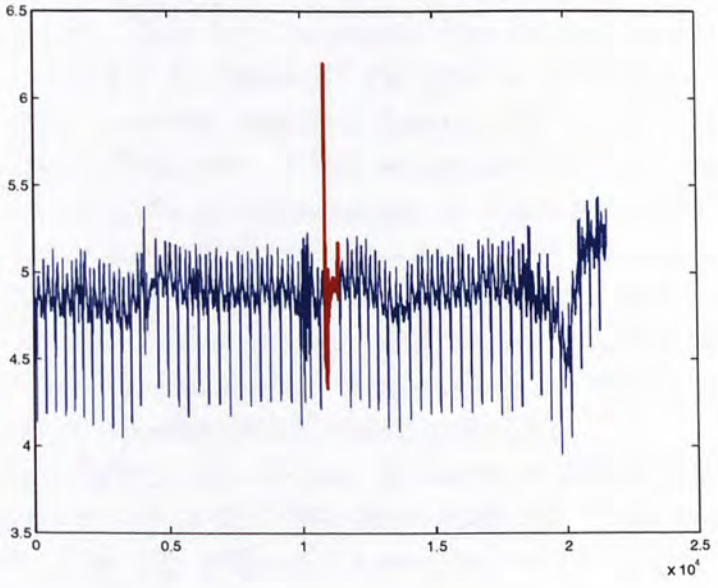


Figure 2.11: The overlapping area (marked in bold line) was found from position 10871 to 11397

In this experiment, we found that the discords always highly overlap. It showed the definition of discord is meaningful, as it could locate the real unusual pattern with different length of discord. In fact, if we look at the figure 2.11 carefully, the highlighted part was the most unusual part comparing with other subsequences. Although, the discord length can not affect the final result very much, it is still important to choose a meaningful discord length. Normally, we want to locate the anomaly in a sequence which may contain certain cycles, so it is good to set the length of discords to be approximately one complete cycle.

If the sequence contains no cycle, it seems that we do not have a good indicator to set the length of discord. However, it is not true. We want to show that even if we just choose the length of discord randomly, we are still able to locate meaningful discords. In this experiment, we tried to find a set of most unusual patterns with different lengths. For example, if we set

the range of length is 50 to 100, we will first find the discord for length 50. Then we will report this discord to the screen. Next, we will find the discord for length 51. If the result discord overlap with previous reported discord, the result discord will be discarded. Otherwise, it will be reported to the screen also.

We randomly chose two sequences in a Random Walk dataset, so there were no cycle in these two sequences. In each sequence, we first found the discord set by setting the length of discord from 50 to 200. Then we repeated the experiment again, but this time we set the length of discord from 100 to 200. The result is shown in figure 2.12 and figure 2.13.

Consider figure 2.12. When the range is 100 to 200, a single discord found at location 268 with length 100. This result shows that even if we use different ranges, we still discover the same discord which is around location 250 to 350. However the range from 50 to 200 finds one more discord at position 423.

Since the application of discords is to locate the anomaly, the areas that are covered by a discord set is more important than the exact location of each discords. From the result, using a larger range of discord lengths can help to locate more anomalies. In the experiment, using the larger range result in a set of discords that totally covers a discord set generated with a sub-range.

2.5 Further Optimization

In the previously mentioned inner loop heuristic, we visit all the similar subsequences mapped to the same node with examining candidate subsequence first. After this step, the rest of the subsequences are visited in a random order. In fact, based on some special properties [27], we now suggest another approach to estimate the similarity between subsequences. This method can further improve the performance of the proposed algorithm.

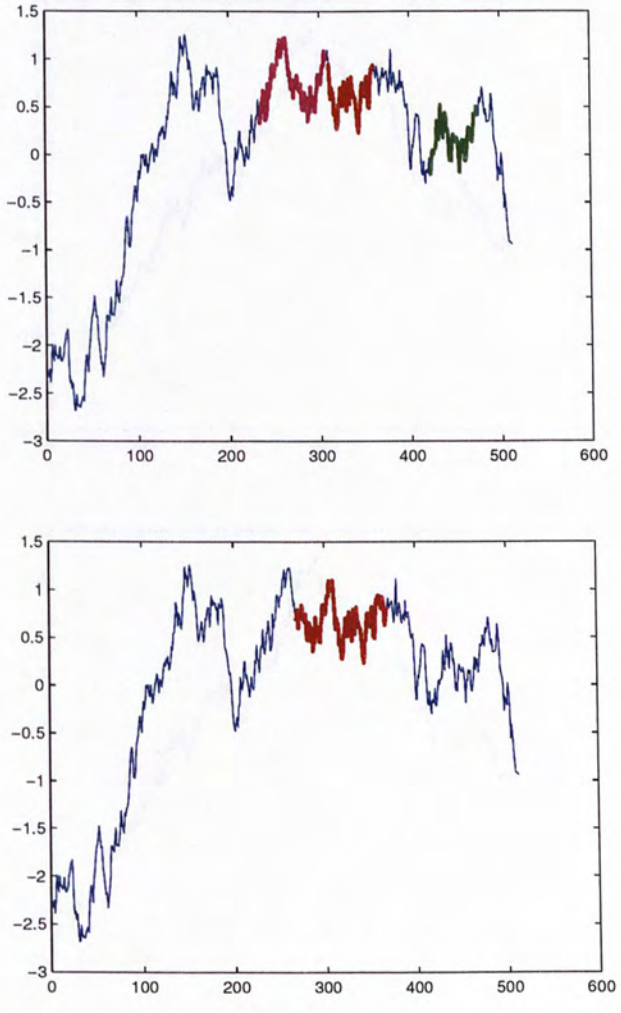


Figure 2.12: (top) Three discords were found. Discord location: 310, length of discord: 50. Discord location: 423, length of discord: 51. Discord location: 233, length of discord: 75. (bottom) One discord was found. Discord location: 268, length of discord: 100

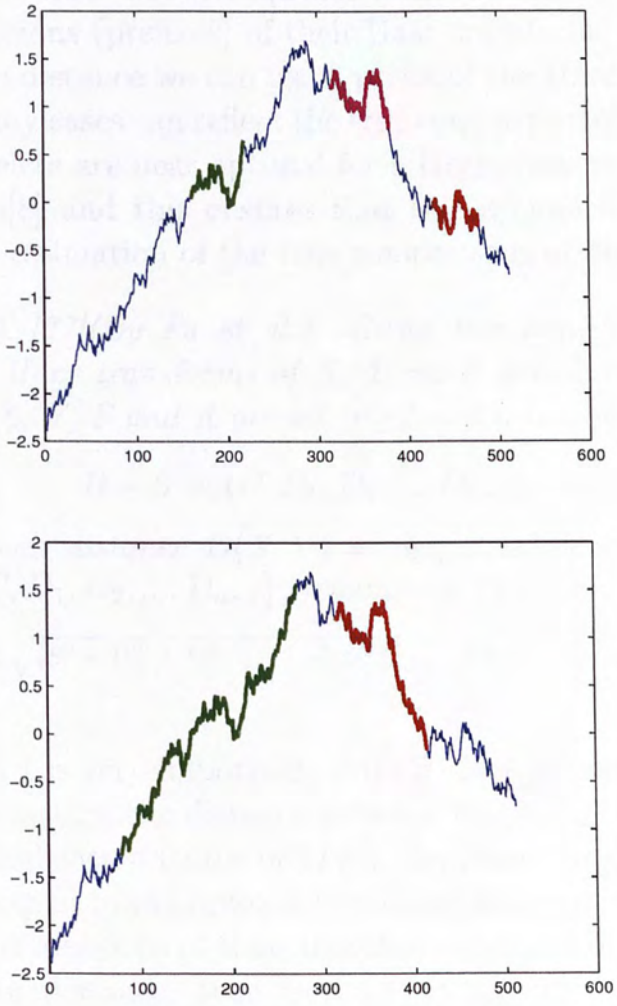


Figure 2.13: (top) Three discords were found. Discord location: 428, length of discord: 50. Discord location: 161, length of discord: 58. Discord location: 315, length of discord: 64. (bottom) Two discords were found. Discord location: 315, length of discord: 100. Discord location: 79, length of discord: 194

2.5.1 Improved Inner Loop Heuristic

Our enhancement is based on the fact that we can estimate the distances between two subsequence from the distance between the compressions (prefixes) of their Haar transforms. When we estimate the distance we can use a prefix of the Haar transform which in many cases can reflect the true comparison of distances. In fact wavelets are near optimal for a large class of signals for compression[8] and this ensures that this approach can often give a good estimation of the true comparisons of distance.

Theorem 1 *[27](By Fu et al.) Given two sequences X and Y, and the Haar transforms of X, Y are S and R respectively. Lengths of X, Y, S and R are all (n≥2 and n is a power of 2).*

$$R - S = \{C, D_1, D_2, ..., D_{n-1}\}$$

The Euclidean distance $D(X,Y) = S_{log_2 n}$ *can be expressed in terms of* $\{C, D_1, D_2, ..., D_{n-1}\}$*, recursively by*

$$\begin{aligned} &S_{i+1} = \sqrt{2} \times \sqrt{(S_i^2 + D_{2^i}^2 + D_{2^i+1}^2 + ... + D_{2^{i+1}-1}^2)} \quad \text{for } 0 \leq i \leq log_2 n - 1 \\ &S_0 = C \end{aligned} \tag{2.4}$$

Theorem 1 is very important, since it gives us some hints for roughly estimating the distance between two sequences. Firstly with a normalization factor of $1/\sqrt{2}$, the Haar based Euclidean distance is equal to the original Euclidean distance. Secondly, if only first h dimensions of Haar transform are used in calculation of Euclidean distance, then we can replace the $h+1^{th}$ to n^{th} coefficients with 0's in the transformed sequences, the result is a lower bound on the actual distance. With Haar transform, a prefix can preserve the low resolution information which is found in many applications to preserve better the Euclidean distance. Hence if two subsequences are very similar in the first h dimensions, it means that very likely the overall structures of two subsequences are very similar too.

Theorem 1 only can handle the distance between two Haar wavelets. As we want to estimate the distance a Haar wavelet and a word, we slightly modified the theorem suggested by Fu et al..

Theorem 2 *Given two sequences X and Y, where the Haar transform of X is R and the word of Y is $\hat{S}$ (the word is obtained by first transforming Y into Haar wavelet S, then S is further symbolized into a word by using the algorithm in section 2.3.2). Lengths of X, Y, R and $\hat{S}$ are all ($n \geq 2$ and n is a power of 2).*
The minimum Euclidean distance $MinD(X,Y)$ is equal to $G_{log_2 n}$ can be expressed in terms of $\{E_0, E_1, E_2, ..., E_{n-1}\}$, recursively by

$$\begin{aligned} G_{i+1} &= \sqrt{2} \times \sqrt{(E_i^2 + E_{2^i}^2 + E_{2^i+1}^2 + ... + E_{2^{i+1}-1}^2)} \quad \text{for } 0 \leq i \leq log_2 n - 1 \\ G_0 &= E_0 \end{aligned} \tag{2.5}$$

In order to improve on the inner search order heuristic, there is no modification on the trie construction and outer loop heuristic. We only need to note the current height of the trie. Imagine that the i^{th} candidate subsequence is considered in the outer loop. We look up the first h coefficients in the i^{th} entry of the array, then we can estimate the distance from the i^{th} subsequence to different leaf nodes by using the first h dimensions (h is the trie height). However, since the subsequences have been encoded in terms of a fixed size alphabet, our computation is based on the cutpoints in the alphabet. Let R be the i-th subsequence and the Haar transformed subsequence for R be given by $r_1, r_2, ...r_n$. We try to estimate the distance between R and a subsequence S that is stored in the trie. The subsequence $S = s_1, ...s_n$ has been mapped by a piecewise linear mechanism to a string of alphabets. Let $\hat{s}_i$ be the alphabet that s_i is being mapped to. For each alphabet a there is a range of possible values for the alphabet, let $U(a)$ be the upper limit of the values

for a and $L(a)$ be the lower limit of the values for a. Based on these values, we calculate the distance vector between R and S, which is estimated by $E_0, E_1, ..., E_h$ by using equation 2.6.

$$E_{j-1} = \begin{cases} r_j - U(\hat{s_j}) & \text{if } r_j > U(\hat{s_j}) \\ r_j - L(\hat{s_j}) & \text{if } r_j < L(\hat{s_j}) \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

For example, $a_1 = 0.5$ and $\hat{b_1} = b$. If the alphabet size is 3, we can look at the upper bound and the lower bound for alphabet b at table 2.2. Then we know the upper bound is 0.43 and lower bound is -0.43. Using equation 2.6, we know E_0 is 0.07.

After finding the value of $E_0, E_1, ..., E_h$, we put zeros into $E_{h+1}, E_{h+2}, ..., E_{n-1}$. Now the minimum distance between a wavelet and leaf nodes can be estimated by using theorem 2. According to the distance from i^{th} subsequence to different leaf nodes, by using theorem 2.

Revised Heuristic: For the inner loop for the i-th subsequence: According to the above Haar transform based distance from the i^{th} candidate subsequence to different leaf nodes, we sort the leaf nodes in ascending order. In this sorted list of leaf nodes, we randomly search all the subsequences in first leaf nodes. After exhausting this set of subsequence, we repeat with the next leaf node in the ordered list.

The reasoning is again that in order to break the inner loop, we need to find a subsequence that has a distance to the i^{th} subsequence p in the outer loop less than the best_so_far discord distance, hence the smallest distance to p will be the best to be try first.

2.5.2 Experimental Result

In figure 2.15, we compared the original heuristic inner loop and improved heuristic inner loop. In this experiment, from 4

datasets, 10 data sequences were picked. For each data subsequence, subsequences of length 512, 1024, 2048, 4096 and 8192 were randomly extracted, forming 4 derived datasets of varying dimensions.

From the experimental results, we found that the performance of improved inner loop was better then the original inner loop in all the cases. This result showed that our idea is correct. That means searching the similar candidate subsequences can provide us better chance to break the inner loop comparing with just searching the candidate subsequences randomly.

2.6 Top K discords

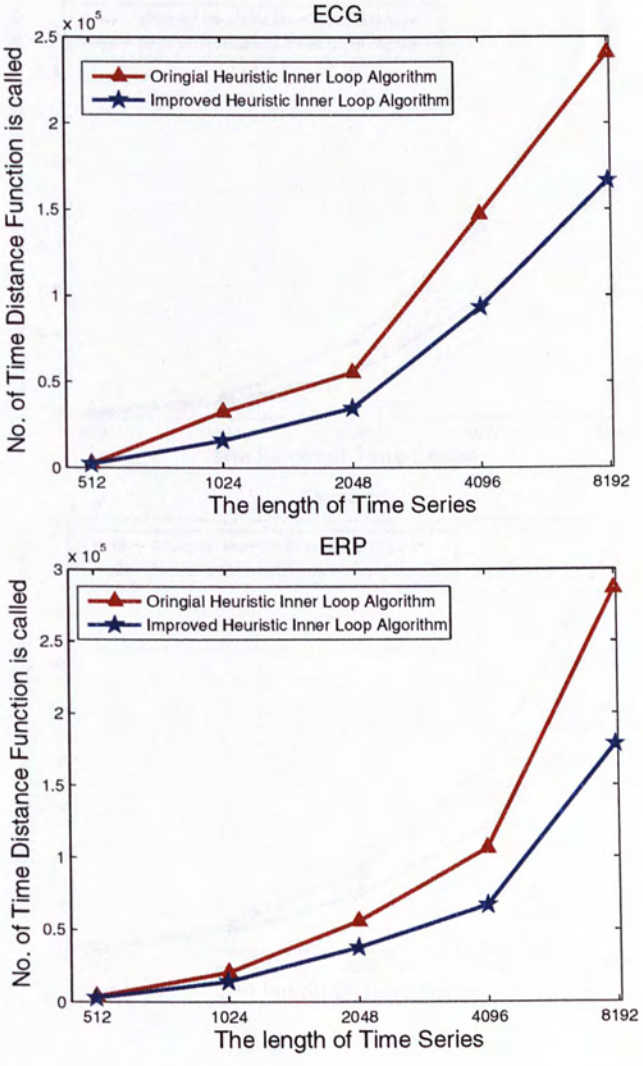
Sometimes, there will be more than one unusual patterns in a given time series. However, until now we only focus on the most unusual pattern. In this section, we will discuss top K discords. First, we will show the utility of top K discords, then we will talk about our algorithm for finding top K discords.

2.6.1 Utility of top K discords

As we have mentioned before, top K discords [24, 19] are very useful in many applications. Before showing examples, we need to formally define top K discords first.

Definition 9 K^{th} ***Time Series Discord*** *(By Keogh et al.): Given a time series T, the subsequence D of length n beginning at position p is the* K^{th} *discord of T if D has the* K^{th} *largest distance to its nearest non-self match, with no overlapping region to the* i^{th} *discord beginning at position* p_i*, for all* $1 \leq i < K$*. That is,* $|p - p_i| \geq n$.

For most applications, we do not only want to find the most unusual subsequence only. Instead, we want to discover all the


ECG
x 10^5
2.5
2
1.5
1
0.5
0
Oringial Heuristic Inner Loop Algorithm
Improved Heuristic Inner Loop Algorithm
No. of Time Distance Function is called
512
1024
2048
4096
8192
The length of Time Series
ERP
x 10^5
3
2.5
2
1.5
1
0.5
0
Oringial Heuristic Inner Loop Algorithm
Improved Heuristic Inner Loop Algorithm
No. of Time Distance Function is called
512
1024
2048
4096
8192
The length of Time Series

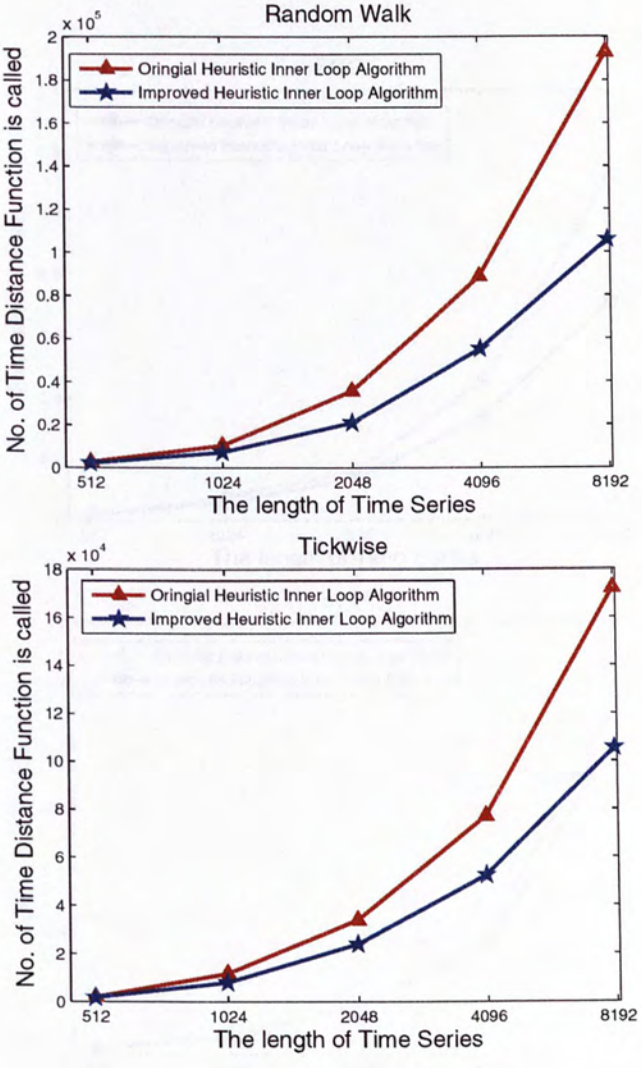
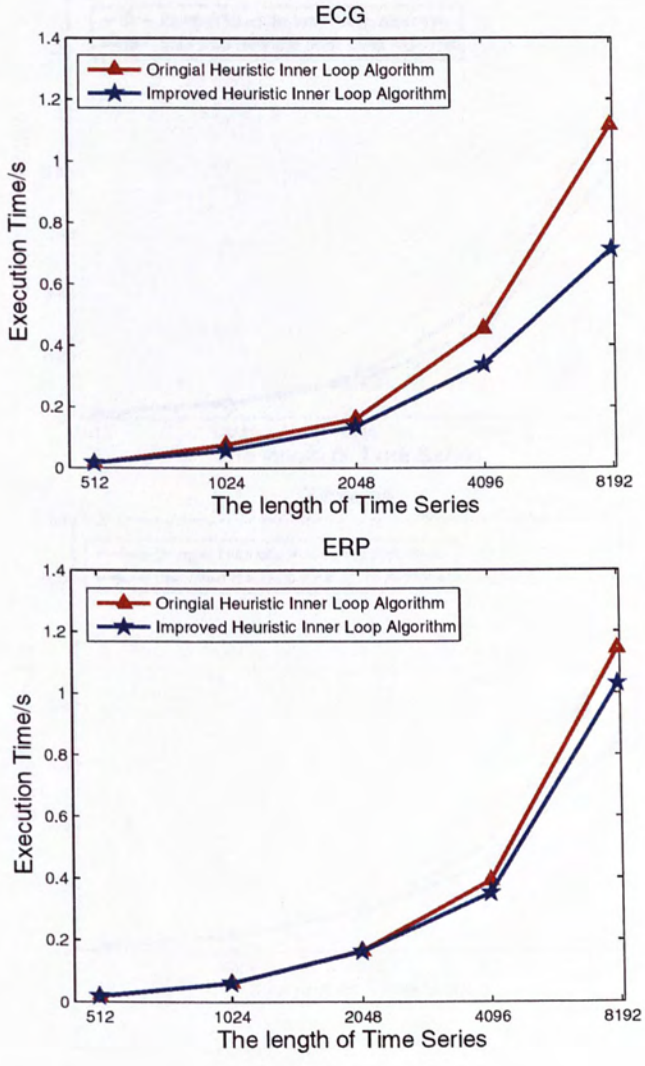


Figure 2.14: Number of times distance function is called by original inner loop heuristic and improved inner loop heuristic


ECG
Oringial Heuristic Inner Loop Algorithm
Improved Heuristic Inner Loop Algorithm
Execution Time/s
The length of Time Series
ERP
Oringial Heuristic Inner Loop Algorithm
Improved Heuristic Inner Loop Algorithm
Execution Time/s
The length of Time Series

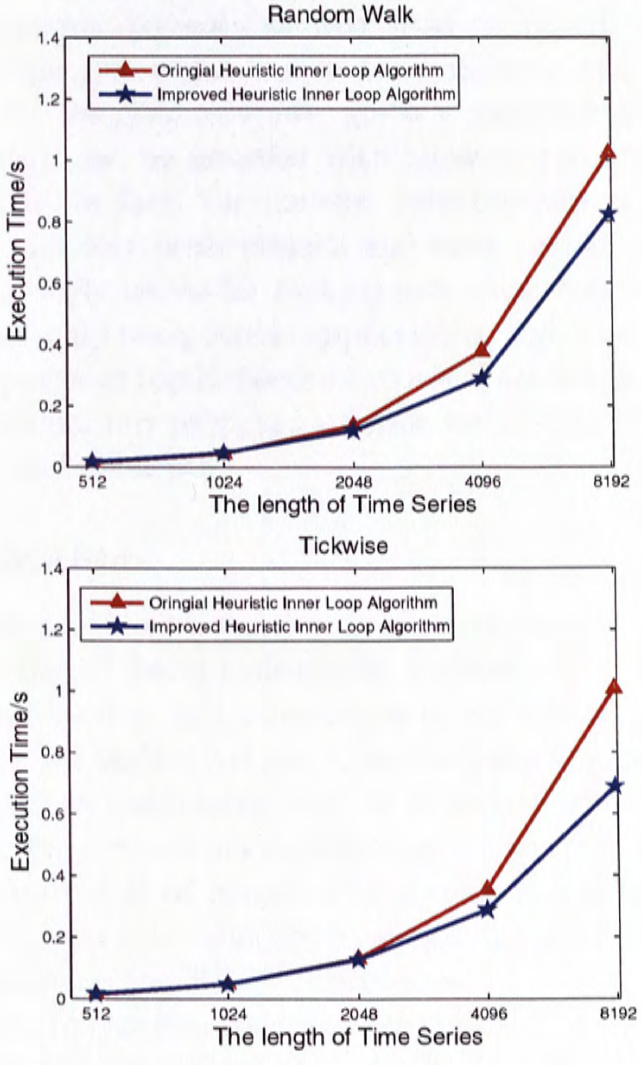


Figure 2.15: Running time of original inner loop heuristic and improved inner loop heuristic

unusual subsequences. As a simple example, given a electrocardiogram of a patient, we may not be satisfied with knowing the most unusual subsequence only, as it may not provide us with enough information to make a diagnosis. In order to make a correct diagnosis, trivially we may need to identify all the unusual subsequences which may be the symptoms of some serious diseases. As another example, given a sales volume per day, again we may not be satisfied with knowing the most unusual subsequence. In fact, the unusual subsequences can probably help us to discover peak periods and slack periods. This data can be extremely useful for making new marketing strategies.

These are only two possible applications. Later we will demonstrate the power of top K discords with real life examples. Before that, we discuss our proposed solution for solving this problem effectively and efficiently.

2.6.2 Algorithm

In fact, our proposed algorithm in previous section can help us to find out the 1^{st} discord effectively. For the 2^{nd}, 3^{rd} discord,etc, however, we need to add some tricks to our original algorithm.

Firstly, after finding a discord, we can remove some candidate subsequences in candidates' list. by definition, there cannot be any overlapping region between the top K discords. That means if a subsequence D of length n beginning at position p is the 1^{st} discord, then subsequences from position p-n+1 to position p+n-1 cannot be the 2^{nd}, 3^{rd} discord, etc.

Secondly, in our proposed algorithm earlier 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.

Based on the above observations, we suggest a new algorithm

for finding top K discords. The pseudo code is shown is Algorithm 6.

Algorithm 6 Finding top K discords

```
for i=1; i ≤ K; i++ do
  discord distance[i] = 0
  discord location [i]= NaN
  //Begin Outer Loop
  for Each p in T ordered by heuristic Outer do
    nearest non-self match distance = table of word[p].nearest so far
    neighbor distance
    if nearest non-self match distance < discord distance[i] then
      continue;
    end if
    //Begin Inner Loop
    for Each unexamined q in T ordered by heuristic Inner do
      if |p − q| ≥ n then
        Dist = Distance Measure(t_p, ...t_{p+n−1}, t_q, ...t_{q+n−1})
        if Dist < nearest non-self match distance then
          nearest non-self match distance = Dist
          table of word[p].nearest so far neighbor = nearest non-self
          match distance
        end if
        if Dist < discord distance[i] then
          break;
        end if
      end if
    end for
    //End For Inner Loop
    if nearest non-self match distance > discord distance[i] then
      discord distance[i] = nearest non-self match distance
      discord location[i] = p
    end if
  end for
  //End for Outer Loop
  remove candidate subsequences(discord location[i], n)
end for

//Return Solution
Return (discord distance, discord location)
```

The basic idea of this algorithm is simple. First we will find the 1^{st} discord, during this process we will record the nearest so far neighbor of each candidate subsequence. The nearest so far neighbor may help us to break the inner loop when we are finding the next discord (The nearest so far neighbor finally must equal to nearest neighbor). And also the inner loop search order of each candidate subsequence is always the same, so we just store the inner loop search order list for each candidate subsequence. Then for every candidate subsequence, we can remember all the examined subsequences in the inner loop and every time we only check the unexamined subsequences. Second, when we know the location of the 1^{st} discord, we will remove the candidate subsequences which cannot be discord anymore from the candidates' list. The algorithm is shown in Algorithm 7.

Algorithm 7 Remove Candidate Subsequences

```
1: start position = discord location - n + 1
2: if start position < 1 then
3:    start position = 1
4: end if
5:
6: end position = discord location + n - 1
7: if start position > N then
8:    start position = N
9: end if
10:
11: for i=start position; i ≤ end position; i++ do
12:    table of word[p].candidate subsequence = false
13: end for
14:
```

At last, we must also modify the outer loop and inner loop heuristic.

Outer Loop Heuristic

For finding the first discord, we use the same procedure shown in section 2.3.4 for approximating the magic outer loop. In fact, we will use the same outer loop ordering for finding the second discord, but the candidate subsequences which overlap the first discord are removed from this outer loop ordering. In other words, subsequences which are not in the candidates' list are removed from the outer loop ordering, as they can not be discord.

The idea is simple. As our proposed outer loop heuristic in section 2.3.4 can effectively separate the discords from the normal subsequences, discords will be mapped to different leaf nodes with single subsequence and different groups of similar normal subsequences will be mapped to different leaf nodes too. It means the further splitting of leaf nodes cannot help us locate the discords, so we will just use the same outer loop heuristic ordering. Since the subsequences which overlap with first discord cannot be second discords, we will remove all these subsequences from the outer loop ordering.

Inner Loop Heuristic

The inner loop heuristic is almost the same as section 2.3.5. The difference is that we will record all the nearest so far distance and the inner loop heuristic search order for each candidate subsequence.

The idea is that in order to break the inner loop, we need to find a subsequence that has a distance to the current examining subsequence in the outer loop less than best so far distance. In fact, we already know the nearest so far neighbor for each subsequence, after finding the first discord. If we use this information for the second discord, the nearest so far distance may be small enough to break the inner loop. Sometimes we may not break the inner loop at once, but we do not need to compare the

current examining subsequence with all the subsequences again.

For example, when the i^{th} candidate subsequence is considered in the outer loop, we look up the word and the the nearest so far neighbor of the i^{th} entry of the array. If the nearest so far neighbor is small enough to break the inner loop, we can break the loop at once. Otherwise, if we already examined the distance between the i^{th} candidate subsequence and the first 10 subsequences in the inner loop heuristic ordering. This time we can continue to search for nearest neighbor starting from the 11^{th} subsequences in the inner loop heuristic. If the distance between the i^{th} candidate subsequence and the 23^{th} subsequence is small enough to break the inner loop, we can record the new nearest so far neighbor and latest searching position of the inner loop heuristic order. In other words, we only examine the unexamined subsequences only.

2.6.3 Experimental Result

In this section, we show that the definition of top K discords is meaningful. We will show this by a real life example. In this example, we tried to find top 2 discords from a dataset that measured the power consumption for a Dutch research facility for the entire year of 1997. Our objective was finding the 2 most unusual weeks. We did not require the week should 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 was longer than the a week, as we needed to ensure that the discord length must long enough to cover the whole week. The result was shown in figure 2.16.

It was not easy to find out the differences between the top 2 discords and the rest of the subsequences by eyes. However, we may find out some interesting patterns, if we compared the top 2 discords with normal subsequences in details. Figure 2.17

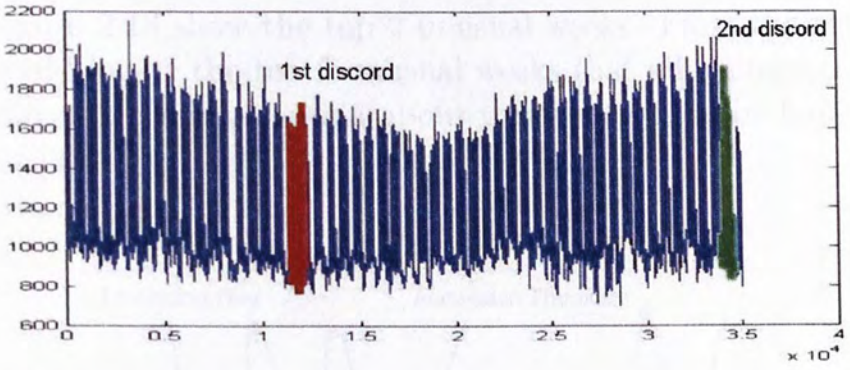


Figure 2.16: The power consumption for a Dutch research facility from 1st January, 1997 to 31st December, 1997

shows the power demands for a normal week, we found that the power consumption from Monday to Friday was relatively high.

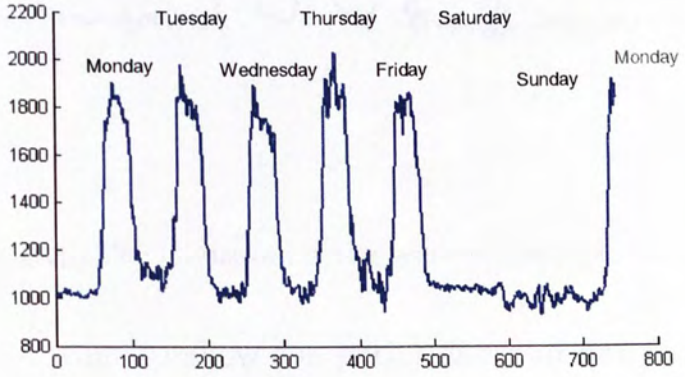


Figure 2.17: The power consumption for a normal week

Figure 2.18 show the top 2 unusual weeks. From the result, we could locate the top 2 unusual weeks that all contained two holidays. More information about this dataset can be found in paper [33].

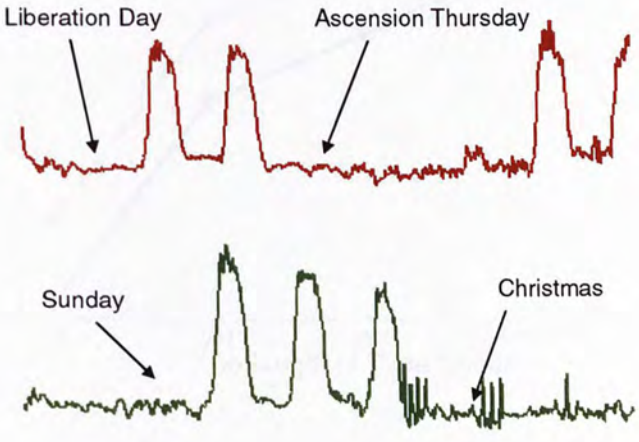
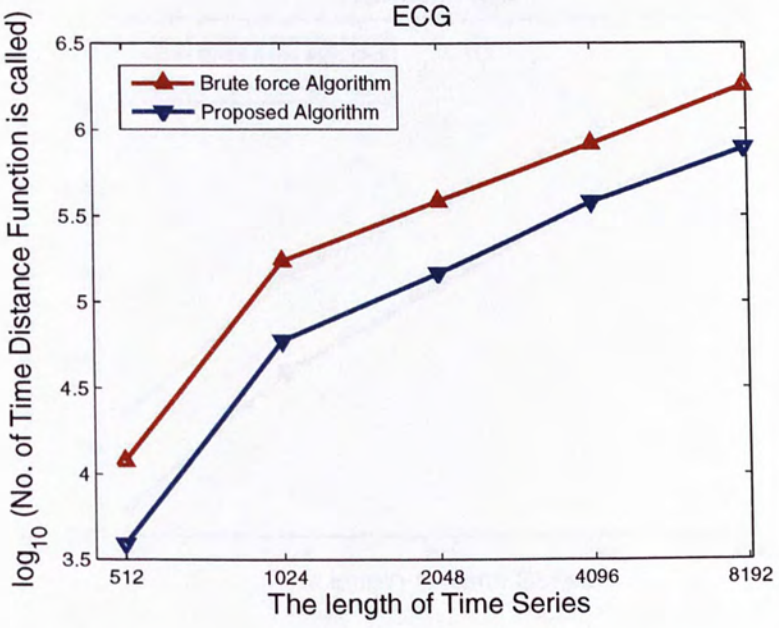


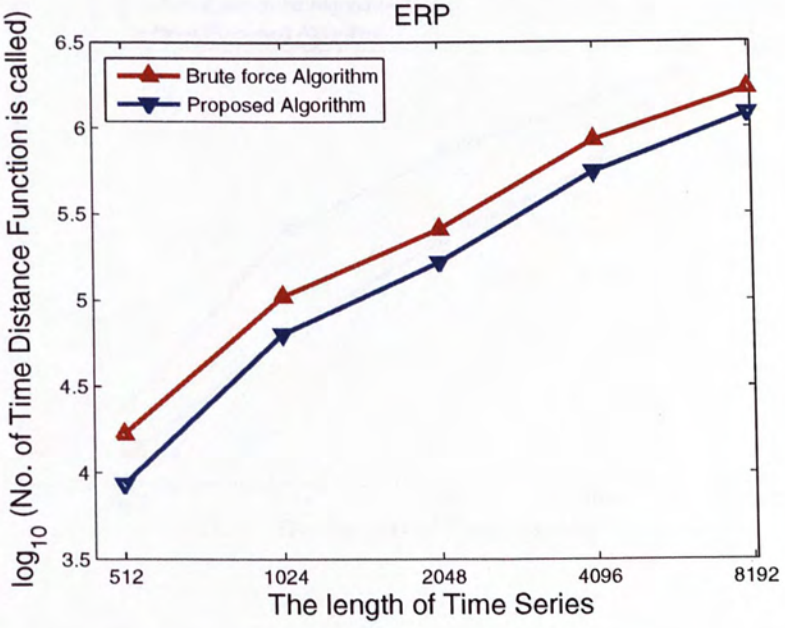
Figure 2.18: (top) The 1st discord of this sequence (bottom) the 2nd discord of this sequence

Then we want to show the performance of our proposed algorithm. As brute force is the only to solve the top K discords problem, we compared brute force algorithm to our proposed algorithm. In this experiment, we continued to find the discords in 4 datasets until there was no potential candidate left in the candidates' list.

2.7 Conclusion

Unusual pattern discovery was discussed in this chapter. We used discords suggested by Keogh et al. as anomaly detectors.


ECG
Brute force Algorithm
Proposed Algorithm
log10 (No. of Time Distance Function is called)
6.5
6
5.5
5
4.5
4
3.5
512
1024
2048
4096
8192
The length of Time Series


ERP
Brute force Algorithm
Proposed Algorithm
log10 (No. of Time Distance Function is called)
6.5
6
5.5
5
4.5
4
3.5
512
1024
2048
4096
8192
The length of Time Series

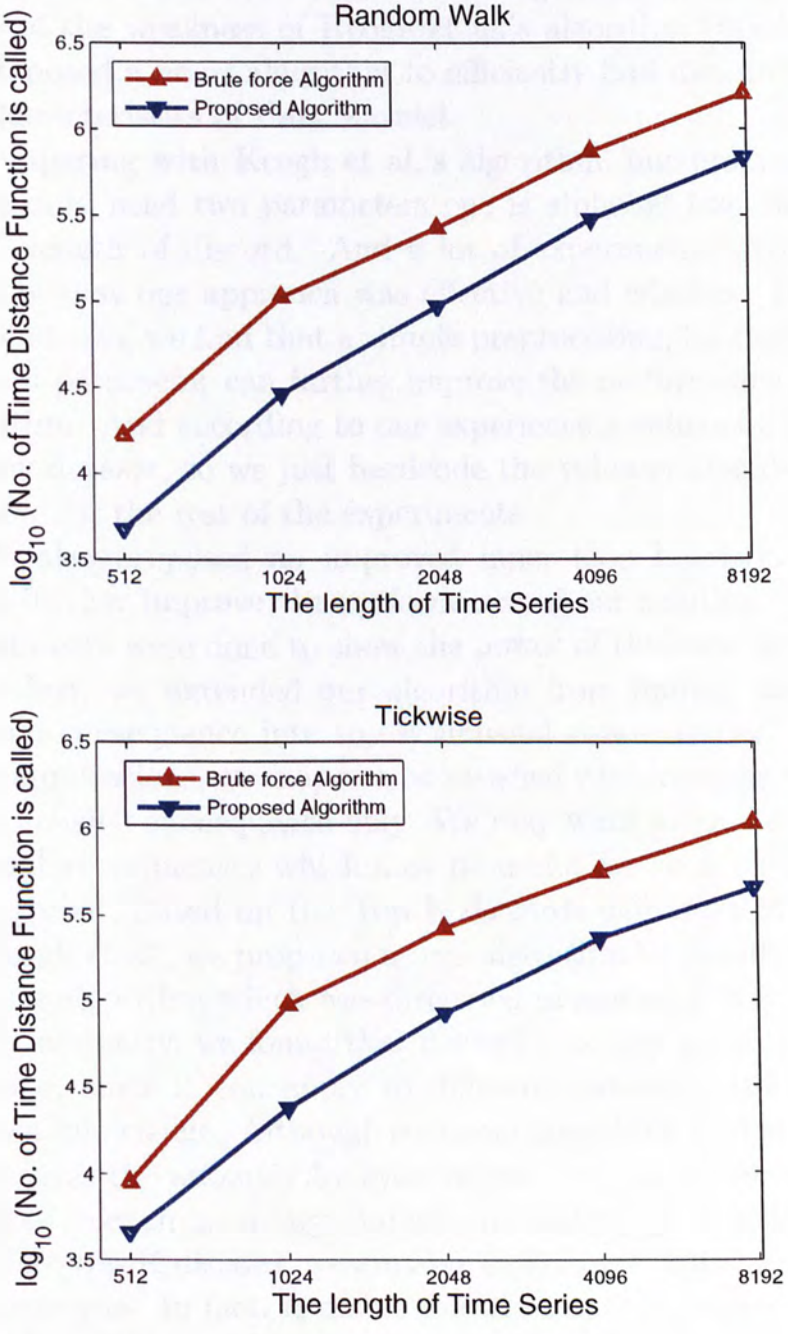


Figure 2.19: Number of times distance function is called by Brute force Algorithm and Our Proposed Algorithm

Then we discussed the Keogh et al.'s algorithm in details. Because of the weakness of Keogh et al.'s algorithm (word size), we proposed a novel algorithm to efficiently find discords using the characteristics of Haar wavelet.

Comparing with Keogh et al.'s algorithm, our proposed solution only need two parameters one is alphabet size, another one is length of discord. And a lot of experiments were done to show that our approach was effective and efficient. For the alphabet size, we find that a simple preprocessing for finding an optimal parameter can further improve the performance of our algorithm. And according to our experience a value of 3 is best for any dataset, so we just hardcode the value of alphabet size to three for the rest of the experiments.

We also proposed an improved inner loop heuristic which could further improve the performance of our solution. Again, experiments were done to show the power of this new heuristic.

At last, we extended our algorithm from finding the most unusual subsequence into top K unusual subsequences. As for most applications, we may not be satisfied with knowing the the most unusual subsequence only. We may want to locate all the unusual subsequences which may be useful for decision making or analysis. Based on the Top K discords definition proposed by Keogh et al., we proposed a new algorithm by modifying the original algorithm which was discussed in section 2.3.

Un summary, we found that discord is a very good anomaly detector, since it can apply to different datasets without any domain knowledge. Although someone may think that it is easy to discover the anomaly by eyes, no one can deny that it is impossible to examine a huge dataset manually. By extending this idea into top K discord, we are able to discover more interesting subsequences. In fact, it makes our algorithm more suitable for solving real life problems.

In this work, we focused on finding unusual time series with

one dimension only. In the future, we want to extend our algorithm for handling time series with more than one dimension. Moreover, we want to extent our algorithm for discovering discords on data streams, as most of the time we need real time anomaly detectors for monitoring different datasets.

□ **End of chapter.**

Chapter 3

Trend Prediction

3.1 Introduction

Time series prediction is not a new topic. People in different areas use different approaches to solve this problem. For example some computer scientists believe that neural network can build up a prediction machine [10] automatically by using massive historical data, stock market analysts believe that moving average [9, 26] can smooth a time series and make it easier to identify trends, and statisticians believe that time series can be modeled with simple equations [32]. In spite of the many previous works, it is still a very difficult problem, especially for financial data such as stock price, bond price and index. It may be because of their high volatility. Some people even suggested that it was impossible to deduce the future from the past. However, people in finance community have shown the power of moving average on trend identification. Unfortunately, they found that using a simple moving average curve sometime may not give them a satisfactory result[12]. We believe that analyzing the basic patterns of a time series can provide us with a lot of information, since data points in a time series must reflect all the underlying generating principal and correlation that exist between data points. In this chapter, we propose an algorithm that applies the previous knowledge about the movement of the stock price

on moving average. And we will show that this algorithm can help develop effective investment strategies.

3.2 Technical Analysis

As we focus on financial data, we first review some commonly used trend prediction methods in the financial community. The methods that make use of the past prices to deduce the future trend are generally called technical analysis. In fact, technical analysis has been used for a long time, and it is still widely used now. In this section, we will introduce several technical analysis tools, which are all easy to use.

3.2.1 Relative Strength Index

Relative Strength Index[2] is a good tool to determine the overbought and oversold position. In technical analysis, overbought means that the price of the asset is overvalued and may experience a pullback. An oversold usually means that the price of the asset is undervalued and the price of the asset my increase later.

Definition 10 *m-relative strength Index (m-RSI)*

$$RSI = 100 - \frac{100}{1 + \frac{average\ of\ m\ days'\ up\ closes}{average\ of\ m\ days'\ down\ closes}} \tag{3.1}$$

The range of RSI is from 0 to 100. If the RSI is smaller than 30, the asset is usually considered as oversold. When the RSI is larger than 70, then asset will be considered as overbought.

3.2.2 Chart Analysis

Some chartists believe that a variety of chart patterns [1, 22] can be good trend indicators. It is because the movement of the

prices can be considered as the change in demand and supply. It means that chart pattern is a concise picture which consolidates the forces of supply and demand, so understanding the chart pattern, we can know the future movement of the prices.

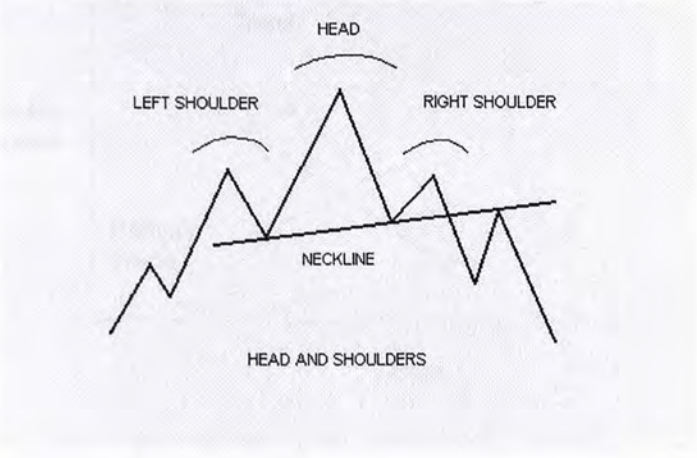


Figure 3.1: The head and shoulders pattern is generally regarded as a reversal pattern

3.2.3 Dow Theory

Dow theory [1] is one of the most famous forecasting methods. It assumes that a stock price reflects everything that is known by general public. It models the stock market in three trends:

- Primary trend: it represents the broad trend of the market. This trend can last for a few months to several years.
- Secondary trend: it runs counter to the primary trend. It can be also treated as the corrective reaction. This trend can last for a few months.
- Third trend: it can view as daily fluctuations. This trend can last for up to a few weeks

Base on this model and a set of guidelines, investors are able to identify the primary trend and react accordingly.

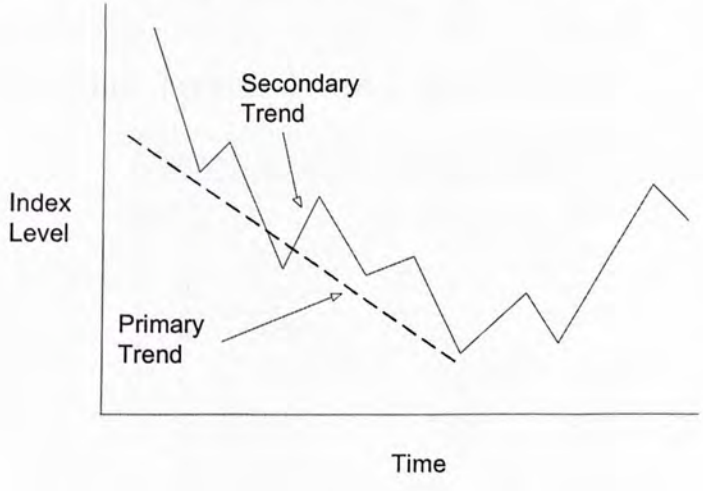


Figure 3.2: In Dow theory, a market can be modeled in three trends. However, the third trends are usually ignored

s

Investors are interested in the primary trends, since they can last for a longer period and easy to identify. On the other hand, investors usually ignore the third trends. Due to the randomness of daily fluctuations, it is difficult to forecast the third trends.

3.2.4 Moving Average

Basic concept of moving average

Moving averages [26, 29, 12] are widely used in the finance community, as they are easy to use. The basic concept is calculating an average of data in a moving sliding window, so the fluctuations are reduced.

Definition 11 *m-moving average: Given a time series* $\{t_1, t_2, t_3, ...t_n\}$,

m-moving average at time q is equals to

$$MA(m,q) = \frac{1}{m}\sum_{i=q}^{q+m-1} t_i \tag{3.2}$$

The procedure for calculation is shown below

Time	Value	4-moving average
1	12.50	-
2	13.40	-
3	11.70	-
4	10.90	12.13
5	15.80	12.95
6	16.80	13.80
7	14.50	14.50
8	19.10	16.55
9	28.10	19.63
10	27.10	22.20
11	7.23	20.38
12	3.13	16.39

For computation convenience, the moving average can be written as a recursion function.

Definition 12 *m-moving average: Given a time series* $\{t_1, t_2, t_3, ...t_n\}$ *and m-moving at time* $q-1$ *is* $MA(m, q-1)$. *The m-moving average at time q is equals to*

$$MA(m,q) = MA(m,q-1) + \frac{1}{m}(t_q - t_{q-m}) \tag{3.3}$$

Some people will call the moving average introduced above the simple moving average. In simple moving average, all the data points have an equal weight. In order to calculate a new moving average, the oldest data point will be discarded and a new data point will be added. Clearly, the discarded data points

have no impact on the current moving average. Because of this disadvantage of simple moving average, exponential moving average was introduced.

Exponential moving average never removed data points in calculation. Instead, it puts more emphasis on recent data. For the less recent data points, they only have a small impact on the moving average. Because of this feature, it can react quicker to recent price changes than a simple moving average and reduce the lag.

Definition 13 *m-exponential moving average: Given a time series* $\{t_1, t_2, t_3, ...t_n\}$ *and m-exponential moving at time* $q-1$ *is* $EMA(m, q-1)$. *The m-exponential moving average at time q is equals to*

$$EMA(m,q) = (1-K)EMA(m,q-1) + Kt_q \tag{3.4}$$

K is equal to $2/(1+m)$ *and* $EMA(m,m) = MA(m,m)$

The procedure for calculation is shown below

Time	Value	Previous 4-exponential moving average	4-exponential moving average
1	12.50	-	-
2	13.40	-	-
3	11.70	-	-
4	10.90	-	12.13
5	15.80	12.13	13.60
6	16.80	13.60	14.88
7	14.50	14.88	14.73
8	19.10	14.73	16.48
9	28.10	16.48	21.13
10	27.10	21.13	23.52
11	7.23	23.52	17.00
12	3.13	17.00	11.45

In exponential moving average. We use the constant weight factor K in calculation and this constant factor only depends on the specified period of the moving average. Imagine that if there is a dramatically change in the price, intuitively we should pay more attention to this change. However, exponential moving average totally ignores the rate of change of the price.

The last type of moving average that we want to introduce is adaptive moving average. It is very similar to exponential moving average, but this time we do not use constant weight factor K. Instead, we update the factor according to the new prices.

Definition 14 *m-adaptive moving average: Given a time series* $\{t_1, t_2, t_3, ...t_n\}$ *and m-adaptive moving at time* $q-1$ *is* $AMA(m, q-1)$. *The m-adaptive moving average at time q is equals to*

$$AMA(m,q) = (1 - k_{m,q})AMA(m, q-1) + k_{m,q}t_q \tag{3.5}$$

$$k_{m,q} = \frac{|t_q - t_{q-m}|}{\sum_{i=q-m+1}^{q} |t_i - t_{i-1}|} \tag{3.6}$$

$AMA(m,m) = MA(m,m)$ *and in order to calculate* $AMA(m,q)$, *we need to calculate* $k_{m,q}$

The procedure for calculation is shown below

We have discussed three different moving average, they are simple moving average, exponential moving average and adaptive moving average. Then we will introduce the usage of moving average in trend identification.

Trend identification using moving average

As past data is used to form moving average, moving average is good for trend identification [26, 3] and trend following. There are many methods for using moving average in trend prediction.

Time	Value	Previous 4-adaptive moving average	k	4-adaptive moving average
1	12.50	-	-	-
2	13.40	-	-	-
3	11.70	-	-	-
4	10.90	-	-	12.13
5	15.80	12.13	0.40	13.59
6	16.80	13.59	0.40	14.89
7	14.50	14.89	0.31	14.77
8	19.10	14.77	0.64	17.54
9	28.10	17.54	0.72	25.23
10	27.10	25.23	0.61	26.37
11	7.23	26.37	0.21	22.33
12	3.13	20.33	0.47	13.30

In this section, we will introduce three simple approaches. The first approach uses the direction of the moving average to determine the trend. In other words, if the price moves in up trend, moving average is also in an up trend. And the price moves in down trend, moving average is also in a down trend. The change in trend can be observed from the turning points (At turning points, the slope of the moving average changes from positive to negative or negative to positive) on the moving average. The turning points could tell us the time for buying and selling. In general, when the slope of the moving average changes from positive to negative, we should sell all the stocks on hand. Otherwise, we should buy the stocks, since an up trend is identified when the slope of the moving average changes from negative to positive.

The second approach uses the location of the price to determine the trend. We will consider the relative position of price to the moving average. When the price is above the moving average, the up trend is identified. Now the market can be viewed in

Figure 3.3: The original time series was smoothed by using 30-moving average. Now the trend of this time series became much easy to observe. And the turning points show us the time for buying and selling the stocks.

bullish behavior, investors can invest in the security. However, when the price is below the moving average, the down trend is identified. The market can be viewed in bearish behavior, it is not a good period for investment.

The third approach uses the relative position of the shorter moving average to the longer moving average. The shorter moving average can show the short term trend of the stock price and the longer moving average can show the long term trend of the stock price. If the shorter moving average is above the longer moving average, it means that comparing with the past, now the stock perform very well. Then an up trend is identified. If the shorter moving average is below the longer moving average, it means that comparing with the past, now the stock perform badly. Then a down trend is identified.

Now, we have introduced three different moving averages and

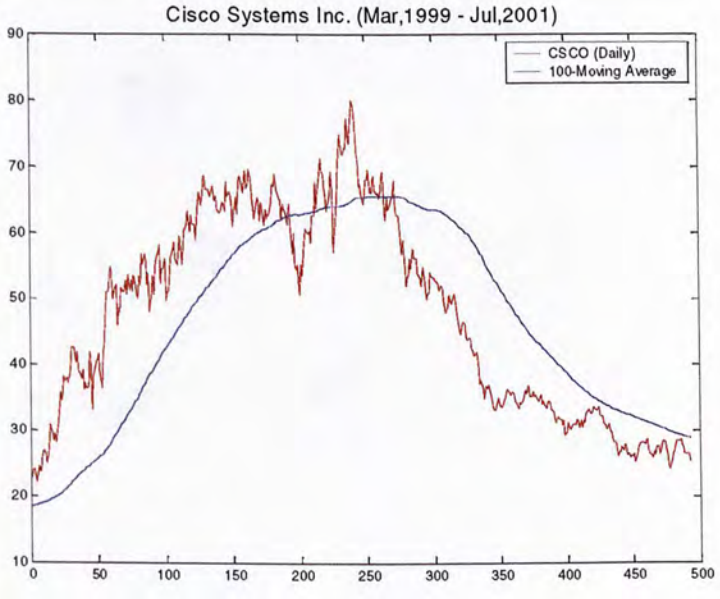


Figure 3.4: The original time series was smoothed by using 100-moving average. When the stock price is above moving average, it is very clear that the stock is in bullish behavior. Then in the second half of moving average, the stock price is below the moving average, so an down trend is identified.

three different methods for trend prediction. For moving averages, both simple moving average and exponential moving average are easy to calculate. Simple moving average cannot capture the chances quickly, but it can generate a smooth curve for analysis. Exponential moving average can capture the chances quickly, but it may be too sensitive and generate false trend signals. Adaptive moving average is good at capturing the chances quickly, but it is not easy to calculate. From these three different trend prediction methods, people in financial community usually suggest it depends on investor's trading and investing style and preferences.

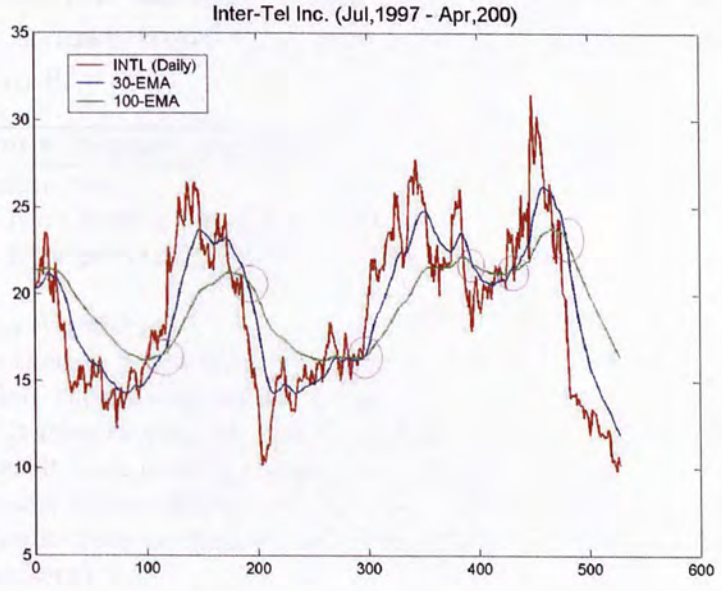


Figure 3.5: Two moving averages were used. One is 30-exponential moving average, another one is 100-exponential moving average. By finding the intersection points by two moving averages, we can identify the up trends and down trends easily.

3.3 Proposed Algorithm

As a moving average is widely used, we want to further improve the performance of an moving average.

Before go into details, we will first give a brief outline about our suggested algorithm. The main idea of our method is that we want to make use of the previous knowledge about the movement of the stock to improve the performance of the moving average. In our algorithm, when there is a new data point, we will update the moving average curve. Then we will check if there any change of trend signal generating by moving average. If yes, we will insert the previous chart pattern into a prediction tree. We then use the current chart pattern and prediction tree to do another prediction. When both moving average and pre-

diction tree give the same answer, we will confirm either an up trend or a down trend is identified. The algorithm is shown in Algorithm 8.

Algorithm 8 Proposed Algorithm

```
1: //Initialization
2: result from moving average = NULL
3: result from prediction tree = NULL
4:
5: //Main Algorithm
6: while there is a new data point do
7:    update the moving average curve
8:    result from moving average = check is there any change in trend
9:    if result from moving average != no change then
10:      insert into prediction tree(previous chart pattern)
11:      result from prediction tree = prediction next trend(current chart
         pattern)
12:      if result from prediction tree == result from moving average then
13:         new trend was identified
14:      end if
15:   end if
16: end while
```

Next, we will discuss using the Piecewise Linear Representation to capture the chart patterns. Then we show the details on inserting chart patterns into the prediction tree and using the prediction tree for trend prediction.

3.3.1 Piecewise Linear Representation

Due to different needs, researchers suggested several representations of time series[25, 16, 27]. We have discussed Symbolic Mapping in previous chapter. In this section, we will focus on the most frequently used representation that is Piecewise Linear Representation.

In Piecewise Linear Representation [37, 36, 5, 17], a time series is approximated by different line segments. Normally, number of line segments will be much less than the number of data

points in the original time series. Because of this observation, this representation makes the storage, transmission and computation of the data more efficient.

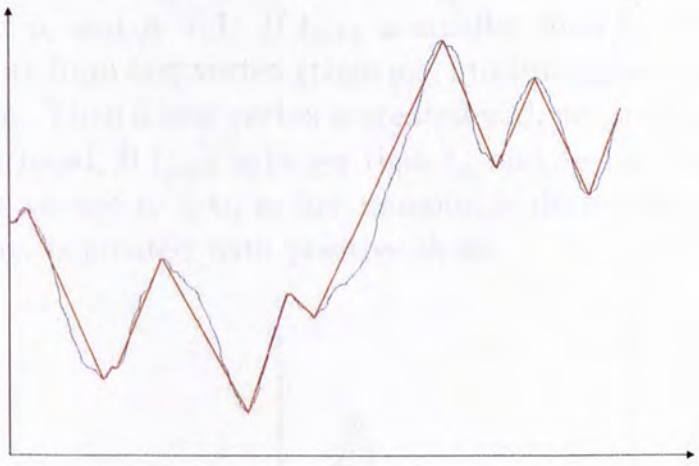


Figure 3.6: Blue line is the original time series. Red line is the Piecewise Linear Representation of the time series.

In this thesis, we propose a new Piecewise Linear Representation method which can represent a time series T in K meaningful line segments. When the slope of the given line segment is positive, it means the time series is in up trend. When the slope of the given line segment is negative, it means the time series is in down trend.

Definition 15 *Vertex: Any K line segments can be represented by a set of vertices. Each vertex is represented by three elements*

$$(p_i, v_i, s_i)$$

p_i is the start time of the $(i+1)^{th}$ line segment and it is also the end time of the i^{th} line segment. v_i is the value of the time series at time p_i. s_i is slope of the $(i+1)^{th}$ line segment, it can be either positive or negative.

Now, we have defined vertex. Then we can introduce our method for finding vertices on an online data stream. First vertex is established at the start of the data stream. Imagine that we are at time $p_i + 1$, we will compare the value of the data stream at p_i and $p_i + 1$. If t_{p_i+1} is smaller than t_{p_i} and all the data points from last vertex (time p_{i-1}) to time p_i are monotonic increasing. Then a new vertex is created with negative slope. On the other hand, If t_{p_i+1} is larger than t_{p_i} and all the data points from last vertex p_{i-1} to p_i are monotonic decreasing. Then a new vertex is created with positive slope.

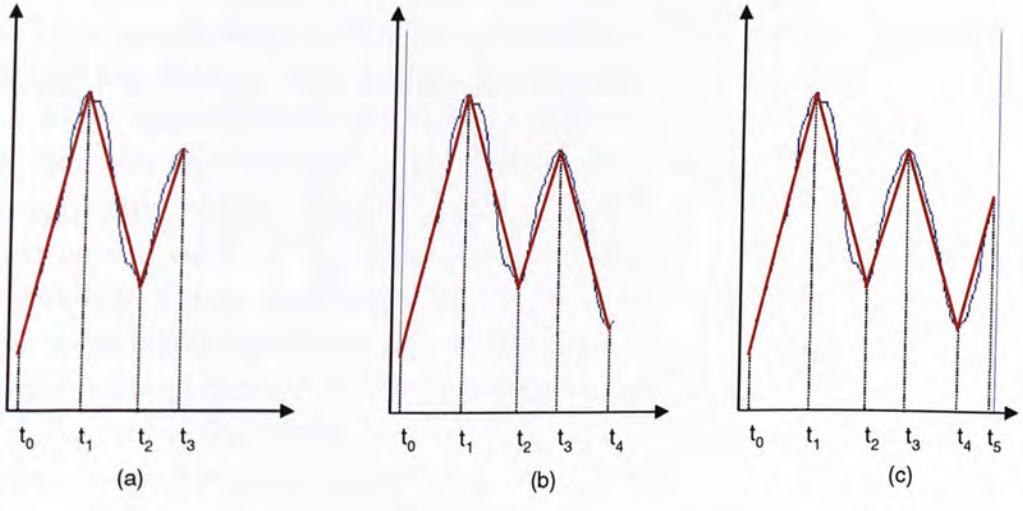


Figure 3.7: Blue line is the original time series. Red line is the Piecewise Linear Representation of the time series. We can continue to add vertices in an online data stream.

3.3.2 Prediction Tree

Using our proposed segmentation algorithm, any time series can be converted into different line segments with zigzag sharp. As we have mentioned before, the line segments can be represented by a set of vertices. In order to study the previous chart pattern

efficiently, the vertices then will be furthered converted into a sequence of symbols using the discretization algorithm in section 2.3.2. After converting vertices into SAX words, we are ready to build a prediction tree.

In our algorithm, we do not only use the original time series to predict the trend, as there are a lot of fluctuations. Instead, we use a moving average, since it can smooth the original time series. It means we build a moving curve on the original time series, then we divide this moving curve into different segments. Each segment can tell use the original time series is moving in up trend or down trend. And in each vertex, we also store the value of the original time series. For example, a vertex created at time p_i will store the value of the moving average v_i at time p_i and the value of the original time series o_i at time p_i.

As we want to make use of k previous vertices to make a prediction, we will extract the k previous vertices (not include the new created vertex which is called as current vertex). Then converting the value of the vertices into a sequence of symbols and insert into a tree. At the leaf nodes, we will keep three counters, one is for counting the number of cases that the the original time series value in current vertex o_i is greater the original time series value in previous vertex o_{i-1} by a user specified threshold f, another one is for counting the number of cases that o_i is smaller than o_{i-1} by a threshold f, the last one is for counting the number of cases which does not fulfill either one of the previous two conditions. This process only occurs when a new vertex is established, so it will not create much computation burden which is one of the big problems for massive stream data management.

In our algorithm, two prediction trees are used, one is for up trend and one is for down trend. If the trend of the previous vertex is up, we will insert the new data into the 'Up Trend Tree'. Otherwise, the new data will be inserted into the 'Down Trend Tree'.

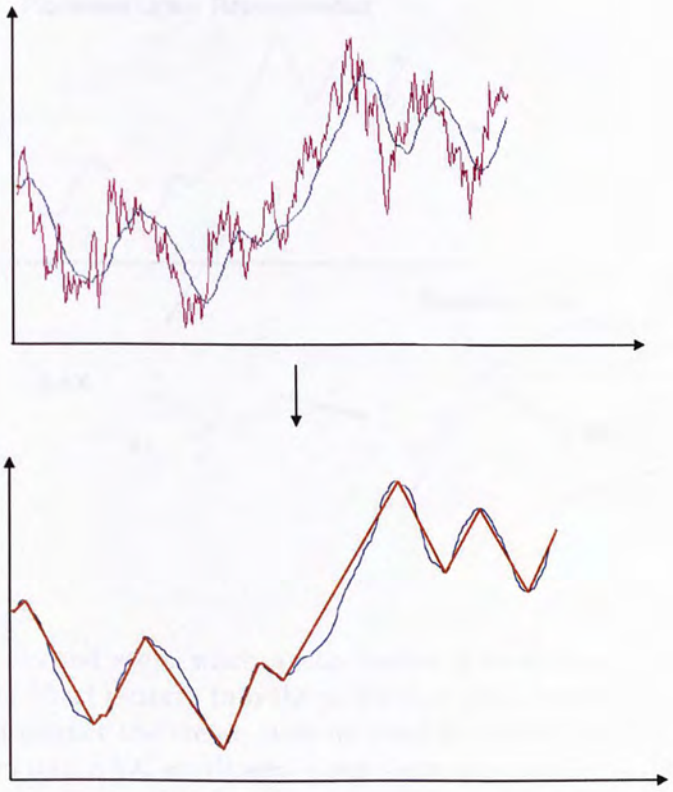


Figure 3.8: Our idea is illustrated in this diagram. First step, we need to generate a moving average for a given time series

3.3.3 Trend Prediction

When a new vertex was established, the current line segment must be either in up trend or down trend. However, it is not enough for financial data stream. The main problem is we do not know how much will the stock price increase. It is difficult to give a precise range for the prediction, but it is important to know if the stock price will decrease or increase by a user specified threshold. This specified threshold can be treated as the user required profit on the investment.

When a new vertex is established, we extract the k-1 previous

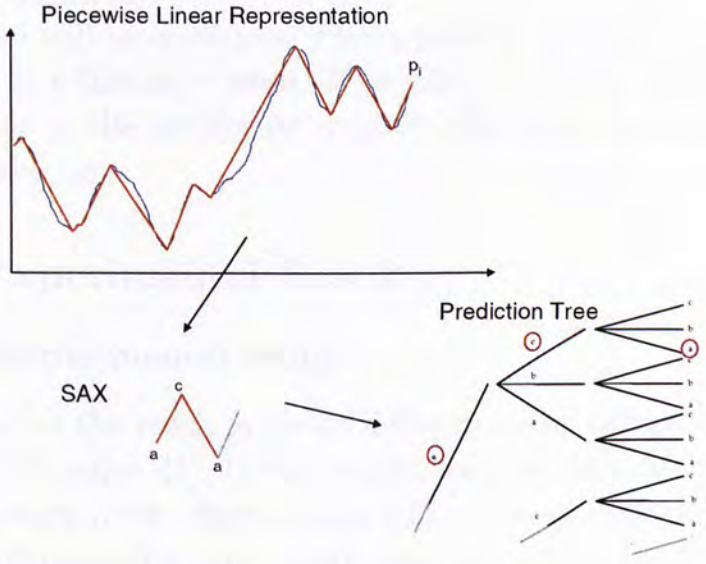


Figure 3.9: Second step, when a new vertex is created at p_i, we need to insert current chart pattern into the prediction tree. Assume we want to use 3 vertices to predict the trend, then we need to extract 3 previous vertices, convert them into SAX words and insert them into predication tree. At the leaf node, we compare the value o_i and value o_{i-1}. If o_i is greater than o_{i-1} by user specified threshold f, we increase the value of 'UP TREND' counter by 1. If o_i is smaller than o_{i-1} by user specified threshold f, we increase the value of 'DOWN TREND' counter by 1. Other, we increase the value of 'NO TREND' counter by 1. In this case, the last vertex is in up trend, so we insert this pattern in 'Up Trend Tree'

vertices and current vertex, convert the values of the vertices into a sequence of symbols. Then we will look up this sequence of symbols in the 'Up Trend Tree', if the trend of the current vertex is up. We will look up this sequence of symbols in the 'Down Trend Tree', if the trend of the current vertex is down. At last, we will compare the value of three counters at the tree node and return the result of the prediction. The final result can be 'UP TREND', 'DOWN TREND' or 'NO TREND', we will choose the one that the counter value is the greatest among

all three counters.

A trend will be confirmed if both moving average and prediction tree give the same result. If we cannot look up the sequence of symbols in the prediction tree, we will trust the result from moving average.

3.4 Experimental Results

3.4.1 Experimental setup

We obtained the stock prices of different listed companies from 'YAHOO Finance' [4]. In our experiment, we only used the adjusted closing price, since closing price is more important comparing with opening price, daily high price and daily low price. We did not use the closing price directly, as we wanted to ignore the fluctuations caused by stock split and stock merge cases.

In stock merge, the current stock price will increase dramatically, so we need to decrease all the data points after the merge point according the to the merge ratio. For example, if two shares of stock will merge into one share of stock, we will decrease all the data points by 50% after the merge point.

In stock split, the current stock price will decrease dramatically, so we need to increase all the data points after the merge point according the to the split ratio. For example, if one shares of stock will split into four share of stock, we will increase all the data points by 400% after the split point.

In our experiments, we may need to set up a prediction tree for our proposed algorithm, so we will use first 1,000 data points to train up a tree. Then the remaining data points will be used to perform our experiments. For the parameters, we fixed the user specified threshold as 10% of the previous price and for the length of moving average, we chose the best length from 20, 30, 50 and 100 by using the training result from the first 1,000 data

points.

And the two experiments are very common in comparing the performance of prediction algorithms [36, 37, 22, 12].

Figure 3.10: (top) Raw data stream (bottom) Adjusted data stream.

3.4.2 Experiment on accuracy

In this experiment, we want to test the prediction accuracy of our algorithm. Both our algorithm and moving average can generate up trend and down trend signals. The signals tell us that the price will move in an up trend or in a down trend. Now we can test the accuracy by using coming trend signal. For example, we receive a down trend signal at time p_x and the stock

price is o_x. The prediction will be correct, if the stock price is smaller than o_x when we receive a new trend signal.

We compared our algorithm with simple moving averages in 5 datasets. The test date range was from the date that the company became a listed company to 31st December, 2005. Our algorithm won in all 5 cases. It suggested that previous knowledge about the movement of the stock price can improve the performance of the moving average.

	Our Proposed Algorithm	Moving Average
ERTS	57.69%	47.88%
COF	56.25%	48.82%
AMGN	47.52%	44.83%
MXIM	51.72%	52.17%
QLGC	57.97%	39.32%
Average	55.55%	48.82%

Table 3.1: Correctness of trend prediction

3.4.3 Experiment on performance

Since we are focusing on financial data, we do not want to develop a trend predication algorithm only. In fact, we want to develop an effective investment strategy. As we have mentioned before, moving averages can be used to determine when we buy the stocks and when we sell the stocks. In this experiment, we want to show that our algorithm is much better than moving averages in term of buy/sell indicators.

In this experiment, we will assume that we have 10,000 dollars on hand. If an up trend is confirmed, we will use all the money on hand to buy the stock and stock price will be equal to the stock price of next day after the up trend is confirmed. And if a down trend is confirmed, we will sell all the stock on hand.

However, we may not sell the stock on the day that the down trend is confirmed. We will sell the stock when the trend is down and the stock price is higher than the previous buying price. At the last data point in our dataset, we will calculate the total value of the asset on hand. It will be equal to the total value of the stock plus the total money on hand.

We compared our algorithm with simple moving averages and 'buy and hold' in 15 datasets. The test date range was from 1st January, 2004 to 31st December, 2005. For 'buy and hold', we use all the money to buy a stock at the beginning and we hold it until the last day. Our algorithm won in most of the cases. It was because our algorithm could successfully avoid making buy/sell decision in a small up trend or down trend.

	Buy and Hold	Moving Average	Our Proposed Algorithm
ATYT	11.91%	1.16%	3.02%
CSCO	-29.38%	-32.80%	-33.14%
HPQ	27.17%	-20.41%	-19.24%
IBM	-8.66%	10.44%	4.08%
MXO	-40.57%	-1.30%	-4.84%
MSFT	7.79%	9.02%	9.83%
ORCL	-7.08%	-10.19%	-17.82%
YHOO	72.51%	79.15%	74.93%
SUNW	23.96%	14.11%	22.18%
CREAF	-14.60%	15.29%	19.26%
ADBE	89.29%	44.26%	36.84%
INTC	-20.83%	-7.47%	-10.46%
DELL	-12.63%	-24.30%	-25.91%
SYMC	0.75%	24.36%	26.45%
TMIC	43.68%	36.14%	65.57%
Average	9.55%	9.16%	10.05%

Table 3.2: Annual rate of return

In fact, it is evident that a stock can go through both trending and trading phases. During a trading phase, the stock price may move in short up trend or short down trend, since the primary trend has not established yet. However, the moving average does not work well in trading period. And our proposed algorithm can also perform well in trading period.

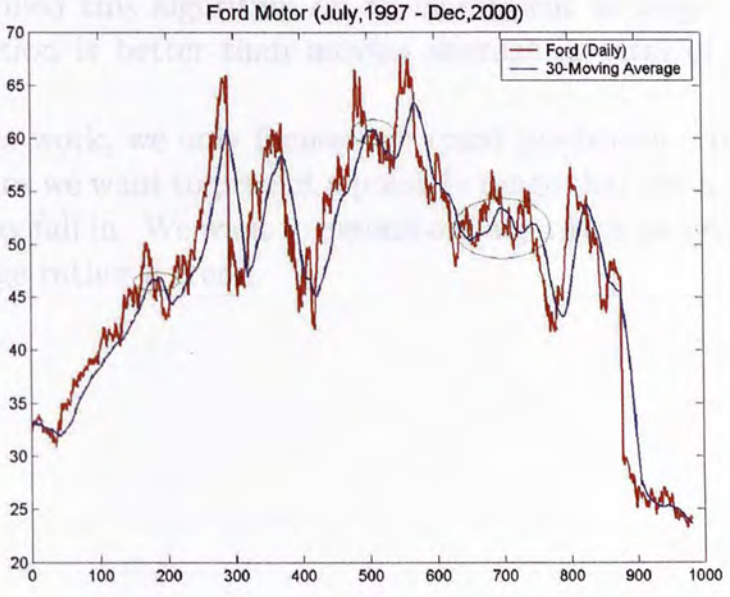


Figure 3.11: The green circles indicate the trading phases.

3.5 Conclusion

Time series trend prediction was discussed in this chapter. We mainly concentrated on financial data such as stock price, index and bond price, as it is nearly a main job for people working in this area. We introduced some common approaches which are widely used in the financial community. Relative strength index, chart analysis, Dow theory and moving average are all widely used, as they are easy to use. Especially moving average,

it can smooth time series and reduce the fluctuations. It can also help the fund manager to determine when he should buy the stock or when he should sell the stock.

We proposed a new algorithm that use the previous moving patterns of a stock to improve the performance of simple moving average. From the experimental results, we showed that our solution was more accurate than the simple moving average. We then applied this algorithm on an investment strategy. Again our solution is better than moving average in term of rate of return.

In this work, we only focused on trend prediction. However, sometimes we want to predict a possible range that the incoming value may fall in. We want to extend our algorithm on prediction of a range rather a trend.

□ **End of chapter.**

Chapter 4

Conclusion

In this thesis, two important problems in time series were addressed. The first one is finding unusual subsequence. According to our definition, the most unusual subsequence D is the one that maximizes the minimum distance between D and any other non-self subsequence E

$$\max_{D}(\min_{E}(Dist(D, E))$$

We proposed a method for solving this problem. We also investigated this problem in different aspects such as the effect of alphabet size and length of discord. The idea of top K discords was introduced in this thesis too. And a lot of experiments were done to prove that our solution is effective and efficient.

The second one is trend prediction. The problem is simple, at every time step we want to know the future data points will move upward or downward. However, the solution is not simple. We proposed an algorithm that make use of the past prices to deduce the future trend. Experiments showed our solution is better than a famous approach, moving average.

We do believe solutions to those two problem are very useful for people who need to handle time series. Imagine, when we need to make decisions by analyzing time series, very often we need to know the future movement of time series and all the existing unusual subsequences. In fact, in this thesis, we showed

a lot of real life applications in different areas. They are useful, as they are core problems of many complicated problems.

□ **End of chapter.**

Bibliography

[1] Chart pattern, http://www.chartpatterns.com.

[2] Investopedia, http://www.investopedia.com.

[3] Stock charts.com, http://www.stockcharts.com.

[4] Yahoo finance, http://finance.yahoo.com.

[5] *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*. IEEE Computer Society, 2004.

[6] B. Chiu, E. Keogh, and S. Lonardi. Probabilistic discovery of time series motifs. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 493–498, New York, NY, USA, 2003. ACM Press.

[7] D. Dasgupta and S. Forrest. Novelty detection in time series data using ideas from immunology, 1995.

[8] D. L. Donoho. Unconditional bases are optimal bases for data compression and for statistical estimation. In *Applied and Computational Harmonic Analysis, 1(1)*, pages 100–115, 1992.

[9] H. Dourra and P. Siy. Investment using technical analysis and fuzzy logic. *Fuzzy Sets Syst.*, 127(2):221–240, 2002.

[10] A. L. Erkki. Time series prediction competition: The cats benchmark.

[11] R. Gesine, S. Sophie, and S. W. Michael. Probabilistic and statistical properties of words: An overview. *Journal of computationnal Biology*, 7(1-2):1–46, 2000.

[12] R. Jiang and K. Szeto. Extraction of investment strategies based on moving average: A gentic algorithm approach. In *CIFEr '03: Proceedings of IEEE International Conference on Computational Intelligence for Financial Engineering*, 2003.

[13] E. Keogh. Exact indexing of dynamic time warping, 2002.

[14] 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.

[15] E. Keogh and T.Folias. The ucr time series data mining archive, http://www.cs.ucr.edu/~eamonn/tsdma/index.html.

[16] E. J. Keogh, S. Chu, D. Hart, and M. J. Pazzani. An online algorithm for segmenting time series. In *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 289–296, Washington, DC, USA, 2001. IEEE Computer Society.

[17] E. J. Keogh, S. Chu, D. Hart, and M. J. Pazzani. An online algorithm for segmenting time series. In *ICDM*, pages 289–296, 2001.

[18] E. J. Keogh and S. Kasetty. On the need for time series data mining benchmarks: A survey and empirical demonstration. *Data Min. Knowl. Discov.*, 7(4):349–371, 2003.

[19] E. J. Keogh, J. Lin, and A. W.-C. Fu. Hot sax: Efficiently finding the most unusual time series subsequence. In *ICDM*, pages 226–233, 2005.

[20] E. J. Keogh, S. Lonardi, and C. A. Ratanamahatana. Towards parameter-free data mining. In *KDD*, pages 206–215, 2004.

[21] E. J. Keogh and C. A. Ratanamahatana. Exact indexing of dynamic time warping. *Knowl. Inf. Syst.*, 7(3):358–386, 2005.

[22] W. Leigh, N. Modani, and R. Hightower. A computational implementation of stock charting: abrupt volume increase as signal for movement in new york stock exchange composite index. *Decis. Support Syst.*, 37(4):515–530, 2004.

[23] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *DMKD '03: Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 2–11, New York, NY, USA, 2003. ACM Press.

[24] 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.

[25] J. Lin, E. J. Keogh, S. Lonardi, and B. Y. chi Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *DMKD*, pages 2–11, 2003.

[26] S. Lofthouse. Investment management.

[27] K. pong Chan and A. W.-C. Fu. Efficient time series matching by wavelets. In *ICDE*, pages 126–133, 1999.

[28] C. A. Ratanamahatana and E. J. Keogh. Making time-series classification more accurate using learned constraints. In *SDM*, 2004.

[29] J. Schwager. Getting started in technical analysis. john wiley and sons inc, 1999.

[30] 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.

[31] V. S.-M. Tseng and C.-H. Lee. Cbs: A new classification method by using sequential patterns. In *SDM*, 2005.

[32] R. van der Weide. Go-garch: a multivariate generalized orthogonal garch model. *Journal of Applied Econometrics*, 17(5):549–564, 2002. available at http://ideas.repec.org/a/jae/japmet/v17y2002i5p549-564.html.

[33] J. J. van Wijk and E. R. van Selow. Cluster and calendar based visualization of time series data. In *INFOVIS*, pages 4–9, 1999.

[34] L. Wei, E. J. Keogh, H. V. Herle, and A. Mafra-Neto. Atomic wedgie: Efficient query filtering for streaming times series. In *ICDM*, pages 490–497, 2005.

[35] L. Wei, N. Kumar, V. Lolla, E. J. Keogh, S. Lonardi, and C. A. Ratanamahatana. Assumption-free anomaly detection in time series. In *SSDBM*, pages 237–240, 2005.

[36] H. Wu, B. Salzberg, G. C. Sharp, S. B. Jiang, H. Shirato, and D. Kaeli. Subsequence matching on structured time

series data. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 682–693, New York, NY, USA, 2005. ACM Press.

[37] H. Wu, B. Salzberg, and D. Zhang. Online event-driven subsequence matching over financial data streams. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 23–34, New York, NY, USA, 2004. ACM Press.

[38] Y. Zhu and D. Shasha. Efficient elastic burst detection in data streams. In *KDD*, pages 336–345, 2003.