HOT SAX: Efficiently Finding the Most Unusual Time Series Subsequence

Eamonn Keogh Jessica Lin Ada Fu

Computer Science & Engineering Department
University of California, Riverside
Riverside, CA 92521
{eamonn, jessica}@cs.ucr.edu

Department of Computer Science and Engineering
The Chinese University of Hong Kong
adafu@cse.cuhk.edu.hk

ABSTRACT

In this work, we introduce the new problem of finding time series *discords*. Time series discords are subsequences of a longer time series that are maximally different to all the rest of the time series subsequences. They thus capture the sense of the most unusual subsequence within a time series. Time series discords have many uses for data mining, including improving the quality of clustering, data cleaning, summarization, and anomaly detection. As we will show, discords are particularly attractive as anomaly detectors because they only require one intuitive parameter (the length of the subsequence) unlike most anomaly detection algorithms that typically require many parameters. We evaluate our work with a comprehensive set of experiments. In particular, we demonstrate the *utility* of discords with objective experiments on domains as diverse as Space Shuttle telemetry monitoring, medicine, surveillance, and industry, and we demonstrate the effectiveness of our discord discovery algorithm with more than one million experiments, on 82 different datasets from diverse domains.

Keywords

Time Series Data Mining, Anomaly Detection, Clustering.

1. INTRODUCTION

The previous decade has seen hundreds of papers on time series similarity search, which is the task of finding a time series that is *most* similar to a particular query sequence [5]. In this work, we pose the new problem of finding the sequence that is *least* similar to all other sequences. We call such sequences time series discords. Figure 1 gives a visual intuition of a time series discord found in a human electrocardiogram.

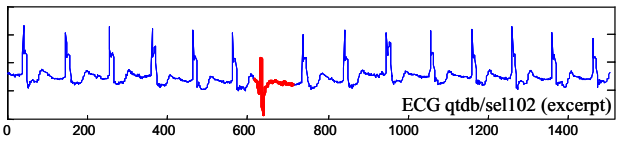

Figure 1: The time series discord found in an excerpt of electrocardiogram qtdb/sel102 (marked in bold line). The location of the discord exactly coincides with a premature ventricular contraction

The fact that the discord in Figure 1 coincides with the location annotated by a cardiologist as containing an anomalous heartbeat hints at one possible use of discords. As we shall show, time series discords are superlative anomaly detectors, able to detect anomalies in domains as diverse as Space Shuttle telemetry, industry, and medicine. One reason why discords are particularly suited for the increasingly important problem of anomaly detection is that they only require a single intuitive parameter, the length of the subsequences to consider. In contrast, many other anomaly detection algorithms require 3 to 7 unintuitive parameters [6]. With so many parameters to set, we need access to huge amounts of training data, even then, avoiding overfitting remains a challenge.

This paper makes two fundamental contributions in discovering unusual time series subsequences. First, while the idea of the "most unusual subsequence" is intuitive, great care must be taken in creating a workable definition, otherwise, we will be plagued with uninteresting pathological solutions. We introduce such a definition here and validate it in domains as diverse as medicine, surveillance, and space telemetry. Second, the brute-force algorithm to discover the most unusual subsequence requires a quadratic "all to all" comparison, which is untenable for large real-world datasets. We introduce a simple algorithm that can achieve 3 to 4 orders of magnitude speedup on real problems.

The rest of the paper is organized as follows. In Section 2, we review related work and discuss some background material before introducing our formal definition of time series discords. In Section 3, we consider the brute-force algorithm for finding discords, and introduce a general framework for speeding up the search based on admissible pruning and reordering the order in which the search examines the subsequences. Section 4 introduces a particular reordering strategy based on examining a symbolic version of the data. We perform an extensive empirical evaluation in Section 5 to demonstrate both the utility of discords and our ability to find them quickly. Finally, Section 6 offers some conclusions and suggestions for future work.

2. RELATED WORK AND BACKGROUND

Our review of related work is exceptionally brief because we are considering a new problem. Most real valued time series problems such as motif discovery [2], longest common subsequence matching, sequence averaging, segmentation, indexing [5], etc. have approximate or exact analogues in the discrete world, and have been addressed by the text processing or bioinformatics communities. However, time series discords do not appear to have a discrete version. Note that the superficially similar sounding Furthest (Sub)String Problem requires us to *build* a string, not to *find* one in the data [9].

2.1 Notation

For concreteness, we begin with a definition of our data type of interest, time series:

Definition 1. *Time Series*: A time series $T = t_1,\ldots,t_m$ is an ordered set of m real-valued variables.

For data mining purposes, we are typically not interested in any of the *global* properties of a time series; rather, we are interested

Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM'05)
1550-4786/05 $20.00 © 2005 IEEE

in *local* subsections of the time series, which are called subsequences.

Definition 2. *Subsequence*: Given a time series T of length m, a subsequence C of T is a sampling of length $n \leq m$ of contiguous position from p, that is, $C = t_p,\ldots,t_{p+n-1}$ for $1 \leq p \leq m - n + 1$.

Since all subsequences may potentially be discords, any algorithm will eventually have to extract all of them; this can be achieved by use of a sliding window.

Definition 3. *Sliding Window*: Given a time series T of length m, and a user-defined subsequence length of n, all possible subsequences can be extracted by sliding a window of size n across T and considering each subsequence C_p.

Since our task is to find the most distant subsequence under some distance measure $Dist(C,M)$, we will take the time to define distance.

Definition 4. *Distance*: *Dist* 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 function *Dist* be symmetric, that is, $Dist(C,M) = Dist(M,C)$.

While the definition of a distance is obvious and intuitive, we need it to exclude *trivial matches*. In general, the best matches to a subsequence (apart from itself) tend to be located one or two points to the left or the right of the subsequence in question. Such matches have previously been called trivial matches [2]. As we shall see, it is critical when finding discords to exclude trivial matches; otherwise, almost all real datasets have degenerate and unintuitive solutions. We will therefore take the time to formally define a non-self match.

Definition 5. *Non-Self Match*: Given a time series T, containing a subsequence C of length n beginning at position p and a matching subsequence M beginning at q, we say that M is a non-self match to C at distance of $Dist(M,C)$ if $|p - q| \geq n$.

We can most easily see the importance of non-self matches for the problem at hand if we consider the analogy of the problem in the discrete world. Consider the following string:

a b c a b c a b c a b c X X X a b c a b c a b a c a b c

The eye is immediately drawn to the subsequence of "**X**", which surely forms the discord here. However, if we assume a sliding window length of 3, and that our distance measure is the hamming distance, then the subsequence that is farthest from its nearest neighbor (i.e. most similar) subsequence is "**bac**". Below, the string is annotated by subscripts that give the distance to the nearest neighbor for each subsequence of length 3:

$$\mathbf{a}_0\mathbf{b}_0\mathbf{c}_0\mathbf{a}_0\mathbf{b}_0\mathbf{c}_0\mathbf{a}_0\mathbf{b}_0\mathbf{c}_0\mathbf{a}_0\mathbf{b}_1\mathbf{c}_1\mathbf{X}_1\mathbf{X}_1\mathbf{X}_1\mathbf{a}_0\mathbf{b}_0\mathbf{c}_0\mathbf{a}_0\mathbf{b}_0\mathbf{c}_0\mathbf{a}_1\mathbf{b}_2\mathbf{a}_1\mathbf{c}_0\mathbf{a}\ \mathbf{b}\ \mathbf{c}$$

This unexpected and unintuitive result is caused by allowing trivial matches. While the subsequence **XXX** may appear unusual, it is only 1 unit distance from the subsequence **XXa**, which shares two elements simply shifted by one place. We can see the difference this makes by annotating the string with the non-self match distance to its nearest neighbor subsequence.

$$\mathbf{a}_0\mathbf{b}_0\mathbf{c}_0\mathbf{a}_0\mathbf{b}_0\mathbf{c}_0\mathbf{a}_0\mathbf{b}_0\mathbf{c}_0\mathbf{a}_0\mathbf{b}_1\mathbf{c}_2\mathbf{X}_3\mathbf{X}_2\mathbf{X}_1\mathbf{a}_0\mathbf{b}_0\mathbf{c}_0\mathbf{a}_0\mathbf{b}_0\mathbf{c}_0\mathbf{a}_1\mathbf{b}_2\mathbf{a}_1\mathbf{c}_0\mathbf{a}\ \mathbf{b}\ \mathbf{c}$$

Here the results are much more intuitive. While it is a simple and contrived example on discrete data, as we shall see, identical remarks apply to real world, real valued data. Note that the idea that one must exclude "partial self" comparisons in order to create meaningful definitions is well known in the bioinformatics community and increasingly understood in the time series data mining community [2][13]. We will therefore use the definition of non-self matches to define time series discords:

Definition 6. *Time Series Discord*: Given a time series T, the subsequence D of length n beginning at position l is said to be the discord of T if D has the largest distance to its nearest non-self match. That is, $\forall$ subsequences C of T, non-self matches M_D of D, and non-self matches M_C of C, $\min(Dist(D, M_D)) > \min(Dist(C, M_C))$

We will denote the location of the discord as $D.l$ and the distance to the nearest non-self matching neighbor as $D.dist$. The length of the discord we denote as D_n.

We may be interested in examining the top K discords, which we define as:

Definition 7. *K^{th} Time Series Discord*: 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, $|p - p_i| \geq n$.

We have deliberately omitted naming a distance function up to this point for generality. For concreteness, we will use the ubiquitous Euclidean distance measure throughout the rest of this paper [2][5].

Definition 8. *Euclidean Distance*: Given two time series Q and C of length n, the Euclidean distance between them is defined as:

$$Dist(Q,C) \equiv \sqrt{\sum_{i=1}^{n}(q_i - c_i)^2}$$

Each time series subsequence is normalized to have mean zero and a standard deviation of one before calling the distance function, because it is well understood that in virtually all settings, it is meaningless to compare time series with different offsets and amplitudes [5].

2.2 Some Properties of Time Series Discords

Here, we discuss some properties of time series discords to enhance the readers' understanding of them and to discount some possible research directions for finding algorithms for quickly locating them.

2.2.1 Discords are not necessary found in sparse space

The idea of considering time series subsequences as points in space has long been exploited by dozens of indexing techniques [5], so one might imagine that such a representation would be useful for the task at hand. We could simply project our time series into n-dimensional space and use existing outlier detection methods [7]. The problem with this idea is the unintuitive fact that discords do not necessarily live in sparse areas of n-dimensional space (Conversely, repeated patterns do not necessarily live in dense parts of the n-dimensional space [2]). We content ourselves here with a visual example and a brief explanation of this observation. In Figure 2, we consider a simple time series consisting of a slightly noisy sine wave. We introduce an "anomaly" of length 50 by shifting the entire second half of the time series.

Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM'05)
1550-4786/05 $20.00 © 2005 IEEE

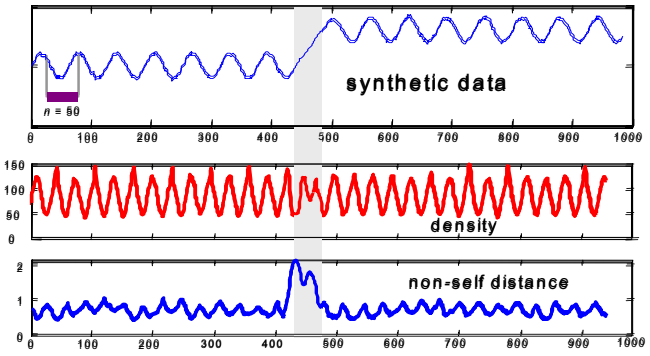

Figure 2: (*Top*) A synthetic time series with an obvious anomaly. (*Middle*) The local density of subsequences of length 50, measured by calculating the number of matching subsequences within a range of 2. (*Bottom*). The non-self match to the nearest neighbor for all subsequences of length 50

We can now extract all subsequences of length 50, project them into 50-dimensional space and measure the local density around each subsequence. Surprisingly, the anomaly is not in the sparsest (or in any other way remarkable) region of space. However, note that the definition of non-self match that is at the heart of time series discords clearly identifies the anomaly.

The explanation of this unintuitive finding harkens back to the idea of trivial matches. Consider a subsequence C located at t_p that is "simple", that is to say it has only one or two features such as peaks or valleys. This simple subsequence is very close in n-dimensional space to the subsequences beginning at t_{p+1}, t_{p-1}, t_{p+2}, etc. In contrast, consider a subsequence M located at t_q that is "complex", that is to say it has many features such as peaks or valleys. This complex subsequence is relatively far from subsequences beginning at t_{q+1}, t_{q-1}, t_{q+2}, etc. In other words, simple (and smooth) shapes appear to be in dense neighborhoods because we over-count shifted versions of them. This problem prevents us from using existing density based algorithms to find time series discords. Note that even if current density based algorithms could be adapted to consider non-self distance, most of them degrade to quadratic time complexity for high dimensionality data.

2.2.2 Discords results are non combinable

Several generic paradigms for solving problems rely on the ability to decompose a problem into smaller sub-problems, which can be solved and admissibly recombined. Depending on the exact definitions, such techniques are variously called dynamic programming, divide and conquer, bottom-up, etc [3]. Unfortunately, as we show below, such ideas are unlikely to help us efficiently find discords.

Imagine that we break a time series T into two sections, A and B, and that we find the discords for both sections, recording their locations as $A.l$, $B.l$ and values as $A.dist$ and $B.dist$, respectively. Furthermore, imagine that we now concatenate A and B to reproduce the original time series T (for simplicity, let us assume that when the discord for T is discovered, it will not span the end of A and the beginning of B). What can we now say about the discord for T? Surprisingly, the answer is very little. We cannot assume that it will be either in location $A.l$ or in location $|A| + B.l$, because both of the two previously discovered discords may have good matches in the other section. All we can do is give weak bounds. The value of $T.dist$ is at most max($A.dist$, $B.dist$). The lower bound of $T.dist$ is a trivial zero (To see this, imagine $A = B$). As to the location of $T.l$, we can say nothing.

If we consider the complementary situation, where we know the discord information $T.l$ and $T.dist$ for T, and we split into two new time series A and B, we are similarly frustrated. Assume that the discord from T happened to fall into A. We can lower bound $A.dist$ as being greater than or equal to $T.dist$, but we cannot provide an upper bound. In addition, we can say nothing about the location of $A.l$. As for $B.dist$ and $B.l$, we can say nothing.

The above results suggest that existing algorithms/paradigms are of little utility for finding discords. This motivates the introduction of an original algorithm in the next section.

3. FINDING TIME SERIES DISCORDS

The brute force algorithm for finding discords is simple and obvious. We simply take each possible subsequence and find the distance to the nearest non-self match. The subsequence that has the greatest such value is the discord. This is achieved with nested loops, where the outer loop considers each possible candidate subsequence, and the inner loop is a linear scan to identify the candidate's nearest non-self match. For clarity, the pseudo code is shown in Table 1.

Table 1: Brute Force Discord Discovery.

```
1  Function [dist, loc ]= Brute_Force(T, n)
2   best_so_far_dist = 0
3   best_so_far_loc = NaN
4
5  For p = 1 to |T| - n  + 1                          // Begin Outer Loop
6    nearest_neighbor_dist = infinity
7    For q = 1 to |T| - n  + 1                        // Begin Inner Loop
8      IF | p - q | ≥ n                               // non-self match?
9        IF Dist(tp,...,tp+n-1, tq,...,tq+n-1) < nearest_neighbor_dist
10         nearest_neighbor_dist = Dist(tp,...,tp+n-1, tq,...,tq+n-1)
11       End
12     End                                            // End non-self match test
13   End                                              // End Inner Loop
14   IF nearest_neighbor_dist > best_so_far_dist
15     best_so_far_dist = nearest_neighbor_dist
16     best_so_far_loc  = p
17   End
18 End                                                // End Outer Loop
19 Return[ best_so_far_dist, best_so_far_loc ]
```

Note that the algorithm requires exactly one parameter, the length of subsequences to consider. The algorithm is easy to implement and produces exact results. However, it has one fatal flaw for data mining. It has $O(m^2)$ time complexity which is simply untenable for even moderately large datasets.

The following two observations offer hope to improve the algorithm's running time.

Observation 1: In the inner loop, we don't actually need to find the true nearest neighbor to the current candidate. As soon as we find any subsequence that is closer to the current candidate than the best_so_far_dist, we can abandon that instance of the inner loop, safe in the knowledge that the current candidate could not be the time series discord.

Observation 2: The utility of the above optimization depends on the order which the outer loop considers the candidates for the discord, and the order which the inner loop visits the other subsequences in its attempt to find a sequence that will allow an early abandon of the inner loop.

While these are simple ideas and only minor modifications of the original algorithm, for concreteness, we will make them clear. The pseudo code is shown in Table 2.

Note that the input has been augmented by two heuristics, one to determine the order in which the outer loop visits the subsequences, and one to determine the order in which the inner loop visits the subsequences. Note that the heuristic for the outer loop is used once, but the heuristic for the inner loop takes the current candidate into account, and is thus invoked to produce a new ordering for every iteration of the outer loop.

We have now reduced the discord discovery problem into a generic framework where all one needs to do is to specify the heuristics. Note that we should not attempt to "cheat" the algorithm. We could provide very good heuristic orderings if we are allowed to completely solve the brute force problem each time the heuristic functions are invoked! However this is simply hiding the time complexly in a different part of the implementation. We must therefore insist that the *Outer* heuristic (invoked only once) takes at most O(m) to calculate and the *Inner* heuristic (invoked m-n times) takes O(1). Note that this requirement precludes the possibility of using R-trees, K-d trees or other classic indexing algorithms [5].

Table 2: Heuristic Discord Discovery.

```
1  Function [dist, loc ]= Heuristic_Search(T, n, Outer, Inner)
2   best_so_far_dist = 0
3   best_so_far_loc = NaN
4
5   For Each p in T ordered by heuristic Outer     // Begin Outer Loop
6      nearest_neighbor_dist = infinity
7      For Each q in T ordered by heuristic Inner // Begin Inner Loop
8         IF | p - q | ≥ n                        // non-self match?
9            IF Dist(t_p,...,t_p+n-1, t_q,...,t_q+n-1) < best_so_far_dist
10              Break                            // Break out of Inner Loop
11           End
12           IF Dist(t_p,...,t_p+n-1, t_q,...,t_q+n-1) < nearest_neighbor_dist
13              nearest_neighbor_dist = Dist(t_p,...,t_p+n-1, t_q,...,t_q+n-1)
14           End
15        End                                    // End non-self match test
16     End                                       // End Inner Loop
17     IF nearest_neighbor_dist > best_so_far_dist
18        best_so_far_dist = nearest_neighbor_dist
19        best_so_far_loc  = p
20     End
21  End                                          // End Outer Loop
22  Return[ best_so_far_dist, best_so_far_loc ]
```

To gain some intuition into our new algorithm, and to hint at our eventual solution to this problem, let us consider 3 possible heuristic strategies:

Random: We could simply have both the *Outer* and *Inner* heuristics randomly order the subsequences to consider. It is difficult to analyze this strategy since its performance is bounded from below by O(m) and from above by O(m^2) (see below for explanation) and depends on the data. However, empirically it works reasonably well. The conditional test on line 9 of Table 2 is often true and the inner loop can be abandoned early, considerably speeding up the algorithm.

Magic: In this hypothetical situation, we imagine that a friendly oracle gives us the best possible orderings. These are as follows: For *Outer*, the subsequences are sorted by descending order of the non-self distance to their nearest neighbor, so that the true discord is the first object examined. For *Inner*, the subsequences are sorted in ascending order of distance to the current candidate. For the **Magic** heuristics, the first invocation of the inner loop will run to completion. Thereafter, all subsequent invocations of the inner loop will be abandoned during the very first iteration. The time complexity is thus one occurrence of m-n+1 steps for the first inner loop, and m-n occurrences of the O(1) step of each subsequent invocation of the inner loop, giving a total time complexity of O(m) + O(m) or just O(m). Note that we have $m \gg n$.

Perverse: In this hypothetical situation, we imagine that a less than friendly oracle gives us the worst possible orderings. These are identical to the **Magic** orderings with ascending/descending orderings reversed. In this case, we are back to the original O(m^2) time complexity, and we waste some time in the conditional tests on line 9 of Table 2.

These results are something of a mixed bag for us. They suggest that a linear time algorithm is possible, but only with the aid of some very wishful thinking. The **Magic** heuristic requires a perfect ordering of subsequences in the inner loop, and any perfect ordering (i.e., sorting) requires at least O($m$log$m$). Furthermore, the only known way to produce the perfect ordering of subsequences in the outer loop requires O(m^2) work, but we are only allowed O(m) time. The following two observations, however, offer us some hope for a fast algorithm:

Observation 3: In the outer loop, we do not actually need to achieve a perfect ordering to achieve dramatic speedup. All we really require is that among the first few subsequences being examined, we have at least one that has a large distance to its nearest neighbor. This will give the best_so_far_dist variable a large value early on, which will make the conditional test on line 9 of Table 2 be true more often, thus allowing more early terminations of the inner loop.

Observation 4: In the inner loop, we also do not actually need to achieve a perfect ordering to achieve dramatic speedup. All we really require is that among the first few subsequences being examined we have at least one that has a distance to the candidate sequence being considered that is less than the current value of the best_so_far_dist variable. This is a sufficient condition to allow early termination of the inner loop.

We can imagine a full spectrum of algorithms, which only differ by how well they order subsequences relative to the **Magic** ordering. This spectrum spans {**Perverse…Random…Magic**}. Our goal then is to find the best possible approximation to the **Magic** ordering, which is the topic of the next section.

At the risk of redundancy, we again emphasize that this search problem requires a specialized solution, and we cannot leverage off the huge literature on time series similarity search [5]. Kd-Trees, R-trees and their many variants require O(log(m)) time per lookup, but we can spare only O(1) time. In any case, these search algorithms support *nearest* neighbor search, whereas all we require here is "*near-enough*" neighbor search, as noted in observation 4.

4. APPROXIMATIONS TO MAGIC

Before we introduce our techniques for approximating the perfect ordering returned by the hypothetical **Magic** heuristics, we must briefly review the **S**ymbolic **A**ggregate Appro**X**imation (SAX) representation of time series introduced in [10]. While there are at least 200 different symbolic approximation of time series in the literature, SAX is unique in that it is the only one that allows both dimensionality reduction and lower bounding of Lp norms. Since its relatively recent introduction, SAX has become an important tool in the time series data mining toolbox. It has been used to find time series motifs [2], to mine rules in health data, for anomaly detection [6], to extract features from a

Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM'05)
1550-4786/05 $20.00 © 2005 IEEE

hepatitis database, for visualization [8][11], and a host of other data mining tasks.

4.1 A Brief Review of SAX

A time series C of length n can be represented in a w-dimensional space by a vector $\overline{C} = \overline{c}_1, \ldots, \overline{c}_w$. The i^{th} element of $\overline{C}$ is calculated by the following equation:

$$\overline{c}_i = \frac{w}{n} \sum_{j=\frac{n}{w}(i-1)+1}^{\frac{n}{w}i} c_j$$

In other words, to transform the time series from n dimensions to w dimensions, the data is divided into w equal sized "frames." The mean value of the data falling within a frame is calculated and a vector of these values becomes the dimensionality-reduced representation. This simple representation is known as Piecewise Aggregate Approximation (PAA).

Having transformed a time series into the PAA representation, we can apply a further transformation to obtain a discrete representation. It is desirable to have a discretization technique that will produce symbols with equiprobability [2][6]. In empirical tests on more than 50 datasets, we noted that normalized subsequences have highly Gaussian distribution [10], so we can simply determine the "breakpoints" that will produce equal-sized areas under Gaussian curve.

Definition 9. *Breakpoints*: Breakpoints are a sorted list of numbers $B = \beta_1, \ldots, \beta_{a-1}$ such that the 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 may be determined by looking them up in a statistical table. For example, Table 3 gives the breakpoints for values of a from 3 to 5.

Table 3: A lookup table that contains the breakpoints that divides a Gaussian distribution in an arbitrary number (from 3 to 5) of equiprobable regions

βi \ a	3	4	5
β_1	-0.43	-0.67	-0.84
β_2	0.43	0	-0.25
β_3		0.67	0.25
β_4			0.84

Once the breakpoints have been obtained we can discretize a time series in the following manner. We first obtain a PAA of the time series. All PAA coefficients that are below the smallest breakpoint are mapped to the symbol "**a**", all coefficients greater than or equal to the smallest breakpoint and less than the second smallest breakpoint are mapped to the symbol "**b**", etc. Figure 3 illustrates the idea.

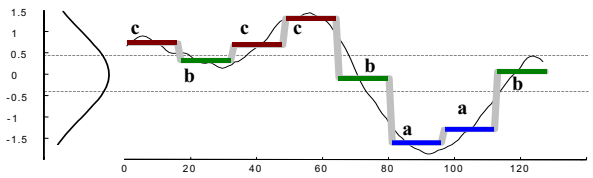

Figure 3: A time series (thin black line) is discretized by first obtaining a PAA approximation (heavy gray line) and then using predetermined breakpoints to map the PAA coefficients into symbols (bold letters). In the example above, with n = 128, w = 8 and a = 3, the time series is mapped to the word cbccbaab

Note that in this example, the 3 symbols, "**a**", "**b**" and "**c**" are approximately equiprobable as we desired. We call the concatenation of symbols that represent a subsequence a *word*.

Definition 10. *Word*: A subsequence C of length n can be represented as a *word* $\hat{C} = \hat{c}_1, \ldots, \hat{c}_w$ as follows. Let α_j denote the j^{th} element of the alphabet, i.e., α_1 = **a** and α_2 = **b**. Then the mapping from a PAA approximation $\overline{C}$ to a word $\hat{C}$ is obtained as follows:

$$\hat{c}_i = \alpha_j \quad iff \quad \beta_{j-1} \le \overline{c}_i < \beta_j$$

We have now completely defined SAX representation. Note that our observation that normalized subsequences have highly Gaussian distribution [10], is not critical to *correctness* of any of the algorithms that use SAX, including the ones in this work. A pathological dataset that violates this assumption will only affect the *efficiency* of the algorithms.

4.2 Approximating the Magic *Outer* Loop

We begin by creating two data structures to support our heuristics. We are given n, the length of the discords in advance, and we must choose two parameters, the cardinality of the SAX alphabet size α, and the SAX word size w. We defer a discussion of how to set these parameters until later in this section, but note that the values of α and w only affect the efficiency of our algorithm, not the final result, which depends *only* on the user supplied length of the discord.

We begin by creating a SAX representation of the entire time series, by sliding a window of length n across time series T, extracting subsequences, converting them to SAX words, and placing them in an array where the index refers back to the original sequence. Figure 4 gives a visual intuition of this, where both α and w are set to 3.

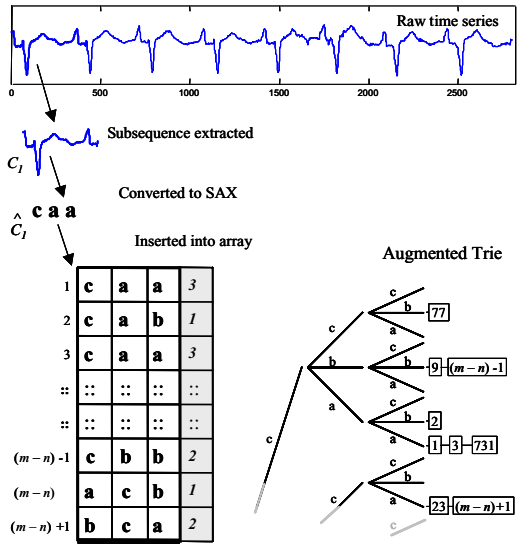

Figure 4: The two data structures used to support the *Inner* and *Outer* heuristics. (*left*) An array of SAX words, where the last column contains a count of how often each word occurs in the array. (*right*) An excerpt of an trie with leaves that contain a list of all array indices that map to that terminal node

Note that the index goes from 1 to $(m - n) + 1$, because the right edge of n-length sliding window "bumps" against end of the m-length time series.

Once we have this ordered list of SAX words, we can imbed them into an augmented trie where the leaf nodes contain a linked list index of all word occurrences that map there. The

Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM'05)
1550-4786/05 $20.00 © 2005 IEEE

count of the number of occurrences of each word can be mapped back to the rightmost column of the array. For example, in Figure 4, if we are interested in the word **caa,** we visit the trie to discover that it occurs in locations 1, 3, and 731. If we are interested in the word that occurs at a particular location, let's say $(m - n)$ - 1, we can visit that index in the array and discover that the word **cbb** is mapped there. Furthermore, we can see by examining the rightmost column that there are a total of 2 occurrences of that particular word (including the one we are currently visiting). However, if we want to know the location of the other occurrence, we must visit the trie.

Surprisingly, both data structures can be created in time and space linear in the length of T [1]. In fact, if we take advantage of the fact that we only need $\lceil \log_2(\alpha) \rceil$ bits for each SAX symbol, then both data structures are significantly smaller than the raw time series data they were derived from.

We can now state our *Outer* heuristic; we scan the rightmost column of the array to find the smallest count *mincount* (its value is virtually always 1). The indices of all SAX words that occur *mincount* times are recorded, and are given to the outer loop to search over first. After the outer loop has exhausted this set, the rest of the candidates are visited in random order.

The intuition behind our *Outer* heuristic is simple. Unusual subsequences are very likely to map to unique or rare SAX words. By considering the candidate sequences that map to unique or rare SAX words early in the outer loop, we have an excellent chance of giving a large value to the best_so_far_dist variable early on, which (as noted in observation 3) will make the conditional test on line 9 of Table 2 be true more often, thus allowing more early terminations of the inner loop.

4.3 Approximating the Magic *Inner* Loop

Our *Inner* heuristic also leverages off the two data structures shown in Figure 4. When candidate i is first considered in the outer loop, we look up the SAX word that it maps to, by examining the i^{th} word in the array. We then visit the trie and order the first items in the inner loop in the order of the elements in the linked list index found at the terminal nodes. For example, imagine we are working on the problem shown in Figure 4. If we were examining the candidate C_{731} in the outer loop, we would visit the array at location 731. Here we would find the SAX word **caa**. We could use the SAX values to traverse the trie to discover that subsequences 1, 3, 731 map here. These 3 subsequences are visited first in the inner loop (note that line 8 of Table 1 prevents 731 from being compared to itself). After this step, the rest of the subsequences are visited in random order.

Because our algorithm works by using heuristics to order SAX sequences, we call it HOT SAX, short for **H**euristically **O**rdered **T**ime series using **S**ymbolic **A**ggregate Appro**X**imation

4.4 Minor Optimizations and Parameter Setting

There are several minor optimizations we can apply to the heuristic search algorithm. For example, imagine we are considering candidate C_i in the outer loop, and as we traverse through the inner loop, we find that subsequence C_j is close enough to it to allow early abandonment. In addition to saving time with the early termination, we can also delete C_j from the list of candidates in outer loop (if it has not already been visited). The key observation is that since we are assuming a symmetric distance measure, if nearness to C_j disqualifies candidate C_i from being the discord, then the same nearness to C_i would also disqualify candidate C_j from being the discord. Empirically, this simple optimization gives a speed-up factor of approximately 2. In addition, there are several well-known optimizations to the Euclidean distance that we can use [5].

As noted above, we must choose two parameters, the cardinality of the SAX alphabet size α, and the SAX word size w. Recall what it is we want to optimize. We would like the distribution of the SAX words to be highly skewed, so that the discord will map to a SAX word that is unique or rare, and all the other subsequences will map to SAX words that are very frequent. This is the best situation for both our heuristics. If we choose very large values of α and/or w, almost all subsequences will map to unique words; if we choose very small values of α and/or w, all subsequences will map to just a small handful of words. Either of these situations is bad for our heuristics.

The good news is that there is little freedom for the a parameter; extensive experiments carried out by the current authors [2] [8][10][11] and dozens of other researchers worldwide [13] suggest that a value of either 3 or 4 is best for virtually any task on any dataset. After empirically confirming this on the current problem with experiments on more than 50 datasets, we will simply hardcode $\alpha = 3$ for the rest of this work. Having fixed α, we performed an exhaustive empirical examination of the role of the w parameter. The best value for this parameter depends on the data. In general, relatively smooth and slowly changing datasets favor a smaller value of w, whereas more complex time series favor a larger value of w. The following observations mitigate the problem of parameter setting:

- The speedup does not critically depend on w parameter. After empirically finding the best value on a particular data we found we could vary the value of w in the range of 60% to 150% with less than a 12% decrease in speedup.
- Once we learn a good setting on a particular data type, say ECGs, that setting will also work well on other datasets of the same type (assuming the sampling rate is the same).

5. EMPIRICAL EVAUALTION

We begin by showing the utility of time series discords for a host of domains, then go on to show that our algorithm is able to find discords very efficiently.

5.1 The Utility of Time Series Discords

In this paper, we will only demonstrate the utility of discords as anomaly detectors. We have done extensive successful experiments in other tasks, such as improving the quality of clustering and summarization; however, anomaly detection is unique in that it allows immediate and intuitive visual confirmation. The additional experiments for other tasks, together with many extra anomaly detection experiments can be found here [4]. We encourage the interested reader to consult this site for additional examples and larger and more detailed figures of the experiments show below.

After much reflection, we have decided *not* to include comparisons to other approaches here[1]. We simply could not make the other approaches work well, and we do not wish to seem to be badgering fellow researchers. To be fair to the other approaches, it is very difficult to make meaningful comparisons between our method, which requires only one intuitive parameter, and some of the rival methods that require 3 to 7 parameters [6], including some parameters for which we may have poor intuition, such as *Embedding dimension*, *Kernel function, SOM topology* or *number of Parzen windows*.

[1] We *have* conducted such experiments and we have made them available here [4].

Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM'05)
1550-4786/05 $20.00 © 2005 IEEE

5.1.1 Anomaly Detection in Space Telemetry

We consider the problem of finding anomalies in sensor time series. In Figure 5, we see an example of a Space Shuttle Marotta Valve time series that was annotated as normal by a NASA engineer.

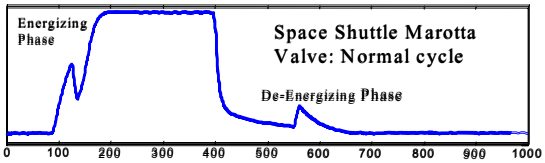

Figure 5: An example of a Space Shuttle Marotta Valve time series that was annotated as *normal*

The engineer further annotated several other traces of the same sensor that have several kinds of faults. We first consider an easy problem; in Figure 6, the expert annotated the last in a series of 5 energize/deenergize cycles as "*Poppet pulled significantly out of the solenoid before energizing*". The problem is immediately obvious to even the untrained eye, and the discord$_{128}$ completely spans the offending section.

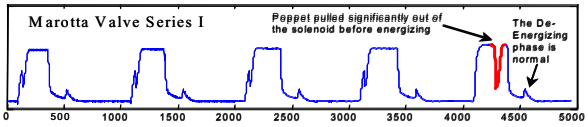

Figure 6: An example of an annotated Marotta Valve time series. The discord discovered (highlighted in bold) exactly corresponds with the expert's annotation

In Figure 7, we consider a much more subtle problem; once again we find the discord$_{128}$, and once again its location exactly coincides with the expert's annotation.

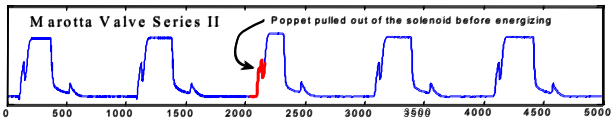

Figure 7: Another example of an annotated Marotta Valve time series. While the discord discovered (highlighted in bold) exactly corresponds with the expert's annotation, it is difficult to see why at this scale

At the scale shown, it is impossible to see what the expert saw to justify her decision; however, in Figure 8, we can see that the discord has a "double hump", whereas the corresponding section of the other four cycles have a single hump.

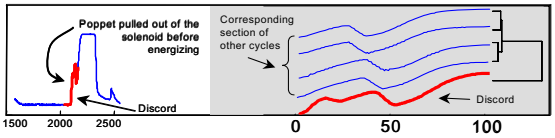

Figure 8: (*Left*) A subsection of Figure 7 showing the discord found. (*Right*) A Zoom-in of the discord, and the four corresponding sections form the normal cycles explains the fault

5.1.2 Anomaly Detection in Electrocardiograms

We have already considered the utility of discords in an Electrocardiogram (ECG) in Figure 1. That was a very simple and "clean" example for clarity; however, it is remarkable how varied and complex normal healthy ECGs can be. For example, Figure 9 shows a very complicated signal with remarkable variability Surprisingly, this ECG contains only one small anomaly, which is easily discovered by a discord.

In Figure 10, we consider an ECG that has several different types of anomalies. Here, the first 3 discords exactly line up with the cardiologist's annotations.

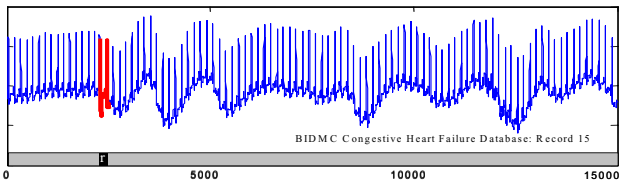

Figure 9: An ECG that has been annotated by a cardiologist (bottom bar) as containing one premature ventricular contraction. The discord$_{256}$ (bold line) exactly coincides with the heart anomaly

In this figure we could perhaps spot the anomalies by eye; however, the full time series is much longer, and impossible to scrutinize without a scrollbar and much patience.

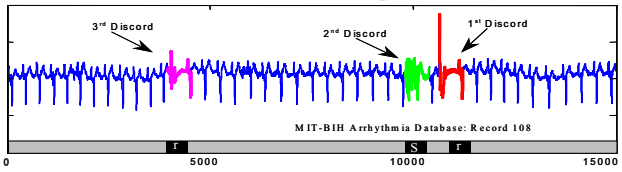

Figure 10: An excerpt of an ECG that has been annotated by a cardiologist (bottom bar) as containing 3 various anomalies. The first 3 discord$_{600}$ (bold lines) exactly coincides with the anomalies

In the above cases, we simply set the length of the discords to be approximately one full heartbeat (note that the two datasets have different sampling rates). Although we found that we could double or half the parameters without affecting the quality of results, on just a handful of the dozens of ECG datasets we examined, the discords had a harder time finding the anomalous heartbeats. We conferred with cardiologist, Dr. Helga Van Herle M.D., who informed us that heart irregularities can sometimes manifest themselves at scales significantly shorter than a single heartbeat. Armed with this knowledge, we searched for discords at approximately ¼ the length of a single heartbeat. In Figure 11, we show the results of such a search.

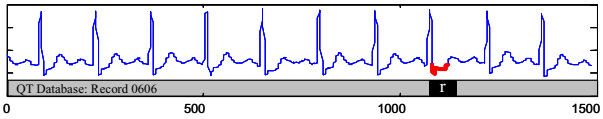

Figure 11: An ECG that has been annotated by a cardiologist (bottom bar) as containing one premature ventricular contraction. The discord$_{40}$ (bold line) exactly coincides with the heart anomaly

While the result is satisfying in that it immediately locates the anomaly, it is not obvious from the figure that the discord is actually different for the other heartbeats. In Figure 12 (*left*) we see a zoom-in of the subsequence surrounding the discord, and we can see that the discord falls over the ST wave. In Figure 12 (*right*), we manually extracted 4 ST waves from the subsequence in Figure 11 and clustered them together with the discord. This makes the source of the anomaly apparent.

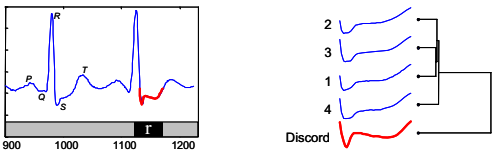

Figure 12: (*left*) A zoom-in of a section of Figure 11. The first heartbeat has been annotated with the classic notation. (*right*) Five ST waves from Figure 11 (including the discord) hierarchically clustered

Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM'05)
1550-4786/05 $20.00 © 2005 IEEE

5.2 The Utility of HOT SAX

It is increasingly recognized that comparing algorithms performance by examining wall clock or CPU time invites the possibility of implementation bias [5], which in turn invites the possibility of irreproducible "improvements." Instead, we measure here the number of times that the distance function is called on line 9 in Table 1 and Table 2. A simple analysis of the pseudocode (confirmed with a profiler) tells us that this single line of code accounts for more than 99% of the running time for both algorithms.

The above metric does not include the time it takes to build the data structures discussed in Section 4.2; however, we note that this is a O(m), one time cost. For datasets of a reasonable size (i.e., the datasets shown in Figures 11 or 12), this overhead takes much less than 0.1% of the total time. Furthermore, as the datasets get larger, it takes an even smaller percentage of time.

In Figure 13, we compare the brute force algorithm to the HOT SAX algorithm in terms of the number of times the Euclidean distance function on line 9 is called. For the HOT SAX we averaged the results for each setting of dataset/length over 100 runs on different subsets of the data.

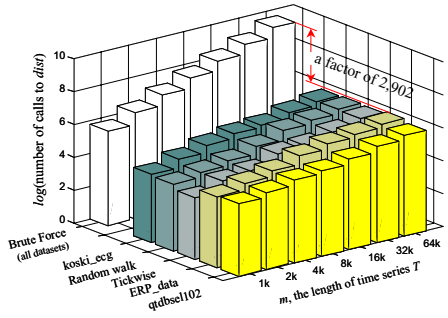

Figure 13: The number of calls to the distance function required by brute force and heuristic search for discord$_{128}$ over a range of data sizes for 5 representative datasets

Note that as the data sizes increase, the differences get larger. For a time series of length 64,000, HOT SAX is almost three thousand times faster than brute force for all datasets. This experiment is actually pessimistic in that we made sure that the test data did not have any obvious anomalies or unusual patterns. In general, if there are truly unusual patterns in the time series, the HOT SAX is even faster.

In general, these results strongly suggest that we can reasonably expect at least 3 orders of magnitude of a speedup for most problems. To concretely ground these numbers, consider the following. While our current implementation is in relatively lethargic Matlab, the experiments shown in Figures 10, 11, and 12 take a few seconds using heuristic search, but several hours using brute force search.

To make sure that the above results were not the result of a happy coincidence of "easy" datasets and the right setting of the single parameter, we repeated the experiment for every dataset in the UCR Time Series Data Mining Archive over a range of values for n. We tested all datasets that have a length of at least 16,000; there are currently 82 such datasets from a diverse set of domains. Figure 14 shows the results.

6. CONCLUSIONS AND FUTURE WORK

In this work, we have defined time series discords, a new primitive for time series data mining. We introduced the HOT SAX algorithm to efficiently find discords and demonstrated its utility of a host of domains. Many future directions suggest themselves; most obvious among them are extensions to multidimensional time series, to streaming data, and to other distance measures. In addition, for truly massive datasets, even the large speedups obtained may be insufficient for real time interaction. We therefore plan to investigate an anytime version of our algorithm.

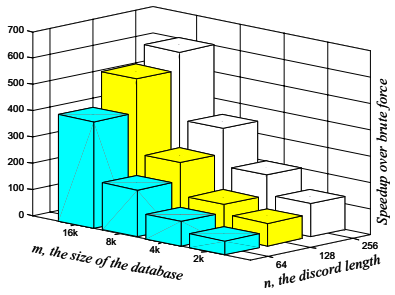

Figure 14: The speed obtained over brute force search for various discord lengths and database sizes, averaged over 82 diverse datasets

7. REFERENCES

[1] Bentley. J. L. & Sedgewick. R. (1997). Fast algorithms for sorting and searching strings. In Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 360-369

[2] Chiu, B., Keogh, E. & Lonardi, S. (2003). Probabilistic Discovery of Time Series Motifs. In the 9th SIGKDD Conference on Knowledge Discovery and Data Mining. pp 493-498.

[3] Coerman, T. H., Leiserson, C. E. &. Rivest R. L. (1990) Introduction to Algorithms, McGraw-Hill Company.

[4] Keogh. E. (2005). www.cs.ucr.edu/~eamonn/discords/

[5] Keogh, E. & Kasetty, S. (2002). On the need for time series data mining benchmarks: A survey and empirical demonstration. In Proc. of SIGKDD. pp 102-111.

[6] Keogh, E., Lonardi, S. & Ratanamahatana, C. (2004). Towards Parameter-Free Data Mining. In proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp 206-215.

[7] Knorr, E., Ng, R. &. Tucakov V. (2000). Distance-Based Outliers: Algorithms and Applications. VLDB J. 8(3-4): 237-253.

[8] Kumar, N., Lolla N., Keogh, E., Lonardi, S., Ratanamahatana, C., & Li, W. (2005). Time-series Bitmaps: A Practical Visualization Tool for working with Large Time Series Databases. SIAM Data Mining Conference.

[9] Lanctot, J. K., Li, M., Ma. B., Wang, S., &. Zhang, L (2003). Distinguishing string selection problems, Information and Computation 185: pp 41–55.

[10] Lin, J., Keogh, E., Lonardi, S., & Chiu, B. (2003). A Symbolic Representation of Time Series, with Implications for Streaming Algorithms. In proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery.

[11] Lin, J., Keogh, E., Lonardi, S., Lankford, J. P. & Nystrom, D. M. (2004). Visually Mining and Monitoring Massive Time Series. In proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp 460-469.

[12] Sadakane, K., (2000) Compressed text databases with efficient query algorithms based on the compressed suffix array, Proceedings of ISAAC'00, LNCS, pp 410–421.

[13] Tanaka, Y. & Uehara, K. (2004). Motif Discovery Algorithm from Motion Data. In proceedings of the 18th Annual Conference of the Japanese Society for Artificial Intelligence (JSAI).

Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM'05)
1550-4786/05 $20.00 © 2005 IEEE