Approximations to Magic: Finding Unusual Medical Time Series

Jessica Lin Eamonn Keogh Ada Fu Helga Van Herle

University of California, Riverside *The Chinese University of Hong Kong* *David Geffen School of Medicine, UCLA*

{jessica , eamonn}@cs.ucr.edu adafu@cse.cuhk.edu.hk hvanherle@mednet.ucla.edu

Abstract

In this work we introduce the new problem of finding time series *discords*. Time series discords are subsequences of 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. While the brute force algorithm to discover time series discords is quadratic in the length of the time series, we show a simple algorithm that is 3 to 4 orders of magnitude faster than brute force, while guaranteed to produce identical results.

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 [3]. 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.

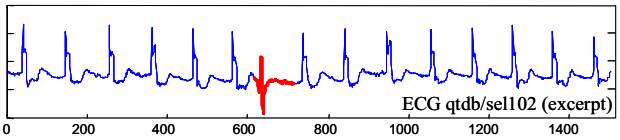

Figure 1: The time series discord found in an excerpt of electrocardiogram qtdb/sel102 (marked in bold line). The location of the discord exactly coincides with a premature ventricular contraction

As we shall show, time series discords are superlative anomaly detectors, able to detect anomalies in diverse medical applications.

2. Related work and background

Our review of related work is exceptionally brief because we are considering a new problem. Some of the notions used throughout this paper are described below.

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 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 $n \le m$ of contiguous position from T, that is, $C = t_p,\ldots,t_{p+n-1}$ for $1 \le p \le m - n + 1$.

One can see that 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 [1][2][5]. 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

Proceedings of the 18th IEEE Symposium on Computer-Based Medical Systems (CBMS'05)
1063-7125/05 $20.00 © 2005 **IEEE**

Definition 3. 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 D(M,C) if $| p - q | >= n$.

We will use the definition of non-self matches to define time series discords:

Definition 4. 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.

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

Definition 5. Euclidean Distance: Given two time series Q and C of length n, the Euclidean distance between them is defined as:

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

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 D(tp,...,tp+n-1, tq,...,tq+n-1) < nearest_neighbor_dist
10             nearest_neighbor_dist = D(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 ]
```

The algorithm is easy to implement, and produces exact results. However, it has $O(m^2)$ time complexity which is simply untenable for even moderately large datasets. The following two observations, however, 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.

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. It is important to 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 inner loop.

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 D(tp,...,tp+n-1, tq,...,tq+n-1) < best_so_far_dist
10          Break                                  // Break out of Inner Loop
11        End
12        IF D(tp,...,tp+n-1, tq,...,tq+n-1) < nearest_neighbor_dist
13          nearest_neighbor_dist = D(tp,...,tp+n-1, tq,...,tq+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 ]
```

We have now reduced the discord discovery problem into a generic framework where all one needs to do is to specify the heuristics. 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 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 abandoned during the very first iteration.
- **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.

These results are something of a mixed bag for us. On one hand they suggest that a linear time algorithm is possible, but only with the aid of some very wishful thinking. The following two observations offer us some hope for a fast algorithm:

- **Observation 3**: In the outer loop, we don't 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 in line 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 don't 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.

4. Approximations to magic

Before we introduce our techniques for approximating the perfect ordering returned by the hypothetical **Magic** heuristics, we briefly review the discretization technique: **S**ymbolic **A**ggregate Appro**X**imation (SAX) [6]. While there are 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 L*p* norms.

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:

$$\overline{c}_i = \frac{w}{n} \sum_{j=\frac{n}{w}(i-1)+1}^{\frac{n}{w}i} c_j \qquad (2)$$

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][4]. In empirical tests on more than 50 datasets we noted that normalized subsequences have highly Gaussian distribution [6], so we can simply determine the "breakpoints" that will produce equal-sized areas under Gaussian curve. These breakpoints may be determined by looking them up in a statistical table. Once the breakpoints have been obtained we can discretize a time series in the following manner. 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 2 illustrates the idea.

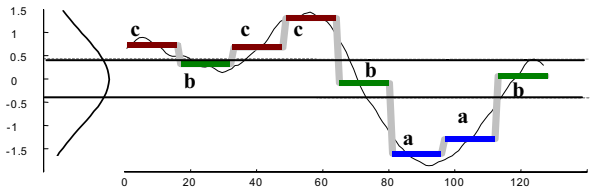

Figure 2: A time series (thin black line) is discretized by first obtaining a PAA approximation (heavy gray line) and then using predetermined breakpoints to map the PAA coefficients into symbols (bold letters). In the example above, with $n = 128$, $w = 8$ and $a = 3$, the time series is mapped to the word *cbccbaab*

4.2 Approximating the magic *outer* loop

We begin by creating two data structures to support our heuristics. First, we create 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 3 gives a visual intuition of this, where both *a* and *w* are set to 3.

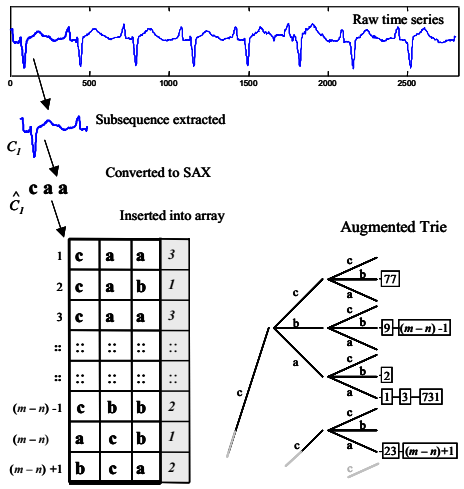

Figure 3: The two data structures used to support the *Inner* and *Outer* heuristics. (*left*) An array of SAX words, where the last column contains a count of how often each word occurs in the array. (*right*) An excerpt of an trie with leaves that contain a list of all array indices that map to that terminal node

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 counts of the numbers of occurrences of each word can be mapped back to the rightmost column of the array. For example, in Figure 3 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, lets 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.

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 of candidates, 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 mapped 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 in line 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

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 3. 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 2 prevents 731 from being compared to itself). After this step, the rest of the subsequences are visited in random order.

The intuition behind our *Inner* heuristic is also simple. Subsequences that have the same SAX encoding as the candidate subsequence are very likely to be highly similar. As noted in observation 4, we just need to find one such subsequence that is similar enough (has a distance to the candidate than the current value of the best_so_far_dist variable) in order to termination the inner loop.

5. Emperical evaualtion

For each example below, we give the speed up factor over brute force search, the only reasonable strawman.

5.1 Anomaly detection in electrocardiograms

Electrocardiograms (ECGs) are a time series of the electrical potential between two points on the surface of the body caused by a beating heart. We have already considered the utility of discords in one ECG in Figure 1. Figure 4 shows a very complicated signal with remarkable variability.

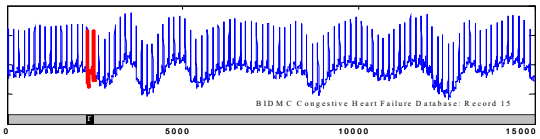

Figure 4: An ECG that has been annotated by a cardiologist (bottom bar) as containing one premature ventricular contraction. The discord$_{256}$ (bold line) exactly coincides with the heart anomaly

Surprisingly, this ECG contains only one small anomaly, which is easily discovered by a discord. In this problem our heuristic search algorithm is 909 times faster than brute force, taking us from an offline ½ hour to a real time few seconds.

In Figure 5 we consider an ECG which has several different types of anomalies. Here the first 3 discords exactly line up with the cardiologists annotations. Here our algorithm was 779 times faster than brute force.

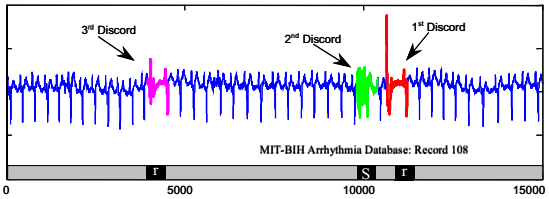

Figure 5: An excerpt of an ECG that has been annotated by a cardiologist (bottom bar) as containing 3 various anomalies. The first 3 discord$_{600}$ (bold lines) exactly coincides with the anomalies

6. References

[1] Duchene, F. Garbayl, C. & Rialle, V. (2004). Mining Heterogeneous Multivariate Time-Series for Learning Meaningful Patterns: Application to Home Health Telecare. Laboratory TIMC-IMAG, Facult'e de m'edecine de Grenoble, France.

[2] Chiu, B., Keogh, E. & Lonardi, S. (2003). Probabilistic Discovery of Time Series Motifs. In the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. August 24 - 27. Washington, DC, USA. pp 493-498.

[3] Keogh, E. & Kasetty, S. (2002). On the need for time series data mining benchmarks: A survey and empirical demonstration. In Proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Edmonton, Canada. July 23-26.

[4] Keogh, E., Lonardi, S. and Ratanamahatana, C. (2004). Towards Parameter-Free Data Mining. In proceedings of the tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Seattle, WA, Aug 22-25.

[5] Kitaguchi, S. (2004). Extracting Feature based on Motif from a Chronic Hepatitis Dataset. In proceedings of the 18th Annual Conference of the Japanese Society for Artificial Intelligence (JSAI). Kanazawa, Japan.

[6] 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.