Disk Aware Discord Discovery: Finding Unusual Time Series in Terabyte Sized Datasets

Dragomir Yankov, Eamonn Keogh
Computer Science & Engineering Department
University of California, Riverside, USA
{dyankov,eamonn}@cs.ucr.edu

Umaa Rebbapragada
Department of Computer Science
Tufts University, Medford, USA
urebbapr@cs.tufts.edu

Abstract

The problem of finding unusual time series has recently attracted much attention, and several promising methods are now in the literature. However, virtually all proposed methods assume that the data reside in main memory. For many real-world problems this is not be the case. For example, in astronomy, multi-terabyte time series datasets are the norm. Most current algorithms faced with data which cannot fit in main memory resort to multiple scans of the disk/tape and are thus intractable. In this work we show how one particular definition of unusual time series, the time series discord, can be discovered with a disk aware algorithm. The proposed algorithm is exact and requires only two linear scans of the disk with a tiny buffer of main memory. Furthermore, it is very simple to implement. We use the algorithm to provide further evidence of the effectiveness of the discord definition in areas as diverse as astronomy, web query mining, video surveillance, etc., and show the efficiency of our method on datasets which are many orders of magnitude larger than anything else attempted in the literature.

1. Introduction

The problem of finding unusual (abnormal, novel, deviant, anomalous) time series has recently attracted much attention. Areas that commonly explore such unusual time series are, for example, fault diagnostics, intrusion detection, and data cleansing. There, however, are other more uncommon yet interesting applications too. For example, a recent paper suggests that finding unusual time series in financial datasets could be used to allow diversification of an investment portfolio, which in turn is essential for reducing portfolio volatility [33].

Despite its importance, the detection of unusual time series remains relatively unstudied when data reside on external storage. Most existing approaches demonstrate efficient detection of anomalous examples, assuming that the time series at hand can fit in main memory. However, for many applications this is not be the case. For example, multi-terabyte time series datasets are the norm in astronomy [23], while the daily volume of web queries logged by search engines is even larger. Confronted with data of such scale current algorithms resort to numerous scans of the external media and are thus intractable. In this work, we present an effective and efficient disk aware algorithm for mining unusual time series. The algorithm is exact and requires only two linear scans of the disk with a tiny buffer of main memory. Furthermore, it is simple to implement and does not require tuning of multiple unintuitive parameters. The introduced method is used to provide further evidence of the utility of one particular definition of unusual time series, namely, the time series *discords*. The effectiveness of the discord definition is demonstrated for areas as diverse as astronomy, web query mining, video surveillance, etc. Finally, we show the efficiency of the proposed algorithm on datasets which are many orders of magnitude larger than anything else attempted in the literature. In particular we show that our algorithm can tackle multi-gigabyte data sets containing tens of millions of time series in just a few hours.

2. Related Work And Background

The time series discord definition was introduced in [20]. Since then, it has attracted considerable interest and follow-up work. For example, [11] provide independent confirmation of the utility of discords for discovering abnormal heartbeats, in [4] the authors apply discord discovery to electricity consumption data, and in [34] the authors modify the definition slightly to discover unusual shapes.

However, all discord discovery algorithms, and indeed virtually all algorithms for discovering unusual time series under any definition, assume that the entire dataset can be loaded in main memory. While main memory size has been rapidly increasing, it has not kept pace with our ability to collect and store data.

There are only a handful of works in the literature that have addressed anomaly detection in datasets of anything like the scale considered in this work. In [12] the authors consider an astronomical data set taken from the Sloan Digital Sky Survey, with 111,456 records and 68 variables. They find anomalies by building a Bayesian network and then looking for objects with a low log-likelihood. Because the dimensionality is relatively small and they only used 10,000 out of the 111,456 records to build the model, all items could be placed in main memory. They report 3 hours of CPU time (with a 400MHz machine). For the secondary storage case they would also require at least two scans, one to build the model, and one to create anomaly scores. In addition, this approach requires the setting of many parameters, including choices for discretization of real variables, a maximum number of iterations for EM (a sub-routine), the number of mixture components, etc.

In a sequence of papers Otey and colleagues [16] introduce a series of algorithms for mining distance based outliers. Their approach has many advantages, including the ability to handle both real-valued and discrete data. Furthermore, like our approach, their approach also requires only two passes over the data, one to build a model and one to find the outliers. However, it also requires significant CPU time, being linear in the size of the dataset but quadratic in the dimensionality of the examples. For instance, for two million objects with a dimensionality of 128 they report needing 12.5 hours of CPU time (on a 2.4GHz machine). In contrast, we can handle a dataset of size two million objects with dimensionality 512 in less than an hour, most of which is I/O time.

Distance based outliers are also the problem of study in Knorr et al. [21] and Tao et al. [31]. Both works discuss a quadratic (in the dataset size) nested loop algorithm for outlier detection and subsequently suggest ways for its improvement. Knorr et al. [21] propose an algorithm that performs three scans through the database and also requires significant amount of main memory. The algorithm further uses a partitioning scheme, whose performance deteriorates in higher dimensions. Tao and colleagues sample the data space to build a set of partitions which they later use to prune non-outlier examples. They explicitly require that the distance function used be a metric, but for time series data non-metric functions have been demonstrated to be often superior [35]. The method again is intended for a lower dimensional data and is demonstrated to require two linear scans of the disk.

Another method that scales to large disk resident datasets, requiring only two linear scans of the disk, was recently proposed by Angiulli and colleagues [6]. An extension of the method for online detection of distance based outliers from streaming data [5] has also been presented by the same authors. To perform faster range queries, the algorithm again builds an index in main memory, based on the concept of pivot points. Apart of the already pointed problems of degrading efficiency in higher dimensions and the requirement of metric distance function, using pivots for indexing poses additional challenges too. For example, selecting good pivot points might itself require to first detect a set of outliers. This is so, because as Shapiro [28] suggests good pivots tend to be points that are far from any dense region in the data.

The problem of outliers detection in higher dimensional spaces was target in an influential paper by Jagadish et al. [18]. They propose to find unusual time series (which they call *deviants*) with a dynamic programming approach. Again this method is quadratic in the length of the time series, and thus it is only demonstrated on kilobyte sized datasets.

The discord introducing work [20] suggests a fast heuristic technique (termed *HOTSAX*) for pruning quickly the data space and focusing only on the potential discords. The authors obtain a lower dimensional representation for the time series at hand and then build a trie in main memory to index these lower dimensional sequences. A drawback of the approach is that choosing a very small dimensionality size results in a large number of discord candidates, which makes the algorithm essentially quadratic, while choosing a more accurate representation increases the index structure exponentially. The datasets used in that evaluation are also assumed to fit in main memory.

In order to discover discords in massive datasets we must design special purpose algorithms. The main memory algorithms achieve speed-up in a variety of ways, but all require random access to the data. Random access and linear search have essentially the same time requirements in main memory, but on disk resident datasets, random access is expensive and should be avoided where possible. As a general rule of thumb in the database community it is said that random access to just 10% of a disk resident dataset takes about the same time as a linear search over the entire data. In fact, recent studies suggest that this gap is widening. For example, [27] notes that the internal data rate of IBM's hard disks improved from about 4 MB/sec to more than 60 MB/sec. In the same time period, the positioning time only improved from about 18 msec to 9 msec. This implies that sequential disk access has become about 15 times faster, while random access has only improved by a factor of two.

Given the above, efficient algorithms for disk resident datasets should strive to do only a few sequential scans of the data.

3. Notation

Let a *time series* $T = t_1, \ldots, t_m$, be defined as an ordered set of scalar or multivariate observations t_i measured at equal intervals in time. When m is very large, looking at the time series as a whole does not reveal much useful information. Instead, one might be more interested in *subsequences* $C = t_p, \ldots, t_{p+n-1}$ of T with length $n << m$ (here p is an arbitrary position, such that $1 \leq p \leq m - n + 1$).

Working with *time series databases* there are usually two scenarios in which the examples in the database might have been generated. In one of them the time series are generated from short distinct events, e.g. a set of astronomical observations (see Section 6.1.1). In the second scenario, the database simply consists of all possible subsequences extracted from the time series of a long ongoing process, e.g. the yearly recordings of a meteorological sensor. Knowing whether the database is populated with subsequences of the same process is essential when performing pattern recognition tasks. The reason for this is that two subsequences C and M extracted from close positions p_1 and p_2 are very likely to be similar to one another. This might falsely lead to a conclusion that the subsequence C is not a rare example in the database. In these cases, when p_1 and p_2 are not "significantly" different, the subsequences C and M are called *trivial matches* [10]. The positions p_1 and p_2 are significantly different with respect to a *distance function* $Dist$, if there exists a subsequence Q starting at position p_3, such that $p_1 < p_3 < p_2$ and $Dist(C, M) < Dist(C, Q)$.

With the above notation in hand, we can now present the formal definition of time series discords:

Definition 1. *Time Series Discord:* Given a database S, the time series $C \in S$ is called the most significant discord in S if the distance to its nearest neighbor (or its nearest non-trivial match in case of subsequence databases) is largest. I.e. for an arbitrary time series $M \in S$ the following holds: $\min(Dist(C, Q)) \geq \min(Dist(M, P))$, where $Q, P \in S$ (and Q, P are non-trivial matches of C and M in case of subsequence databases).

Similarly, one could define the second-most significant or higher order discords in the database. To capture the case of a small group of examples in the space that are close to each other but far from all other examples, we might want to generalize Definition 1 so that the distance to the k-th instead of the first nearest neighbor is considered:

Definition 2. K^{th} *Time Series Discord:* Given a database S, the time series $C \in S$ is called the most significant k-th discord in S if the distance to its k-th nearest neighbor (or its k-th nearest non-trivial match in case of subsequence databases) is largest.

The generalized view of discords (Definition 2) is equivalent to another notion of unusual time series that is frequently encountered in the literature, i.e. the *distance based outliers* [21]. The definition can be generalized further to compute the average distance to all k nearest neighbors, which is in fact the non-parametric density estimation approach [29]. The algorithm proposed in this work can easily be adapted with any of these outlier definitions. We use Definition 1 because of its intuitive interpretation. Our choice is further justified by the effectiveness of the discord definition demonstrated in Section 6.1.

Unless otherwise specified we will use as a distance measure the Euclidean distance, still the derived algorithm can be utilized with any distance function which may not necessarily be a metric. In computing $Dist(C, M)$ we expect that the arguments have been normalized to have mean zero and a standard deviation of one. Throughout the empirical evaluation we assume that all subsequences are stored in the database in the above normalized form. This requirement is imposed so that the nearest neighbor search is invariant to transformations, such as shifting or scaling [19].

4. Finding Discords In Secondary Storage

So far we have introduced the notion of time series discords, which is the focus of the current work. Here, we are going to present an efficient algorithm for detecting the top discords in a dataset. Firstly, the simpler problem of detecting what we call *range discords* is addressed, i.e. given a range r the presented method efficiently finds all discords at distance at least r from their nearest neighbor. As providing r may require some domain knowledge, the next section will demonstrate a sampling procedure that will solve the more general problem of detecting the top dataset discords without knowing the range parameter.

The discussion is limited to the case where the database S contains $|S|$ separate time series of length n. If instead the database is populated with subsequences from a long time series the fundamental algorithm remains unchanged, with some additional minor bookkeeping to discount trivial matches.

4.1. Discord Refinement Phase

The range discord detection algorithm has two phases: a candidate selection phase (phase1), and a discord refinement phase (phase2). For clarity of exposition we first outline the second phase of the algorithm.

The discord refinement phase accepts as an input a subset $C \subset S$ (built in phase1), which is assumed to contain all discords C_j at distance $C.dist_j \geq r$ from their nearest neighbor in S, and possibly some other time series from S. If this is case, then the following simple algorithm can be used to prune the set C to retain only the true discords with respect to the range r:

Although all discords are assumed to be in C, prior to starting Algorithm1 it is unknown which items in C are true discords, and what their actual discord distances are. Initially, all these distance are set to infinity (line 2). The above algorithm simply scans the disk resident database, comparing the list of candidates to each item on disk. The actual distance is computed with an optimized procedure which uses an upper bound for early termination [20] (line 9). For

```
Algorithm 1 Discord Refinement Phase
procedure [$C, C.dist$]=DC_Refinement(S, C, r)
in:  $S$: disk resident dataset of time series
     $C$: discord candidates set
     $r$: discord defining range
out: $C$: list of discords
     $C.dist$: list of NN distances to the discords
 1: for $j = 1$ to $|C|$ do
 2:   $C.dist_j = \infty$
 3: end for
 4: for $\forall S_i \in S$ do
 5:   for $\forall C_j \in C$ do
 6:     if $S_i == C_j$ then
 7:       $continue$
 8:     end if
 9:     $d = EarlyAbandon(S_i, C_j, C.dist_j)$
10:     if $(d < r)$ then
11:       $C = C \setminus C_j$
12:       $C.dist = C.dist \setminus C.dist_j$
13:     else
14:       $C.dist_j = min(C.dist_j, d)$
15:     end if
16:   end for
17: end for
```

example, in the case of Euclidean distance, the *EarlyAbandon* procedure will stop the summation $Dist(S_i, C_j) = \sum_{k=1}^{n} \sqrt{(s_{ik} - c_{jk})^2}$ if it reaches $k = p$, such that $1 \leq p \leq n$ for which $\sum_{k=1}^{p} (s_{ik} - c_{ik})^2 \geq C.dist_j^2$. If this happens then the new item S_i obviously cannot improve on the current nearest neighbor distance $C.dist_j$, and thus the summation may be *abandoned*.

Based on the distance calculations, for each S_i there are three situations:

1. The distance between the discord candidate in C and the item on disk is greater than the current value of $C.dist_j$. If this is true we do nothing.

2. The distance between the discord candidate in C and the item on disk is less that r. If this happens it means that the discord candidate can not be a discord, it is a false positive. We can permanently remove it from the set C (line 11 and line 12).

3. The distance between the discord candidate in C and the item on disk is less than the current value of $C.dist_j$ (but still greater than r, otherwise we would have removed it). If this is true we simply update the current distance to the nearest neighbor (line 14).

It is straightforward to see that upon completion of Algorithm1 the subset C contains only the true discords at range at least r, and that no such discord has been deleted from C, provided that it has already been in it. The time complexity for the algorithm depends critically on the size of the subset size $|C|$. In the pathological case where $|C| = |S|$, it becomes a brute force search, quadratic in the size $|S|$. Obviously, such candidate set could be produced if the range parameter r is equal to 0. If, however, the candidate set C contains just one item, the algorithm becomes essentially a linear scan over the disk for the nearest neighbor to that one item. A very interesting observation is that if the candidate set C contains two or three items instead of one, this will most likely not change the time for the algorithm to run. This is so, because for a very small $|C|$ the CPU required calculations will execute faster than the disk reading operations, and thus the running time for the algorithm is just the time taken for a linear scan of the disk data. To summarize, the efficiency of Algorithm1 depends on the two critical assumptions that:

1. For a given value of r, we can efficiently build a set C which contains all the discords with a discord distance greater than or equal to r. This set may also contain some non-discords, but the number of these "false positives" must be relatively small.

2. We can provide a "good" value for r which allows us to do '1' above. If we choose too low of a value, then the size of set C will be very large, and our algorithm will become slow, and even worse, the set C might no longer fit in main memory. In contrast, if we choose too large a value for r, we may discover that after running the algorithm above the set C is empty. This will be the correct result; there are simply no discords with a distance of that value. However, we probably wanted to find a handful of discords.

4.2. Candidates Selection Phase

In this section we address the first of the above assumptions, i.e. given a threshold r we present an efficient algorithm for building a compact set C with a small number of false positives. A formal description of this candidate selection phase is given as Algorithm2.

The algorithm performs one linear scan through the database and for each time series S_i it validates the possibility for the candidates already in C to be discords (line 5). If a candidate fails the validation, then it is removed from this set. In the end, the new S_i is either added to the candidates list (line 11), if it is likely to be a discord, or it is omitted. To show the correctness of this procedure, and hence of the overall discord detection algorithm, we first point out an observation that holds for an arbitrary distance function:

Proposition 1. *Global Invariant.* Let S_i be a time series in the dataset S and d_{s_i} be the distance from S_i to its nearest neighbor in S. For any subset $C \subset S$ the distance d_{c_i} from

Algorithm 2 Candidates Selection Phase

```
procedure  $[C]$=DC_Selection(S, r)
in:   $S$: disk resident data set of time series
      $r$: discord defining range
out:  $C$: list of discord candidates
 1: $C = \{S_1\}$
 2: for $i = 2$ to $|S|$ do
 3:    $isCandidate = true$
 4:    for $\forall C_j \in C$ do
 5:       if $(Dist(S_i, C_j) < r)$ then
 6:          $C = C \setminus C_j$
 7:          $isCandidate = false$
 8:       end if
 9:    end for
10:    if $(isCandidate)$ then
11:       $C = C \cup S_i$
12:    end if
13: end for
```

S_i to its nearest neighbor in C is larger or equal to d_{s_i}, i.e. $d_{c_i} \geq d_{s_i}$.

Indeed, if the nearest neighbor of S_i is part of C then $d_{s_i} = d_{c_i}$. Otherwise, as C does not contain elements outside of S, the distance d_{c_i} should be larger than d_{s_i}.

Using the above global invariant, we can now easily justify the following proposition:

Proposition 2. Upon completion of Algorithm2, the candidates list C contains all discords S_i at distance $d_{s_i} \geq r$ from their nearest neighbors in S.

Proof. Let S_i be a discord at distance $d_{s_i} \geq r$ from its nearest neighbor in S. From the global invariant it follows that the distance d_{c_i} from S_i to its nearest neighbor in C is larger or equal to d_{s_i}. Therefore, the condition on line 5 of the algorithm will never be satisfied for S_i and hence it will be added to the candidates list (line 11). □

Proposition 2 together with the analysis presented for the refinement phase demonstrate the overall correctness of the algorithm. More formally, the following proposition holds:

Proposition 3. *Correctness.* The candidates selection and the refinement steps detect the discords and only the discords at distance $d_{s_i} \geq r$ from their nearest neighbor in S.

The time complexity of the presented discord detection algorithm is upper-bounded by the time necessary to scan the database twice plus the time necessary to perform all distance computations, which has complexity $O(f{=}\max(|C|)|S|)$. In the experimental evaluation we will demonstrate that, for a good choice of the range parameter, the function f is essentially linear in the database size $|S|$.

5. Finding a Good Range Parameter

The range discord detection algorithm presented in the previous section is deterministic in the sense suggested by Proposition 3, i.e. it finishes by either identifying all discords at range r, or by returning an empty set which indicates that no elements have the required property. Providing a good value for the threshold parameter, however, may not be very intuitive. Furthermore, it may also be the case that the users would like to detect the top k discords regardless of the distance to their neighbors. In those cases, specifying a large threshold will result in an empty set, while a very small range parameter may have high time and space complexity. With this in mind, a reasonable strategy to detect the top k discords would be to start with a "relatively large" r and if in the end $|C| < k$, to restart the algorithm with a smaller parameter. Such iterative restarts will increase the number of database scans, yet we argue that with a sampling procedure we can obtain a good estimate for r that decreases the probability of having multiple scans of the database. We further provide a way to reevaluate the range parameter, so that if a second run of the algorithm is required, the new value of r with high probability will lead to a solution.

A good estimate for the range parameter can easily be obtained by studying the nearest neighbor distance distribution (*nndd*) of the dataset, and more precisely the number of elements that fall in its tail. Computing the *nndd*, however, is hard, especially in high dimensional spaces as is the case with time series [8][30]. The available methods require that random portions of the space are sampled and the nearest neighbor distances in those portions to be computed. Unfortunately, for a robust estimate, this requires scanning the entire database once, regardless of whether an index is available, and also involves some extensive computations [30]. Another drawback of this approach is that the *nndd* is also dependent on the number of elements in the data, which means that if new sequences are added to the dataset the whole evaluation procedure should be performed again. Consider for example the graphs in Figure 1.

Both graphs show the *nndd* for a normally distributed two dimensional dataset $S \in \mathcal{N}(0,1)$. Graph A represents the probability density function when $|S| = 10^3$, while graph B shows the function when $|S| = 10^4$. Intuitively, the mean of the distribution shifts to zero as new points are added, because for larger percentage of the points their nearest neighbors are likely to be found in close proximity to them. For infinite tail data distributions though (as the normal), increasing the sample size also increases the chance of having elements sampled from its tail. These elements will be outliers and are likely to be far from the other examples. Therefore, their nearest neighbor distances will fall in the tail of the corresponding distance distribution too.

Using the above intuition, rather than sampling from the distance distribution, we perform the less expensive sam-

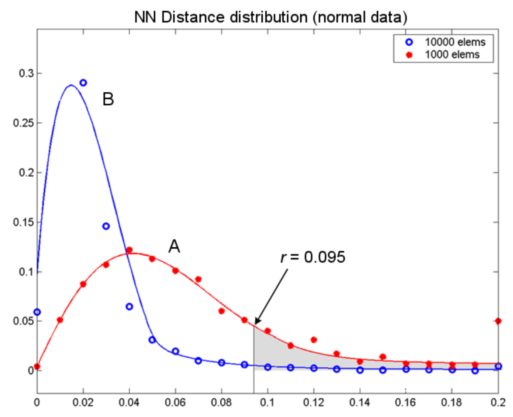

Figure 1: Points sampled from the same normal distribution produce different nearest neighbor distance distributions. The mean and the volume of the tail cut by r decrease with adding more data.

pling from the data distribution and compute the *nndd* of this sample. The exact steps of the sampling procedure are:

1. Select a uniformly random sample S' from S. In the evaluation, for datasets of size $|S| \geq 10^6$ we choose $|S'| = 10^4$. For the smaller datasets we use $|S'| = 10^3$.

2. If the user requires that k discords are detected in their data, then using a fast memory based discord detection method (e.g. [34]) detect the top k discords in S'. Order the nearest neighbor distances $d_i, i = 1..k$ for these discords in S'. I.e. we have $d_1 \geq d_2 \geq \ldots \geq d_k$.

3. Set $r = d_k$.

Note that S' is an unbiased sample from the data and it can be used if new examples generated by the same underlying process are added to the database. This means that we do not need to run the sampling procedure every time that the dataset is updated.

It is relatively easy to see that the above procedure is unlikely to overflow the available memory, regardless of the data distribution. To demonstrate this, consider for example the case when $|S| = 10^6$ and $|S'| = 10^4$. The probability that none of the top 10^3 discords fall in S' is $\hat{p} = \binom{10^6-10^3}{10^4}/\binom{10^6}{10^4}$, which using Stirling's approximation gives $\hat{p} \sim e^{-10}$. This implies that S' almost certainly contains one of the top 10^3 discords. If that discord is S_i, from the global invariant in Section 4.2 it follows that its nearest neighbor distance $d_{s'_i}$ in S' is larger or equal to its nearest neighbor distance d_{s_i} in S. But we also have that $d_1 \geq d_{s'_i}$, which leads to $d_1 \geq d_{s_i}$. This means that if we set r to d_1 (or equivalently to d_k, for small k), it is very likely that r will be larger than the nearest neighbor distance of the 10^3-th discord in S. As will be demonstrated in the experimental evaluation, the majority of the time series that are not discords and enter C during the candidate selection phase get removed from the list very quickly which restricts its maximum size to at most several orders of magnitude the size of the final discord set. Therefore, for the above example the maximum amount of memory required will be linear in the amount of memory necessary to store 10^3 time series. Slightly relaxed, but still reasonable, upper bounds can be demonstrated even when S contains an order of 10^8 examples.

The more challenging case is the one when at the end of the discord detection algorithm we have $|C| < k$. In this situation we will need to restart the whole algorithm, yet this time a better estimate for the threshold r can be computed, so that no other restarts are necessary. For the purpose, prior to running the algorithm, a second sample S'' of size 100 is drawn uniformly at random from S'. During the candidates selection phase, for every element S_i in the database, apart of updating the candidates list C, we also update the nearest neighbor distances $S''.dist_q, q = 1..100$. As the size of S'' is relatively small, this will not increase significantly the computational time of the overall algorithm. At the same time, the list $S''.dist$ will now contain an unbiased estimate of the true nearest neighbor distance distribution. Selecting a threshold $r' = \max_{q=1..100}(S''.dist_q)$ will lead to C having on average 1% of the examples. Finally, if k is much smaller than 1% the size of S, but still larger than the size $|C|$ obtained for the initial parameter r, we might further consider an intermediate value r'', such that $r' < r'' < r$ and one that will increase sufficiently the initial size $|C|$.

6. Empirical Evaluation

In this section we conduct two kinds of experiments. Although the utility of discords has been noted before, e.g. in [4][11][15][20][34], we first provide additional examples of its usefulness for areas where large time series databases are traditionally encountered. Then we empirically demonstrate the scalability of our algorithm.

6.1. The Utility of Time Series Discords

6.1.1 Star Light-Curve Data

Globally there are myriads of telescopes covering the entire sky and constantly recording massive amounts of valuable astronomical data. Having humans to supervise all observations is practically impossible [23].

The goal for this evaluation was to see to what extent the notion of discords, as specified in Definition 1, agrees with the notion of astronomical anomalies as suggested by methods used in the field. The data used in the evaluation are light-curve time series from the Optical Gravitational Lensing Experiment [1]. A light-curve is a real-valued time series of light magnitude measurements. The series are derived from telescopic images of the night sky taken over time. Astronomers identify each star in the image and convert the star's manifestation of light into a light magnitude measurement. The set of measurements from all images for a given star results in a light-curve. The light-curves that we obtained for this study are pre-processed (containing a uniform number of points) by domain experts.

The entire dataset contains 9236 light-curves of length 1024 points. The curves are produced by three classes of star objects: *Eclipsed Binaries* - EB (2580 examples); *Cepheids* - Ceph (1329), and *RR Lyrae variables* - RRL (5326) (see Figure 2). Both Ceph and RRL stars have very similar pulsing pattern which explains the similarity in their light-curve shape.

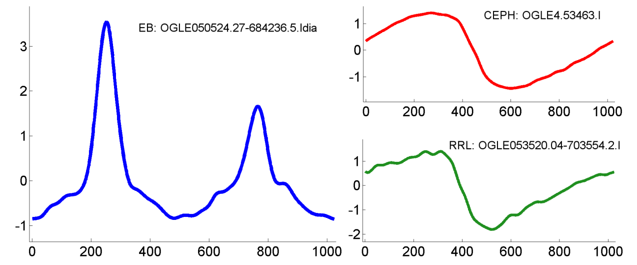

Figure 2: Typical examples from the three classes of lightcurves: Left) Eclipsed Binary, Right Top) Cepheid, Bottom) RR Lyrae.

For each of the three classes we also compute the ranking of their examples for being anomalous. For instance, the topmost anomaly in every class has ranking 0, the second anomaly has ranking 1, and so on. This ordering is based on the results of the first method presented in [26]. The method is an $O(n^2)$ algorithm that exhaustively computes the similarity (via cross correlation) between each pair of light-curves. The anomaly score for each light-curve is simply the weighted average of its $n-1$ similarity scores.

We further compute the top ten discords in each of the three classes and compare them with the top ten anomalies inferred with the above ranking. The sampling procedure described in Section 5 is performed with a set S' of size 10^3 elements and the threshold r is selected so that at least ten elements from each class fall in the tail of the distance distribution computed on S' (we obtained $r = 6.22$ using Euclidean distance). Running the discord finding algorithm produces a discord set C of size 1161. Figure 3 shows several examples of the most significant discords in each class.

One of the top ten EB discords is also among the top ten EB anomalies, three of the top ten RRL discords are among the top ten RRL anomalies and six of the CEPH discords are among the corresponding anomalies. The poor consensus between the one nearest neighbor discords and the anomalies for the EB class results from the fact that the Euclidean distance does not account well for the small amount of warping that is present between the two magnitude spikes. Substituting the Euclidean distance with a phase invariant or a dynamic time warping distance function

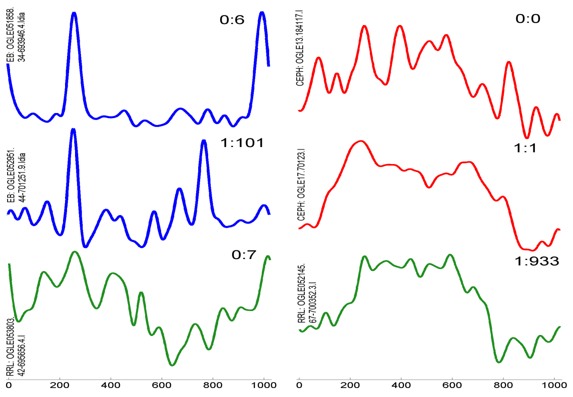

Figure 3: Top light-curve discords in each class. For each time series on the top right corner are indicated its *discord rank : anomaly rank*.

may improve on this problem. For the other two classes the discord definition is more consistent with the expert opinion on the outliers. Even for elements where they disagree significantly, the discord algorithm still returns some intuitive results. For example, the second most significant RRL discord (see Figure 3, bottom right) deviates greatly from the expected RRL shape.

6.1.2 Web Query Data

Another domain where large scale time series datasets are observed daily are the search engines query logs. For example, we studied a dataset consisting of MSN web queries made in 2002. A casual inspection reveals that most web query logs seem to fall into a handful of patterns. Most have a ”background” periodicity of seven days, which reflects the fact that many people only have access to the web during the workweek. This background weekly pattern is sometimes augmented by seasonal effects or bursts due to news stories. The two curves labeled ”Stock Market” and ”Germany” in Figure 4 are such examples. Another common type of pattern we call the *anticipated burst*; it consists of a gradual build up, a climax and a fall off. This is commonly seen for seasonally related items (”Easter”, ”Tour de France”, ”Hanukkah”) and for movie releases as in ”Spiderman” and ”Star Wars”.

Also common is the *unanticipated burst*, which is seen after an unexpected event, such as the death of a celebrity. This pattern is characterized by a near instantaneous burst, followed by a tapering off. Given that both anticipated and unanticipated bursts can happen at any point in the year, we use phase invariant Euclidian distance as discord distance measure. The number one discord is shown in Figure 5.

This discord makes perfect sense with a little hindsight. Unlike weather or cultural events which are intrinsically local, the phases of the moon are perhaps the only changing phenomena that all human beings can observe. While some

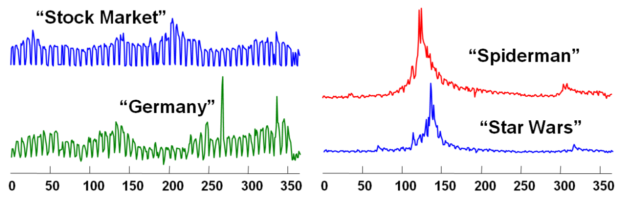

Figure 4: Some examples of typical patterns in web query logs in 2002. Most patterns are dominated by a weekly cycle, as in "stock market" or "Germany", with seasonal deviations and bursts in response to news stories. The "anticipated burst" is seen for movie releases such as "Spiderman/Star Wars", or for seasonal events.

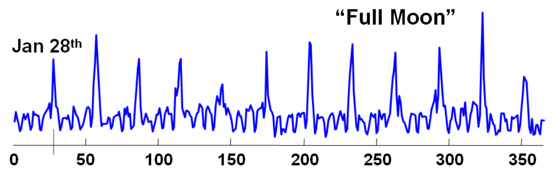

Figure 5: The number one discord in the web query log dataset is "Full Moon". The first full moon of 2002 occurred on January 28th at 22:50 GMT. The periodicity of the subsequent spikes is about 29.5 days, which is the length of the synodic month.

other queries have a weak periodicity corresponding to calendar months, this query has a strong periodicity of 29.5 days, corresponds to the synodic month.

6.1.3 Population Growth Data

We can further demonstrate the utility of discords by examining datasets for which we need external sources of knowledge to evaluate findings. We examined a data set of population growth rates of 206 countries covering 1965 to 2005. We wanted to know the most dramatic 5-year events in this data set, so we queried the data for the top 5-year discords. Figure 6 shows the top two such discords, together with some other representative counties (Argentina, Belgium, Cameroon, Canada, Honduras, Hong Kong, Iceland, India, Indonesia, Ireland) for contrast.

The dramatic differences shown by the discords have intuitive and poignant explanations [2]. The extreme drop in population is clearly understood, but why is it followed in both cases by a spike in growth rate? We conjecture that this corresponds to refugees that fled during the worst of the crises later returning to their homelands. Note that in this case the brute force algorithm would be fast enough to produce these results in reasonable time.

6.1.4 Trajectory Data

We obtained two trajectory datasets used in [24] and [25] respectively, which have been purposefully created to test anomaly detection in video sequences. The time series are two dimensional (comprised of the x and y coordinates for

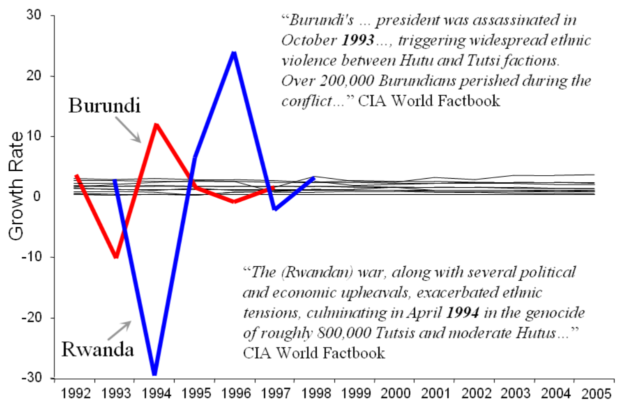

Figure 6: The top two 5-year discords discovered in a growth rate database covering 206 counties over 40 years. Ten other counties are shown for contrast (thin lines).

each data point), and are further normalized to have the same length. In both datasets several deliberately anomalous sequences are created to have a ground truth. The datasets contain 156 [24] and 239 [25] trajectories, with 4 and 2 annotated anomalous sequences respectively. Figure 7 shows the number one discord (2D version of the Euclidean distance has been used) found in the dataset of [24]. It is one of the labeled anomalies too.

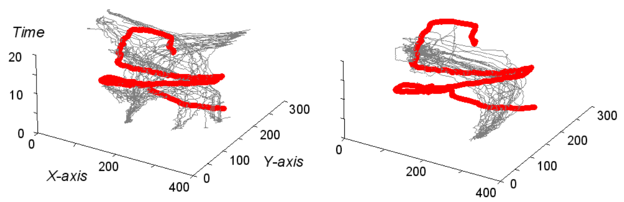

Figure 7: Left) The number one discord found in a trajectory data (bold line) with 50 trajectories. It is difficult to see why the discord is singled out unless we cluster all the non-discord trajectories and compare the discord to the clustered sets. Right) When the discord is shown with the clustered trajectories, its unusual behavior becomes apparent (just one cluster is shown here).

On both datasets the discord definition achieves perfect accuracy, as do the original authors. Since all the data can easily fit in main memory our algorithm takes much less than one second. We do not compare efficiency directly with the original works, but note that [24] requires building a SOM, which are generally noted for being lethargic, while [25] is faster, requiring $O(m \log(m) n)$ time, with m being the number of time series and n their dimensionality. Neither algorithm considers the secondary storage case.

Throughout this paper we have omitted discussion of determining when a discord is truly anomalous/unusual. We plan to address this issue in a separate work. However, in Figure 8 we hint at one possible line of research. The Pokrajac video surveillance dataset [24] is created with two anomalous trajectories (sequences 225 and 237). If we run our algorithm to discover the top sixteen discords, we find that the top two have significantly greater discord distances than all the rest.

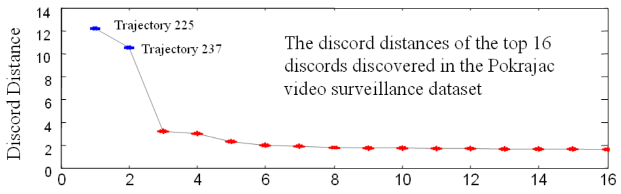

Figure 8: The discord distances for the planted anomalies differ notably from the discord distances of the rest 14 top discords. The fact can be used to evaluate the significance of the detected discords.

6.2. Scalability of the Discord Algorithm

We test the scalability of the method on a large heterogeneous dataset of real-world time series and on three synthetically generated datasets of size up to a third of a terabyte. Two aspects of the algorithm were the focus of this evaluation. Firstly, whether the threshold selection criterion from Section 5 can be justified empirically (at least for certain underlying distributions) for data of such scale. Secondly, we were interested on how efficient our algorithm is, provided that a good threshold is selected. For both, the synthetic and the real time series datasets, the data are organized in pages of size 10^4 examples each. All pages are stored in text format on an external Seagate FreeAgent hard drive of size 0.5Tb with 7200 RPM and a USB2.0 connection to a computer using Pentium D 3.0 GHz processor. Our implementation of the algorithm loads one page for 5.6 secs.: 0.4 secs. for reading the data and 5.2 secs. for parsing the text matrices. The algorithm was coded in Java.

6.2.1 Random Walk Data

We generated three datasets with random walk time series. The datasets contain 10^6, 10^7 and 10^8 examples respectively. The length of the time series is set to 512 points. Additionally, six non-random walk time series are planted in each of the datasets (see Figure 9).

To compute the threshold a sample of size $|S'| = 10^4$ is used. We set the threshold to the nearest neighbor distance of the tenth discord, hoping to detect some of the planted anomalies among the top ten discords in the entire datasets. Thus it was obtained $r = 21.45$. The time series in the three datasets come from the same distribution and therefore, as mentioned in Section 5, the same sample S' (and hence the same threshold r) can be used for all of them. Note that this threshold selection procedure requires less than a minute. After the discord detection algorithm finishes, the set C contains 24 discords for the dataset of size 10^6, 40 discords for the dataset of size 10^7 and 41 discords for the dataset of size 10^8. The running time for the three cases is summarized in Table 1.

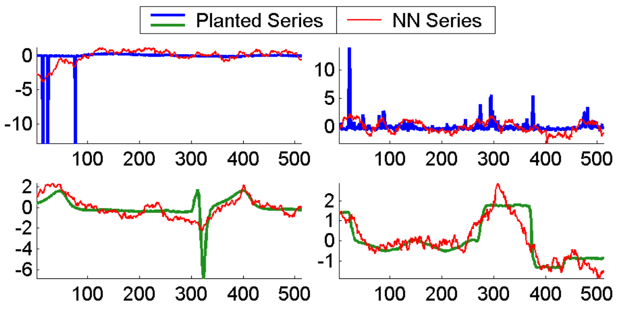

Figure 9: Planted non-random walk time series with their nearest neighbors. The top two time series are among the top discords, the bottom two time series fail the range threshold. $|S| = 10^6$

Table 1: *Randomwalk data.* Time efficiency of the algorithm.

Examples	Disk Size	I/O time	Total time
1 million	3.57Gb	19min	28min
10 million	35.7Gb	3h 12min	5h 43min
100 million	0.35Tb	31h 18min	66h 17min

In all cases the list C contains the required number of 10 discords, so no restart is necessary. From the planted time series three are among the top 10 discords and for the other three a random walk nearest neighbor is found that is relatively close (see Figure 9 for examples). This does not decrease the utility of the discord definition, and is expected as the random walk time series exhibit some extreme properties with respect to the discord detection task, i.e. they cover almost the entire data space that can be occupied by all possible time series of the specified length.

We further note, that the time necessary to find the nearest neighbor for an arbitrary example is 8.5 minutes for the dataset of size one million and approximately 18 hours for the dataset of size 100 million. This means that our algorithm detects the most significant discords in less than four times the time necessary to find the nearest neighbor of a single example only.

Figure 10 demonstrates the size $|C|$ after processing each database page. The graphs also show how the size varies when changing the threshold. The plots demonstrate that with a $2\% - 5\%$ change in its values we still detect the required 10 discords with just two scans, while the maximum memory and the running time do not increase drastically. It is interesting to note how quickly the memory drops after the refinement step is initiated. This implies that most

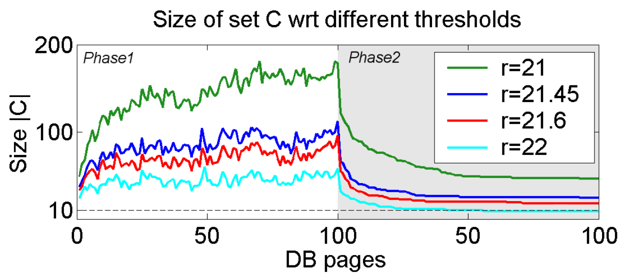

Figure 10: *Randomwalk Data ($|S| = 10^6$).* Number of examples in C after processing each of the 100 pages during the two phases of the algorithm. The method remains stable even if we select a slightly different threshold r during the sampling procedure.

of the non-discord elements in the candidates list get eliminated after scanning just a few pages of the database. From this point on the algorithm performs a very limited number of distance computation to update the nearest neighbor distances for the remaining candidates in C. Similar behavior was observed throughout all datasets studied.

6.2.2 Heterogeneous Data

Finally we check the efficiency of the discord detection algorithm on a large dataset of real-world time series coming from a mixture of distributions. To generate such dataset we combined three datasets each of size $4\text{x}10^5$ (1.2 million elements in total). The time series have length of 140 points. The three datasets are: motion capture data, EEG recordings of a rat, and meteorological data from the Tropical Atmosphere Ocean project (TAO) [3].

Table 2: *Heterogeneous data.* Time efficiency of the algorithm.

Examples	Disk Size	Time(Phase1)	Time(Phase2)
1.2 mill.	1.17Gb	8min. 45secs.	9min. 15secs.

Table 2 lists the running time of the algorithm on the heterogeneous dataset. Again we are looking for the top 10 discords in the dataset. On the sample the threshold is estimated as $r = 12.86$. After the candidate selection phase the set C contains 690 elements, and at the end of the refinement phase there are 59 elements that meet the threshold r. No restarts of the algorithm were necessary for this dataset either. The discords detected are mostly from the TAO class as its time series exhibit much larger variability compared to the time series for the other two classes.

6.2.3 Parallelization of the Discord Detection Algorithm

Recently there has been an emerging interest in scaling some of the best off-the-shelf machine learning algorithms, through the means of parallelization across grids of multiple computers. A limited number of works target the parallel outlier detection problem too. For example, Hung et al. [17] introduce a parallel version of the quadratic nested loop algorithm for distance based outliers, discussed in [21]. Lozano et al. [22] propose two algorithms for parallel mining of distance and density based outliers. The distance based algorithm is a parallel modification of Bay's randomized nested loop algorithm [7], and the density based version is a modification of the popular local outlier factor (LOF) algorithm [9]. Both parallel variants proceed in a similar fashion: First the data space is partitioned across different computers and outliers, local for each computer, are identified. Subsequently, the results from all computers are merged within a 'master process' and a global anomaly score is assigned to each outlier.

A common problem with many parallel data mining approaches is that they require implementing specific distributed architectures, as well as distributed indexing structures. The technical overhead of this prohibits the adoption of these methods across the global data mining community. Recently, however, an intuitive yet extremely scalable framework for parallel data mining has emerged, namely the *MapReduce* framework [13]. *MapReduce* is quickly turning into a parallel data mining standard and is already adopted by large companies, such as Google, Yahoo! and Microsoft. The framework operates in two steps. All examples in the data that can be part of the final solution are first *mapped* to some corresponding keys. Then a user specified function is called to *reduce* the key-value pairs to a set that contains only the final answer.

We conclude the efficiency and scalability evaluation of our algorithm by demonstrating that both of its phases can easily fit into the *MapReduce* framework, where the overhead of the parallelization remains relatively small and a nearly linear (in the number of machines used) speed-up is achieved.

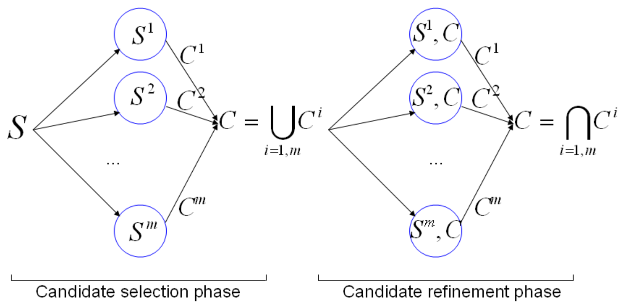
S
S^1
S^2
S^m
C^1
C^2
C^m
$C = \bigcup_{i=1,m} C^i$
S^1, C
S^2, C
S^m, C
$C = \bigcap_{i=1,m} C^i$
Candidate selection phase
Candidate refinement phase

Figure 11: Parallelization of the disk aware discord detection algorithm with m computers.

For the experiments, we assume that the input dataset S is split evenly across m computers as shown in Figure 11. The candidate selection phase of the discord detection algorithm is then run simultaneously on all computers with input parameters $S = S^i$ and the same threshold parameter r, producing m distinct candidate sets C^i. This concludes the mapping function of the first phase, and the reducing function then simply combines all candidate sets into one $C = \bigcup_{i=1..m} C^i$. Note that when constructing C^i we use only the data from S^i, which means that it might introduce to the union C examples that would be pruned, had we used a single computer that scans the entire dataset S. Therefore, the combined candidate set C for m computers is actually larger than the one that would be obtained with one computer and below we show the overhead introduced by this increased candidate set size. What is essential at this point, though, is the fact that Proposition 2 remains valid, i.e. no false dismissals are introduced by the parallel modification of the candidate selection phase.

Once the union operation is performed, the combined candidate set C is distributed to all computers and the candidate refinement phase is run simultaneously on all of them with input parameters $S = S^i$, C, and r. This can again be represented by a mapping function that produces refined sets C^i, $i = 1..m$. We now make the observation that the true discords, and only the true discords, with respect to the range parameter r and the entire dataset S, will be present in every refined set C^i. Hence, the final result requires that we *reduce* the refined candidates C^i by performing an intersect among all of them (see Figure 11). The final discords are thus given by the set $C = \bigcap_{i=1..m} C^i$.

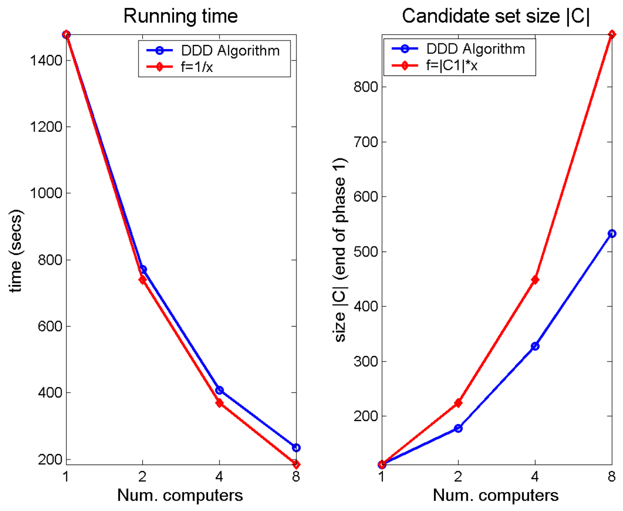

Figure 12: *Randomwalk Data ($|S| = 10^6$). Left:* Running time of the disk aware discord detection algorithm (DDD) with 1, 2, 4 and 8 computers. *Rgith:* Size of the discord candidates set after the first phase of the algorithm. Refer to the text for details.

We simulate the above parallel implementation with $m = 1, 2, 4, 8$ computers each having the same specification as indicated in the beginning of Section 6.2. Figure 12 demonstrates the running time of the entire algorithm and the candidate set size $|C|$ after the first phase, for the random walk dataset with $|S| = 10^6$ elements. The running time (Figure 12, left) is the combined running time for the slowest computer during the first phase, plus the running time for the slowest computer during the second phase. The communication time (broadcasting C^i at the end of each phase, and distributing C at the beginning of the second phase) is not included, yet it is negligible as $|C|$ is much smaller than $|S|$ and also we do not need to broadcast the entire time series but only their indices. The union and intersect operations can also be ignored. In our implementation for both of them we used hashing which requires time linear in the size $|C|$.

Figure 12 left, shows that with 8 computers the algorithm finishes in 240 seconds (the same threshold $r = 21.45$ was used as in Section 6.2.1). The red curve with diamonds on the plot shows simply a division of the time for the algorithm on one computer by x (i.e. if we had x computers and no parallelization overhead). From the plot it is visible that, for example, with 8 computers this 'perfect' running time would be 180 seconds. This means that the increased candidate set size contributes for approximately 30% running time loss when the discord detection algorithm is run with 8 computers. This is due to the approximately five times larger candidate size, obtained at the end of the first phase (see Figure 12 right, the blue curve for 8 computers). Just for comparison we have also plotted the function $f = |\mathrm{C}_1| * x$, where $|\mathrm{C}_1|$ is the candidate set size when using only one computer, to indicate that adding x computers does not necessarily increase the candidate set size x times. Overall the main memory requirement remains admissible for a fairly large number of computers, while the gain in speed-up is enormous. With a single computer, as indicated in Section 6.2.1, the total running time is approximately 28 minutes while now with 8 computers it takes only 4 minutes.

7. Discussion

In a sense, the approach taken here may appear surprising. Most data mining algorithms for time series use some approximation of the data, such as DFT, DWT, SVD etc. Previous (main memory) algorithms for finding discords have used SAX [20][34], or Haar wavelets [15]. However, we are working with just the raw data. It is worth explaining why. Most time series data mining algorithms achieve speed-up with the Gemini framework (or some variation thereof) [14]. The basic idea is to approximate the full dataset in main memory, approximately solve the problem at hand, and then make (hopefully few) accesses to the disk to confirm or adjust the solution. Note that this framework requires one linear scan just to create the main memory approximation, and our algorithm requires a total of two linear scans. So there is at most a factor of two possibility of improvement. However, it is clear that even this cannot be achieved. Even if we assume that some algorithm can be created to approximately solve the problem in main memory. The algorithm must make some access to disk to check the raw data. Because such random accesses are ten times more expensive than sequential accesses [27], if the algorithm must access more that 10% of the data it can no longer be competitive. In fact, it is difficult to see how any algorithm could avoid retrieving 100% of the data in the second phase. For all time series approximations, it is possible that two objects appear arbitrarily close in approximation space, but be arbitrarily far apart in the raw data space. Most data mining algorithms exploit lower bound pruning to find the nearest neighbor, but here upper bounds are required to prune objects that cannot be the furthest nearest neighbor. While there has been some work on providing upper bounds for time series, these bounds tend to be exceptionally weak [32]. Intuitively this makes sense, there are only so many ways two time series can be similar to each other, hence the ability to tightly lower bound. However, there is a much larger space of possible ways that two time series could be different, and an upper bound must somehow capture all of them. In the same vein, it is worth discussing why we do not attempt to index the candidate set C in main memory, to speed up both the phase one and phase two of our algorithm. The answer is simply that it does not improve performance. The many time series indexing algorithms that exist [14][32] are designed to reduce the number of disk accesses, they have little utility when all the data resides in main memory (as with the candidate set C). For high dimensional time series in main memory it is impossible to beat a linear scan; especially when the linear scan is highly optimized with early abandoning. Furthermore, in phase one of our algorithm every object seen in the disk resident data set is either added to the candidate set C or causes an object to be ejected from C, this overhead in maintaining the index more than nullifies any possible gain.

8. Conclusions

The work introduced a highly efficient algorithm for mining range discords in massive time series databases. The algorithm performs two linear scans through the database and a limited amount of memory based computations. It is intuitive and very simple to implement. We further demonstrated, that with a suitable sampling technique the method can be adapted to robustly detect the top k discords in the data. The utility of the discord definition combined with the efficiency of the method suggest it as a valuable tool across multiple domains, such as astronomy, surveillance, web mining, etc. Experimental results from all these areas have been demonstrated.

We are currently exploring adaptive approaches that allow for the efficient detection of statistically significant discords when the time series are generated by a mixture of different processes. In these cases alternating the range parameter according to the distribution of each example turns out to be essential when looking for the top discords with respect to the individual classes.

9. Acknowledgements

We would like to thank Dr. M. Vlachos for providing us the web query data, Dr. P. Protopapas for the light-curves, Dr. A. Naftel and Dr. L. Latecki for the trajectory datasets.

References

[1] http://bulge.astro.princeton.edu/∼ogle/.

[2] http://www.cia.gov/cia/publications/factbook/.

[3] http://www.pmel.noaa.gov/tao/index.shtml.

[4] J. Ameen and R. Basha. Mining time series for identifying unusual sub-sequences with applications. *1st International Conference on Innovative Computing, Information and Control*, 1:574–577, 2006.

[5] F. Angiulli and F. Fassetti. Detecting distance-based outliers in streams of data. In *CIKM '07: Proc. of the sixteenth ACM conference on Conference on information and knowledge management*, pages 811–820, 2007.

[6] F. Angiulli and F. Fassetti. Very efficient mining of distance-based outliers. In *CIKM '07: Proc. of the sixteenth ACM conference on Conference on information and knowledge management*, pages 791–800, 2007.

[7] S. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *KDD '03: Proc. of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 29–38, 2003.

[8] S. Berchtold, C. Böhm, D. Keim, and H. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proc. of the 16th ACM Symposium on Principles of database systems (PODS)*, pages 78–86, 1997.

[9] M. Breunig, H. Kriegel, R. Ng, and J. Sander. Lof: identifying density-based local outliers. *SIGMOD Rec.*, 29(2):93–104, 2000.

[10] B. Chiu, E. Keogh, and S. Lonardi. Probabilistic discovery of time series motifs. In *Proc. of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'03)*, pages 493–498, 2003.

[11] M. Chuah and F. Fu. ECG anomaly detection via time series analysis. Technical Report LU-CSE-07-001, 2007.

[12] S. Davies and A. Moore. Mix-nets: Factored mixtures of gaussians in bayesian networks with mixed continuous and discrete variables. In *Proc. of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 168–175, 2000.

[13] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proc. of the 6th conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[14] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. *SIGMOD Record*, 23(2):419–429, 1994.

[15] A. Fu, O. Leung, E. Keogh, and J. Lin. Finding time series discords based on Haar transform. In *Proc. of the 2nd International Conference on Advanced Data Mining and Applications*, pages 31–41, 2006.

[16] A. Ghoting, S. Parthasarathy, and M. Otey. Fast mining of distance-based outliers in high dimensional datasets. In *Proc. of the 6th SIAM International Conference on Data Mining*, 2006.

[17] E. Hung and D. Cheung. Parallel mining of outliers in large database. *Distrib. Parallel Databases*, 12(1):5–26, 2002.

[18] H. Jagadish, N. Koudas, and S. Muthukrishnan. Mining deviants in a time series database. In *Proc. of the 25th International Conference on Very Large Data Bases*, pages 102–113, 1999.

[19] E. Keogh and S. Kasetty. 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*, pages 102–111, 2002.

[20] E. Keogh, J. Lin, and A. Fu. Hot sax: Efficiently finding the most unusual time series subsequence. In *Proc. of the 5th IEEE International Conference on Data Mining*, pages 226–233, 2005.

[21] E. Knorr and R. Ng. Algorithms for mining distance-based outliers in large datasets. In *Proc. of the 24rd International Conference on Very Large Data Bases (VLDB)*, pages 392–403, 1998.

[22] E. Lozano and E. Acuna. Parallel algorithms for distance-based and density-based outliers. In *ICDM '05: Proc. of the Fifth IEEE International Conference on Data Mining*, pages 729–732, 2005.

[23] K. Malatesta, S. Beck, G. Menali, and E. Waagen. The AAVSO data validation project. *Journal of the American Association of Variable Star Observers (JAAVSO)*, 78:31–44, 2005.

[24] A. Naftel and S. Khalid. Classifying spatiotemporal object trajectories using unsupervised learning in the coefficient feature space. *Multimedia Syst.*, 12(3):227–238, 2006.

[25] D. Pokrajac, A. Lazarevic, and L. Latecki. Incremental local outlier detection for data streams. In *IEEE Symposium on Computational Intelligence and Data Mining*, pages 504–515, 2007.

[26] P. Protopapas, J. Giammarco, L. Faccioli, M. Struble, R. Dave, and C. Alcock. Finding outlier light-curves in catalogs of periodic variable stars. *Monthly Notices of the Royal Astronomical Society*, 369:677–696, 2006.

[27] M. Riedewald, D. Agrawal, A. Abbadi, and F. Korn. Accessing scientific data: Simpler is better. In *Proc. of the 8th International Symposium in Spatial and Temporal Databases*, pages 214–232, 2003.

[28] M. Shapiro. The choice of reference points in best-match file searching. *Commun. ACM*, 20(5):339–343, 1977.

[29] B. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC, 1986.

[30] D. Stoyan. On estimators of the nearest neighbour distance distribution function for stationary point processes. *Metrica*, 64(2):139–150, 2006.

[31] Y. Tao, X. Xiao, and S. Zhou. Mining distance-based outliers from large databases in any metric space. In *KDD '06: Proc. of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 394–403, 2006.

[32] C. Wang and X. Wang. Multilevel filtering for high dimensional nearest neighbor search. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 37–43, 2000.

[33] D. Wang, P. Fortier, H. Michel, and T. Mitsa. Hierarchical agglomerative clustering based t-outlier detection. *6th International Conference on Data Mining - Workshops*, 0:731–738, 2006.

[34] L. Wei, E. Keogh, and X. Xi. SAXually explicit images: Finding unusual shapes. In *Proc. of the 6th International Conference on Data Mining*, pages 711–720, 2006.

[35] X. Xi, E. Keogh, C. Shelton, L. Wei, and C. Ratanamahatana. Fast time series classification using numerosity reduction. In *ICML '06: Proc. of the 23rd international conference on Machine learning*, pages 1033–1040, 2006.