Finding Time Series Motifs in Disk-Resident Data

Abdullah Mueen, Eamonn Keogh
Department of Computer Science and Engineering
University of California, Riverside, USA
{mueen, eamonn}@cs.ucr.edu

Nima Bigdely-Shamlo
Swartz Center for Computational Neuroscience
University of California, San Diego, USA
nima@sccn.ucsd.edu

***Abstract*—Time series motifs are sets of very similar subsequences of a long time series. They are of interest in their own right, and are also used as inputs in several higher-level data mining algorithms including classification, clustering, rule-discovery and summarization. In spite of extensive research in recent years, finding *exact* time series motifs in massive databases is an open problem. Previous efforts either found *approximate* motifs or considered relatively small datasets residing in *main memory*. In this work, we describe for the first time a disk-aware algorithm to find *exact* time series motifs in multi-gigabyte databases which contain on the order of *tens of millions* of time series.**

We have evaluated our algorithm on datasets from diverse areas including medicine, anthropology, computer networking and image processing and show that we can find interesting and meaningful motifs in datasets that are many orders of magnitude larger than anything considered before.

Keywords-time series motif; exact method; disk aware; EEG

I. INTRODUCTION

Time series motifs are sets of very similar subsequences of a long time series, or from a set of time series. As in other domains, (i.e. text, DNA, video [3]) this approximately repeated structure may be conserved for some reason that is of interest to domain specialists. In addition, time series motifs are also used as inputs in several higher-level data mining algorithms, including classification [19], clustering and rule-discovery. Since their formalization in 2002, time series motifs have been used to solve problems in domains as diverse as human motion analyses, medicine, entertainment, biology, telepresence and weather prediction.

In spite of extensive research in recent years [4][9][19][28], finding *exact* time series motifs in massive databases is an open problem. Previous efforts either found *approximate* motifs or considered relatively small datasets residing in *main memory* (or in most cases, both). In this work, we describe for the first time a disk–aware algorithm to find *exact* time series motifs in multi-gigabyte databases containing *tens of millions* of time series. As we shall show, our algorithm allows us to tackle problems previously considered intractable, for example finding near duplicates in a dataset of forty-million images.

The rest of this paper is organized as follows. In Section II we review related work. In Section III we introduce the necessary notation and background to allow the formal description of our algorithm in Section IV. Sections V and VI empirically evaluate the scalability and utility of our ideas, and we conclude with a discussion of future work in Section VII.

II. RELATED WORKS

Many of the methods for time series motif discovery are based on searching a discrete approximation of the time series, inspired by and leveraging off the rich literature of motif discovery in discrete data such as DNA sequences [4][22][9][16][29][24]. Discrete representations of the real-valued data must introduce some level of *approximation* in the motifs discovered by these methods. In contrast, we are interested in finding *exact* motifs with respect to the raw time series. It has long been held that the exact motif discovery is intractable even for datasets residing in main memory. In a recent work the current authors have shown that motif discovery is tractable for large in-core datasets [19]; however, in this work we plan to show that motif discovery can be made tractable even for massive disk resident datasets.

The literature is replete with different ways of defining time series "motifs." Motifs are defined and categorized using their support, distance, cardinality, length, dimension, underlying similarity measure, etc. Motifs may be restricted to have a minimum count of participating similar subsequences [4][9] or may only be a single closest pair [19]. Motifs may also be restricted to have a distance lower than a threshold [4][9][28] or restricted to have a minimum density [17]. Most of the methods find fixed length motifs [4][9][19][28], while there are a handful of methods for variable length motifs [17][29][25]. Multidimensional motifs and subdimensional motifs are defined [16] and heuristic methods to find them are explored in [17][16][29]. Depending on the domain in question, the distance measures used in motif discovery can be specialized, such as allowing for "don't cares" to increase tolerance to noise [4][24]. In this paper we explicitly choose to consider only the simplest definition of a time series motif, which is the closest pair of time series. Since virtually all of the above approaches can be trivially calculated with inconsequential overhead using the closest pair as a *seed*, we believe the closest pair is *the* core operation in motif discovery. Therefore, we ignore other definitions for brevity and simplicity of exposition.

To the best of our knowledge, the closest-pair/time series motif problem in high dimensional disk resident data has not been addressed. There has been significant work on spatial closest-pair queries [20][5]. These algorithms used indexing techniques such as R-tree and R*-tree which have the problem of high creation and maintenance cost for multidimensional data [12]. A possible approach could be to use high dimensional self similarity join algorithms [12]. If the data in hand is joined with itself with a low similarity

threshold we would get a motif set, which could be quickly refined to find the true closest pair. Indeed, [12] does consider (an approximation of) stock market time series as one of their examples. However, the datasets we wish to consider in this work have three orders of magnitude more objects than any of the datasets considered in [5][12][20]. More critically, the dimensionality of our data is up to two orders of magnitude larger, and ultra-high dimensionality is the death knoll for R-tree based algorithms.

It has long been known that the closest pair problem in multidimensional data has a lower bound of $\Omega(n \log n)$. The optimal algorithm is derived from the divide-and-conquer paradigm and exploits two properties of virtually all real datasets: sparsity of the data and the ability to select a "good" cut plane [1]. This algorithm recursively divides the data by a cut plane. At each step it projects the data to the cut plane to reduce the dimensionality by one and then solve the subproblems in the lower dimensional space. Unfortunately the optimal algorithm hides a very high constant factor in the complexity expression, which is of the order of 2^d. In addition, the large worst-case memory requirement and essentiality random data accesses made the algorithm impractical for disk-resident applications.

In this paper we employ a bottom-up search algorithm that simulates the merge steps of the divide-and-conquer approach. Our contribution is that we created an algorithm whose worst-case memory and I/O overheads are practical for implementation on *very* large-scale databases. The key difference with the optimal algorithm that makes our algorithm amenable for large databases is that we divide the data *without* reducing the number of dimensions and *without* changing the data order at any divide step. This allows us to do a relatively small number of batched sequential accesses, rather than a huge number of random accesses. As we shall see, this can make a three- to four-order-of- magnitude difference in the time it takes to find the motifs on disk-resident datasets.

III. NOTATION AND BACKGROUND

As described in the previous section we focus on the simplest "core" definition of time series motifs. In this section we define the terms formally.

Definition 1: *A time series* is a sequence $T=(t_1,t_2,...,t_m)$ of m real valued numbers.

The sequence of real valued numbers (t_i) is generally a temporal ordering. Other well-defined ordering such as shapes [28], spectrographs, handwritten text, etc. can also be fruitfully considered as "time series." As with most time series data mining algorithms, we are interested in local, not global properties of the time series and therefore, we need to extract subsequences from it.

Definition 2: A *subsequence* of length n of a time series $T=(t_1,t_2,...,t_m)$ is a time series $T_{i,n} = (t_i,t_{i+1},...,t_{i+n-1})$ for $1 \le i \le m-n+1$.

A time series of length m has $m-n+1$ subsequences of length n. We naturally expect that adjacent subsequences will be similar to each other; these subsequences are known as trivial matches [4]. However, subsequences which are similar to each other, yet at least some minimum value w apart suggest a recurring pattern, an idea we formalize as a time series motif.

Definition 3: The *time series motif* is a pair of subsequences $\{T_{i,n}, T_{j,n}\}$ of a long time series T of length m that is the most similar. More formally, $\forall a,b,i,j$ the pair $\{T_{i,n}, T_{j,n}\}$ is the time series motif iff $\text{dist}(T_{i,n}, T_{j,n}) \le \text{dist}(T_{a,n}, T_{b,n})$, $|i-j| \ge w$ and $|a-b| \ge w$ for $w > 0$.

Note the inclusion of the separation window w. This means that a motif must contain subsequences separated by at least w positions. The reason behind this separation constraint is to prevent trivial matches from being reported as motif [22]. Thus, the total number of possible motif pairs is exactly $\frac{(m-n-w+1)(m-n-w)}{2}$.

There are a number of distance measures to capture the notion of similarity between subsequences. In [7] and elsewhere it has been empirically shown that simple Euclidean distance is competitive or superior to many of the complex distance measures and has the very important triangular inequality property. Note, however, that our method can work with any distance measure that is *metric*. Additional reasons for using Euclidean distance are that it is parameter-free and its calculation allows many optimizations, including the classic trick of *early abandoning* [19].

One potential problem with Euclidean distance is that it is very sensitive to offset and scale (amplitude). Two subsequences of a time series may be very similar but at different offsets and/or scales, and thus report a larger distance than warranted. Normalization before comparison helps to mitigate this effect. To avoid renormalizing subsequences multiple times, we compute and store all of the $(m-n+1)$ normalized subsequences just once, store them in a database of time series and use these normalized sequences when computing the distances.

Definition 4: A *time series database* is an unordered set of normalized time series stored in one or multiple disk blocks of fixed or varying sizes.

Although the most typical applications of motif discovery involve searching subsequences of a long time series, one other possibility is that we may simply have m individual and independent time series to examine, such as m gene expression profiles or m star light curves. With the exception of the minor overhead of keeping track of the trivial matches [4] in the former case, our algorithm is agnostic to which situation produces the time series and it assumes to have a *time series database* as the input and to output the *time series motif* found in the database.

As noted earlier, there are many other definitions of time series motifs [14][28][25][16]. In general, these definitions impose conditions on two major features of a set of subsequences: similarity and support. We argue that all such conditions can easily be obtained with a single pass through the data using our definition as a seed. As such the definition above is *the* core task of motif discovery.

For meaningful motif discovery the motif pair should be significantly more similar to each other than one would expect in a random database. In this work we gloss over the

problem of assessing significance, other than to show that the motifs have an interpretation in the domains in question. In the next section we describe an algorithm which allows us to efficiently find the motifs in massive time series databases.

IV. Our Algorithm

A set of time series of length n can be thought of as a set of points in n-dimensional space. Finding the time series motif is then equivalent to finding the pair of points having the minimum possible distance between any two points. Before describing the general algorithm in detail, we present the key ideas of the algorithm with a toy example in 2D.

A. A Detailed Intuition of Our Algorithm

For this example, we will consider a set of 24 points in 2D space. In Figure 1A the dataset is shown to scale. Each point is annotated by an id beside it. A moment's inspection will reveal that the closest pair of points is {4,9}. We assume that a disk block can store at most three points (i.e. their co-ordinates) and their ids. So the dataset is stored in the disk in eight blocks.

We begin by randomly choosing a reference point ***r*** (see Figure 1A). We compute the distance of each data point from ***r*** and sort all such distances in ascending order. As the data is on the disk, any efficient external sorting algorithm can be used for this purpose [18][21]. A snap shot of the database after sorting is shown in Figure 1B. Note that our closest pair of points is separated in two different blocks. Point 4 is in the fourth block and point 9 is in the fifth block.

Geometrically, this sorting step can be viewed as projecting the database on one line by rotating all of the points about ***r*** and stopping when every point is on that line. We call this line the *order line* since it holds all of the points in the increasing order of their distances from ***r***. The *order line* shown in Figure 1C begins at the top, representing a distance of 0 from ***r*** and continues downward to a distance of infinity. Note that the *order line* shown in Figure 1C is representative, but does not strictly conform to the scale and relative distances of Figure 1A. Data points residing in the same block after the sorting step are consecutive points in the *order line* and thus, each block has its own interval in the *order line*. In Figure 1C the block intervals are shown beside the *order line*. Note that, up to this point, we have not compared any pairs of data points. The *search* for the closest pair (i.e. comparisons of pairs of points) will be done on this representation of the data.

Our algorithm is based upon the following observation. If two points are close in the original space, they must also be close in the *order line*. Unfortunately, the opposite is not true; two points which are very far apart in the original space might be very close in the *order line*. Our algorithm can be seen as an efficient way to weed out these false positives, leaving just the true motif.

As alluded to earlier, we search the database in a bottom-up fashion. At each iteration we partition the database into consecutive groups. We start with the smallest groups of size 1 (i.e. one data point) and iteratively double the group size (i.e. 2,4,8,…).

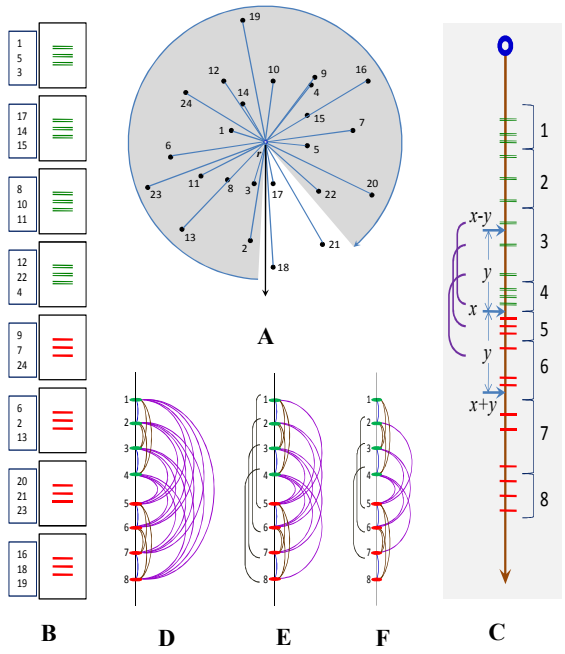


Figure 1: (A) A sample database of 24 points. (B) Disk blocks containing the points sorted in the order of the distances from *r*. The numbers on the left are the ids. (C) All points projected on the *order line*. (D) A portion of an *order line* for a block of 8 points. (E) After pruning by a current motif distance of 4.0 units. (F) After pruning by 3.0 units.

At each iteration we take pairs of disjoint consecutive groups one at a time and compare all pairs of points that span those two groups. Figure 1D shows a contrived segment of an *order line* unrelated to our running example, where a block of eight points is shown. An arc in this figure represents a comparison between two points. The closest four arcs to the *order line* {(1,2),(3,4),(5,6),(7,8)} are computed when the group size is 1. The following eight arcs {(1,3),(1,4),(2,3),(2,4), (5,7),(5,8),(6,7),(6,8)} are computed when the group size is 2 and the rightmost sixteen arcs are computed when the group size is 4. Note that each group is compared with one of its neighbors in the *order line* at each iteration. After the in-block searches are over, we move to search across blocks in the same way. We start by searching across disjoint consecutive pairs of blocks and continually increasing the size of groups like 2 blocks, 4 blocks, 8 blocks, and so on. Here we encounter the issue of accessing the disk blocks efficiently, which is discussed later.

As described thus far, this is clearly a brute force algorithm that will eventually compare all possible pairs of objects. However, we can now explain how the *order line* helps to prune the vast majority of the calculations.

Assume A and B are two objects, and B lies beyond A in the *order line*; i.e. $dist(A,\mathbf{r}) \leq dist(B,\mathbf{r})$. By the triangular inequality we know that $dist(A,B) \geq dist(B,\mathbf{r}) - dist(A,\mathbf{r})$. But $dist(B,\mathbf{r}) - dist(A,\mathbf{r})$ is the distance between A and B in the *order line*. Thus, the distance between two points in the *order line* is a lower bound on their true distance. Therefore, at some point during the search, if we know that the closest pair found so far has a distance of y, we can safely ignore all

pairs of points that are more than *y* apart on the *order line*. For example, in Figure 1E, if we know that the distance between the best pair discovered so far is 4.0 units, we can prune off comparisons between points {(1,6),(1,7),(1,8),(2,7),(2,8),(3,8)}, since they are more than 4.0 units apart on the *order line.* Similarly, if the best pair discovered so far had an even tighter distance of 3.0 units, we would have pruned off four more pairs. (see Figure 1F).

More critically, the *order line* also helps to minimize the number of disk accesses while searching across blocks. Let us assume that we are done searching all possible pairs (i.e. inside and across blocks) in the top four blocks and also in the bottom four blocks (see Figure 1B). Let us further assume that the smaller of the minimum distances found in each of the two halves is *y*. Let *x* be the distance of the cut point between the two halves from ***r***. Now, all of the points lying within the interval (*x-y,x*] in the *order line* may need to be compared with at least one point from the interval [*x,x+y*). Points outside these two intervals can safely be ignored because they are more than *y* apart from the points in the other half. Since in Figure 1C the interval (*x-y,x*] overlaps with the intervals of blocks 3 and 4 and the interval [*x,x+y*) overlaps with the intervals of blocks 5 and 6, we need to search points *across* block pairs {3,5},{3,6},{4,5} and {4,6}. Note that we would have been forced to search all 16 possible block pairs if there were no *order line*.

Given that we are assuming the database will not fit in the main memory, the question arises as to *how we should load these block pairs when the memory is very small.* In this work, we assume the most restrictive case, where we have just the memory available to store exactly two blocks. Therefore, we need to bring the above four block pairs {{3,5},{3,6},{4,5},{4,6}} one at a time. The number of I/O operations depends on the order in which the block pairs are brought into the memory. For example, if we search the above four pairs of blocks in that order, we would need exactly six block I/Os: two for the first pair, one for the second pair since block 3 is already in the memory, two for the third pair and one for the last pair since block 4 is already in the memory. If we choose the order {{3,5},{**4**,6},{**3**,6},{4,5}} we would need seven I/Os. Similarly, if we chose the order {{3,5},{4,5},{**4**,6},{**3**,6}}, we would need five I/Os. In the latter two cases there are reverse scans; a block (**4**) is replaced by a previous block (**3**) in the *order line.* We will avoid reverse scans and avail of sequential loading of the blocks to get maximum help from the *order line*.

Let us consider *how the order line helps in pruning pairs across blocks*. When we have two blocks in the memory, we need to compare each point in one block to each point in the other block. In our running example, during the search across the block pair {3,6}, the first and second data points in block 3 (i.e. 8 and 10 in the database) have distances larger than *y* to any of the points in block 6 in the *order line* (see Figure 1C). The third point (i.e. 11) in block 3 has only the first point (i.e. 6) in block 6 within *y* in the *order line*. Thus, for block pair {3,6}, instead of computing distances for all nine pairs of data points we would need to compute the distance for only one pair, <11,6>.

At this point, we have informally described *how a special ordering of the data can reduce block I/Os as well as reduce pair- wise distance computations*. With this background, we hope the otherwise daunting detail of the technical description in the next section will be less intimidating.

B. A Formal Description of Our Algorithm

For the ease of description, we assume the number of blocks (***N***) and the block size (***m***) are restricted to be powers of two. We also assume that all blocks are of the same size and that the main memory stores only two disk blocks with a small amount of extra space for the necessary data structures. Readers should note that all variable and method names are in *italics* and globally accessible elements are in **bold**. Shaded lines denote the steps for the pruning of pairs from being selected or compared. We call our algorithm **DAME,** *Disk Aware Motif Enumeration.*

Our algorithm is logically divided into subroutines with different high-level tasks. The main method that employs the bottom-up search on the blocks is *DAME_Motif*. The input to this method is a set of blocks ***B*** which contains every subsequence of a long time series or a set of independent time series. Each time series is associated with an id used for finding its location back in the original data. Individual time series are assumed to be *z*-normalized. If they are not, this is done in the sorting step.

DAME_Motif first chooses ***R*** random time series as reference points from the database and stores them in ***Dref***. These reference time series can be from the same block, allowing them to be chosen in a single disk access. It then sorts the entire database, residing on multiple blocks, according to the distances from the first of the random references named as ***r***. The reason for choosing ***R*** random reference time series/points will be explained shortly. The *computeInterval* method at line 5 computes the intervals of the sorted blocks. For example, if s_i and e_i are respectively the smallest and largest of the distances from ***r*** to any time series in B_i, then $[s_i,e_i]$ is the block interval of B_i. Computing these intervals is done during the sorting phase, which saves a few disk accesses. Lines 7-14 detail the bottom-up search strategy. Let *t* be the group size, which is initialized to one, and iteratively doubled until it reaches $\frac{N}{2}$.

Procedure	*DAME_Motif* (***B***)
1	***bsf*** ← INF, $L_1 \leftarrow -1$, $L_2 \leftarrow -1$
2	***Dref***←Randomly pick ***R*** time series
3	***r*** ←$Dref_1$
4	*sort*(***B,r***)
5	***s,e***←*computeInterval*(***B***)
6	*t*←*1*
7	**while** $t \leq \frac{N}{2}$
8	*top*←*1*
9	**while** *top* < ***N***
10	*mid*←*top*+*t*
11	*bottom*←*top*+*2t*
12	*searchAcrossBlocks*(*top,mid,bottom*)
13	*top*←*bottom*
14	*t*←*2t*

For each value of *t,* pairs of time series across pairs of successive *t*-groups are searched using the *searchAcrossBlock* method.

The *searchAcrossBlocks* method searches for the closest pair across the partitions [*top,mid*) and [*mid,bottom*). The order of loading blocks is straightforward (lines 1 and 8). The method sequentially loads one block from the top partition, and for each of them it loads all of the blocks from the bottom partition one at a time (lines 4 and 11). *D1* and *D2* are the two memory blocks and are dedicated for the top and bottom partitions, respectively. A block is loaded to one of the memory blocks by the *load* method. *load* reads and stores the time series and computes the distances from the references.

DAME_Motif and all subroutines maintain a variable ***bsf*** (*best so far*) that holds the minimum distance discovered up to the current point of search. We define the distance between two blocks p and q by $\boldsymbol{s_q}$-$\boldsymbol{e_p}$ if $p<q$. Lines 2-3 encode the fact that if block p from the top partition is more than ***bsf*** from the first block (*mid*) of the bottom partition, then p cannot be within ***bsf*** of any other blocks in the bottom partition. Lines 9-10 encode the fact that if block q from the bottom partition is not within ***bsf*** of block p from the top partition, then none of the blocks after q can be within ***bsf*** of p. These are the pruning steps of *DAME* that prune out entire blocks.

Lines 5-6 and 12-13 check if the search is at the bottom-most level. At that level, *searchInBlock* is used to search within the loaded block. Lines 14-15 do the selection of pairs by taking one time series from each of the blocks. Note the use of *istart* at lines 14 and 19. *istart* is the index of the last object of block p which finds an object in q located farther than ***bsf*** in the *order line*. Therefore, the objects indexed by $i \leq istart$ do not need to be compared to the objects in the blocks next to q. So, the next time series to *istart* is the starting position in p when pairs across p and the next of q are searched. For all the pairs that have escaped from the pruning steps, the *update* method is called.

```
Procedure   searchAcrossBlocks (top,mid,bottom)
1   for p←top to mid–1
2       if s_mid – e_p ≥ bsf and mid–top != 1
3           continue
4       D1,Dist1←load(B_p)
5       if mid–top = 1
6           searchInBlock(D1,Dist1)
7       istart←0
8       for q←mid to bottom–1
9           if s_q – e_p ≥ bsf and bottom–mid != 1
10              break
11          D2,Dist2←load(B_q)
12          if bottom–mid = 1
13              searchInBlock (D2,Dist2)
14          for i←istart+1 to m
15              for j←1 to m
16                  if Dist1_{1,i} – Dist2_{1,i} < bsf
17                      update(D1,D2,Dist1,Dist2,i,j)
18                  else
19                      istart←i
20                      break
```

```
Procedure   load(b)
1   D←read(b)
2   for i←1 to R
3       for j←1 to m
4           Dist_{i,j}←dist(Dref_i,D_j)
5   return D,Dist
```

The method *searchInBlock* is used to search within a block. This method employs the same basic bottom-up search strategy as the *DAME_Motif*, but is simpler due to the absence of a memory hierarchy. Similar to the *searchAcrossBlocks* method, the search across partitions is done by simple sequential matching with two nested loops. The pruning step at lines 11-12 terminates the inner loop over the bottom partition at the j^{th} object which is the first to have a distance larger than ***bsf*** in the *order line* from the i^{th} object. Just as with *searchAcrossBlocks* method, every pair that has escaped from pruning is given to the *update* method for further consideration.

```
Procedure   searchInBlock(D,Dist)
1   t←1
2   while t ≤ m/2
3       top←1
4       while top < m
5           mid←top+t
6           bottom←top+2t
7           for i←top to mid–1
8               for j←mid to bottom–1
9                   if Dist_{1,i} – Dist_{1,j} < bsf
10                      update(D,D,Dist,Dist,i,j)
11                  else
12                      break
13          top←bottom
14      t←2t
```

The *update* method does the distance computations and updates the ***bsf*** and the motif ids (i.e. $\boldsymbol{L_1}$ and $\boldsymbol{L_2}$). The pruning steps described in the earlier methods essentially try to prune some pairs from being considered as potential motifs. When a potential pair is handed over to *update,* it also tries to avoid the costly distance computation for a pair. In Section A, it is shown that distances from a *single* reference point ***r*** provides a lower bound on the true distance between a pair. In *update,* distances from multiple (***R***) reference points computed during *load*s are used to get ***R*** *lower_bound*s, and *update* rejects distance computation as soon as it finds a *lower_bound* larger than ***bsf***. Although ***R*** is a preset parameter like ***N*** and ***m***, its value is not very critical to the performance of the algorithm. Any value from five to sixty produces near identical speedup, regardless of the data ***R*** [19].

```
Procedure   update(D1,D2,Dist1,Dist2,x,y)
1   reject←false
2   for i←2 to R
3       lower_bound←|Dist1_{i,x} – Dist2_{i,v}|
4       if lower_bound > bsf
5           reject←true, break
6   if reject = false and trivial(D1_x,D2_y)= false
7       if dist(D1_x,D2_y) < bsf
8           bsf ←dist(D1_x,D2_y)
9           L_1← id(D1_x), L_2←id(D2_y)
```

Note that the first reference time series ***r*** is special in that it is used to create the *order line*. The rest of the reference points are used only to prune off distance computations. Also note the test for trivial matches [4][19] at line 6. Here, a pair of time series is not allowed to be considered if they overlapped in the original time series from which they were extracted.

C. Correctness of DAME

The correctness of the algorithm can be described by the following two lemmas. Note that pruning steps are marked by the shaded regions in the pseudocode of the previous section.

Lemma 1: *The bottom-up search compares all possible pairs if the pruning steps are removed.*

Proof: In *searchInBlock* we have exactly $\boldsymbol{m}$ time series in the memory block *D*. The bottom-up search does two-way merging at all levels for partition sizes $t=1, 2, 4, \ldots, \frac{\boldsymbol{m}}{2}$ successively. For partitions of size *t,* while doing the two-way merge, the number of times *update* is called is $\frac{\boldsymbol{m}\,t}{2}$. Therefore, the total number of calls to *update* is $\{2^0+2^1+2^2+\ldots+2^{x-1}\}\ \frac{\boldsymbol{m}}{2}$, where $\boldsymbol{m}=2^x$. This sum exactly equals the total number of possible pairs $\frac{\boldsymbol{m}(\boldsymbol{m}-1)}{2}$. Similarly, *DAME_Motif* and *searchAcrossBlocks* together do the rest of the search for partition sizes $t=\boldsymbol{m}, 2\boldsymbol{m}, 4\boldsymbol{m}, \ldots, \frac{\boldsymbol{N}\boldsymbol{m}}{2}$ to complete the search over all possible pairs. ■

Lemma 2: *Pruning steps ignore pairs safely.*

Proof: Follows from Section B. ■

Before ending the description of the algorithm, we describe the worst-case scenario for *seachAcrossBlocks.* If the motif distance is larger than the spread of the data points in the *order line,* then all possible pairs are compared by *DAME* because no pruning happens in this scenario. Therefore, *DAME* has the worst-case complexity of $O(n^2)$. Note, however, that this situation would require the most pathological arrangement of the data, and hundreds of experiments on dozens of diverse real and synthetic datasets show that average cost is well below n^2.

V. Scalability Experiment

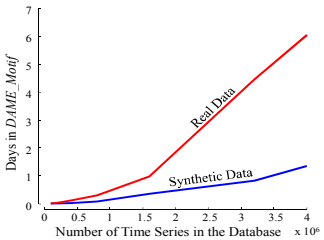
In this section we describe experimental results to demonstrate *DAME*'s scalability and performance. Experiments in sections V.A-V.C are performed in a 2.66GHz Intel Q6700 and the rest of the experiments are performed on an AMD 2.1GHz Turion-X2. We use internal hard drives of 7200 rpm. For the ease of reproducibility, we have built a webpage [30] that contains all of the code, data files for real data, data generators for synthetic data and a spreadsheet of all the numbers used to plot the graphs in this paper. In addition, the webpage has experiments and case studies which we have omitted here due to space limitations.

Note that some of the large-scale experiments we conduct in this section take several days to complete. This is a long time by the standards of typical empirical investigations in data mining conferences; however, we urge the reader to keep in mind the following as they read on:

- Our longest experiment (the "*tiny images*" dataset [26]) looks at 40,000,000 time series and takes 6.5 days to finish. However, a brute force algorithm would take 124 years to produce the same result. Even if we could magically fit all of the data in main memory, and therefore bypass the costly disk accesses, the brute force algorithm would require (40,000,000*39,999,999)/2 Euclidean comparisons, and require 8 years to finish.
- Our largest experiment finds the *exact* motif in 40,000,000 time series. If we sum up the sizes of the largest datasets considered in papers [25][19][9][4][22] which find only *approximate* motifs, they would only sum to 400,000. So we are considering datasets of at least two orders of magnitude larger than anything attempted before.
- In many of the domains we are interested in, practitioners have spent weeks, months or even years collecting the data. For example, the "*tiny images*" dataset [26] took eight months to collect on a dedicated machine running 24 hours a day [8]. Given the huge efforts in both money and time to collect the data, we believe that the practitioners will be more than willing to spend a few days to uncover hidden knowledge from it.

A. Sanity Check on Large Databases

We begin with an experiment on random walk data. Random walk data is commonly used to evaluate time series algorithms, and it is an interesting contrast to the real data (considered below), since there is no reason to expect a particularly close motif to exist. We generate a database of four million random walks in eighty disk blocks. Each block is identical in size (400MB) and can store 50,000 random walks of length 1024. The database spans more than 320GB of hard drive space. We find the closest pair of random walks using *DAME* on the first 2, 4, 8, 16, 32, 64 and 80 blocks of this database. Figure 2:*left* shows the execution times against the database size in the number of random walks.



Algorithm	Largest Dataset Tested (thousands)	Time for the Largest Dataset	Estimated Time for 4.0 million
NoAdditionalStorage	200	4.82 days	5.28 years
CompletelyInDisk	200	1.50 days	1.65 years
CompletelyInMemory	100	35 minutes	37.8 days
DAME (Synthetic Data)	4,000	1.35 days	--

Figure 2: (*left*) Execution times in days on random walks and EOG data. (*right*) Comparison of the three different versions of brute-force algorithm with *DAME*.

In another independent experiment we use *DAME* on a very long and highly oversampled real time series (EOG trace, cf. section VI.C) to find a subsequence-motif of length 1024. We start with a segment of this long time series created by taking the first 100,000 data points, and iteratively double the segment-sizes by taking the first 0.2, 0.4, 0.8, 1.6, 3.2 and 4.0 million data points. For each of these segments, we run *DAME* with blocks of 400MBs, each containing 50,000 time series, as in the previous

experiment. Figure 2:*left* also shows the execution times against the lengths of the segments. Because of the oversampled time series, the extra "noise" makes the motif distance larger than it otherwise would be, making the ***bsf*** larger and therefore reducing the effectiveness of pruning. This is why *DAME* takes longer to find a motif of the same length than in a random-walk database of similar size.

Since no other algorithm is reported to find motifs *exactly* on such *large* databases, we have implemented three possible versions of the naïve brute force algorithm to compare with *DAME*:

- *CompletelyInMemory*: The database is in the main memory.
- *CompletelyInDisk*: The database is in the disk and memory is free enough to store only two disk blocks and negligible data structures for comparisons.
- *NoAdditionalStorage*: There is no database. Only the base time series is stored in the main memory. Subsequences must be normalized again and again every time they are extracted.

Figure 2:*right*. tabulates the performances of the above three algorithms as well as DAME's. *CompletelyInMemory* algorithm has been run until memory allocation request is honored; therefore, estimated time for four million time series is *never* attainable, as it would need 320GB of main memory. The other two algorithms have been executed until time becomes critical. The estimated execution times demonstrates that *DAME* is the *only* tractable algorithm for *exact* discovery of time series motifs.

B. *Performance for Different Block Sizes*

As *DAME* has a specific order of disk access, we must show how the performance varies with the size of the disk blocks. We have taken the first one million random walks from the previous section and created six databases with different block sizes. The sizes we test are 40, 80, 160, 240, 320 and 400 MBs containing 5, 10, 20, 30, 40 and 50 thousand random walks, respectively. Since the size of the blocks is changed, the number of blocks also changes to accommodate one million time series. We measure the time for both I/O and CPU separately for *DAME_Motif* (Figure 3:*left*) and for *searchAcrossBlocks* (Figure 3:*right*).

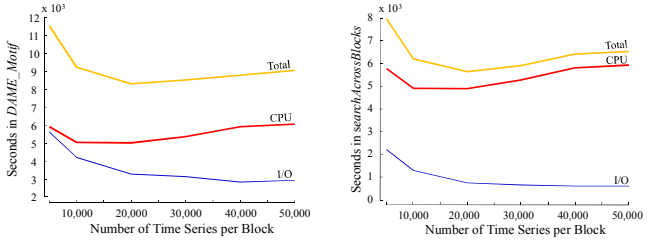


Figure 3: Total execution times with CPU and I/O components recorded on one million random walks for different block sizes (*left*) for the *DAME_Motif* method and (*right*) for the *searchAcrossBlocks* method.

Figure 3:*left* shows that I/O time decreases as the size of the blocks gets larger and the number of blocks decreases. On the other hand, the CPU time is worse for very low or very high block sizes. Ideally it should be constant, as we use the same set of random walks. The two end deviations are caused by two effects: When blocks are smaller, block intervals become smaller compared to the closest pair distance, and therefore, almost every point is compared to points from multiple blocks and essentially *istart* loses its role. When the blocks become larger, consecutive pairs on the *order line* in later blocks are searched after the distant pairs on the *order line* in an earlier block. Therefore, ***bsf*** decreases at a slower rate for larger block sizes.

Figure 3:*right* shows that the search for a motif using the *order line* is a CPU-bound process since the gap between CPU time and I/O time is large, and any effort to minimize the number of disk loads by altering the loading order from the current sequential one will make little difference in the total execution time.

C. *Performance for Different Motif Lengths*

To explore the effect of the motif length (i.e. dimensionality) on performance, we test *DAME* for different motif lengths. Recall that the motif length is the only user-defined parameter. We use the first one-million random walks from Section A. They are stored in 20 blocks of 50,000 random walks, each of length 1024. For this experiment, we iteratively double the motif length from 32 to 1024. For each length x, we use only the first x temporal points from every random walk in the database. Figure 4:*left* shows the result, where all the points are averaged over five runs.

The linear plot demonstrates that *DAME* is free of any exponential constant (2^d) in the complexity expression, as in the optimal algorithm. The linear increase in time is due to the distance computation, which needs a complete scan of the data. Note the gentle slope indicating a sub-linear scaling factor. This is because longer motifs allow greater benefit from *early abandoning* [19].

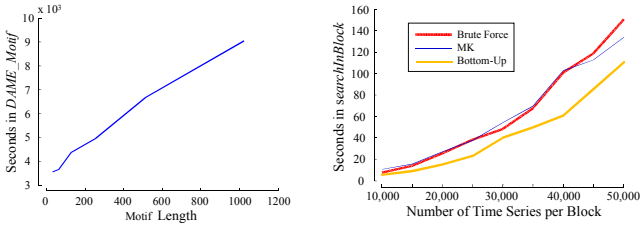


Figure 4: (left) Execution times on one million random walks of different lengths. (right) Comparison of in-memory search methods.

D. *In-Memory Search Options*

While searching within a memory block, *DAME* does a bottom-up search starting with pairs of consecutive time series, continuing until it covers all possible pairs. Thus, *DAME* has a consistent search hierarchy from within blocks to between blocks. There are only two other *exact* methods we could have used, the classic brute-force algorithm or the recently published MK algorithm [19]. We measure the time that each of these methods takes to search in-memory blocks of different sizes, and experiment on different sizes of blocks ranging from 10,000 to 50,000 random walks of length 1024. For all of the experiments, the databases are

four times the block sizes and values are averaged over ten runs.

From the Figure 4:*right*, brute force search and the MK algorithm perform similarly. The reason for MK not performing better than brute force here is worth considering. MK performs well when the database has a wide range and uniform variability on the *order line*. Since the database in this experiment is four times the block size, the range of distances for one block is about one fourth of what it would be in an independent one-block database of random walks. Therefore, MK cannot perform better than brute force. The bottom-up search performs best because it does not depend on the distribution of distances from the reference point, and moreover, prunes off a significant part of the distance computations.

E. *Lower Bound Options*

In our algorithm we compute distances from ***R*** reference points to all of the objects. In the *update* method, we use each reference point one at a time to compute a lower bound on the distance between a pair and check to see if the lower bound is greater than the current best. This lower bound is a simple application of the triangular inequality computed by circularly projecting the pair of objects onto any line that goes through the participating reference point. We call this idea the "linear bound" for clarity in the following discussion. Since we pre-compute all of the distances from ***R*** reference points, one may think about getting a tighter lower bound by combining these referenced distances. In the simplest case, to find a "planar bound" for a pair of points using *two* reference points, we can project both the points (x and y) onto any 2D plane, where two reference points (r_1 and r_2) reside, by a circular motion about the axis connecting the reference points.

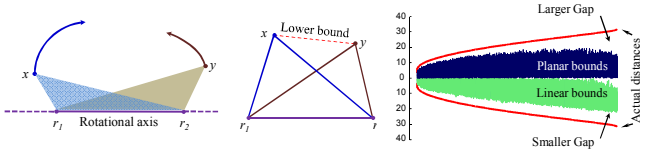


Figure 5: (*left*) Two points x and y are projected on a plane by a rotation around the axis joining two reference points r_1 and r_2. (*mid*) Known distances and the *lower bound* after the projection. (*right*) Planar and linear bound are plotted against true distances for 40,000 random pairs.

After that, simple 2D geometry is needed to compute the lower bound (dashed line) using five other pre-computed distances (solid lines in Figure 5:*mid*).

We have computed both the bounds on one million pairs of time series of length 256. In 56% of the pairs, planar bounds are larger than linear bounds. Intuitively it seems from this value that the planar bound is tighter. But the true picture is more complex. The average value of linear bounds is 30% larger than that of planar bounds and standard deviation of linear bounds is 37% larger than that of planar bounds. In Figure 5:*right,* it is clear that the linear bound is significantly tighter than the planar one when the actual distances between pairs are larger. Moreover, the planar bound is complex to compute compared to a simple subtraction in the case of a linear bound. Therefore, we opt to use the linear bound in *update* to prune off distance computations.

VI. EXPERIMENTAL CASE STUDIES

In this section we consider several case studies to demonstrate the utility of motifs in solving real-world problems.

A. *Motifs for Brain-Computer Interfaces*

Recent advances in computer technology make sufficient computing power readily available to collect data from a large number of scalp electroencephalographic (EEG) sensors and to perform sophisticated spatiotemporal signal processing in near-real time. A primary focus of recent work in this direction is to create brain-computer interface (BCI) systems.

In this case study, we apply motif analysis methods to data recorded during a target recognition EEG experiment [2]. The goal of this experiment is to create a real-time EEG classification system that can detect "flickers of recognition" of the target in a rapid series of images and help Intelligence Analysts find targets of interest among a vast amount of satellite imagery. Each subject participates in two sessions: a training session in which EEG and behavior data are recorded to create a classifier and a test session in which EEG data is classified in real time to find targets. Only the training session data is discussed here.

In this experiment, overlapping small image clips from a publicly available satellite image of London are shown to a subject in 4.1 second bursts comprised of 49 images at the rate of 12 per second. Clear airplane targets are added to some of these clips such that each burst contains either zero (40%) or one (60%) target clip. To clearly distinguish target and non-target clips, only complete airplane target images are added, though they can appear anywhere and at any angle near the center of the clip. After viewing the RSVP burst the subject is asked to indicate whether or not he/she has detected a plane in the burst clips, by pressing one of two (yes/no) finger buttons.

In training sessions only, visual error/correct feedback is provided. The training session comprises of 504 RSVP bursts organized into 72 bouts with a self-paced break after each bout. In all, each session thus includes 290 target and 24,104 non-target image presentations. The EEG from 256 scalp electrodes at 256 Hz and manual responses are recorded during each session.

Each EEG electrode receives a linear combination of electric potentials generated from different sources in and outside the brain. To separate these signals, an extended-infomax Independent Component Analysis (ICA) algorithm [13][6] is applied to preprocessed data from 127 electrodes to obtain about 127 maximally independent components (ICs). The ICA learns spatial filters in the form of an unmixing matrix separating EEG sensor data into temporally maximally independent processes, most appearing to predominantly represent the contribution to the scalp data of one brain EEG or non-brain artifact source, respectively.

It is known that among ICs representing brain signals, some show changes in activity after the subject detects a target. However, the exact relationships are currently unknown. In an ongoing project, we attempt to see if the occurrences of motifs are correlated with these changes.

We use *DAME* to discover motifs of length 600 ms (153 data points), on IC activity from one second *before* until 1.5 second *after* image presentation. Figure 6 shows the discovered motif.

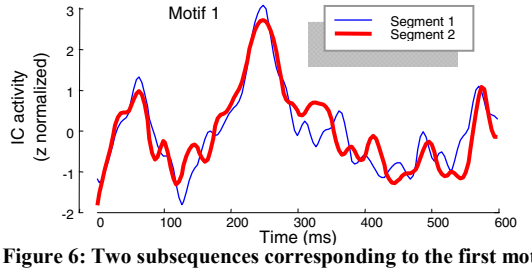


Figure 6: Two subsequences corresponding to the first motif.

Figure 7 shows the start latencies of the first motif for a cluster with a radius of twice the motif distance, i.e. twice the Euclidean distance between the two time series shown in Figure 6. Note that the distribution of these latencies is *highly* concentrated around 100 ms after target presentation. This is significant because no information about the latency of the target has been provided beforehand, and thus the algorithm finds a motif that is highly predictive of the latency of the target.

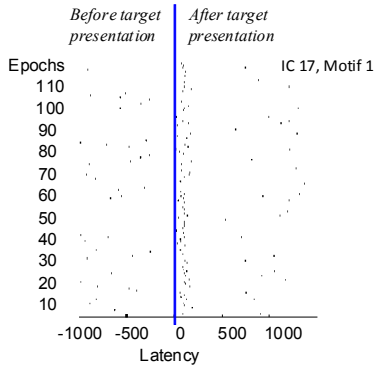


Figure 7: Motif 1 start latencies in epochs.

Motifs can also be used as classifier features. Figure 8 shows the Euclidean distance between the first motif and subsequences starting at all latencies in each epoch. A distinct pattern of decreased distance can be seen in target epochs but not in non-target epochs. If the minimum distance to motif 1 is used as a feature, an area of 0.83 under ROC curve can be achieved in epochs shown in Figure 8.

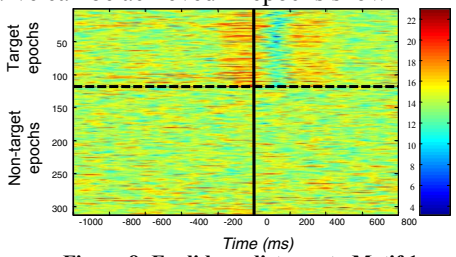


Figure 8: Euclidean distance to Motif 1.

B. *Detecting Near-Duplicate Images*

Finding near-duplicate images in an image database can be used to summarize the database, identify forged images and clean out distorted copies of images. If we can convert the two-dimensional ordered images into one-dimensional (possibly unordered) vectors of features, we can use our motif discovery algorithm to find near-duplicate images.

To test this idea, we use the 40 million images from the dataset in [26]. We convert each image to a pseudo time series by concatenating its normalized color histograms for the three primary colors [10]. Thus, the lengths of the "time series" are exactly 768. We run *DAME* on this large set of time series and find 1,719,443 images which have at least one and on average 1.231 duplicates in the same dataset. We also find 542,603 motif images which have at least one *non-identical* image within 0.1 Euclidean distances of them. For this experiment, *DAME* has taken 6.5 days (recall that a brute-force search would take over a century, cf. Section V). In Figure 9, samples from the sets of duplicates and motifs are shown. Subtle differences in the motif pairs can be seen; for example, a small "dot" is present next to the dog's leg in one image but not in the other. The numbers in between image pairs are the ids of the images in the database.

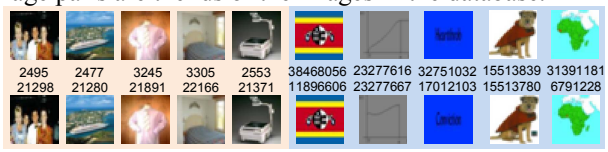


Figure 9: (left) Five identical pairs of images. (right) Five very similar, but non-identical pairs.

C. *Discovering Patterns in Polysomnograms*

In polysomnography, body functions such as pulse rate, brain activity, eye movement, muscle activity, heart rhythm, breathing, etc. are monitored during a patient's sleep cycle. To measure the eye movements an Electrooculogram (EOG) is used. Eye movements do not have any periodic pattern like other physiological measures such as an ECG and respiration. Repeated patterns in the EOG of a sleeping person have attracted much interest in the past because of their potential relation to dream states. We use *DAME* to find a repeated pattern in the EOG traces from the "Sleep Heart Health Study Polysomnography Database" [10]. The trace has about 8,099,500 temporal values at the rate of 250 samples per second. Since the data is oversampled, we downsample it to a time series of 1,012,437 points. A subset of 64 seconds is shown in Figure 10.

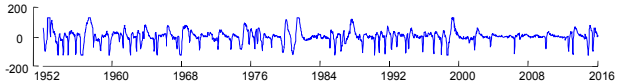


Figure 10: A section of the EOG from the polysomnogram traces.

After a quick review of the data, one can identify that most of the natural patterns are shorter in length (i.e. 1 or 2 seconds) and are visually detectable locally in a single frame. Instead of looking for such shorter patterns, we search for longer patterns of 4.0 seconds long with the hope of finding visually undetectable and less frequent patterns. *DAME* has finished the search in 10.5 hours (brute force

search would take an estimated 3 months) and found two subsequences shown in Figure 11 which have a common pattern, and very unusually this pattern does not appear anywhere else in the trace. Note that the pattern has a plateau in between seconds 1.5 and 2.0, which might be the maximum possible measurement by the EOG machine.

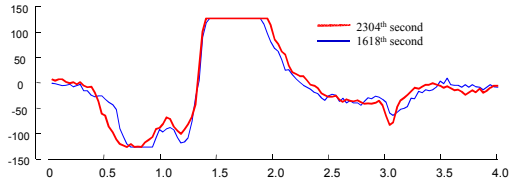


Figure 11: Motif of length 4.0 seconds found in the EOG.

We map these two patterns back to the annotated dataset. Both the subsequences are located at points in the trace just as the person being monitored was going back and forth between arousal and sleep stage 1, which suggests some significance to this pattern.

VII. CONCLUSION AND FUTURE WORK[1]

In this paper we introduced the first scalable algorithm for *exact* motif discovery. Our algorithm can handle databases of the order of *tens of millions* of time series, which is at least two orders of magnitude larger than anything attempted before. We used our algorithm in various domains and discovered significant motifs. To facilitate scalability to the handful of domains that are larger than those considered here (i.e. star light curve catalogs), we plan to consider parallelization, given that the search for different group sizes can easily be delegated to different processors. Because it has recently been shown that DNA can be meaningfully transformed into time series [23], we plan to use DAME to Explore DNA (DAME EDNA), in particular, the DNA of Phalangeroidea (possums). Another avenue of research is to modify the algorithm to find multidimensional motifs in databases of similar scale.

REFERENCES

[1] Bentley, J. L. *Multidimensional divide-and-conquer*, Communications of the ACM, v.23 n.4, p.214-229, 1980.

[2] Bigdely-Shamlo, N., Vankov, A., Ramirez, R. and Makeig, S. *Brain Activity-Based Image Classification from Rapid Serial Visual Presentation.* IEEE Transactions on Neural Systems and Rehabilitation Engineering, 2008, vol. 16, no 4.

[3] Cheung, S. S. and Nguyen, T. P. *Mining Arbitrary-Length Repeated Patterns in Television Broadcast.* ICIP (3) 2005: 181-184.

[4] Chiu, B., Keogh, E. and Lonardi, S. *Probabilistic Discovery of Time Series Motifs.* ACM SIGKDD 2003. pp 493-498..

[5] Corral, A., Manolopoulos, Y., Theodoridis, Y. and Vassilakopoulos, M. *Closest Pair Queries in Spatial Databases*. SIGMOD 2000.

[6] Delorme, A. and Makeig, S. *EEG Changes Accompanying Learning Regulation of the 12-Hz EEG Activity.* IEEE Transactions on Rehabilitation Engineering, 11(2), 2003: 133-136

[7] Ding, H., Trajcevski, G., Scheuermann, P., Wang X. and Keogh E. *Querying and Mining of Time Series Data: Experimental Comparison of Representations and Distance Measures*. VLDB'08.

[8] Fergus, R. *Personal Communication.* Email on 12/28/08

[9] Ferreira, P., Azevedo, P.J., Silva, C. and Brito, R. *Mining Approximate Motifs in Time Series*, Discovery Science, 2006.

[10] Goldberger, A. L., Amaral, L. A. N., Glass, L., Hausdorff, J. M., Ivanov, P. Ch., Mark, R. G., Mietus, J. E., Moody, G. B., Peng, C.-K. and Stanley, H. E. *PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals*. Circulation 101(23) pp. e215-e220, 2000.

[11] Hafner, J., Sawhney, H. et al. *Efficient Color Histogram Indexing for Quadratic Form Distance Functions*, IEEE PAMI, 17(7): 729-736, 1995.

[12] Koudas, N. and Sevcik, K.C. *High Dimensional Similarity Joins: Algorithms and Performance Evaluation.* IEEE TKDE, Volume 12, Issue 1, Pages: 3-18, 2000.

[13] Lee, T., Girolami, M. and Sejnowski, T. J. *Independent Component Analysis Using an Extended Infomax Algorithm for Mixed Subgaussian and Supergaussian Sources.* Neural Computation, Feb. 15, 1999, vol. 11, no. 2, pp. 417-441.

[14] Lin, J., Keogh, E., Lonardi, S., and Patel, P. *Finding Motifs in Time Series*, 2nd Workshop on Temporal Data Mining (KDD'02).

[15] Liu, Z., YU, J. X., Lin, X., Lu, H. and Wang, W. *Locating Motifs in Time-Series Data*. PAKDD. pp. 343-353, 2005.

[16] Minnen, D., Isbell, C.L., Essa, I. and Starner, T. *Detecting Subdimensional Motifs: An Efficient Algorithm for Generalized Multivariate Pattern Discovery.* ICDM 2007.

[17] Minnen, D., Isbell, C.L., Essa, I. and Starner, T. *Discovering Multivariate Motifs Using Subsequence Density Estimation and Greedy Mixture Learning.* 22th Conf. on A.I., 2007.

[18] Motzkin, D. and Hansen C. L. *An Efficient External Sorting with Minimal Space Requirement*, International Journal of Parallel Programming, 11(6), 381-396, 1982.

[19] Mueen, A., Keogh, E., Zhu, Q., Cash, S. and Westover, B. *Exact Discovery of Time Series Motif*. SDM 2009.

[20] Nanopoulos A., Theodoridis Y. and Manolopoulos, Y. *C2P: Clustering based on Closest Pairs.* International Conference on Very Large Data Bases, pp. 331 – 340, 2001.

[21] Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J. and Lomet, D. *Alphasort: A Cache-Sensitive Parallel External Sort*, The VLDB Journal, 4(4), 603-628, October, 1995.

[22] Patel, P., Keogh, E., Lin, J. and Lonardi, S. *Mining Motifs in Massive Time Series Databases.* ICDM, 2002.

[23] Shieh, J. and Keogh E. J. *iSAX: disk-aware mining and indexing of massive time series datasets.* DMKD 19(1): 24-57 (2009).

[24] Simona, R. and Giorgio, T. *Discovering Representative Models in Large Time Series Databases.* International conf. on query answering systems, vol. 3055, 2004: 84-97.

[25] Tang, H. and Liao, S. S., *Discovering Original Motifs with Different Lengths from Time Series Source*. Knowledge-Based Systems, 21(7), pp. 666-671, 2008.

[26] Torralba, A., Fergus, R. and Freeman, W. T. *80 Million Tiny Images: a Large Database for Non-Parametric Object and Scene Recognition*, IEEE PAMI, 30(11), 2008: 1958-1970.

[27] Weber R., Schek, H-J. and Blott, S., *A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces*. VLDB, pp. 194 – 205, 1998.

[28] Yankov, D., Keogh, E., Medina, J., Chiu, B. and Zordan B. *Detecting Motifs under Uniform Scaling*. SIGKDD 2007.

[29] Yoshiki, T., Kazuhisa, I. and Kuniaki, U. *Discovery of Time-Series Motif from Multi-Dimensional Data Based on MDL Principle.* Machine Learning, 58(2-3), pp. 269-300, 2005.

[30] Supporting webpage for this paper http:\\www.cs.ucr.edu\~mueen\DAME\index.html

[1] Dr. Keogh's work was funded by NSF 0803410 and 808770.