UNIVERSITY OF CALIFORNIA
RIVERSIDE

Exact Primitives for Time Series Data Mining

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Abdullah Al Mueen

March 2012

Dissertation Committee:

Dr. Eamonn Keogh, Chairperson
Dr. Vassilis Tsotras
Dr. Stefano Lonardi

The Dissertation of Abdullah Al Mueen is approved:

_____

_____

_____

Committee Chairperson

University of California, Riverside

Acknowledgements

I would like to take this opportunity to express a deep gratitude to my adviser Dr. Eamonn Keogh for guiding me into research on knowledge discovery from data. He gracefully granted me immense freedom to work on my own interest and above all, blessed me with his philosophy of doing scientific research. I humbly thank Dr. Vassilis Tsotras and Dr. Stefano Lonardi for their generous support in compiling this thesis.

I express gratitude to my colleagues in the data mining lab at UCR: Bilson Campana, Jin Shieh, Qiang Zhu, Lexiang Ye, Xiaoue Wang, Art Rakthanmanon, Yuan Hao, Bing Hu and Jesin Zakaria. Together we traveled through many memorable moments of our PhD studies. I particularly thank Jin for bringing the Tiny Images dataset in the lab, Qiang for his original experiments in section 2.4.3 and Lexiang for providing her shapelet code and supporting me in further development.

I thank the donors of the datasets. I am grateful to M. Brandon Westover and Sydney Cash for the EEG dataset which setup the basis of this thesis. I particularly thank Gregory Walker and Candice Stafford of the Entomological Dept. of UCR for the Beet Leafhopper data and their assistance with interpreting the data. I thank Antonio Torralba, Rob Fergus, and William Freeman for the Tiny Images dataset. I am also grateful to Nima Bigdely-Shamlo for the Brain Activity data and the magnificent results in section 3.4.1.

Finally I would like to thank my wife, Shahani Noor, for being my constant inspiration.

ABSTRACT OF THE DISSERTATION

Exact Primitives for Time Series Data Mining

by

Abdullah Al Mueen

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2012
Dr. Eamonn Keogh, Chairperson

Data mining and knowledge discovery algorithms for time series data use primitives such as bursts, periods, motifs, outliers and shapelets as building blocks. For example a model of global temperature considers both bursts (i.e. solar fare) and periods (i.e. sunspot cycle) of the sun. Algorithms for finding these primitives are required to be *fast* to process large datasets. Because *exact algorithms* that guarantee the optimum solutions are very slow for their immense computational requirements, existing algorithms find primitives *approximately*. This thesis presents *efficient* exact algorithms for two primitives, *time series motif* and *time series shapelet*. A time series motif is any repeating segment whose appearances in the time series are too similar to happen at random and thus expected to bear important

information about the structure of the data. A time series shapelet is any subsequence that describes a class of time series differentiating from other classes and thus can be used to classify unknown instances.

We extend the primitives for different environments. We show exact methods to find motifs in three different types of time series data. They are the in-memory datasets suitable for batched processing, the massive archives of time series stored in hard drives and finally, the streaming time series with limited storage. We also describe an exact algorithm for logical-shapelet discovery that combines multiple shapelets to better describe complex concepts.

We use efficient bounds to the goodness measures to increase the efficiency of the exact algorithms. The algorithms are orders of magnitude faster than the trivial solutions and successfully discover motifs/shapelets of real time series from diverse sensors such as EEG, ECG, EPG, EOG, Accelerometers and Motion captures. We show applicability of these algorithms as subroutines in high-level data mining tasks such as summarization, classification and compression.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

A time series is an ordered sequence of real valued numbers. Typically, the numbers are uniformly sampled measurements of an event or quantity e.g., temperature of a room, pH of water supply, acceleration of a robot arm, price of a stock option etc.

Numerous commercial sensors exist that record and transmit time series. Any of them can potentially produce massive amount of time series data. For example echocardiogram (ECG), a widely used diagnostic test, is recorded at 150Hz or above. A patient can generate upto 50MB of ECG time series per day. "*An estimated 300 million ECGs are recorded each year*" [64]. Therefore, the total ECG data produced can easily reach hundreds of terabytes a year.

Data mining practitioners have long been developing algorithms for processing such diverse and large scale time series data. Initial goal was to generate tools or primitives to extract important features of a time series such as bursts [54][118], periods [43][33], anoma-

lies [19][110], motifs [59], shapelets [115] and discords [114]. A relatively recent trend is to use the primitives in algorithms for high-level tasks. For example, Minnen et al. [67] recognizes activity and say, "*We approach the activity discovery problem as one of sparse motif discovery in multivariate time series*." McGovern et al. [65] predicts severe weather and say, "*Our approach first identifies . . . the temporal ordering of the motifs necessary for prediction*." Vlachos et al. [108] predicts search-query demand and say, "*Next we devise a test to separate the important periods from the unimportant ones*."

The algorithms for data mining should generally have two properties. First, the algorithms should be very efficient to process large scale data and second, should guarantee the *exact* optimality of the output. In addition to that, a primitive for time series data mining should be exact and efficient to facilitate the high-level algorithms that use it. For example, a high-level algorithm may call or invoke the primitive multiple times and may use the quality of the primitive to decide the stopping criterion.

Typically the primitives' algorithms avoid the massive computations required for exactness and content with suboptimal/close solutions. Such approximate solutions lack any guarantee on the quality of the output and in consequence are not widely usable in high-level algorithms for diverse domains. For example, many researchers have used time series motifs in domains as diverse as medicine [4][5], entertainment [18], biology [47], telemedicinse [40], telepresence [7], television broadcast [21] and severe weather prediction [65]. All of these algorithms are approximate and highly optimized for respective domains.

In this thesis, we develop efficient exact algorithms for two time series primitives: time series motif and time series shapelet. The algorithms are comparable to the approximate algorithms in speed and always guarantee the optimal answer. The algorithms possess all the benefits of exactness. We use the algorithms in classic data mining tasks such as classification, summarization and compression. We extend the algorithms for different environments (i.e. disk resident data and online data) and diverse data sources such as ECG (*Electrocardiogram*), EEG (*Electroencephalogram*), EOG (*Electrooculogram*), EPG (*Electrical Penetration Graph*), Accelerometer, Motion Capture, Audio and Video.

In the next two sections of this chapter, we briefly introduce time series motif and time series shapelet. In Chapter 2 we describe the in-memory algorithm for finding time series motif with a detailed background. In Chapter 3 and 4 we describe two extensions of motif discovery. In Chapter 5 we present the algorithm for finding shapelets. Finally in Chapter 6 we conclude with a summary and describe future possibilities.

1.1 Time Series Motif

A time series motif is a set of subsequences (i.e. segments) of a time series, which are very similar to each other [59] in their shapes. Figure 1.1 illustrates an example of a motif discovered in an industrial dataset. The time series shown in the *top* is a trace of steam flow from an industrial valve. The red and blue time series shown overlapped on one another are the motifs. The motifs are so similar that it is implausible that they happened at random and

therefore, deserve a further exploration. In reality, the similarity of the motifs was directed by a control valve that is always operated by the same machineries.

"*Time series motifs generally form the primitives for many data mining tasks*" [83] and it has been demonstrated in many domains. For example, [85] recently investigated a motif-based algorithm for controlling the performance of data center chillers, and reported "*switching from motif 8 to motif 5 gives us a nearly $40,000 in annual savings!*". Motif discovery is a core subroutine in many research projects on activity discovery [42][67], with applications in elder care [105], surveillance and sports training. In addition, there has been a recent explosion of interest in motifs from the graphics and animation communities, where motifs are used for finding transition sequences to allow just a few motion capture sequences to be stitched together in an endless cycle [9].

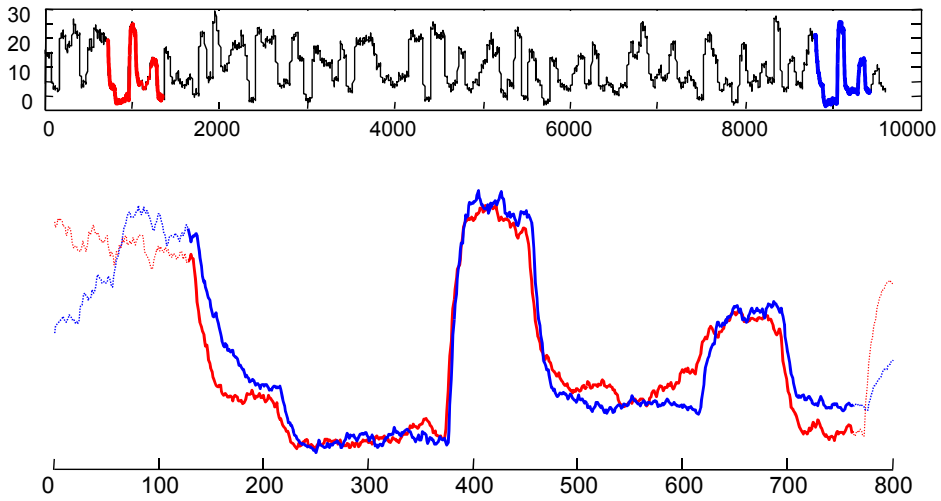

Figure 1.1: (*top*) The output steam flow telemetry of the Steamgen dataset has a motif of length 640 beginning at locations 589 and 8,895. (*bottom*) by overlaying the two motifs we can see how remarkably similar they are to each other.

To find time series motif, one can imagine the inherent necessity of all-to-all comparisons of the contiguous subsequences of the given time series. Therefore, the obvious algorithm for finding motif is quadratic on the size (i.e. length) of the given time series. For meaningful motif discovery, the larger the time series the better is the chance of capturing similar occurrences and therefore, a simple quadratic algorithm is computationally expensive and close to intractable. Researchers have long abandoned the hope of computing the exact solution (i.e. *the most similar motif*) to the motif discovery problem. Instead, many approximate algorithms to discover motifs have been proposed in the past that can handle some large datasets [9][22][40][66][68][88][98].

In this thesis, for the first time, we show an *exact* algorithm to find time series motifs. While our exact algorithm is still quadratic in the worst case, we show that we can reduce the time required for finding motif by three orders of magnitude. In fact, under most realistic conditions our exact algorithm is faster than the current linear time approximate algorithms and other exact algorithms with better guarantees in the worst case. The reason is either they have very large constant overheads or their best cases are the same as the worsts. As we shall show, our algorithm allows us to tackle problems which have previously thought intractable, for example, automatically constructing *dictionaries* of recurring patterns from electroencephalographs.

The challenges involved in finding motifs exactly vary with the type and size of the time series data. If the given time series fits in the main memory, the critical part (i.e. cost unit) of the algorithm is the computation of similarities (i.e. the Euclidean distance) between

subsequences. Therefore, the main focus of an in-memory algorithm is to reduce the number of comparisons. In contrast if the data does not fit in the memory and is stored in the disk, there is an additional challenge of carefully organizing the disk accesses. Similarly when the data is streaming in real time, the speed of the data imposes an upper limit on the computation time per sample. In this thesis we investigate all of the three scenarios described above and propose efficient solutions for each of them. We also demonstrate real applications that generate the above scenarios and our algorithms discover significant motifs in all of them.

1.2 Time Series Shapelet

Time series shapelets were introduced in 2009 as a primitive for time series data mining [115]. Shapelets are small subsequences that separate the time series into two classes by asking the question *"Does this unknown object have a subsequence that is within τ of this shapelet?"* Where there are three or more classes, repeated application of shapelets can be used (i.e. a decision tree-like structure) to predict the class label. Figure 1.2 shows examples of shapelets found in a dataset of accelerometer signals [60]. Every time series in the dataset corresponds to one of two hand motions performed by an actor tracing a *circular* or *rectangular* path through the air with an input device. The shapelet denoted by **P** in the figure is the one that maximally separates the two classes when used with a suitable threshold. In essence, the shapelet **P** captures the sinusoidal acceleration pattern of the circular motion along the Z-axis.

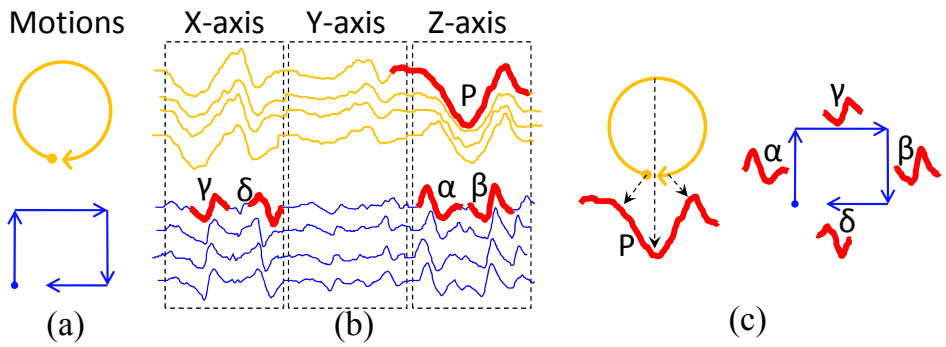

Figure 1.2: (a) Idealized motions performed with a Wii remote. (b) The concatenated accelerometer signals from recordings of actors performing the motions (c) Examples of Shapelets that describe each of the motions.

Time series shapelets are generating increasing interest among researchers [44] [65] [112] for at least two reasons. First, in many cases time series shapelets can learn the inherent structure of the data in a manner that allows intuitive interpretation. For example, beyond classifying, say, normal/abnormal heartbeats, shapelets could tell a cardiologist that the distinguishing feature is at the beginning of the dicrotic pulse. Second, shapelets are usually much shorter than the original time series, and unlike instance based methods (i.e. k-NN) that require comparison to the entire dataset, we only need one shapelet at classification time. Therefore, shapelets create a very compact representation of the class concept, and this compactness means that the time and space required for classification can be significantly reduced, often by at least two orders of magnitude. This is a particularly desirable property in resource limited systems such as sensor nodes, cell phones, mobile robots and smart toys.

Despite the above promising features of time series shapelets, the current algorithm [115] for discovering them is still relatively lethargic and, therefore, does not scale up to use on

real-world datasets, which are often characterized by being noisy, long, and non-uniformly sampled.

In addition, the current definition of shapelets is not expressive enough to represent certain concepts that seem quite common in the real world (examples appear in Chapter 5). In particular, the expressiveness of shapelets is limited to simple binary presence/absence questions. While recursive application of these binary questions can form a decision tree-like structure, it is important to recognize that the full expressive power of a classic, machine-learning decision tree is not achieved (recall a decision tree represents the concept space of disjunction of conjunctions). For example, differentiating classes by using only binary questions is not possible if, the classes differ only in the *number* of occurrences of a specific pattern rather than presence/absence of a pattern.

In this thesis, we address the problem of scalability and show how an efficient algorithm allows us to define a more expressive shapelet representation. We introduce two novel techniques to speedup the search for shapelets. First, we precompute sufficient statistics [90] to compute the distance (i.e. similarity) between a shapelet and a subsequence of a time series in amortized constant time. In essence, we trade time for space, finding that a relativity small increase in the space required can help us greatly decrease the time required. Second, we use a novel admissible pruning technique to skip the costly computation of entropy (i.e. the goodness measure) for the vast majority of candidate shapelets. Both of the speedup techniques are admissible hence the algorithm discovers the best shapelet *exactly*.

We further show that we can combine multiple shapelets in logic expressions such that complex concepts can be described. For example, in the dataset shown in Figure 1.2, there are discontinuities in the rectangular motion. It will not be possible to describe the rectangular class using one shapelet if there are variable pause times at the corners in different instances. In such cases, we can express the rectangular class by the logical-shapelet "α **and** β **and** γ **and** δ." Here, each literal corresponds to one of the straight line motions as shown in Figure 1.2(c). In addition, our algorithm is able to find shapelet that increases the gap/margin between classes even if they are already separated by other candidates. This allows for more robust generalization from the training to test data.

We show the efficiency of our algorithm on twenty-four datasets and achieve up to 27$\times$ speedup over the current algorithm experimentally. We demonstrate that combination of shapelets can better describe classes than single shapelets can do in multiple datasets such as accelerometer signals from sports automation, pen-based biometrics and accelerometer signals from a mobile robot.

Before we end this chapter, we state our experimental philosophy. We have designed all experiments such that they are not only reproducible, but easily reproducible. To this end, we have built several webpages [2] which contain all datasets and code used in this thesis, together with spreadsheets which contain the raw numbers displayed in all the figures. In addition, the webpage contains many additional experiments which we could not fit into this thesis; however, we note that this thesis is completely self-contained.

Chapter 2

Exact Discovery of Time Series Motifs

In this chapter, the time series motif is defined formally and the in-memory exact algorithm is presented. This is the first efficient algorithm to find time series motif exactly. We break down the chapter in five major sections. Section 2.1 defines the problem and describes underlying assumptions. Section 2.2 describes the intuition behind the algorithm and formally proves the correctness. Section 2.3 shows the scalability experiments and necessary discussions on certain choices we made. Section 2.4 describes independent case studies on several datasets. We also describe the related work in Section 2.5 of this chapter.

2.1 Definitions and Background

Before describing our algorithm, we define the key terms for this chapter.

Definition 2.1 [TIME SERIES] A Time Series is a sequence $T = (t_1, t_2, \ldots, t_m)$ which is an ordered set of m real valued numbers.

The ordering is typically temporal; however other kinds of data such as color distributions [41], shapes [104] and spectrographs [117] also have a well defined ordering and can fruitfully be considered "*time series*" for the purpose of indexing and mining. It is possible there could be variable time spacing between successive points in the series. For simplicity and without loss of generality we consider only equispaced data in this thesis. In general, we may have many time series to consider and thus need to define a time series database.

Definition 2.2 [TIME SERIES DATABASE] A Time Series Database D is an unordered set of n time series possibly of different lengths.

Again for simplicity, we assume that all the time series are of same length and D fits in the main memory (a disk-aware version of our algorithm is given in Chapter 3). Thus D is a matrix of real numbers where D_i is the ith row in D as well as the ith time series T_i in the database and $D_{i,j}$ is the value at time j of T_i. Having a database of time series, we are now in a position to define time series motifs.

Definition 2.3 [TIME SERIES MOTIF] The Time Series Motif of a time series database D is the unordered pair of time series $\{T_i, T_j\}$ in D which is the most similar among all

possible pairs. More formally, the pair $\{T_i, T_j\}, i \neq j$ is the motif IFF $\forall_{a,b,a\neq b}\, dist(T_i, T_j) \leq dist(T_a, T_b)$.

Note that our definition excludes the trivial match of a time series with itself by not including $i = j$. We can extend motifs to subsequences of a very long time series by treating every subsequences of length m ($m << n$) as an object in the time series database. Motifs in such a database are subsequences that are conserved locally in the long time series. More formally,

Definition 2.4 [SUBSEQUENCE] A subsequence of length m of a time series $T = (t_1, t_2, \ldots, t_m)$ is a time series $T_{i,m} = (t_i, t_{i+1}, \ldots, t_{i+m-1})$ for $1 \leq i \leq n - m + 1$.

Notice the term subsequence is used for *contiguous* subsequences of the time series as opposed to arbitrary subsequences in discrete strings.

Definition 2.5 [SUBSEQUENCE MOTIF] The Subsequence Motif is a pair of subsequences $T_{i,m}, T_{j,m}$ of a long time series T that are most similar. More formally, the pair $\{T_{i,m}, T_{j,m}\}, |i - j| \geq w > 0$ is the motif IFF

$$\forall_{a,b,|a-b|\geq w}\, dist(T_{i,m}, T_{j,m}) \leq dist(T_{a,m}, T_{b,m})$$

We impose a limit on the relative positions of the subsequences in the motif. This says that there should be a gap of at least w places/samples between the subsequences. For example, $w = m$ gives us non-overlapping motifs only. For a given separation window w, the total

number of possible motif pairs is exactly $\frac{1}{2}(m-n-w+1)(m-n-w)$, which is slightly less than the number of otherwise possible pairs (i.e. $\frac{1}{2}(m-n+1)(m-n)$).

This restriction helps to prune out the trivial subsequence motifs [84]. For example (and considering discrete data for simplicity), if we were looking for motifs of length four in the string:

*sjdb*bnvfdfpqoeutyvnABABABmbzchslfkeruyou*sjdq*

Then in this case we probably don't want to consider the pair {ABAB,ABAB} to be a motif, since they share 50% of their length (i.e **AB** is common to both). Instead, we would find the pair {*sjdb, sjdq*} to be a more interesting approximately repeated pattern. In this example, we can enforce this by setting the parameters $w = 4$.

With the exception of the minor overhead of keeping track of the trivial matches [22] in finding subsequence motif, our algorithm is agnostic to how the time series in the database are produced (i.e. subsequences or independent 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. There are some obvious possible generalizations of the above definition of time series motifs.

Definition 2.6 [kTH MOTIF] The kth-Time Series motif is the kth most similar pair in the database D.

Instead of dealing with pairs only we can also extend the notion of motif to sets of time series that are very similar to each other.

Definition 2.7 [RANGE MOTIF] The Range motif with range r is the maximal set of time series that have the property that the maximum distance between them is less than $2r$.

The range motif corresponds to dense regions or high dimensional "bumps." Finding the range motif is equivalent to finding a maximal clique in the high dimensional space where there is an edge between a pair of time series if their distance is less than $2r$. A reasonable approximation of range motif can be to find the first motif and then selecting all the time series within r of the first motif.

In all the definitions given above we assumed there is a meaningful way to measure the distance between two time series. There are several such ways in the literature and our method is valid for any distance measure that is a metric. We use z-normalized Euclidean distance and define it as below.

Definition 2.8 [Z-NORMALIZED EUCLIDEAN DISTANCE] The z-normalized Euclidean distance $d(X, Y)$ between two time series X and Y of length m is $d(X, Y) = \sqrt{\sum_{i=1}^{m}(\hat{x}_i - \hat{y}_i)^2}$ where $\hat{x}_i = \frac{x_i - \mu_x}{\sigma_x}$ and $\hat{y}_i = \frac{y_i - \mu_y}{\sigma_y}$. Here μ and σ are the mean and standard deviation computed from the population.

Recently an extensive empirical comparison has shown that the Euclidean distance is competitive with more complex measures on a wide variety of domains [29]. A detail reasoning about this choice of distance measure is given in Section 2.3.6. Z-normalized Euclidean

distance is essentially the positive correlation between the time series [72] and invariant to scale and offset of the comparing time series. In Section 2.3.4, we describe experiments on real data demonstrating the choice of normalization.

Computing Euclidean distance between two time series of length m takes a full pass over the two time series and thus has $O(m)$ time complexity. However when searching for the nearest neighbor for a particular time series Q, it is possible to abandon the Euclidean distance computation as soon as the cumulative sum goes beyond the current $best\text{-}so\text{-}far$, an idea known as *early abandoning* [6]. For example assume the current $best\text{-}so\text{-}far$ has a distance of 13.93. If, as shown in Figure 2.1, the next item to be compared is further away, then at some point the sum of the squared error will exceed the current minimum distance $r = 12$. So the rest of the computation can be abandoned since this pair can't have the minimum pairwise distance.

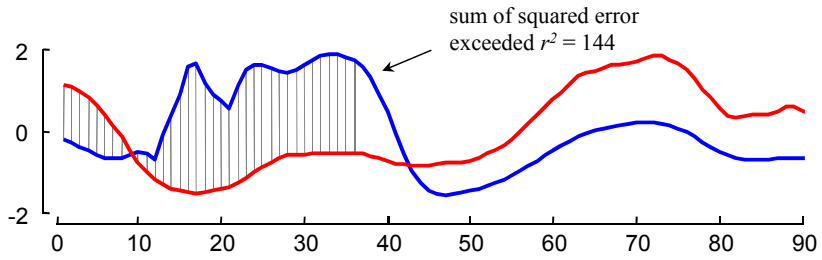

Figure 2.1: A visual intuition of early abandoning. Once the squared sum of the accumulated gray hatch lines exceeds r^2, Its confirm that the full Euclidean distance exceeds r

It has long been known that early abandoning reduces the amortized cost of computing the distances to less than $O(m)$, however, in this work we show for the first time, and explain why, early abandoning is particularly effective for motif discovery (Section 2.3.6). The

algorithm presented in the next section is designed for the original definition of time series motif (Definition 2.3). Once this problem can be solved quickly, the generalizations to the kth motif and the range motif are trivial and incur only a tiny additional overhead. Therefore, for simplicity, we will ignore these extensions in our description of the algorithm.

2.2 The MK Algorithm

In Section 2.2.2 we have a detailed formal explanation of our exact motif discovery algorithm. However, for clarity and ease of exposition the next section contains a simple visual intuition of the underlying ideas that the algorithm exploits. We call our algorithm **Motif Kymatology** (MK), as we aim to find motifs in waveforms, rather than in strings.

2.2.1 The Intuition behind MK

In Figure 2.2(A) we show a small dataset of two-dimensional time series objects. We are interested in finding the motifs, which we can see here are objects O4 and O5.

Before our algorithm begins, we must assume the $best\text{-}so\text{-}far$ distance for the motif pair to be infinity. As shown in Figure 2.2(B), we can choose a random object (in this case O1), as a reference point, and we can order all other objects by their distances to that point. As a side effect of this step, we can use the distance between O1 and its nearest neighbor O8, to update the $best\text{-}so\text{-}far$ distance to be 23.0.

Note that in the act of sorting the objects we can record the distances between adjacent pairs, as shown in Figure 2.2(C). It is critical to recall that these distances are not (in general) the true distances between objects in the original space, rather they are lower bounds to those true distances.

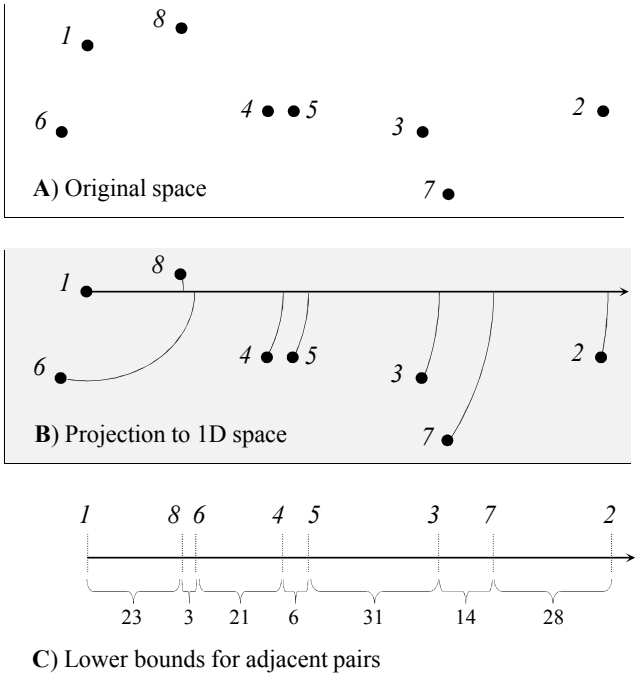

Figure 2.2: (A) A small database of two-dimensional time series objects. (B) The time series objects can be arranged in a one-dimensional representation by measuring their distance to a randomly chosen point, in this case O1. (C) The distances between adjacent pairs along the linear projection is a (generally very weak) lower bound to the true distance between them

The key insight of our algorithm is that this linear ordering of data provides us with some useful heuristic information to guide our motif search. The observation is that if two objects are close in the original space, they must also be close in the linear ordering. Note that the contrapositive is not true. Two objects can be arbitrarily close in the linear ordering but very far apart in the original space.

In the next stage of our algorithm we can scan across the linear ordering and measure the true distances between adjacent pairs. If, while doing this we encounter a pair that has a distance less than the current $best\text{-}so\text{-}far$, we can update it, as shown in Figure 2.3. In our example we slide from left to right, updating the estimated distance between O8 and O6 of 3.0 to the correct distance of 42.0. Similarly we update the estimated distance between O6 and O4 to 49.0. In our next update, we find the true distance between O4 and O5 is only 7.0. Since this is less than our current $best\text{-}so\text{-}far$, we update it.

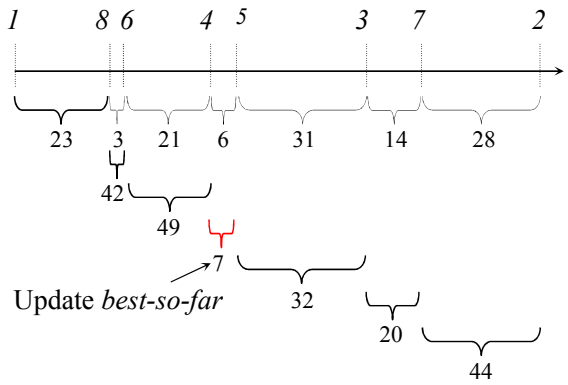

Figure 2.3: We scan the objects from left to right, measuring the true distances between them. Note that just for the first pair O1, O8 the linear distance is the true distance. In all other cases the linear distance is a lower bound. For example, the lower bound distance between O8, O6 is 3, but our test of the true distance reveals dist(O8, O6) = 42.0

In our contrived example, we have already found the true motif. However this may not be true in general. Moreover, we do not know at this point that the current $best\text{-}so\text{-}far$ refers to the true motif. However we can now use the linear representation combined with the $best\text{-}so\text{-}far$ to prune off large fraction of the search space.

For example, could the pair O8 and O3 be closer than our $best\text{-}so\text{-}far$? We can answer that question without having to actually measure the true distance between them. The lower bound distance in our linear representation is 60.0, but our $best\text{-}so\text{-}far$ is only 7.0. Given that, we can be sure that the pair O8 and O3 is not a candidate to be the motif. More generally, we can take a sliding window of exactly width 7 (the $best\text{-}so\text{-}far$), and slide it across the linear order testing for possible pairs of objects that could be the true motif. As shown in Figure 2.4 a necessary condition for two objects to be the motif is that both of them intersect the sliding window at the same time.

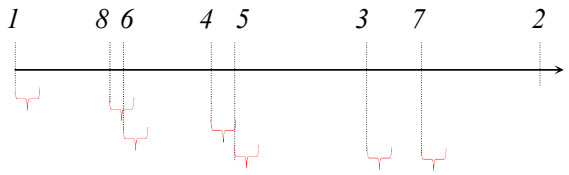

Figure 2.4: A necessary condition for two objects to be the motif is that both of them intersect a sliding window, of width $best\text{-}so\text{-}far$, at the same time. Only pairs O8, O6 and O4, O5 survive the sliding window pruning test

In this example, only pairs O8, O6 and O4, O5 could be the true motif, but in this case we already know the true distances for these pairs, so we are done. More generally, we may have additional true and/or false positives not pruned by this test, and we would need to check all of them in the original space.

This then, is the major intuition behind our approach. The full algorithm differs in several ways: Not all objects are equally good as a reference point, we use a simple heuristic to find a good reference points. In the above exposition we did one round of pruning. However for

Algorithm 2.1 $[L_1, L_2] = BruteForce\_Motif(D)$

Require: A database D of n time series
Ensure: Locations L_1 and L_2 of a motif

```
1: best-so-far ← ∞
2: for i ← 1 to n do
3:    for j ← i + 1 to n do
4:       if d(D_i, D_j) < best-so-far then
5:          best-so-far ← d(D_i, D_j)
6:          L_1 ← i, L_2 ← j
7: return L_1, L_2
```

large datasets this may still leave a huge number of candidate pairs to check. Instead, we can run multiple pruning steps with multiple reference points to do additional pruning. In the next section we consider a more formal and detailed discussion of these points.

2.2.2 A Formal Statement of MK

For ease of exposition we will first consider the brute force motif discovery algorithm, and then show how our algorithm can be obtained by modifying it. The brute force algorithm as outlined in 2.1 has a worst case complexity of $O(n^2)$. The algorithm maintains a running minimum $best\text{-}so\text{-}far$ and updates it whenever the algorithm finds a pair of time series having a smaller distance smaller between them.

The algorithm is simple a pair of nested loops which tests every possible combination of pairs of time series, and reports the pair $\{L_1, L_2\}$ which has the minimum distance between them.

Speeding up the Brute Force Algorithm

In order to significantly speed up the brute force algorithm we must find a way of pruning off as many distance computations so that the algorithm does not need to compare all pairs. This is because each distance computation takes $O(m)$ time which makes the brute force algorithm precisely an $O(n^2m)$ algorithm. Algorithm 2.2 shows the version of our algorithm after the speeding up techniques have been applied. We recommend the reader to match this with Algorithm 2.1 to identify the changes. To prune off a possible distance computation for a pair we would need to be sure that the pair can't be a motif. The obvious way is to compute the actual distance and compare it with the $best\text{-}so\text{-}far$ as done in the brute force algorithm. But we can do it in a better way by having cheaply computed lower bounds on the distances for all possible pairs. Since we are using distance metric like Euclidean distance, we can use the triangular inequality property of Euclidean distance to find lower bounds. Let ref is a reference time series which may or may not be an object in D. Let, $\{D_i, D_j\}$ is the pair whose distance we don't want not compute. Now by triangular inequality $|d(ref, D_i) - d(ref, D_j)| \leq d(D_i, D_j)$. Thus if we know the distances on the left, we can use them as lower bound for the $d(D_i, D_j)$ after a cheap subtraction. If this lower bound happens to be larger than the $best\text{-}so\text{-}far$ (the running minimum), we can safely avoid computing the $d(D_i, D_j)$ (lines 10 in Algorithm 2.2). Otherwise $\{D_i, D_j\}$ remain as a potential motif to be compared. The distances of the time series in D from ref can be computed and saved (in a single column table called $Dist$) before the search starts and it takes only $O(n)$ time which is small enough compared to the $O(n^2)$ search time (lines 3-4 in Algorithm 2.2). Hence

computing the lower bounds needs only two table look ups. Note that we prefer having a time series in D as the reference because it helps the algorithm to start with a good initial $best\text{-}so\text{-}far$ (instead of infinity) before the start of the search. This also needs attention to prevent the $\{ref, ref\}$ from occurring by assigning a large value in Dist at the entry for ref. Doing this trick reduces the number of distance computations, but still requires searching all possible pairs.

To get rid of all-to-all comparisons, we need an alternative search strategy that can stop well before seeing all pairs. Our strategy is the ordered search. Ideally if we sort the lower bounds for every pair in ascending order and compare pairs in that order, we can stop our search as soon as we get a pair whose lower bound is greater than the $best\text{-}so\text{-}far$ at that time. This is unrealistic because it will take $O(n^2 \log n^2)$ to sort the lower bounds and surely worse than the brute force algorithm. Rather than sorting lower bounds for every pairs we can sort the distances from the reference time series which is the linear ordering as mentioned in the previous section. This sorting can also be performed before the search starts and thus costs reasonably small time of $O(n \log n)$ (lines 5 in Algorithm 2.2). Since we look up the table $Dist$ to avoid distance computations, instead of sorting all the distances in $Dist$, we can sort the indices to the rows of $Dist$. This ordered array of indices is named as I in Algorithm 2.2 at line 5. Technically, I is the sorted order of the time series in D where $d(ref, D_{I(i)}) \leq d(ref, D_{I(j)})$ iff $i \neq j$. Now the question is how the ordering would guide the search and help to stop the search early. The following two lemmas have the answer.

Algorithm 2.2 $[L_1, L_2] = Speedup\_Motif(D)$

Require: A database D of n time series
Ensure: Locations L_1 and L_2 of a motif

```
best-so-far ← ∞
ref ← randomly chosen time series D_r from D
for j ← 1 to n do
    Dist_j ← d(ref, D_j)
find an ordering I of the indices to the time series in D such that Dist_I(j) ≤ Dist_I(j+1)
offset ← 0, abandon ← false
while abandon = false do
    offset ← offset + 1, abandon ← true
    for j ← 1 to n − offset do
        if |Dist_I(j) − Dist_I(j+offset)| < best-so-far then
            abandon ← false
            x ← d(D_I(j), D_I(j+offset))
            if x < best-so-far then
                best-so-far ← x
                L_1 ← I(j), L_2 ← I(j + offset)
return L_1, L_2
```

Lemma 2.1 *If* $D_{I(j+offset)} - D_{I(j)} > best\text{-}so\text{-}far$ *for all* $1 \leq j \leq n - offset$ *and* $offset > 0$ *then* $D_{I(j+w)} - D_{I(j)} > best\text{-}so\text{-}far$ *for all* $1 \leq j \leq n - w$ *and* $w > offset$.

Proof: For a positive integer $offset$ and for $j = 1, 2, \ldots, n - offset$ if $\{D_{I(j)}, D_{I(j+offset)}\}$ fail to have their lower bounds less than the $best\text{-}so\text{-}far$ then for all positive integers $w > offset$ and for all $j = 1, 2, \ldots, n - w$, $\{D_{I(j)}, D_{I(j+w)}\}$ will also fail to have their lower bounds less than the $best\text{-}so\text{-}far$. This is true from the definition of I that says

$$d(ref, D_{I(j)}) \leq d(ref, D_{I(j+offset)}) \leq d(ref, D_{I(j+w)})$$

This can be rewritten as

$$d(ref, D_{I(j+offset)}) - d(ref, D_{I(j)}) \leq d(ref, D_{I(j+w)}) - d(ref, D_{I(j)})$$

Therefore if the left part is larger than $best\text{-}so\text{-}far$ the right part will obviously be larger. ∎

Lemma 2.2 *If* $offset = 1, 2, \ldots, n-1$ *and* $j = 1, 2, \ldots, n-offset$ *then* $\{D_{I(j)}, D_{I(j+offset)}\}$ *generates all the possible pairs.*

If we search the database D for all possible offsets by which two time series can stay apart in the linear ordering, we must encounter all the possible pairs. Since I has no repetition, it is obvious that $\{D_{I(j)}, D_{I(j+offset)}\}$ will generate all the pairs with no repetition. Hence this lemma states the exactness of our search strategy.

With the help of these two lemmas we can build the search strategy. The algorithm starts with an initial $offset$ of 1 and searches pairs that are $offset$ apart in I ordering. After searching all pairs of offset apart it increases the offset and searches again (for the next round). The algorithm continues till it reaches an $offset$ for which there is no pair having lower bound larger than the $best\text{-}so\text{-}far$ and staying $offset$ apart in the I ordering. In Algorithm 2.2, lines 7-13 incorporate this search strategy. Obviously it is true that this strategy has the worst case complexity $O(n^2)$ equal to the brute force algorithm. But this only occurs in the cases where the "motif has distance larger than any lower bound computed using a random reference." With no doubt it is very unlikely to happen in real time series databases.

Generalization to multiple reference points

The speeding up that we gained in the previous section can be extended to multiple reference time series and use them to have tighter lower bounds. Using multiple reference time series

Algorithm 2.3 $[L_1, L_2] = MK\_Motif(D, R)$

Require: A database D of n time series
Ensure: Locations L_1 and L_2 of a motif

```
best-so-far ← ∞
for i ← 1 to R do
    ref_i ← randomly chosen time series from D
    for j ← 1 to n do
        Dist_{i,j} ← d(ref_i, D_j)
    S_i = standard_deviation(Dist_i)
find an ordering Z of the indices to the references in ref such that S_{Z(i)} < S_{Z(i+1)}
find an ordering I of the indices to the D such that Dist_{Z(1),I(j)} < Dist_{Z(1),I(j+1)}
offset ← 0, abandon ← false
while abandon = false do
    offset ← offset + 1, abandon ← true
    for j ← 1 to n − offset do
        if |Dist_{Z(1),I(j)} − Dist_{Z(1),I(j+offset)}| < best-so-far then
            abandon ← abandon∧false
            reject ← false
            for i = 1 to R do
                lower_bound = |Dist_{Z(i),I(j)} − Dist_{Z(i),I(j+offset)}|
                if lower_bound > best-so-far then
                    reject ← true , break
            if reject = false then
                x ← d(D_{I(j)}, D_{I(j+offset)})
                if x < best-so-far then
                    best-so-far ← x
                    L_1 ← I(j),L_2 ← I(j + offset)
        else
            abandon ← abandon∧true
return L_1,L_2
```

raises some issues in our previous version. We show the final version of our algorithm in algorithm 2.3 and again recommend the user to watch it for the changes.

To get the tighter lower bounds we use multiple reference time series randomly chosen from D as before. The number of references is a parameter to our algorithm represented by R. As a consequence of using multiple references, $Dist$ becomes a two dimensional table that stores distances between any reference time series to any time series in D. Again note

that we will need to prevent invalid motifs of same time series by assigning large values in $Dist$.

The way multiple references help tightening the lower bounds is very simple. In effect we will use the maximum of the lower bounds. This is correct because if one lower bound is larger than $best\text{-}so\text{-}far$ the maximum would also be larger and there is no way the pair would become a motif. Since the lower bounds are not stored anywhere, the algorithm needs to compute all the R lower bounds for every single pair (lines 16-17 in Algorithm 2.3). Rather than computing the maximum which would take $O(R)$ time, we compare each bound with the current $best\text{-}so\text{-}far$ and reject (line 19 in Algorithm 2.3) computing the true distance as soon as one bound is higher than the $best\text{-}so\text{-}far$. Thus amortized cost of the combined lower bound is smaller than $O(R)$.

Although multiple reference time series tightens the lower bounds, all of them can't be used in the search strategy. This is because it needs to follow exactly one ordering (Lemma 2.2). To choose a reference time series for ordering the time series in D, we select the one ($Z(1)$ in line 10) with the largest standard deviation in the distances from itself to others in D (lines 7-8 in Algorithm 2.3). The intuition behind it is - the larger the deviation is the larger the lower bounds would be and the more probable the "early stop" and "reject comparisons" would be. Since we need to sort the standard deviations in descending order for all the references, we can use this order of the references to reject a pair earlier instead of a random order.

2.3 Experiments

In this section, we compare MK with two available trivial counter parts because MK is first of its kind. We demonstrate the parameter sensitivity and discuss some of the characteristics of the algorithm.

We performed the scalability experiments on both synthetic and real data. All the experiments are performed on a computer with an AMD 2.1GHz Turion X2 Ultra ZM-80 processor and 3.0GB of DDR2 memory. The algorithm is coded in C and compiled with gcc.

2.3.1 Performance Comparison

The two trivial algorithms we consider comparing with MK are the divide and conquer algorithm [11] and the standard brute force algorithm described in Algorithm 2.1.

The divide and conquer based algorithm for finding the closest-pair of points in space is prevalent in the text books of computational geometry. The divide and conquer (DaC) approach for finding closest pair in multidimensional space is described in great detail in [11]. DaC works in $O(n \log^d n)$ time for any data distribution, which is expensive for large d. For "sparse" data DaC works in $O(n \log n)$. The relevant definition of sparsity is given "as the condition that no d-ball in the space (that is, a sphere of radius d) contains more than some constant c points."[11] This condition ensures that the conquer step remains a linear operation with no more than cn pairs of close points to be compared. But subsequences of a long time series form a trail in the high dimensional space which may cross itself arbitrary number

of times to violate the sparsity condition for efficient DaC algorithm [35]. A simple 3D demonstration is shown in Figure 2.5(left) by plotting all triplets of successive real numbers in an ECG time series.

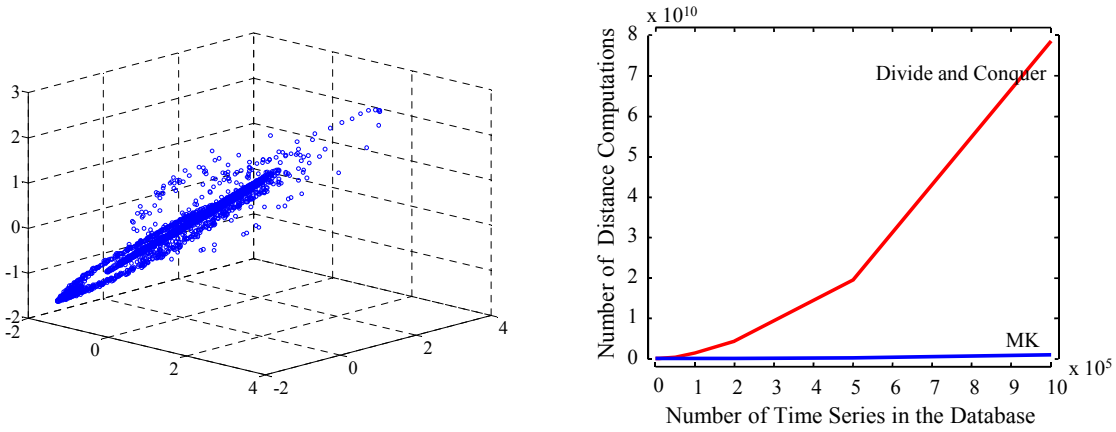

Figure 2.5: (*left*)Plot of successive pairs of numbers of a time series. (*right*) Comparison of DAME with divide and conquer approach

Considering the above two observations, we may expect MK to perform much better than the divide and conquer algorithm. As shown in Figure 2.5(*right*), this is the case. For a motif length of 128, we tested up to one million points of EEG time series and DaC performs 100 times more distance computations than MK for the larger datasets. These results are in spite of the fact that we allowed DaC to "cheat", by always using the best axis to divide the data at each step.

To compare with the standard brute force algorithm, we use random walk time series to test our algorithm. Random walk data is a difficult case for our algorithm, since we should not expect a very close motif pair to exist in the data. We produced 10 sets of random walks of different sizes containing from 10,000 to 100,000 time series, all of length 1024. We ran

our algorithm 10 times on each of these datasets and took the average of the execution times. Figure 2.6 shows a comparison of the brute force algorithm with our MK algorithm in terms of the execution time.

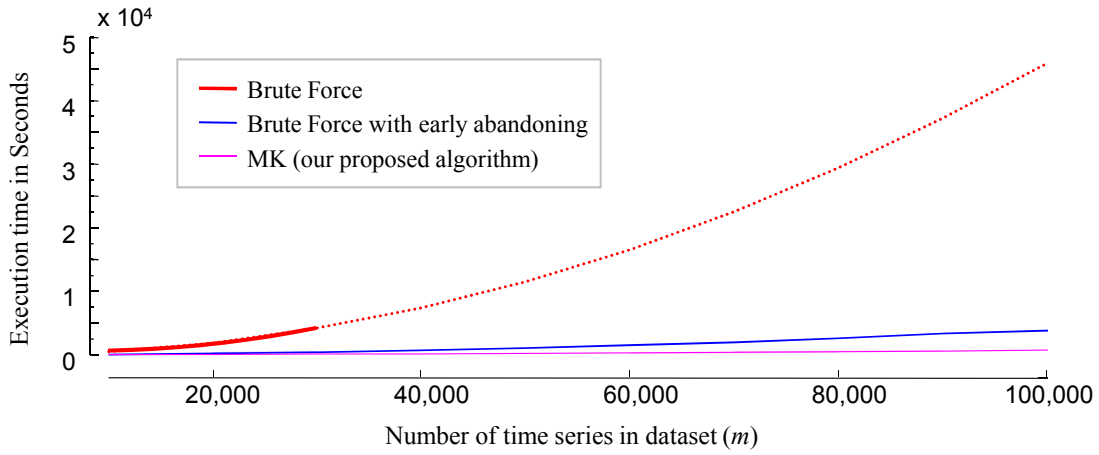

Figure 2.6: A comparison of three algorithms in the time taken to find the motif pair in increasingly large random walk databases. For the brute force algorithm, values for dataset sizes beyond 30,000 are extrapolated

The difference in execution times is quite dramatic, for 100,000 objects brute force takes 12.7 hours, but our algorithm takes only 12.4 minutes (with a standard deviation of 55 seconds).

As dramatic as this speedup is, it is in fact the worst case for our algorithm. This is because there is no reason to expect a particularly close pair of objects in a random walk dataset. In real datasets the results are generally significantly better. For example, we repeated the experiment with an EEG dataset s shown in Figure 2.7.

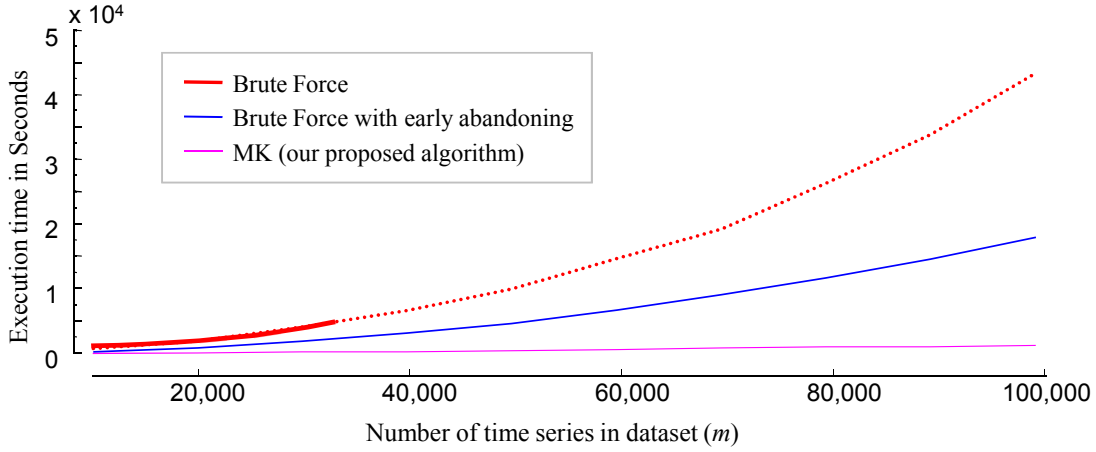

Figure 2.7: A comparison of three algorithms in the time taken to find the motif pair in increasingly large electroencephalograph databases (all subsets of the EEG dataset). For the brute force algorithm, values for dataset sizes beyond 30,000 are extrapolated

Here the brute force time is essentially unchanged, but the time for our algorithm is only 2.17 minutes (with a standard deviation of 13.5 seconds).

2.3.2 Choosing the number of reference points

Our algorithm has one input parameter, the number of reference time series used. Up to this point we have not discussed the parameter in detail, however it is natural to ask how critical its setting is. A simple thought experiment tells us that a too small and too large a value should produce a slower algorithm. In the former case, if few reference time series are used, most candidate pairs are not pruned, and must be examined by brute force. In the latter case, we may have only $O(n)$ pairs of candidates left to check, but the time to create a one-dimensional representation from a reference time series is $O(n)$, so we may not break even

and we may have been better off to just brute force the remaining time series. This reasoning tells us that a plot of execution time vs. number of reference time series should be U-shaped plot, and we should target a parameter that gives us the bottom of the curve. In Figure 2.8 we illustrate this observation with an experiment in varying the number of reference points and measuring the execution time. Note that the rightmost value, corresponding to zero reference points is equivalent to the special case of brute force search.

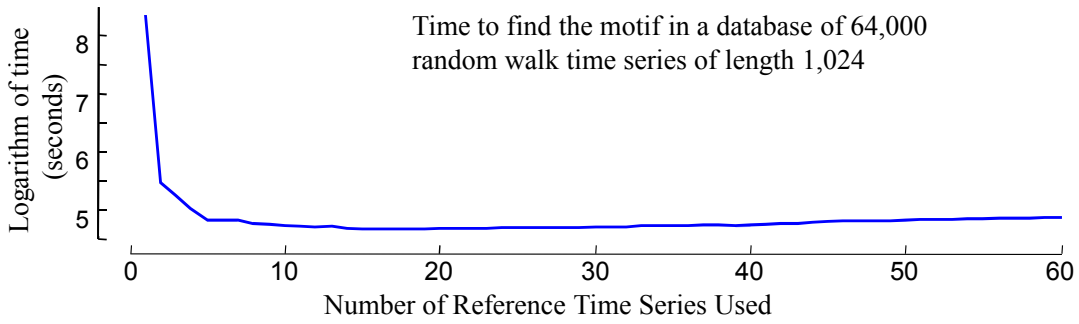

Figure 2.8: A plot of execution time vs. the number of reference points. Note that once the number of reference points is beyond say five, its exact value makes little difference. Note the log scale of the time axis

This plot suggests that the input parameter is not critical. Any value from five to sixty gives two orders of magnitude speedup. Moreover, this is true if we change the length of the time series, the size of the database (i.e n) or the data type. For this reason, we fixed the number of reference points to be eight in all the experiments in this chapter.

2.3.3 Why not use other lower bounding techniques?

We consider two other ways of using the distances from multiple reference points for tighter lower bounds.

In our algorithm we compute distances from R reference points to all of the objects. In the line 16 of Algorithm 2.3, 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 ($best\text{-}so\text{-}far$). 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.

After that, simple 2D geometry is needed to compute the lower bound (dashed line) using five other pre-computed distances (solid lines in Figure 2.9(*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 2.9(*right*), it is clear that the linear bound is

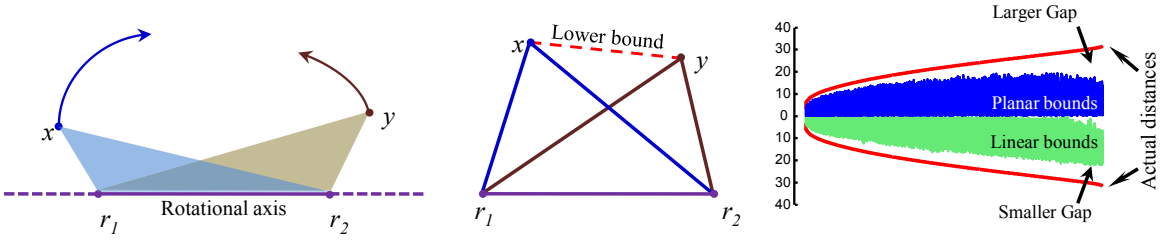

Figure 2.9: (*left*) Two points x and y are projected on a plane by a rotation around the axis joining two reference points r_1 and r_2. (*middle*) Known distances and the lower bound after the projection. (*right*) Planar and linear bound are plotted against true distances for 40,000 random pairs

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.

The second way of using two reference points for computing lower bounds is by using *ptolemaic bound* [46]. Ptolemy's inequality says $xr_1 * yr_2 \leq xy * r_1r_2 + xr_2 * yr_1$ for the four points shown in Figure 2.9(*left*). In our effort to test if ptolemaic bound is better than the linear bound based on the triangular inequality, we compute the number of times either of the bounds successfully prunes a pair while the other doesn't in line 18 of Algorithm 2.3. We plot the counts in Figure 2.10 for various values of n and m. Since there is no clear winner and linear bound is simpler, we opt to use the linear bound for pruning.

There are many other ways of computing lower bounds by using various projections (e.g. random [23], orthogonal [82] etc.) and transforms (e.g. fourier [107], wavelet [16] etc.). However, they are primarily developed for similarity search or indexing large databases and

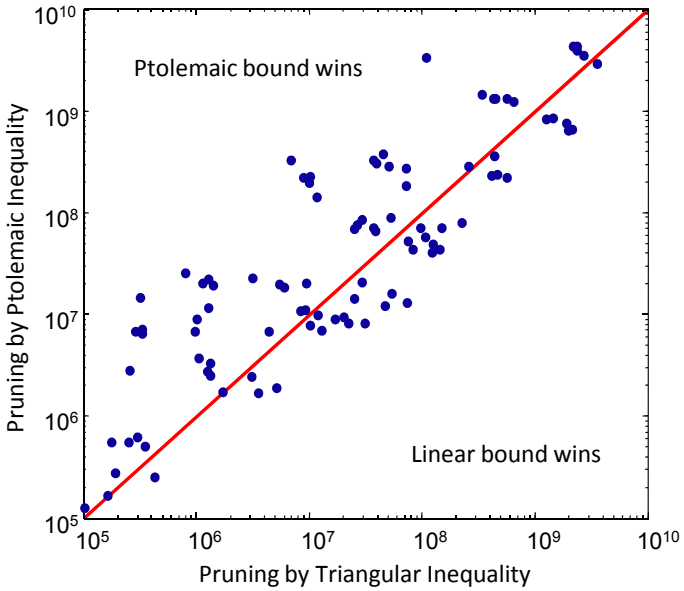

Figure 2.10: Comparison of the number of times ptolemaic bound prunes a distance computation to that of linear bound for various values of n and m

have significant preprocessing overhead. Motif discovery is an unsupervised process and cannot afford to precompute complex projections/transforms for in-memory datasets. Therefore we opt for simple linear ordering of the subsequences that adds insignificant overhead.

2.3.4 z-Normalizing the time series

The algorithms we describe in this thesis perform explicit z-normalizations of the time series in the database. At this point we demonstrate the necessity of such normalization and discuss the possible impact on the design of the algorithms.

We begin by downloading a long ECG sequence [38] from a 61-year-old female and manually extracting a segment of interest of length 2155 as shown in Figure 2.11. We take the first beat (shown by the thick line) of the segment as a query. We set a conservative

threshold of 50% correlation equivalent to a distance of 13. Figure 2.11(*middle*,*bottom*) show the distances of every subsequence from the query. We see no missing beat for the z-normalized distance whereas the un-normalized Euclidean distance misses three beats out of twelve. How does that happen? The reason is that the un-normalized distance is *not* shift/offset and scale invariant.

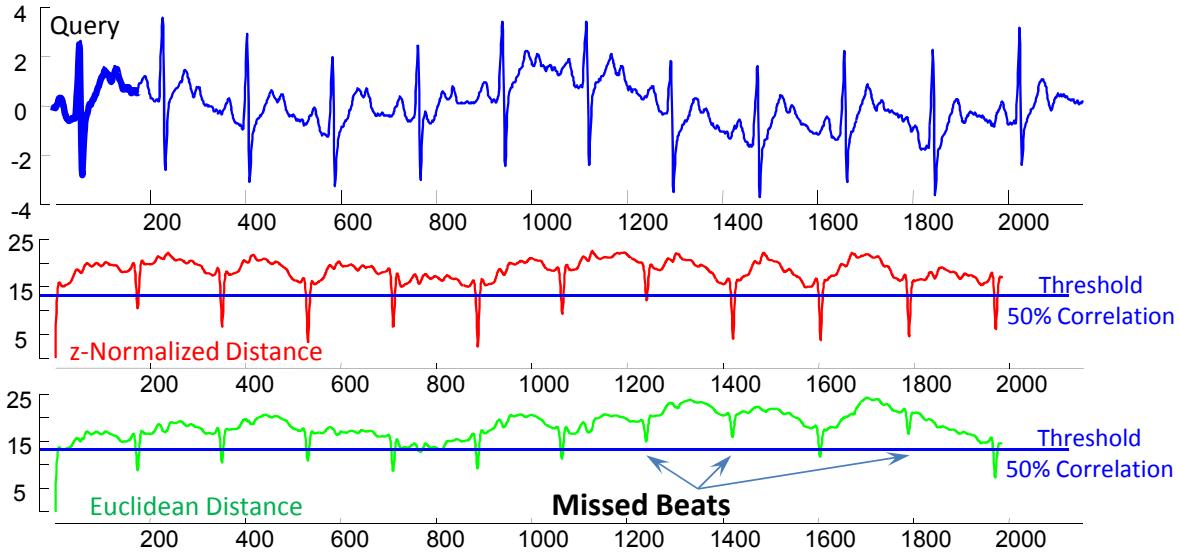

Figure 2.11: (*top*) A segment of ECG with a query. (*middle*) All the twelve beats are detected. Plotting the z-normalized distance from the query to the relevant subsequence (*bottom*) Three of the twelve beats are missed. Plotting the un-normalized Euclidean distance reveals that slight differences in a subsequence's mean value (offset) completely dominate the distance, dwarfing any contribution from the similarity of the shape

Note that while the query and the first subsequence of the trace both have identical local mean, the trace slowly rises and falls to have local means very different from that of the query. This feature is named as the "*Wandering Baseline*" and contributes to the larger Euclidean

distances in Figure 2.11(*bottom*) hence, the missing beats. Similar case exists where the scale of the matching subsequences pushes the distances beyond the threshold.

Readers may find some domains such as body temperature, inflation rate etc. where offset and scale invariances are meaningless. We argue that simple threshold based methods can solve almost all the problems in those domains because offset is all that matters. For example, body temperature of over $100\,^{\circ}\mathrm{F}$ can easily be classified as "fever." Therefore, complex data mining techniques are not very useful for such time series. In general z-normalization works more accurately in mining tasks for many other domains (i.e. on 37 out of 39 problems considered in [29]).

Apart from the benefits of z-normalization, it introduces computational difficulty in all of the algorithms presented in this thesis. Consider two successive subsequences of a time series of length m. They share a subsequence of length $m-1$ in the original time series. Notice that the same pair of subsequences may not have any identical value after they are z-normalized just because they may have different means and standard deviations. Therefore, in none of the algorithms in this thesis we could utilize the overlaps between subsequences.

Since we don't reuse computation in evaluating the distances, we treat a distance computation as the unit of computation in this thesis unless otherwise specified. Therefore an algorithm of $O(n^2)$ requires quadratic number of distance computations.

2.3.5 Extension to Multidimensional Motifs

Because MK works with any metric distance measure, it is very easily extendible to multidimensional time series, so long as we use a metric to measure distances among them. We can use multidimensional Euclidean distance for multidimensional time series which is computed by taking the Euclidean distance for all the values along each of the dimensions. The squared errors from different dimensions can be weighted by their relative importance. Any weighting scheme preserves the metric property of Euclidean distance and thus our *exact* algorithm is directly applicable to multidimensional data. Previously researchers have worked on approximate methods for finding repeated patterns in multidimensional time series [32] and in motion capture data [73]. Being exact, our algorithm bears good promise to be useful in these domains too.

A good example of multidimensional time series is human motion capture data where the 3D positions and angles of several joints of the body are recorded while a subject performs a specific motion. The positions of different joints are synchronous time series and can be considered as different dimensions of a multidimensional time series. While computing the similarity of motion segments, we must define the relative importance (weights) of different body parts. For example, to compare Indian dances we may need to put larger weights on the hands and legs than head and chest. To test the applicability of MK on multidimensional data, we use two dance motions from the CMU motion capture databases and identified the motion-motif shown in Figure 2.12. Each of the dance motions are more than 20 seconds

long and we search for motif of length 1 second. The motion-motif we found is a dance segment that denotes “joy” in Indian dance.

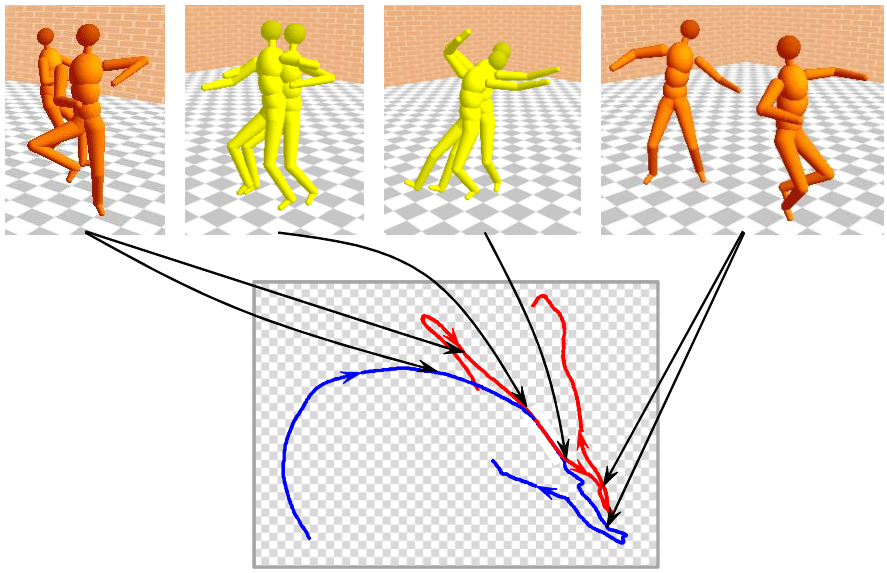

Figure 2.12: An example of multidimensional motif found in the motion captures of two different Indian dances. In the top row, four snapshots of the motions aligned at the motif are shown. In the bottom, the top-view of the dance floor is shown and the arrows show the positions of the subjects

2.3.6 Discussion and Interpretation of Results

In the following sections we interpret and explain the results of the scalability results in more detail.

Why is Early-Abandoning so Effective?

While the results in the previous section bode well for the MK algorithm, a perhaps unexpected result is that just using early-abandoning can make brute force search significantly

faster. While it has been known for some time that early-abandoning can speed up nearest neighbor search; most work suggests that the speedup is a small constant, in the range of two to three [53]. However, at least for the random walk experiment shown in Figure 2.6 it appears early-abandoning can produce at least a ten-fold speed up. It is informative to consider why this is so. The power of early-abandoning comes from the (relative) value of the $best\text{-}so\text{-}far$ variable during search. If it has a small value early on in a search, then most items can be abandoned very quickly. However, we typically have no control over how fast the $best\text{-}so\text{-}far$ decreases; we simply hope that a relatively similar object will be encountered early in the search.

The key insight into explaining why early-abandoning works so well for motif discovery is to note that there are simply many more possibilities for the $best\text{-}so\text{-}far$ to decrease early in the (quadratic) search for a motif, than during the (linear) search for a nearest neighbor. To see this, we performed a simple experiment. We measured the average distance between two time series, the nearest neighbor distance for ten randomly chosen time series, and the motif distance, all for increasingly large instantiations of a database of random walks of length 128. The results are shown in Figure 2.13.

Note that average distance is essentially independent of the dataset size. The mean distance of a query to its nearest neighbor decreases with database size as we would expect, however note that the motif distance decreases more dramatically, and is significantly smaller. This effect is like a real-valued version of the familiar birthday paradox. In a dataset consisting of 23 people, the chance that one of them will share your birthday (the analogue to

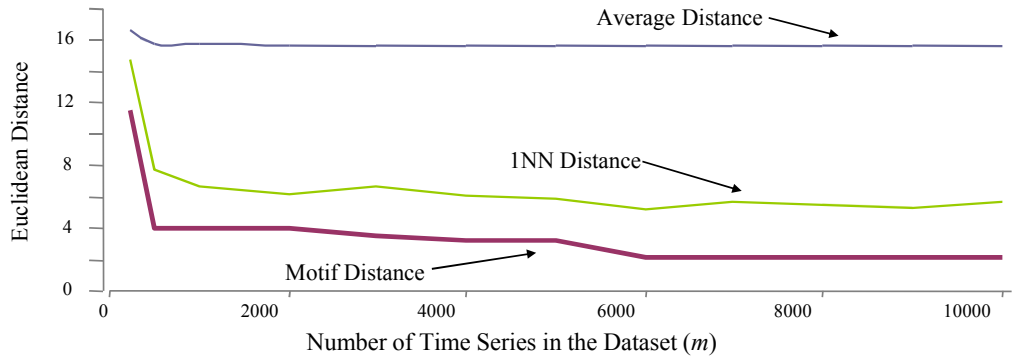

Figure 2.13: How the size of the dataset effects the average, nearest neighbor and motif distances

linear nearest neighbor search) is just 6.1%. However, the chance of any two people sharing a birthday (the analogue to quadratic motif search) is 50.7%. There are simply many more possibilities in the latter case. Likewise, for motif search, there are so many possible ways for pairs to be similar that we can be confident to find very low $best\text{-}so\text{-}far$ early on, and therefore extract the most benefit out of early abandoning.

Why not use DTW or Uniform Scaling?

In this work we have used the classic Euclidean distance as the underlying distance measure. However one could imagine using Dynamic Time Warping (DTW) or Uniform Scaling Euclidean distance (US) instead. In many independent works it has been shown that DTW and US can produce superior classification/clustering accuracy and superior subjective judgments of similarity in diverse time series domains [53][104].

However recent work has forcefully shown that for DTW, it's superiority over Euclidean distance for nearest neighbor classification is an inverse function of the dataset size. As the datasets gets larger, the difference between DTW and Euclidean distance rapidly decreases [92]. To see this, we performed 1NN classification with both DTW and Euclidean distance for increasing large instantiations of the Two-Pattern dataset, a highly "warped" publicly available time series dataset. Figure 2.14 shows the results.

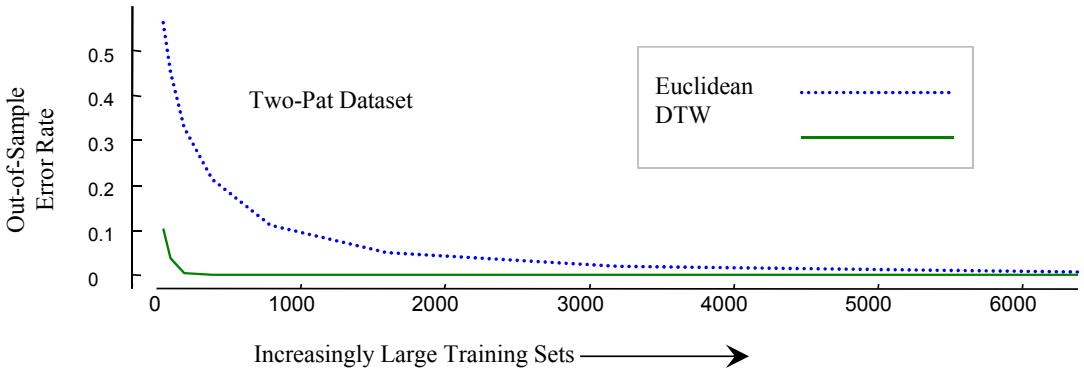

Figure 2.14: The error rate of DTW and ED on increasingly large instantiations of the Two-Pat problem

We have performed similar experiments on 20 other time series datasets, this example is interesting only in that it is the slowest to converge. Upon reflection, this result is unsurprising, as the datasets get larger, the expected distance (under any measure) to the nearest neighbor will decrease (c.f. Figure 2.13). Given this fact, the Euclidean distance is more likely to find a nearest neighbor so near that "warping" the distance (and therefore decreasing the distance) is unlikely to change the rankings of nearest neighbors, and therefore unlikely to change the class prediction.

Given that this is true for 1NN classification, we can expect it to be even more of a factor for motif discovery, since motif discovery has allows many more distance comparisons, and the smallest of them (the motif distance) is likely to be so small that DTW and Euclidean distance will be essentially identical. To see this, we randomly created 300 pairs of random walks, and measured the distance between them using DTW and Euclidean distance. The results are shown in a scatter plot in Figure 2.15.

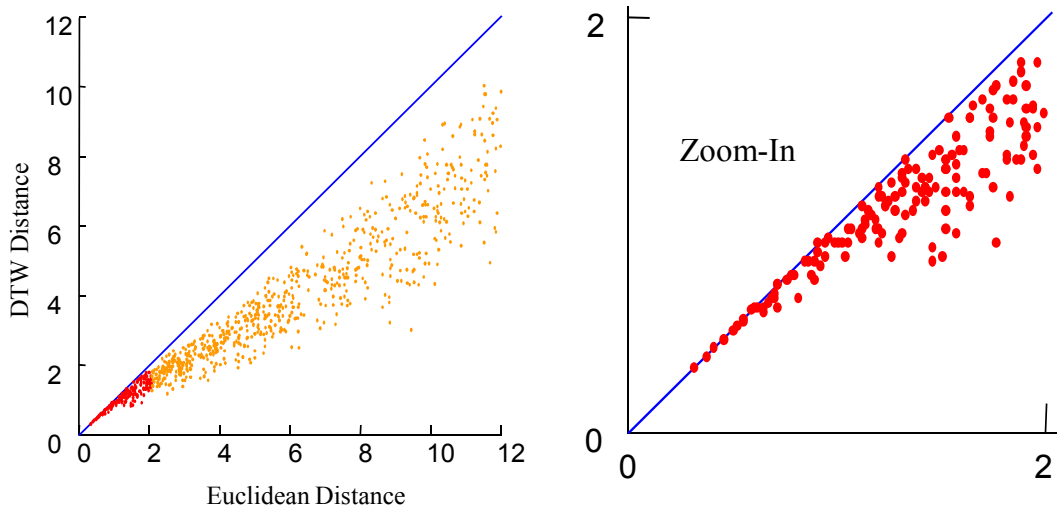

Figure 2.15: (*left*) A scatter plot where each point represents the Euclidean distance (x-axis) and the DTW distance (y-axis) of a pair of time series. Some data points had values greater than 12, they were truncated for clarity (*right*) a zoom-in of the plot on the *left*

We can see that if two objects are relatively far apart under the Euclidean distance, then using DTW can make them appear closer, and change possibility the nearest neighbor ranking. However, as objects get relatively close under the Euclidean distance, the difference between the Euclidean distance and DTW diminishes. In this example, for values under 1.0,

both measures are near perfectly correlated. Empirically we find that for random walks of this length, by they time we have a mere 100,000 objects in the dataset, the average motif distance is usually much less than 0.25.

Given these facts, we can now succinctly answer the question as to why we do not use the DTW distance to find motifs. The answer is that for the very large datasets we consider, it does not make any difference to the result. Identical remarks apply to uniform scaling.

2.4 Experimental Case Studies

Having demonstrated the scalability of our algorithm in the previous section, we now turn our attention to demonstrate the utility of time series motifs in various domains.

2.4.1 Finding Repeated Insect Behavior

In the arid to semi-arid regions of North America, the Beet leafhopper (*Circulifer tenellus*) shown in figure 2.16, is the only known vector (carrier) of curly top virus, which causes major economic losses in a number of crops including sugar-beet, tomato, and beans [49]. In order to mitigate these financial losses, entomologists at the University of California, Riverside are attempting to model and understand the behavior of this insect [94].

It is known that the insects feed by sucking sap from living plants; much like the mosquito sucks blood from mammals and birds. In order to understand the insect's behaviors, entomologists glue a thin wire to the insect's back, complete the circuit through a host plant and

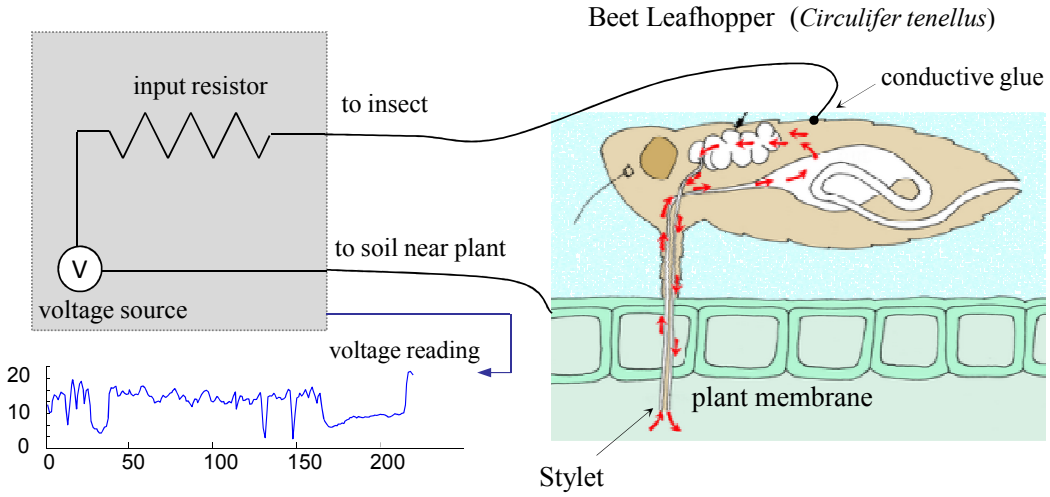

Figure 2.16: A schematic diagram showing the apparatus used to record insect behavior

then measure fluctuations in voltage level to create an Electrical Penetration Graph (EPG) as shown in Figure 2.16.

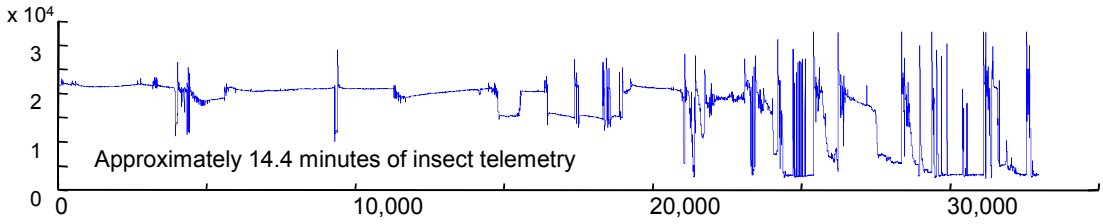

Figure 2.17: An Electrical Penetration Graph of insect behavior. The data is complex and highly non-stationary, with wandering baseline, noise and dropouts

This method of data collection produces large amounts of data, in Figure 2.17 we see about a quarter hour of data, however the entomologists data archive currently contains thousands of hours of such data, collected in a variety of conditions. Up to this point, the only

analysis of this data has been some Fourier analyses, which has produced some suggestive results [94]. However Fourier analysis is somewhat indirect and removed from the raw data. In contrast motif discovery operates on the raw data itself and can potentiality produce more intuitive and useful knowledge. In Figure 2.18 we show the motif of length 480 discovered in the entire 33,021 length time series shown in Figure 2.17.

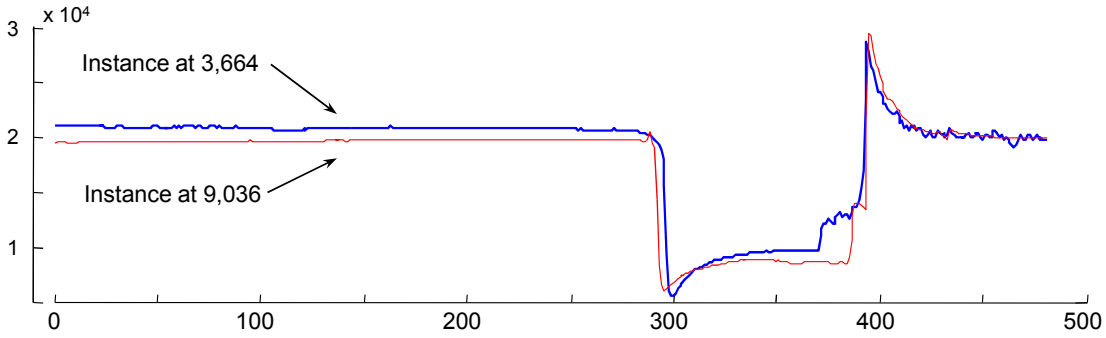

Figure 2.18: The motif of length 480 found in the insect telemetry shown in Figure 2.17. Although the two instances occur minutes apart they are uncannily similar

As we can see, the motifs are uncannily similar, even though they occur minutes apart. Having discovered such a potentially interesting pattern, we follow up to see if it is really significant. The first thing to do is to see if it occurs in other datasets. We have indexed the entire archive with an iSAX index [92] so we quickly determined the answer to be affirmative, this pattern does appear in many other datasets, although the "plateau" region (approximately from 300 to 380 in Figure 2.18) may be linearly scaled by a small amount [94]. We recorded the time of occurrence and looked at the companion video streams which were recorded synchronously with the EPGs. It appears that the motif occurs immediately after phloem (plant sap) ingestion has taken place. The motif discovered in this stream happens to be usually

smooth and highly structured, however motifs can be very complex and noisy. Consider Figure 2.19 which shows a motif extracted from a different trace of length 78,254.

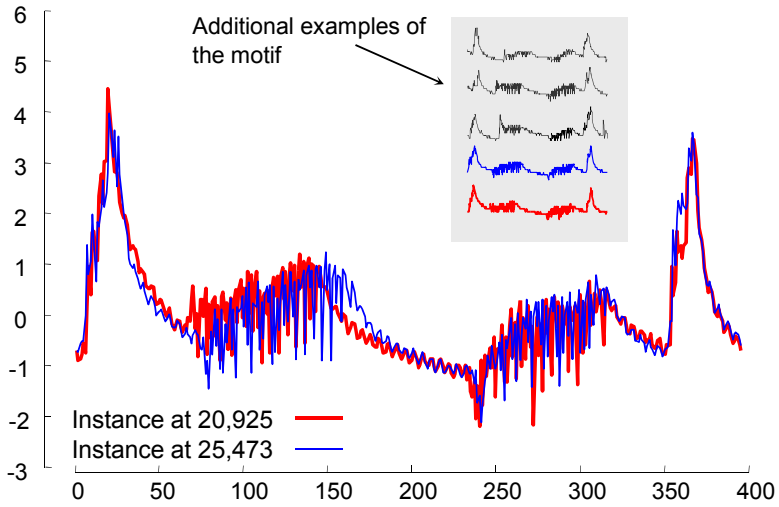

Figure 2.19: The motif of length 400 found in an EPG trace of length 78,254 . (inset) Using the motifs as templates, we can find several other occurrences in the same dataset

In this case, examination of the video suggests that this is a highly ritualized grooming behavior. In particular, the feeding insect must get rid of honeydew (a sticky secretion, which is by-product of sap feeding). As a bead of honeydew is ejected, it temporarily forms a highly conductive bridge between the insect and the plant, drastically affecting the signal.

Note that these examples are just a starting point for entomological research. It would be interesting to see if there are other motifs in the data. Having discovered such motifs we can label them, and then pose various hypotheses. For example: "Does motif A occur more frequently for males than females?". Furthermore, an understanding of which motifs correlate with which behaviors suggests further avenues for additional data collection and experiments. For example, it is widely believed that Beet leafhoppers are repelled by the

presence of marigold plants (Tagetes). It may be possible to use the frequency of (now) known motifs to detect if there really is a difference between the behavior of insect with and without the presence of marigolds. We defer further discussion of such issues to future and ongoing work.

2.4.2 Automatically Constructing EEG Dictionaries

In this example of the utility of time series motifs we discuss an ongoing joint project between the authors and Physicians at Massachusetts General Hospital (MGH) in automatically constructing "dictionaries" of recurring patterns from electroencephalographs. The electroencephalogram (EEG) measures voltage differences across the scalp and reflects the activity of large populations of neurons underlying the recording electrode [76]. Figure 2.20 shows a sample snippet of EEG data. Medical situations in which EEG plays an important role include, but are not limited to, diagnosing and treating epilepsy; planning brain surgery for patient's with intractable epilepsy, monitoring brain activity during cardiac surgery and in certain comatose patients; and distinguishing epileptic seizures from other medical conditions (e.g. "psudoseizures"). The interpretation of EEG interpretation data involves inferring information about the brain (e.g. presence and location of a brain lesion) or brain state (e.g. awake, sleeping, having a seizure) from various temporal and spatial patterns, or graphoelements (which we see as motifs), within the EEG data stream. Over the roughly 100 years since its invention in the early 1900s, electroencephalographers have identified a small collection of clinically meaningful motifs, including entities named "spike-and-wave

complexes", "wicket spikes", "K-complexes", "sleep spindles" and "alpha waves", among many others examples. However, the full "dictionary" of motifs that comprise the EEG contains potentially many yet-undiscovered motifs. In addition, the current, known motifs have been determined based on subjective analysis rather than a principled search. A more complete knowledge of the full complement of EEG motifs may well lead to new insights into the structure of cortical activity in both normal circumstances and in pathological situations including epilepsy, dementia and coma.

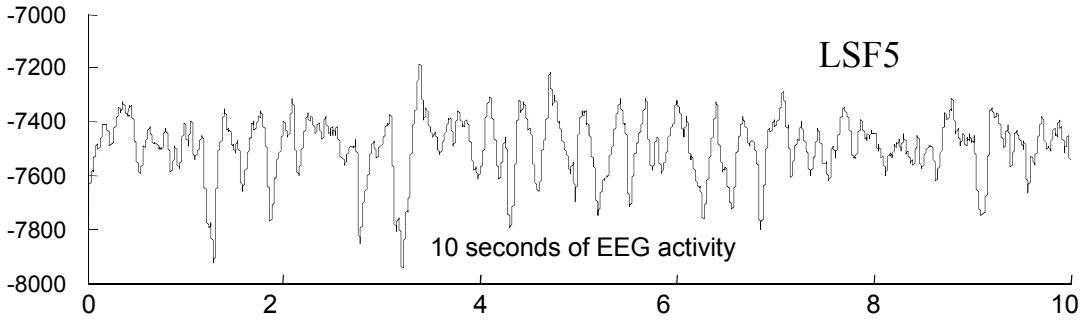

Figure 2.20: The first ten seconds of an EEG trace. In the experiment discussed below, we consider a full hour of this data

Much of the recent research effort has focus on finding typical patterns that may be associated with various conditions and maladies. For example, [96] attempts to be an "Atlas of EEG patterns". However, thus far, all such attempts at finding typical patterns have been done manually and in an ad-hoc fashion. A major challenge for the automated discovery of EEG motifs is large data volumes. To see this, consider the following experiment. We conducted a search for the motif of length 4 seconds, within a one hour EEG from a single channel in a sleeping patient. The data collection rate was 500 Hz, yielding approximately

2 million data points, after domain standard smoothing and filtering, an 180,000 data point signal was produced. Using the brute force algorithm (c.f. Algorithm 2.1), finding the motif required over 24 hours of CPU time. By contrast, using the MK algorithm described in this chapter, the same result requires 2.1 minutes, a speedup of about factor of about 700. Such improvements in processing speed are crucial for tackling the high data volume involved in large-scale EEG analysis. This is especially the case in attempting to complete a dictionary of EEG motifs which incorporates multi-channel data and a wide variety of normal situations and disease states. Having shown that automatic exploration of large EEG datasets is tractable, our attention turns to the question, is it useful? Figure 2.21(*left*) shows the result of our first run of our algorithm and Figure 2.21(*right*) shows a pattern discussed in a recent paper [95].

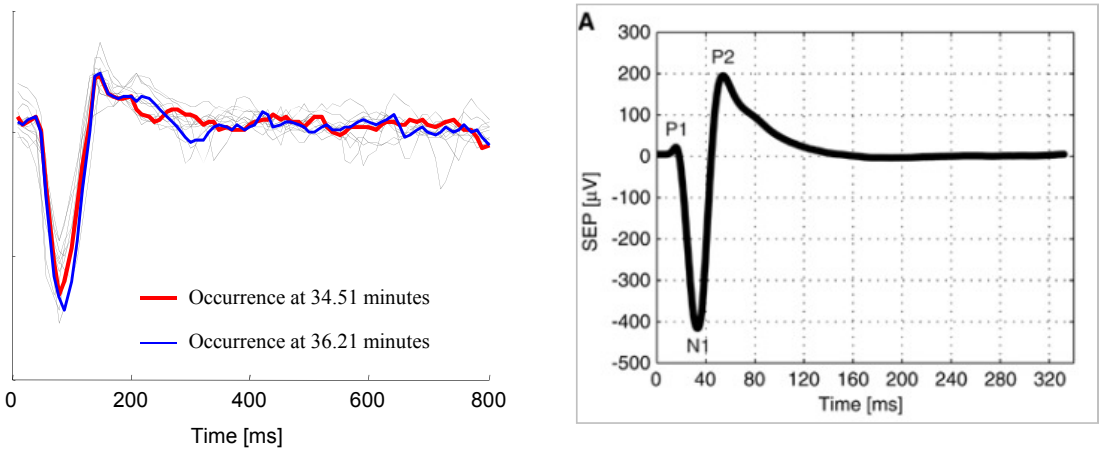

Figure 2.21: (*left*) Bold Lines: The first motif found in one hour of EEG trace LSF5. Light Lines: The ten nearest neighbors to the motif. (*right*) A screen dump of Figure 6.A from paper [95]

It appears that this automatically detected motif corresponds to a well-known pattern, the K-complex. K-complexes were identified in 1938 [76][62] as a characteristic event during the sleep. This figure is at least highly suggestive that in this domain, motif discovery can really find patterns that are of interest to the medical community. In ongoing work we are attempting to see if there are currently unknown patterns hiding in the data.

2.4.3 Motif-based Anytime Time Series Classification

We conclude our experimental section with an example of a novel use for time series motifs. There has been recent interest in converting classic batch data mining algorithms to anytime versions [104]. In some cases this is trivial, for example we can frame the nearest-neighbor classification algorithm as an anytime algorithm simply by conducting a sequential search for the unlabeled items nearest neighbor in the labeled dataset [104]. If the algorithm is interrupted before completing the full search, then the label of the $best\text{-}so\text{-}far$ nearest neighbor is returned as the class label. This simple idea can be enhanced by sorting the labeled instances such that the most useful instances are seen early in the sequential search. In all work that we are aware of, "most useful" is determined by some estimate of how often each instance is used to correctly predict, as opposed to incorrectly predict, unknown instances [104][111]. The astute reader will immediately see a potential weakness here. Suppose we happen to have two nearly identical instances with the same class label in the training dataset. Furthermore, suppose they both happen to useful instances (in the sense discussed above). In this case, both of the instances will be pushed to the head of the sequential search array. However,

this is clearly redundant; we should push either one, but not both top, of the sequential search array. Time series motifs potentially allow a fix for this problem. We can discover the 1st motif, and then move one of the pair to the head of the sequential search array. Then we can rerun motif discovery on the remaining $n - 1$ time series (excluding the recently moved object) again move one of the motif pair to the front, and begin motif discovery on $n - 2$ objects and so on. This strategy should ensure high diversity of the first few training examples encountered by the anytime classification algorithm. We tested this simple idea against random ordering, and a well known ordering algorithm called Drop3 [111]. We considered two publicly available datasets, CBF and Face4.

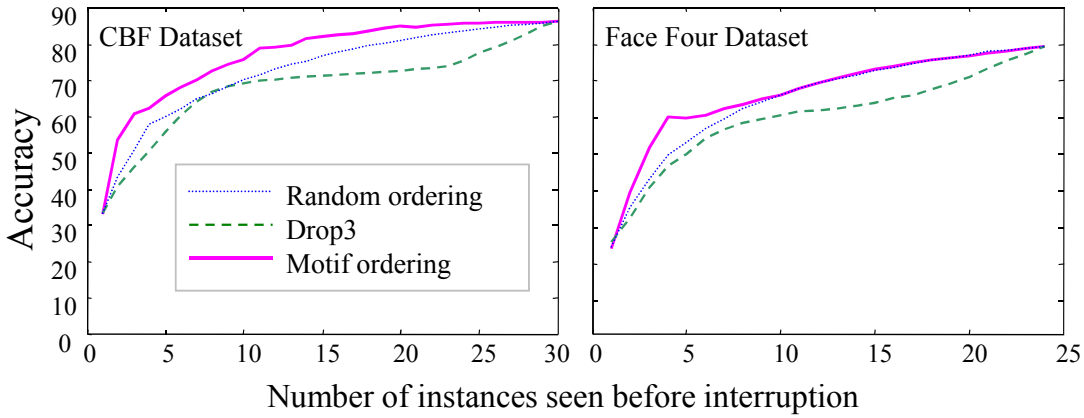

Figure 2.22: The out-of-sample accuracy of three different ordering techniques on two benchmark time series datasets. The y-axis shows the accuracy of 1NN if the algorithm is interrupted after seeing x objects

The results are quite surprising. The motif ordering algorithm is significantly better that Drop3, even though it does not consider any information about useful any individual instance

is, it is merely enhancing the diversity seen by the classifier in the early part of the nearest neighbor search.

2.5 Prior and Related Work

A theoretical lower bound on finding the closest pair of points in 2-dimensional space is $O(n \log n)$ [97]. The trivial extension to an arbitrary d-dimensional space gets a hidden constant term with this lower bound which is exponential to d. For high dimensional data (i.e. long time series) this constant overhead dominates the running time.

The divide-and-conquer algorithm in [11] exploits two properties of virtually all real datasets: sparsity of the data and the ability to select a "good" cut plane to get a true $O(n \log n)$ algorithm. 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 this algorithm also hides a very high constant factor, which is of the order of d. In addition, the large worst-case memory requirement and essentiality random data accesses made the algorithm impractical for disk-resident applications. We compare both the brute force and the divide-and-conquer algorithm to MK and discuss the reason for the amazing speedup we get in Section 2.3.1.

The literature is replete with different ways of defining time series "motifs." Motifs are defined and categorized using their support, distance, cardinality, length, dimension and underlying similarity measure. Motifs may be restricted to have a minimum count of participat-

ing similar subsequences [22][36] or may only be a single closest pair [70]. Motifs may also be restricted to have a distance lower than a threshold [22][36][113] or restricted to have a minimum density [68]. Most of the methods find fixed length motifs [22][36][70][113], while there are a handful of methods for variable length motifs [68][98][99]. Multidimensional motifs and subdimensional motifs are defined [68] and heuristic methods to find them are explored in [68][68][98][32]. 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 [22][88]. In this thesis, 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.

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 [22][84][36][68][98][88]. 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 motifs exactly with respect to the raw time series. More precisely, we want to do an exact search for the most similar pair of subsequences (i.e. the motif) in the raw time series. It has long been held that the exact motif discovery is intractable even for datasets residing in main memory. Note that in the previous chapter we show that motif discovery is tractable for large in-core datasets.

Given this, the most of the literature has focused on fast approximate algorithms for motif discovery [9][22][40][66][68][88][98], however there is one algorithm that proposes to exactly solve the time series motif problem. The FLAME algorithm of [100] is designed to find motifs in discrete strings (i.e. DNA), but the authors show that time series can be discretized and feed into their algorithm. They note that their algorithm "is guaranteed to find the motif"s. However, this is only true with respect to the discrete representation of the data, not with the raw time series itself. Thus the algorithm is approximate for real-valued time series. Furthermore, the algorithm is reported to take about 16 seconds to find the motif of length eleven in a financial time series of length 8,400, whereas our algorithm only takes 0.53 seconds (on average) to find the exact motif, with similar hardware.

Approximate algorithms fair little better. For example, a recent paper on finding approximate motifs reports taking 343 seconds to find motifs in a dataset of length 32,260 [66], in contrast we can find exact motifs in similar datasets, and on similar hardware in under 100 seconds. Similarly, another very recent paper reports taking 15 minutes to find approximate motifs in a dataset of size 111,848 [9], however we can find exact motifs in similar datasets in under 4 minutes. Finally, paper [61] reports five seconds to find approximate motifs in a stock market dataset of size 12,500, whereas our exact algorithm takes less than one second. Bearing these numbers in mind, we will simply ignore other motif finding approaches for the rest of this work.

To the best of our knowledge, our algorithm is completely novel. However there are related ideas in the literature. For example, [20] also exploits the information gained by the

relative distances to randomly chosen reference points. However they use this information to solve the approximate similarity search problem, whereas we use it to solve the exact closest-pair problem.

2.6 Conclusion

In this chapter, we have introduced the first exact motif search algorithm which is significantly faster than the brute force search. We have further demonstrated the utility of motif discovery in a variety of data domains. In the next chapters, we extend this in-memory algorithm for disk resident data and online data.

Chapter 3

Extension to Disk Resident Time Series

In spite of extensive research in recent years [22][36][70][113], finding exact time series motifs in massive databases is an important 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 chapter is organized as follows. We review some of the related work from the database community. We mostly follow the definitions and notations of the Section 2.1 and specify clearly if otherwise. We formally describe our algorithm with elegant examples and empirically evaluate the scalability and utility of our ideas.

3.1 Related Work

To the best of our knowledge, the closest-pair/time series motif problem in high dimensional (i.e. hundreds of dimensions) disk resident data has not been addressed. There has been significant work on spatial closest-pair queries [74][24]. These algorithms use indexing techniques such as R-tree [39] and R*-tree [10] which have the problem of high creation and maintenance cost for multidimensional data [56]. In [109], it has been proved that there is a dimensionality beyond which every data or space partitioning method degenerates into sequential access. Another possible approach could be to use high dimensional self similarity join algorithms [56][30][8]. 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, [56] does consider (an approximation of) stock market time series as one of their examples. However, the threshold must be at least as big as the distance between the closest pair to filter it from the self-join results. This is a problem because most of the time users do not have any idea about a good threshold value. Obviously, user can choose a very large threshold for guaranteed results, but this degrades the performance a lot. In this regard, our method is parameter free and serves the exact purpose of finding the closest pair of time series in disk resident data. In addition, the datasets we wish to consider in this chapter have three orders of magnitude more objects than any of the datasets considered in [24][56][74] and dimensionality (i.e length) of the motifs are from several hundreds to a thousand whereas

in [56] the maximum dimensionality is thirty. Therefore, our algorithm is the first algorithm to find exact time series motifs in disk resident data.

In this chapter we employ a bottom-up search algorithm that simulates the merge steps of the divide-and-conquer approach [11]. 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 orders of magnitude difference in the time it takes to find the motifs on disk-resident datasets.

To the best of our knowledge, our algorithm is completely novel. However we leverage off related ideas in the literature [20][48][116]. In particular, the *iDistance* method of Jagadish et al. [48] introduces the idea of projecting data on to a single line, a core subroutine in our algorithm. Other works, for example [20] also exploits the information gained by the relative distances to randomly chosen reference points. However they use this information to solve the approximate similarity search problem, whereas we use it to solve the exact closest-pair problem. In [48][116], reference objects have been used for each partition of a B+ tree index which is adapted for different data distribution. However, we use only one reference object to do the data ordering. In [30], reference objects (pivots) are used to build an index for similarity joins. While we exploit similar ideas, the design of the index is less useful

for the closest-pair problem because of data replication and parameter setting described previously. Both [48] and [116] use the idea of pivots to do K-nearest neighbor search, and report approximately one order of magnitude speedup over brute force. However we use the idea of pivots for motif discovery and report four to five orders or magnitude. What explains this dramatic difference in speedup? A small fraction can be attributed to the simple fact that we consider significantly larger datasets, and pivot-based pruning is more effective for larger datasets. However, most of the difference can be explained by our recent observation that the speed up of pivot-based indexing depends on the value of the $best\text{-}so\text{-}far$ variable [70]. While this value does decrease with datasets size for K-nearest neighbor search or full joins [30], it decreases much faster for motif discovery, allowing us to prune over 99.99% of distance computations for real-world problems.

3.2 DAME: Disk Aware Motif Enumeration

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.

3.2.1 A Detailed Intuition of Our Algorithm

For this example, we will consider a set of 24 points in 2D space. In Figure 3.1(A) 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 3.1(A)). 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 [69][78]. A snap shot of the database after sorting is shown in Figure 3.1(B). 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 will be using this line in the next chapters; we name 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 3.1(C) 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 3.1(C) is representative, but does not strictly conform to the scale and relative distances of Figure 3.1(A). 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 3.1(C) 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 same principal as the MK algorithm. 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,. . .).

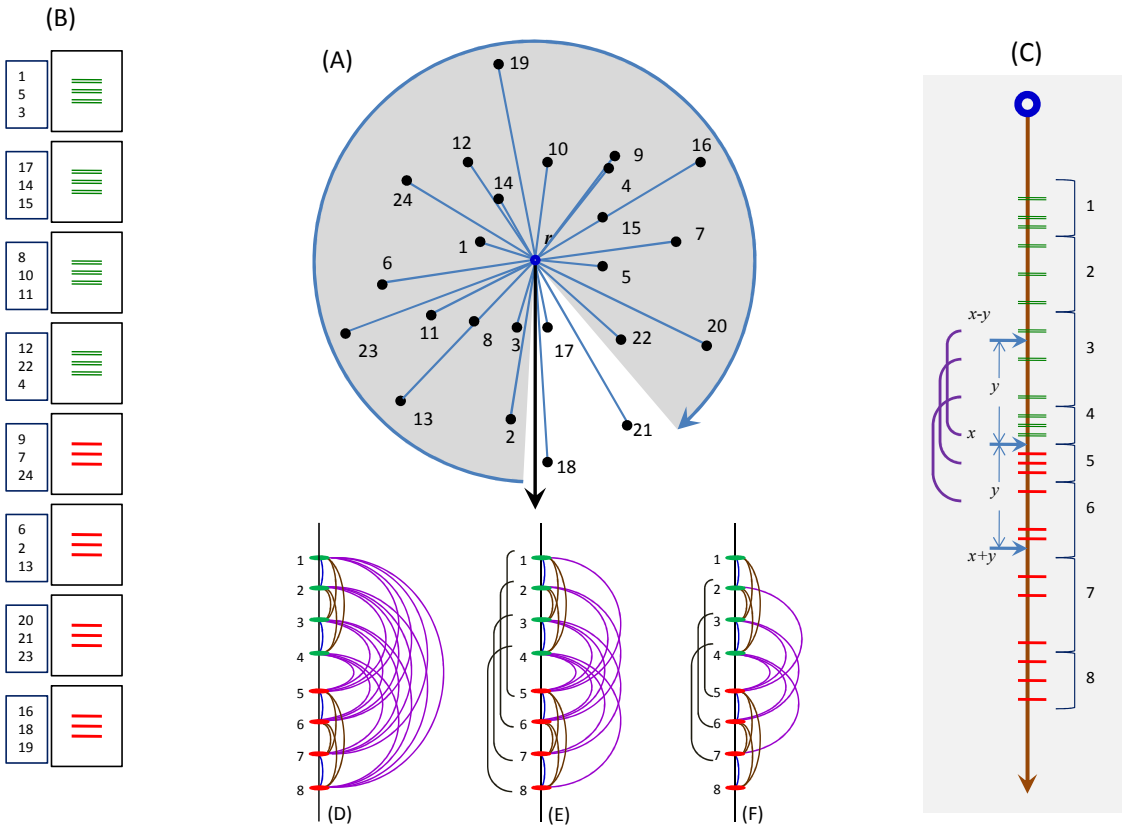

Figure 3.1: (A) A sample database of 24 points. (B) Disk blocks containing the points sorted in the order of the distances from r. The numbers on the left are the ids. (C) All points projected on the order line. (D) A portion of an order line for a block of 8 points. (E) After pruning by a current motif distance of 4.0 units. (F) After pruning by 3.0 units

At each iteration we take disjoint pairs of consecutive groups one at a time and compare all pairs of points that span those two groups. Figure 3.1(D) 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 pairs of consecutive 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. $dist(A, r) \leq dist(B, r)$. By the triangular inequality we know that $dist(A, B) \leq dist(B, r) - dist(A, r)$. But $dist(B, r) - dist(A, r)$ 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 3.1(E), 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 3.1(F)). 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 3.1(B)). 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 3.1(C) 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 3.1(C)). 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.

3.2.2 A Formal Description of DAME

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 some of the variables in the algorithms are assumed to be global variables accessible from all the algorithms. These are $bsf, B, M, R, Dref, L_1$ and L_2. 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 .

Algorithm 3.1 $[L_1, L_2] = DAME\_Motif(B)$

Require: A set B of N blocks each containing M time series
Ensure: Locations L_1 and L_2 of the motif

```
 1: bsf ← ∞
 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 ≤ N/2 do
 8:    top ← 1
 9:    while top < N do
10:       mid ← top + t
11:       bottom ← top + 2t
12:       searchAcrossBlocks(top, mid, bottom)
13:       top ← bottom
14:    t ← 2t
15: return L1,L2
```

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 (short form of "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 $s_q - e_p$ 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.

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 jth object which

Algorithm 3.2 $searchAcrossBlocks(top, mid, bottom)$

Require: top, mid and $bottom$ blocks from B
Ensure: Update L_1 and L_2 if necessary

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

Algorithm 3.3 $[D, Dist] = load(b)$

Require: A block id b
Ensure: Data D and referenced distances in $Dist$

```
1: D ← read(b)
2: for i ← 1 to R do
3:   for j ← 1 to M do
4:     Dist_{i,j} ← distance(Dref_i, D_j)
```

is the first to have a distance larger than bsf in the order line from the ith object. Just as with $searchAcrossBlocks$ method, every pair that has escaped from pruning is given to the update method for further consideration.

The update method does the distance computations and updates the bsf and the motif ids (i.e. L_1 and L_2). The pruning steps described in the earlier methods essentially try to prune

Algorithm 3.4 $searchInBlock(D, Dist)$

Require: Data D and the distances $Dist$ from the references
Ensure: Update L_1 and L_2 if necessary

```
 1: t ← 1
 2: while t ≤ M/2 do
 3:    top ← 1
 4:    while top < M do
 5:       mid ← top + t
 6:       bottom ← top + 2t
 7:       for i ← top to mid − 1 do
 8:          for j ← mid to bottom − 1 do
 9:             if Dist_{1,i} − Dist_{1,j} < bsf then                prune step
10:                update(D, D, Dist, Dist, i, j)
11:             else                                                 prune step
12:                break                                             prune step
13:       top ← bottom
14:    t ← 2t
```

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 the previous section, 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 loads are used to get R $lower\_bounds$, 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 (c.f. section 2.3.2).

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 [22][70] 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.

Algorithm 3.5 $update(D1, D2, Dist1, Dist2, x, y)$

Require: Data $D1, D2$, the distances $Dist1, Dist2$ and the locations x, y
Ensure: Update L_1 and L_2 if necessary

```
1: reject ← false
2: for i ← 2 to R do                                              prune step
3:    lower_bound ← |Dist1_{i,x} − Dist2_{i,y}|                   prune step
4:    if lower_bound > bsf then                                   prune step
5:       reject ← true, break                                     prune step
6:       if reject = false and trivial(D1_x, D2_y) = false then   prune step
7:          if distance(D1_x, D2_y) < bsf then
8:             bsf ← distance(D1_x, D2_y)
9:             L_1 ← id(D1_x), L_2 ← id(D2_y)
```

3.2.3 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 3.1 *The bottom-up search compares all possible pairs if the pruning steps are removed.*

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

Lemma 3.2 *Pruning steps ignore pairs safely.*

Proof: Follows from Section 3.2.2. ∎

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.

3.3 Scalability Experiments

In this section we describe experimental results to demonstrate $DAME$'s scalability and performance. Experiments in Sections 3.3.1 to 3.3.3 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. As before, we have built a webpage 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 thesis [2]. 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 [102]) 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)*0.5 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 [99][70][36][22][84] 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 in [102] took eight months to be collected on a dedicated machine running 24 hours a day. 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.

3.3.1 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 in ascii format. 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 3.2 shows the execution times against the database size in the number of random walks.

Figure 3.2: Execution times in days on random walks and EOG data

In another independent experiment we use $DAME$ on a very long and highly oversampled real time series (EOG trace, cf. Section 3.3.3) 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 3.2 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.

3.3.2 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 3.3(*left*)) and for $searchAcrossBlocks$ (Figure 3.3(*right*)).

Figure 3.3(*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

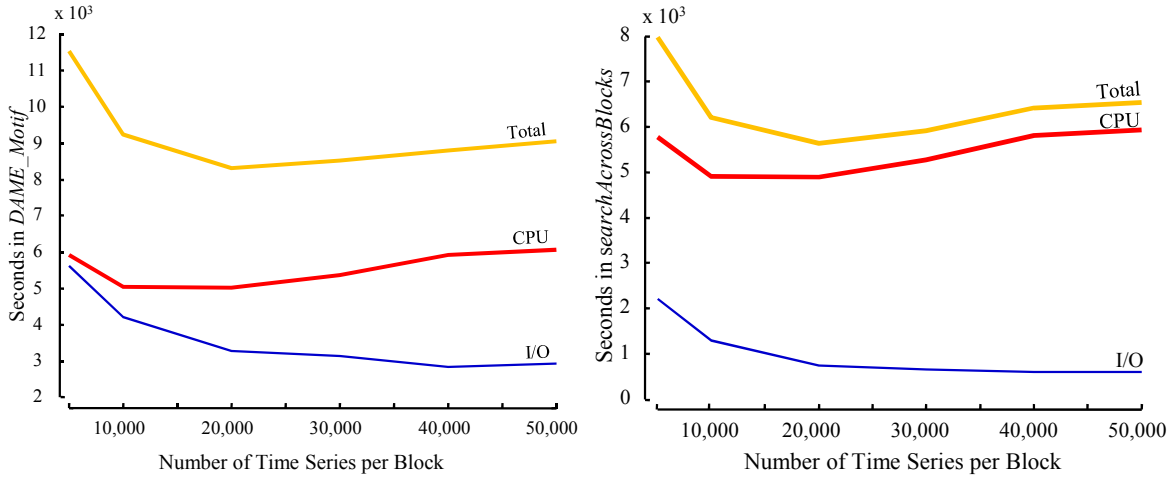

Figure 3.3: Total execution times with CPU and I/O components recorded on one million random walks for different block sizes (*left*) for the DAME_Motif method and (*right*) for the searchAcrossBlocks method

point is compared to points from multiple blocks and larger number of disk accesses is required. 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 3.3(*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 one adopted in $DAME$ will make little difference in the total execution time.

3.3.3 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 3.3.1. 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 3.4(*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 [70].

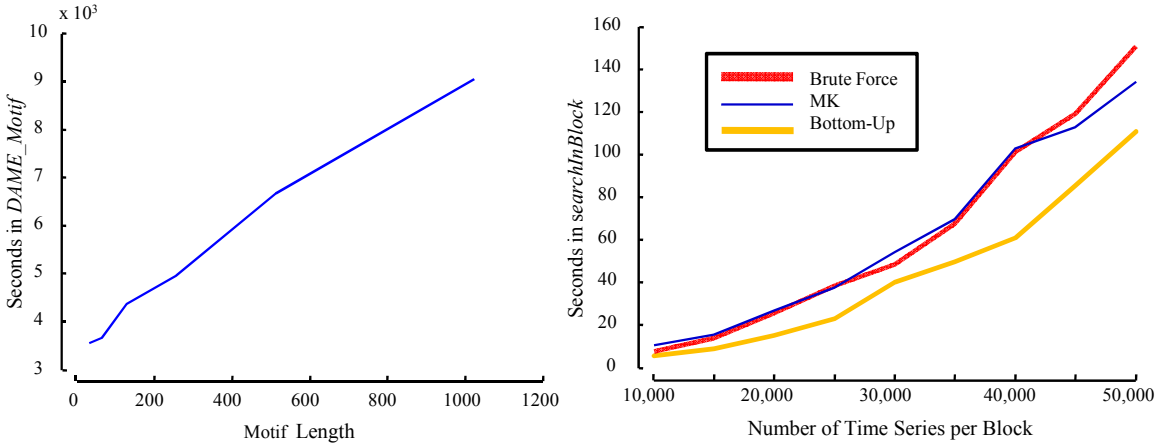

Figure 3.4: (*left*) Execution times on one million random walks of different lengths. (*right*) Comparison of in-memory search methods

3.3.4 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 [70]. 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 3.4(*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.

3.4 Experimental Case Studies

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

3.4.1 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 [15]. 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 [58][28] 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 3.5 shows the discovered motif.

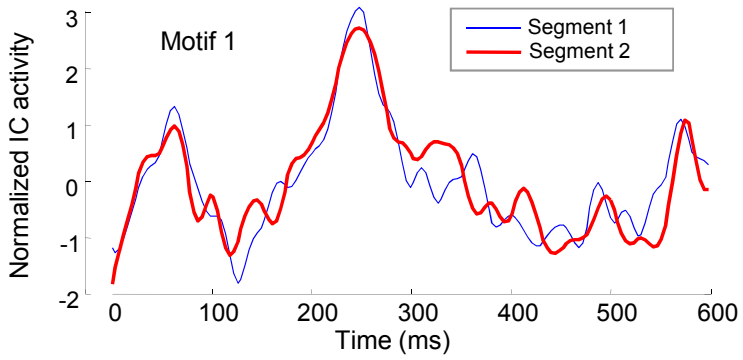

Figure 3.5: Two subsequences corresponding to the first motif

Figure 3.6 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 3.5. 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.

Motifs can also be used as classifier features. Figure 3.7 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 3.7.

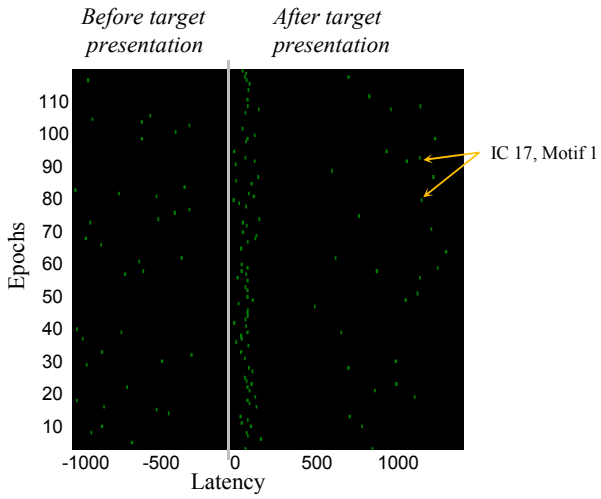

Figure 3.6: Motif 1 start latencies in epochs

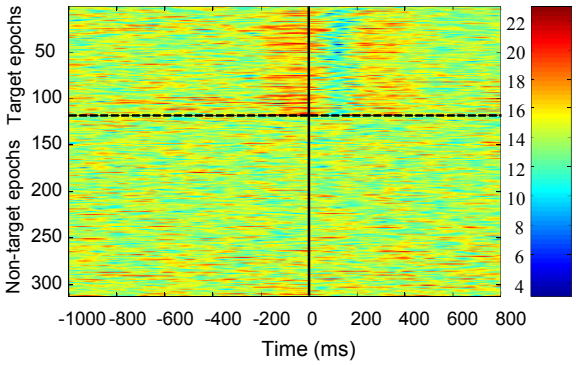

Figure 3.7: Euclidean distance to Motif 1

3.4.2 Detecting Near-Duplicate Images

Algorithms for near-duplicate detection in images are useful for finding copyright violations, detecting forged images, and summarizing large image collections (by showing only one example from a set of near duplicates). These algorithms can be seen as two-dimensional analogues of time series motif discovery. While many specialized algorithms exist for this

problem, it is clear that time series motif discovery could be used for this task, if we can find a way to meaningfully representation images as "time series." While there are many possible ways to convert two-dimensional images to a one-dimensional signal, the most obvious is the classic trick of treating the color distribution histogram as a time series [41].

To test this idea, we use the 40 million images from the dataset in [102]. We convert each image to a pseudo time series by concatenating its normalized color histograms for the three primary colors [38]. 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). In Figure 3.8, 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.

3.4.3 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 ECG and respiration. Repeated

Figure 3.8: (*left*) Five identical pairs of images. (*right*) Five very similar, but non-identical pairs

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" [38]. 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 3.9.

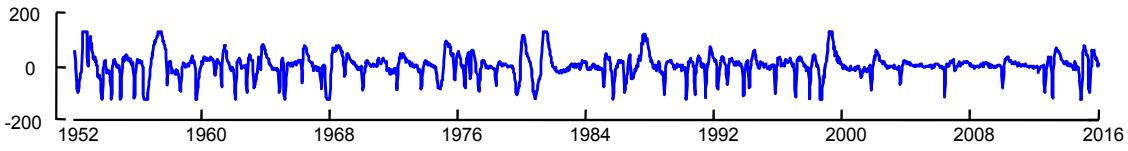

Figure 3.9: A section of the EOG from the polysomnogram traces

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 3.10 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.

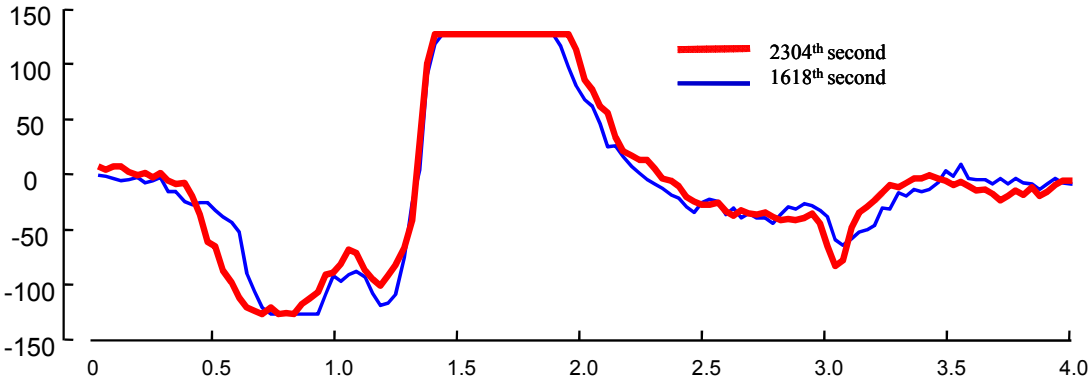

Figure 3.10: Motif of length 4.0 seconds found in the EOG

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.

3.5 Conclusion

In this chapter we have introduced the disk aware 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 may consider parallelization, given that the search for different group sizes can easily be delegated to different processors.

Chapter 4

Extension to Streaming Time Series

Time series motifs are approximately repeated subsequences of a longer time series stream. Figure 4.1 shows an example of a ten-minute long motif discovered in telemetry from a shuttle mission.

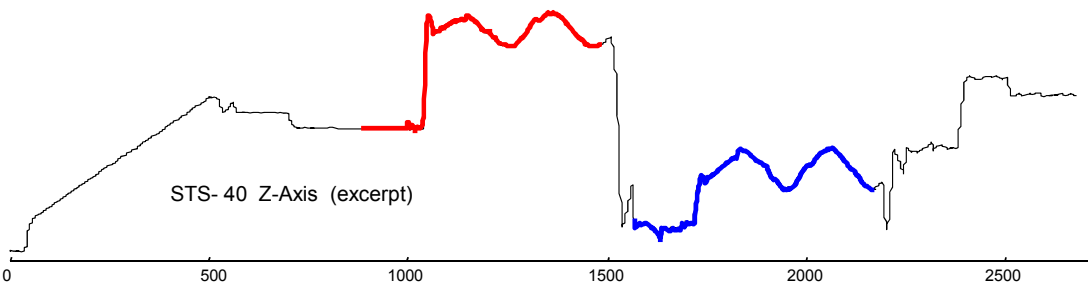

Figure 4.1: Forty-five minutes of Space Shuttle telemetry from an accelerometer. The two occurrences of the best ten-minute long motif are highlighted

Whenever a repeated structure is discovered, it immediately suggests some underlying reason for the conservation of the pattern. In this case a little investigation tells us that this

pattern is indicative of a "correction burn" subroutine to compensate for random drift in the orbiter.

However, as others have observed in many other settings, most data sources are not static but dynamic, and data may stream in effectively forever. This suggests two obvious questions: is it possible to discover and maintain motifs on streaming data, and is it meaningful and useful to do so? In this thesis we answer both questions in the affirmative. We develop the first online motif discovery algorithm which monitors and maintains exact motifs in real time over the most recent history of a stream. While we defer a formal definition of the problem until later, Figure 4.2 gives a visual intuition of the problem .

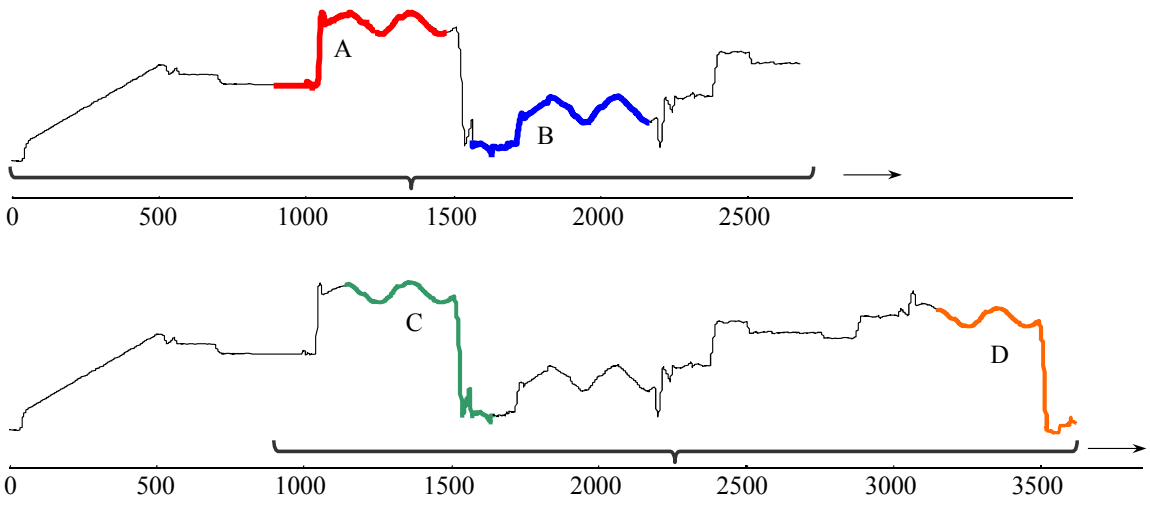

Figure 4.2: Maintaining motifs on a forty-five minute long sliding window. (*top*) Initially A and B are the motif pair. (*bottom*) but at time 790, two new subsequences C and D become the new motif pair of subsequences

Our algorithm has a worst-case update time which is linear to the window size, allowing deployment in realistic settings with current off-the-shelf hardware. As to the utility of

streaming motif discovery, we show empirically its usefulness on several real world problems in the domains of robotics, wildlife monitoring, and online compression. In addition, we show that our core ideas allow useful extensions of our algorithm to deal with data sources that produce data at changing rates and discovering motifs in multidimensional streams.

The rest of this chapter is organized as follows. In Section 2 we introduce necessary background materials and notation and in Section 3 we discuss related work. In Section 4 we introduce our algorithm, and in Section 5 evaluate its performance. Section 6 we consider some extensions to allow us to solve related problems. Section 7 sees an extensive testing of our ideas in several diverse domains, and we offer conclusions and directions in Section 8.

4.1 Notation and Background

Our algorithm considers real time streaming environments. In this section we define the environment in which our algorithm works and the notion of online motif. We begin by defining the data type of interest, a streaming time series:

Definition 4.1 [STREAMING TIME SERIES] A streaming time series is a continuous sequence $x = (x_1, x_2, \ldots, x_t)$ of real-valued numbers where x_t is the most recent value. The numbers are generated at a rate of λ, which can be constant or variable within a range.

We are not interested in the entire history of the time series, but rather the recent history, which we capture in a sliding window:

Definition 4.2 [SLIDING WINDOW] A sliding window (W) is the latest w numbers $(x_{t-w+1}, x_{t-w+2}, \ldots, x_t)$ in the streaming time series x.

Within a sliding window, we are interested in motifs, which informally are repeated subsequences. We restate the definition of a subsequence for completeness.

Definition 4.3 [SUBSEQUENCE] A subsequence of length m of a streaming time series $x = (x_1, x_2, \ldots, x_t)$ is a time series $x_{i,m} = (x_i, x_{i+1}, \ldots, x_{i+m-1})$ for $1 \leq i \leq t - m + 1$.

We are now in a position to define the online motif. We define the online motif of length m in the most recent sliding window as the most similar non-overlapping pair of subsequences.

Definition 4.4 [ONLINE MOTIF] The online motif of length m of a time series $x = (x_1, x_2, \ldots, x_t)$ is a pair of subsequences $(x_{i,m}, x_{j,m})$ for $1 \leq i < i + m \leq j \leq t - m + 1$ such that $d(x_{i,m}, x_{j,m})$ is the smallest among all such pairs.

The reason for considering only the non-overlapping sequences is to avoid trivial matches that are inherently similar because they share most of their values [22]. Glancing back at Figure 4.2, we can see the examples of online motifs, which changed as time passed. Now we can define the class of algorithms our method belongs to.

Definition 4.5 [EXACT ONLINE MOTIF] The exact search for the online motif of length m of a time series $x = (x_1, x_2, \ldots, x_t)$ finds the pair of subsequences $(x_{i,m}, x_{j,m})$ for $1 \leq$

$i < i+m \leq j \leq t-m+1$ such that Euclidean distance between $x_{i,m}$ and $x_{j,m}$ is the smallest among all such pairs.

Note that there is always a motif pair under the definition of an exact search. We denote the output produced by an exact search as the exact motif, as opposed to the approximate motifs, which may not be the most similar pair under Euclidean distance.

For ease of presentation we only discuss the case of maintaining a single motif pair. However, it is simple to modify our algorithm to maintain a pattern that appears k-times or to maintain all pairs having distances smaller than a threshold.

To measure the distance between subsequences we use the ubiquitous z-normalized Euclidean distance as before.

At every time tick, a new subsequence $x_{t-m+2,m}$ of length m is generated in W and the oldest subsequence $x_{t-w,m}$ is deleted from W. Therefore, in our model of online motif discovery we assume that at every time tick a new object/point (i.e. subsequence) is generated and the oldest object/point is deleted. Objects may have an exclusion condition (for example, to avoid trivial matches) specifying the objects with which it should not be compared. This model is general enough to discover the online motif in streams of independent objects such as individual images, video frames, transactions and motion poses.

Our algorithm requires $O(wm)$ arithmetic operations to compute all distances in one update. The most costly part of this is floating point multiplication. Let's assume a pessimistic constant of b which roughly denotes the amount of time each floating point operation takes. In current computers, b can be close to 10^{-8} seconds. Given this and a user-given (m, w)

pair, we can easily compute the maximum rate ($\frac{1}{bmw}$) at which our algorithm guarantees to operate. We assume that λ is below this maximum rate until Section 4.6.1, where we remove the restriction.

4.1.1 Why is this Problem Hard?

Here we explicitly state why this problem does not lend itself to simple or "off-the-shelf" solutions. The issues are well known in the general context of dynamic closest pair [34], but are worth restating here.

Assume that we have identified the motif pair in a window W. We know the exact locations of the two occurrences of the motif, and their exact Euclidean distance D. If we now insert a single data point at the head of the queue, what do we now know? The answer is very little; the motif pair may have changed, and if it has, then all we know is that the new motif pair has a distance of at most D. The locations of the new motif pair can be anywhere. Suppose instead that we delete a single data point from the tail of the queue, what do we now know? The answer is again, very little, as the new locations of the motif pair can be anywhere. All we know is the (now) lower bound D on the motif distance. However, in the case we are considering, we both insert (enqueue) and delete (dequeue) at each time tick, so we have neither an upper nor lower bound on the motif distance, nor any constraint on where they might be. So in principle, we can be forced to completely "resolve" the motif discovery at each time step, with a $O(w^2m)$ cost, even though our data has changed only a tiny amount,

say, 0.0001%. However, as we shall show in the next section, by caching some computations we can guarantee that we can maintain the motifs in just $O(wm)$ time.

4.2 Related Work

Eppstein describes an algorithm for maintaining the closest pair of points under any distance measure [34]. This algorithm solves a slightly more general problem than the one we consider, in that it can have any arbitrary order of insertion and deletion, and it does not require metric properties in the distance measure. It has found use mainly in speeding up agglomerative clustering and in some other offline applications. There is a subtle but critical difference between the dynamic closest pair maintained in [34][74] and the online motif discovery we consider here. In our case we are in a streaming environment where for every update we have a fixed time until the next value comes in. As such we must optimize the worst-case time at each insertion for an application that runs forever. In contrast, the method in [34] optimizes the total running time after all of the updates (i.e. the clustering) for an application that runs for a finite time. These distinctions are critical, and cannot be removed by assuming a buffer in which we temporarily cache difficult cases, since an arbitrary number of difficult cases may arrive one after another to overflow the buffer.

Another approach in dealing with dynamic closest pair maintenance is commonly known as the "lazy approach" [17]. Here the data structure is not updated until the closest pair changes. In a streaming scenario we are interested in, this idea reduces the amortized time

costs, but does not allow us to tightly bound the time per individual object arrival on arbitrary streams.

In [12] an optimal algorithm for maintaining the closest pair of points is described. It runs in logarithmic time with linear space. This algorithm works by hierarchically dividing the space into sub-spaces and has a problematic exponential constant (2^d) where d is the dimensionality of the objects. It is well understood that space/data partitioning methods do not work well beyond dimensionality on the order of eight to ten [109]. However, the time series motifs that we are trying to maintain can be of any length from hundreds to thousands.

Eppstein actually introduces two different types of data structures in [34], a *quadratic space-linear update* time and a *linear space-*$O(w \log^2 w)$ *update* time considering constant m. We believe that for the general dynamic closest pair problem these are currently the best two choices. Our algorithm falls in the first category and utilizes the temporal ordering of updates to have an amortized $O(w^{\frac{3}{2}})$ space complexity.

In [26], statistics such as average, sum, minimum, maximum etc. are maintained over a sliding window. Their objective is to approximate these statistics in bounded space and time, whereas we are dealing with higher level statistics ,i.e. the closest pair. Our work can be seen as an attempt to add motif to the set of statistics that can be maintained; however none of the techniques in [26] are of direct help to us. In summary, to the best of our knowledge, none of this work, nor the rest of the literature on maintaining the closest pair of points has direct bearing on the exact search problem.

4.3 Online Monitoring of Motif

In this section we describe our algorithm with a running example. Assume that we are given a set of eight points in 2D as shown in Figure 4.3(*left*) (for now ignore the connecting arrows). Every point is numbered by the timestamp of their time of arrival. Recall that our task is to find the closest pair of points (currently 4 and 1), and maintain the closest pair as we simultaneously delete 1 and insert 9, then delete 2/insert 10, then delete 3/insert 11 and so on. We will begin with a naive version and revise it to define our algorithm.

4.3.1 The First Solution

First note that the closest pair in Figure 4.3(*left*) can be changed by one or both of the following two events (see Figure 4.3(*right*)):

- **Deletion**: If one of the objects in the closest pair is deleted, there must be a new closest pair having a distance not less than that of the departing closest pair. For example, after 1 is deleted (8,2) is the new closest pair.

- **Insertion**: If the new object is closer to any object than the current closest pair, the motif pair must be updated to reflect that. For example, (6,9) is the new closest pair after the insertion of 9.

Note that in our example, the closest pair has been changed by both the insertion and deletion.

Now, the arrows connecting the points in Figure 4.3 represent the nearest neighbor relation. For example, the arrow from 5 to 2 denotes that 2 is the nearest neighbor of 5. To

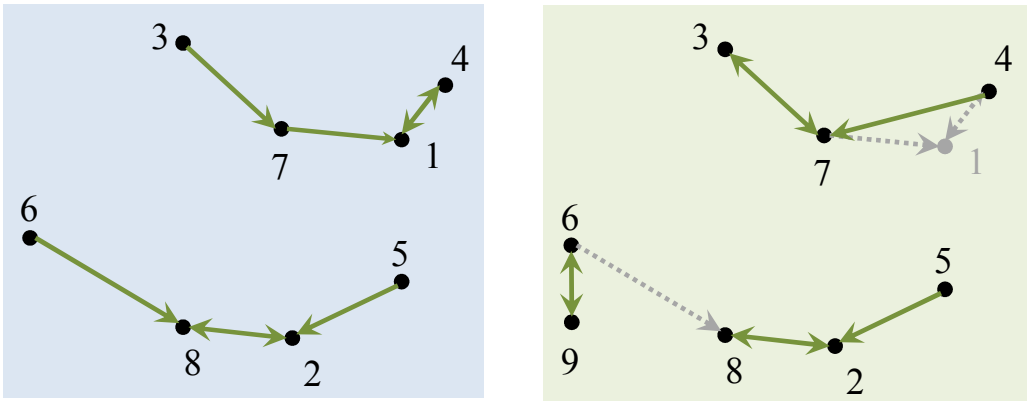

Figure 4.3: (*left*) A set of 8 points. (*right*) At a certain time tick 1 is deleted and 9 is inserted

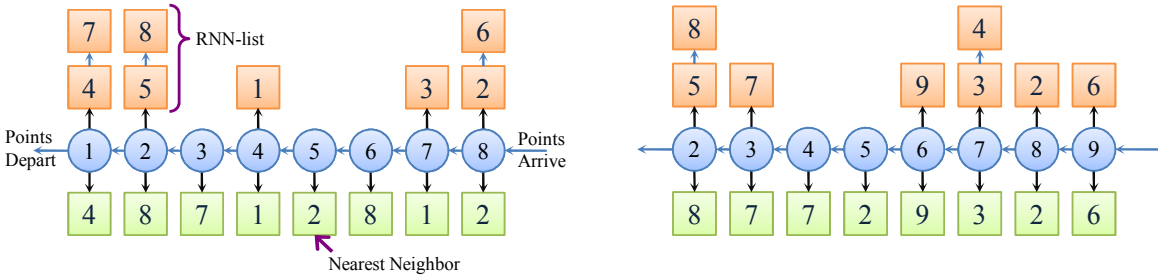

Figure 4.4: (*left*) The data structure of points. (*right*) The data structure after the update (1 is deleted and 9 is inserted)

maintain the closest pair online, our first choice is to track the nearest neighbors of all of the objects. We use the data structure shown in Figure 4.4(*left*) for this purpose. Here the horizontal arrows show the direction of insertion and deletion of normalized subsequences represented as points. Each data point is associated with a list of pointers to the reverse nearest neighbors, the **RNN-list**. RNN-list is not ordered therefore insertion to it is a constant time operation. A data point also has a pointer to its nearest neighbor, **NN**. With each pointer the distance associated with the pair is also stored. If we can maintain such a data structure,

we can answer the closest pair query for this sliding window efficiently simply by finding the minimum of the NN distances. Next we show how we update this data structure.

Update upon insertion: When a new point 9 is inserted, the distances to all of the existing points (1-8) from 9 are computed to find its NN (i.e. 6). While computing the distances we may find that the new point is nearest to an older point. Therefore, we may need to reset an older point's NN as well as the new point's RNN-list. For example, after 9 is inserted, the NN of 6 is changed to 9 from 8 (Figure 4.4(*right*)), and also, 6 is inserted in the RNN-list of 9. After the nearest neighbor x of the new object is found we need to update the RNN-list of x. For example, the NN of 9 is 6 and therefore, 9 is added in the RNN-list of 6 (Figure 4.4(*right*)). The update upon insertion is $O(wm)$, as we have no way to avoid those $O(m)$ distance computations.

Update upon deletion: To handle deletion we need to look at the RNN-list of the departing point. For each of those reverse nearest neighbors, we need to find their new nearest neighbors. For example, after 1 is deleted, both 4 and 7 have been assigned new nearest neighbors (Figure 4.4(*right*)). In the worst case, a point can have $O(w)$ reverse nearest neighbor and thus the naive approach to handle the deletion would take $O(w^2m)$ time.

Counting both insertion and deletion, the naive algorithm needs $O(w^2m)$ update time. The space complexity is $O(w)$ since each point appears exactly once in all of the RNN-lists. In the next version of our algorithm we reduce the update time complexity to $O(w^2)$. As visually hinted at in Figure 5.2(*left*), we create a huge space overhead in addressing the problem, which we will mitigate later.

The squared space version:

In this version we change the data structure to store a complete neighbor list (N-list) instead of just the nearest neighbor (NN). The N-list entries are sorted by the distances from the owner of the list (Figure 5.2(*left*)). Here also the closest pair is the minimum of the first points of the N-lists.

Update upon insertion: The new object needs to be compared with every old object and be inserted in every old object's N-list in distance order. If we implement N-list by *min-heap* then insertion in an N-list is $O(\log w)$. As a whole, the insertion cost can be as low as $O(wm)$. If N-lists are simple linked-lists, the insertion cost would be $O(w^2)$ since ordered insertion in is $O(w)$ and $w > m$.

Update upon deletion: For every reverse nearest neighbor x of the departing point p, we delete the first few entries (including the departing one) from the N-list of x to get the next nearest neighbor y within the sliding window. We also insert x in the RNN-list of y. For example, when 1 goes out of the sliding window (Figure 5.2(*left*)), 1 is deleted from the heads of the N-lists of all of its RNNs (7 and 4). Then 7 and 4 are inserted in the RNN-lists of 3 and 7 respectively. Similarly, when 2 departs, 2 is deleted from 5's N-list leaving 1 in the head of 5's N-list. Since 1 would be an invalid entry as it is already out of the sliding window, it is also deleted for consistency. 2 is then deleted from 8's N-list leaving 6 in the head which is a valid entry as 6 is not yet departed.

If we use min-heap we may need to heapify after every deletion to get the next minimum distance. Therefore, min-heap increases the deletion cost to $O(w^2 \log w)$. For simple linked-

list, the worst case is $O(w^2)$ as we may need to delete w^2 entries from an overgrown $2w^2$ sized data structure.

Altogether, we opt for simple linked-list as the data structure for the N-list and can perform an update in $O(w^2)$ time.

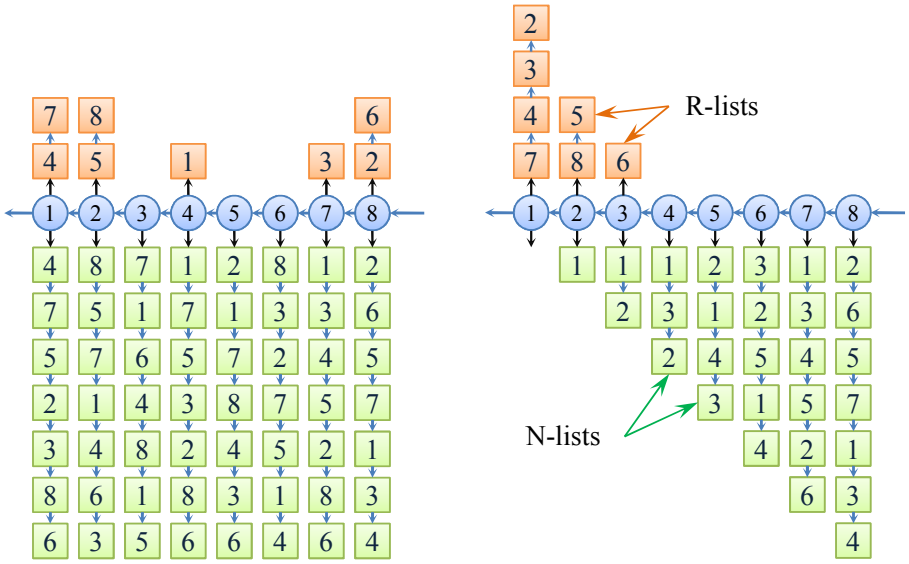

Figure 4.5: (*left*) The squared space structure. Each point has one RNN-list (upper part) and one N-list (lower part). Both of the lists are in order of the distances. (*right*) The reduction of space using observation 4.1

4.3.2 Reducing Space and Time Complexity

We use two observations stated below for further refinement.

Observation 4.1 *Every pair of points appears twice in the data structure. If we keep just one copy of each, it is still possible to retrieve the closest pair from this data structure.*

To exploit the above observation, we can skip updating the old N-lists during insertion even if the new point becomes the nearest neighbor of an older point. That way the insertion involves only building the N-list of the new point and inserting into exactly one RNN list. This is clearly $O(wm)$ as we can sort the list after inserting all of the old objects. Figure 5.2(*right*) shows the data structure after applying observation 4.1. Note that the N-list of a point now only holds points that arrived earlier than it. Also note that the RNN-lists contain only later points. For example, the RNN-list of 7 does not have 3 although 7 is the NN of 3. The RNN-list of a point is built when subsequent points are added and we will denote it as R-list (Reverse list) from this point on. The reason is that R-list points to the opposite direction of N-list and stores the pointer to the later/newer N-lists where its owner is in the head.

Deletion is still $O(w^2)$. Since the N-lists are always kept sorted and valid, the motif pair is guaranteed to be among the first points of the N-lists as before.

Observation 4.2 *A point x can never make a motif (x, y) with a later point y if there is a point $x < z < y$ such that $d(x, y) \leq d(z, y)$.*

This is because (z, y) would remain the closest pair when x goes out of the sliding window. The direct implication of the above is that the points in an N-list can be stored in the strict increasing order of their timestamps starting with the nearest neighbor. Obviously the distance ordering must be preserved.

For example, (6,4) will never get a chance to be the motif because (6,5) has smaller distance than (6,4) (Figure 5.2(*right*)) and we can safely skip (6,4) when the N-list of 6 is

created in the newer version (Figure 4.6(*left*)). Note that $< 2, 5 >$ is a strictly increasing sub-list of the N-list of 6, but it does not start with the nearest neighbor (3) and so it would be an erroneous N-list. The correct N-list for 6 is $< 3, 5 >$ as shown in Figure 4.6(*left*).

After building the N-list, we can use observation to delete some of the older points safely and build a strictly time ordered list by only one pass over the N-list. Therefore, it does not increase the insertion cost. As a benefit of the strict temporal ordering, now a departing point can only occur in the head of the N-lists of the points in its R-lists and nowhere else. This removes the burden of deleting extraneous pointers after the heads at deletion time and reduces the deletion cost to $O(w)$. The update cost is dominated by the distance computations upon insertion which is $O(wm)$.

The space complexity still appears as worst-case-quadratic with the above two observations. In the worst-case, the N-list of every point could contain all of the previous points exactly in the order of their arrival. However, we argue that such a pathological worst case can never occur. In terms of amortized space cost, we can prove that our algorithm needs $O(w^{\frac{3}{2}})$ amortized space. The proof is the following.

The N-list of a point arriving at time t can be any of the random permutations of all of the objects preceding it i.e. $t - w + 1, t - w + 2, \ldots, t - 1$. There are $O(w)$ preceding objects and $w!$ possible permutations. Now, we are storing the neighbors in the ascending order of their arrivals in the NN-list. Therefore, the average length of an NN-list is at most as large as the average length (L_n) of the longest increasing subsequence of a random permutation of length w. There have been many conjectures about the exact distribution of L_n but all agree

that the expected value is $O(n^{1/2})$ [79]. Therefore, the expected space needed for the data structure is $O(w^{\frac{3}{2}})$.

Reducing Time to create N-list:

To further reduce the update time we need to reduce the number of distance computations upon insertion. We can use an order line [70] to order the points on a 1D line. The order line is just a circular projection of all of the points around a reference/pivot point [30]. The relative distance between a pair of points on the order line is a lower bound on their true distance in the original space. Thus, for every pair of points we now know a lower bound on their true distance, which can be used to decide if we will compare and insert a point into the N-list of the newly added point. To facilitate this, we first find an allowable upper limit of the distance between an older point and the new point and then check if the lower bound for this pair is larger than this upper limit. Given any growing N-list, the allowable upper limit of the distance between a point x and the new point n is the minimum of $d(n, y)$ for $y > x$.

To illustrate this idea, in Figure 4.7 the evolving N-list of point 6 is shown. On the *left* the order line is shown with points 1 through 6 and their positions/referenced distances illustrated. Starting from 6 the algorithm walks both directions on the order line and compares every new point encountered with point 6. Thus the order line provides a specific order of the points within the sliding window to be compared with the new point. In this example the order can be 3, 2, 1, 4 and 5. The state of the N-list after each of the points is considered is shown by Figure 4.7. First of all, 3 is inserted as $UL(3, 6) = \infty$. Now, 2 has a lower bound

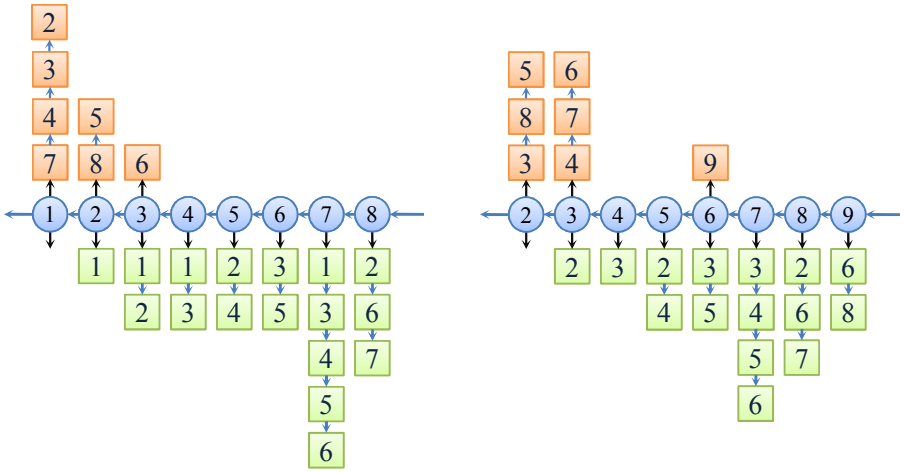

Figure 4.6: (*left*) The space reduction using the temporal ordering of the neighbors. (*right*) In the next time tick 1 is deleted from all of the lists and 9 is inserted

$LB(2,6) = 1$, which is smaller than the upper limit $UL(2,6) = d(3,6) = 1.5$. Therefore, we compute $d(2,6) = 3$, which is larger than the UL, and so 2 is not inserted. Similarly, 1 has a lower bound $LB(1,6) = 2$ which is larger than $UL(1,6) = d(3,6) = 1.5$ and therefore, 1 is not inserted. After that, 4 is inserted, as it has $LB(4,6) = 2$ smaller than $UL(4,6) = \infty$. Finally, 5 is inserted for the same reason in the list. The last step is to sort the list and remove out-of-order points. For example, 4 is knocked out of the N-list at this step.

4.4 Online MK Algorithm

With the above example elucidated, we can complete the description of the subsequent modifications made to the naive algorithm to produce our final algorithm named **Online MK**. Algorithm 4.1 through 4.3 show the pseudocode of our algorithm. There are two subrou-

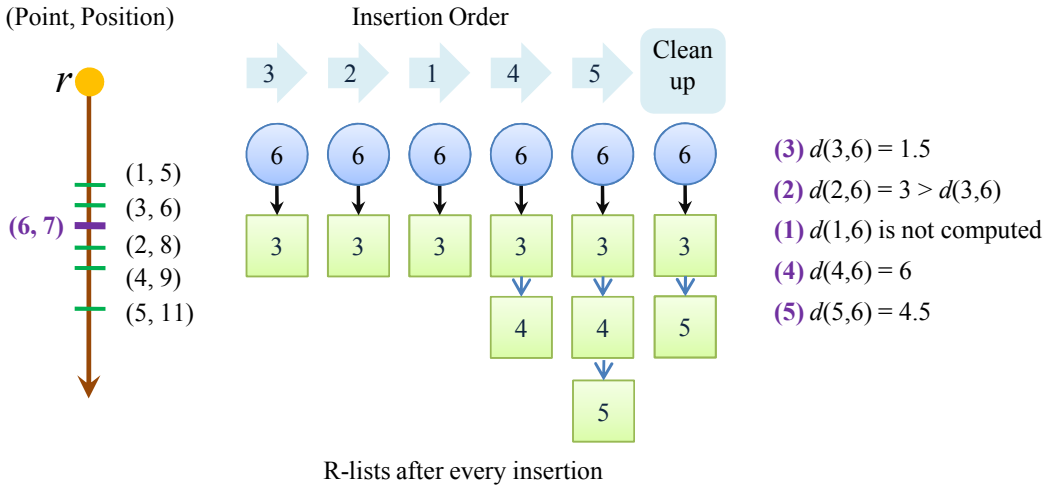

Figure 4.7: Building the Neighbor list of point 6. (*left*) The order line while 6 is being inserted. (*middle*) The states of the N-lists after each insertion. (*right*) The distance values assumed in this example

tines for insertions and deletions made to the sliding window. Each of them takes in a point (p) as the argument and performs the necessary operations on the data structure. At every time tick, $insertPoint(latestpoint)$ and $deletePoint(oldestpoint)$ are called to keep the data structure updated. The locations and the distance of the motif pair are always available after these two operations. The data structure is assumed to be accessible by every subroutine.

When $insertPoint(p)$ (Algorithm 4.1) is called with the new point p, p is compared with the reference point (randomly generated or chosen from the database [70]). By projecting p on the order line (line 1) we mean computing the referenced distance (i.e. $d(p, r)$) and inserting it in the sorted order-line (which is simply a doubly-linked list of pointers). After that, the $buildNeighborList(p)$ (Algorithm 4.2) is called to insert p and create its N-list in the data structure. As described earlier, the points are considered in the order of the distance

Algorithm 4.1 $insertPoint(p)$

Require: An order line
Ensure: p is inserted in the data structure
1: Project p on the order line
2: $buildNeighborList(p)$
3: Sort p.N-list in ascending order of distances from p
4: Remove all x from p.N-list such that $x.timeStamp < prev_{N-list(x)}.timestamp$
5: Insert p in the R-list of p.N-list.head
6: **if** $d(p, p$.N-list.head$)<$ motif distance **then**
7: Update motif pair with (p,p.N-list.head)

from p on the order line (line 2 in Algorithm 4.2). Before inserting a point n in the N-list, the algorithm finds the allowable upper limit u by looking at the current N-list (line 3) and compares it with the lower bound which is the same as the difference between the referenced distances of p and n (i.e. $LB(n, p) = |d(n, r) - d(p, r)|$). If the lower bound is smaller than the upper limit, the algorithm computes the distance $d(p, n)$ and again compares this with u. In case of $d(p, n) < u$, n has to be inserted in the N-list of p. The loop (line 1) finishes when the immediately previous point in the time order of p is already inserted and the lower bound of a point is larger than the $d(prev_{time(p)}, p)$. The reason for this is that all points that would be considered if the loop were not broken must have $u < d(prev_{time(p)}, p)$ and therefore would never succeed in the if statement at line 4.

When $buildNeighborList(p)$ returns, the N-list is sorted according to the distances from p (line 3 of Algorithm 4.1) and all the points that meet observation 4.3.2 are removed from the N-list (line 4). Then, p is inserted in the R-list of the first point of its own N-list (line 5). At line 6 the algorithm checks if the new point forms a motif and updates the motif pair if it is so. Note that the computation of upper limit should be efficient enough to preserve the benefit

Algorithm 4.2 $buildNeighborList(p)$

Require: An order line
Ensure: Neighbor list of p is built
1: **while true do**
2: $n \leftarrow$ next point from p on the order line
3: $u \leftarrow UL(n, p$.N-list)
4: **if** $LB(n, p) < u$ **then**
5: **if** $d(n, p) < u$ **then**
6: insert n in p.N-list at the head
7: **else if** $LB(n, p) \geq d(prev_{time(p)}, p)$ **then**
8: **break**

Algorithm 4.3 $deletePoint(p)$

Require: An order line
Ensure: p is deleted from the data structure
1: **for** all points q in the R-list of p **do**
2: Remove q.N-list.head
3: Insert q into R-list of q.N-list.head
4: Remove p from the order line
5: **if** p is one of the motif pair **then**
6: Find x for which $d(x, x$.N-list.head) is minimum
7: Update motif pair with $(x, x$.N-list.head)

of reduction in distance computation. We leave it as a design choice for the practitioners for brevity and lack of space.

When the $deletePoint(p)$ (Algorithm 4.3) is called with the oldest point p, all of its reverse neighbors (q) will lose their nearest neighbor which is p itself (line 2). Since q is a later point than q.N-list.head, the algorithm inserts q into the R-list of q.N-list.head. If p is one of the motif pair, the algorithm finds a new motif by finding the minimum of all of the nearest neighbor distances (lines 6-7).

4.5 Performance Evaluation

We have used four very different datasets in our experiments, EEG trace [70], EOG trace [71], insect behavior trace [70] and a synthetic random walk (RW). All datasets, codes, videos and numbers used to generate the figures in this section are available to be downloaded from the supporting webpage [2]. We use a 2.67 GHz Intel quad core processor with 6GB RAM.

To the best of our knowledge there is no other algorithm that discovers time series motifs online[1], although there are works on dynamic maintenance of the closest pair in high dimensionality. It is possible to trivially modify any of these algorithms to perform the online closest pair problem. We have selected the highly optimized implementation of the well referenced work [34] for this purpose. To be fair to the authors of [34], we note that we made changes to the implementation to specialize it for time series motif discovery, and the original code is more general than our problem requires, as it allows arbitrary insertions and deletions, whereas we only need to be able to support insertions at the "head" and deletions at the "tails."

We have used the implementation of the $FastPair$ data structure as it performs best in most of the applications [34]. Figure 4.8 shows that our algorithm grows a lot more slowly than FastPair if we change both of the parameters w and m while fixing the other at a specific value. For different datasets FastPair performs almost identically, so we show only the best one. The speedup in average update time is guaranteed as we compute $O(w)$

[1]Based just the title, the reader may imagine that *On-line motif detection in time series with SwiftMotif* [37] discovers time series motifs *online*. However this work finds approximate motifs *offline* then approximately filters them *online*

distances per update while FastPair computes $O(w \log^2 w)$ distances. Although we cache more statistics and thus use more space per point, in Figure 4.9 we can see an almost flat average space usage per point over a large range of window sizes and motif lengths. This is significantly less than the worst case space needed per point, which is $O(w)$. Note that random walk needs significantly larger N-lists to accommodate more neighbors. The reason for this is the prominent low-varying trends of random walk. For any m, a new subsequence becomes neighbor to a relatively larger set of subsequences that just show the same trend after normalization even if they have different slopes and variances.

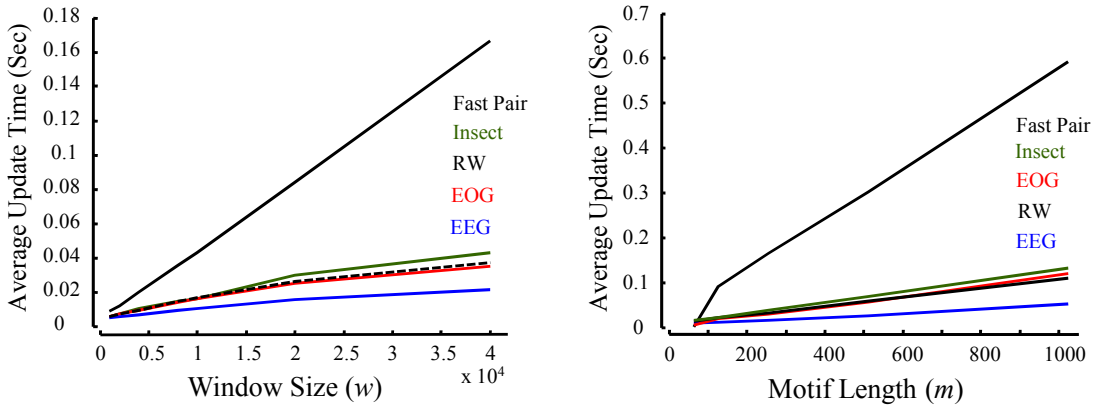

Figure 4.8: Empirical demonstration of the slow growth of average update time with respect to window size (w varies, $m = 256$) and motif length (m varies $w = 40,000$)

We have two parameters to be set by the users, w and m. Optimum values of (w, m) significantly depend on the domain and are very easy for the practitioners to interpret as both can be measured in seconds or in the number of samples. In Figure 4.10 we show the average update time per point for every combination of two sets of possible values of w and m (Figure 4.10). Although the Figure shows values for the EEG dataset, other datasets exhibit a similar

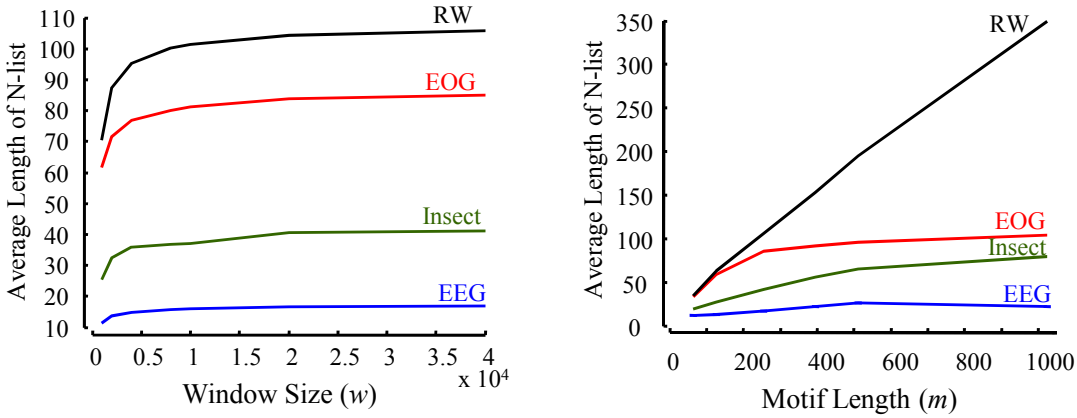

Figure 4.9: Empirical demonstration of the slow growth of average length of N-list with respect to window size (w varies, $m = 256$) and motif length (m varies $w = 40,000$). Labels are in order of the heights of the right-most points of the curves

shape. Figure 4.10 shows the space used per point for the EOG dataset. Note that there are three zero values showing the invalid combinations where a motif cannot be defined such as w=1000, m=512.

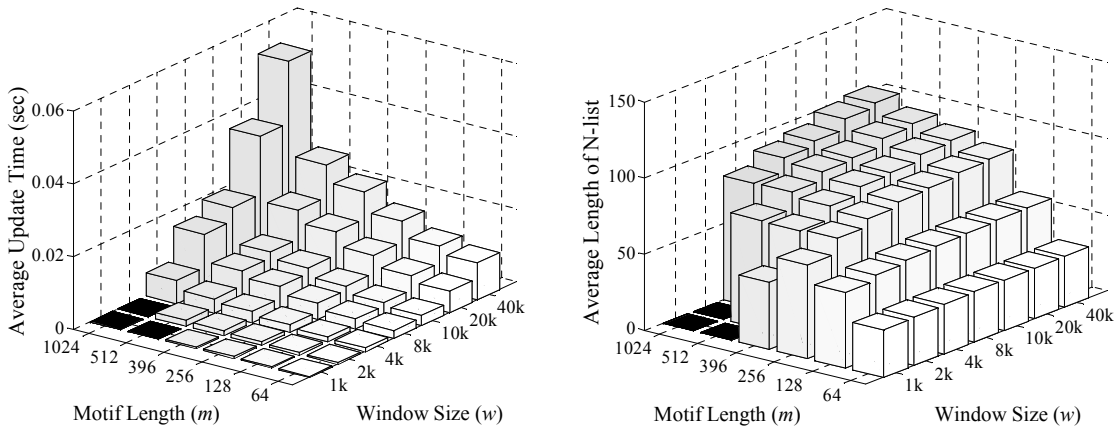

Figure 4.10: (*left*) Time usage per point in EEG dataset with varying w and m. (*right*) Space usage per point in EOG dataset with varying w and m

As impressive as these results are, the following observation allows us to further improve them. In most applications, we can define the maximum distance (d_m) beyond which no pair can be meaningfully called a motif simply because they are not similar enough to be of interest in that domain. As a concrete example, in the wildlife monitoring application discussed in Section 4.7.2, we found that motifs that had a value greater than about 12.0 did not correspond to sounds that people found to be subjectively similar. Therefore, we can ask the algorithm not to even bother finding the motif pair, if they would have a distance of more than d_m=12.0. To incorporate dm in our algorithm, only line 8 in Algorithm 4.2 needs to be changed, to test if $LB(n,p) \geq min(d(prev_{time(p)}, p), d_m)$. If we can obtain a reasonable dm from domain experts, it can reduce the number of distance computations performed per point with the help of the order line. The reason for this is that our algorithm can prune off all of the pairs having distances $> d_m$ without computing the true distances.

Consequently, it makes our algorithm faster. Figure 4.11(*left*) shows that when we use d_m=0.4m (equivalent to 80% correlation) and $0.2m$ (equivalent to 90% correlation) then the average number of distance computation in the EEG dataset has been reduced for every window size. The speedup is generic for all of the datasets, as shown in Figure 4.11(*right*).

4.6 Extending Online MK

The basic online motif discovery algorithm described above can be extended, augmented and modified in numerous ways. We shared a very early draft of this work with domain

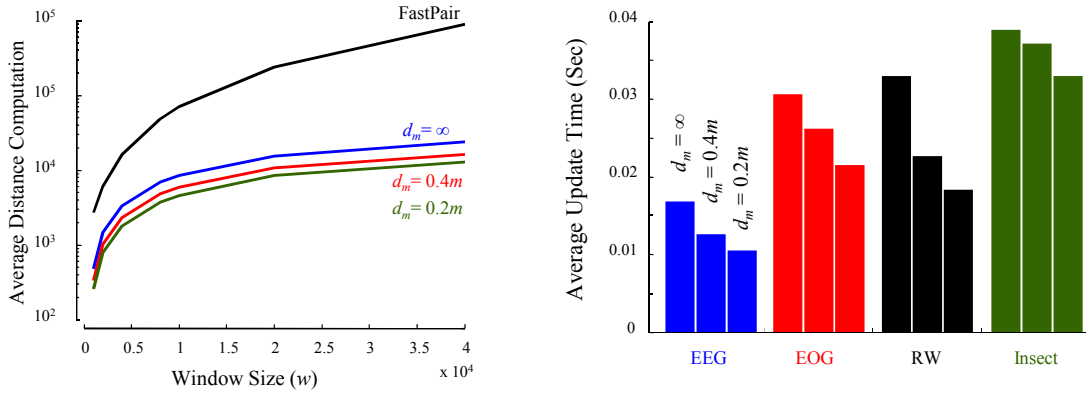

Figure 4.11: (*left*) The average amount of distance computation is much less in our algorithm than FastPair for EEG and further decreases with decreasing dm. (*right*) Speedup is consistent over all of the datasets for m=256 and w=40,000

experts in motion capture, medicine, robotics and agricultural monitoring, and asked them to suggest a "wish list" of extensions. The top two requests were adapting to variable data rates (robotics and agricultural monitoring) and handling multidimensional motifs (motion capture, robotics). In the next two subsections we show how this can be accomplished.

4.6.1 Adapting to Variable Data Rate

Recall that our framework allows a guaranteed performance rate. That is to say, given values for m, w and a time to compute one distance calculation, we can compute the fastest arrival rate λ that we can a guarantee to handle. However, even if asked to perform at exactly this rate, we can generally expect to have idle CPU cycles, simply because there is a gap between the pathological worst case we must consider and the average performance on real datasets. An obvious question is whether we can we bridge this gap between our average

performance, and the worst-case situation we must guarantee to handle, but expect to rarely if ever encounter in the real world. The problem is exasperated by the fact that up to this point we are assuming constant arrival rates. For example, suppose that a stream produces data at 100Hz 99.999999% of the time, but very occasionally produces a burst at 120Hz. If we can just handle 100Hz with an off-the-shelf processor, must we really spend $300 for a faster processor that can handle the rarely seen faster rate? Much of the literature on monitoring streams for various events makes the constant arrival rate assumption [93]. However, variable arrival rates are common in many domains. Previously, similar problems have been dealt with by load shedding in Data Stream Management systems with techniques that allow dropping operators [101], while still maintaining the quality of the results. We believe that skipping points is also the best solution in the current context.

Concretely, we *skip* every point that arrives within the current update operation (one insertion and one deletion). For example, for a 100Hz stream, if the update for $x_{i,m}$ takes 30ms then our algorithm would skip two immediate points ($x_{i+1,m}$ and $x_{i+2,m}$) and would start updating from the third point ($x_{i+3,m}$) on. However, if an update takes less than 10ms then we would not skip the following point. Therefore, for a smaller average update time (i.e. 6ms in a 100Hz stream) a whole range of data usage (amount of data not skipped) is possible. For example, if all of the updates take 6ms then 100% data points are used and nothing is skipped. In contrast, about 50% of the data will be skipped if there are oscillating update times of 1ms, 11ms, 1ms, 11ms and so on. Figure 4.12(*left*) shows the fraction of the stream that is not skipped for different data rates with of w=32,000. For most of our datasets, our

algorithm can process at 200Hz while skipping every alternate point. Most real time sensors work on less than 100Hz, a rate at which we process more than 60% of the data.

Obviously there is a chance that one of the skipped points is a potential motif. There is no way to predict if a skipped point would be a motif with some future subsequence. Therefore, we accept this potential loss for the sake of an infinitely running system. In Figure 4.12(*right*) we show that although we skipped 30-40% of the points in high data rates (i.e. over 100Hz), we did not miss many of the motifs. The drop rate of the number of motifs discovered is slower than the drop in data usage.

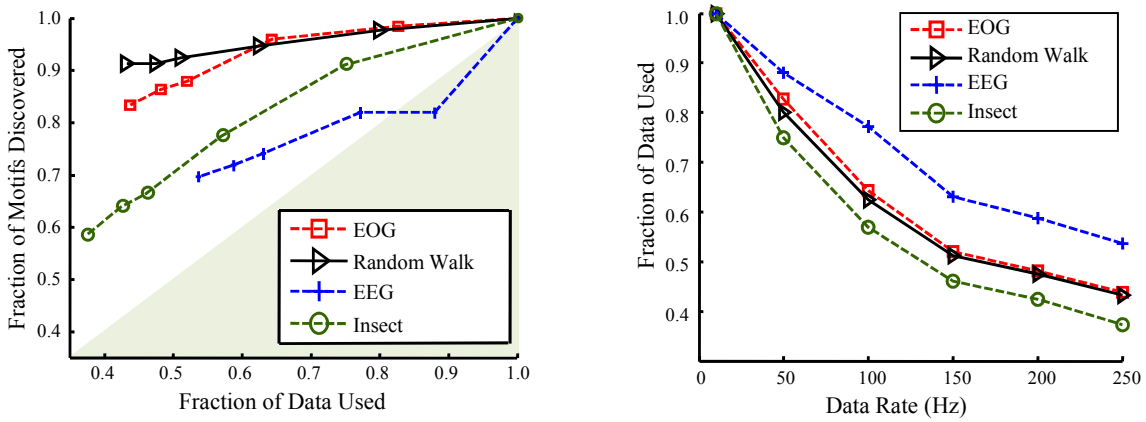

Figure 4.12: (*left*) Fraction of Data Used (the amount of subsequences considered) plotted against the varying data rate for w=32,000. Our algorithm can operate at 200Hz while skipping roughly every other point. (*right*) The fraction of the motifs discovered drops more slowly than the fraction of data used

If we considered the unique motifs (non-overlapping) only, our algorithm would rarely miss any of them. The reason for this is the following: "A skipped subsequence is very similar to the previous and following non-skipped subsequences" (i.e. they are "trivial matches"

[22]). Thus, even if we skipped a subsequence, its trivial mates would get a chance to form a motif that is almost identical to the non-skip version.

4.6.2 Monitoring Multidimensional Motifs

Our algorithm is trivially extendible to multidimensional time series motifs. For simplicity, let's consider the two-dimensional case of online motif discovery. At every time tick here we have exactly two points arriving and two points departing. For the two time series we keep two separate data structures, each similar to Figure 4.6(*left*). Depending on the application we can ignore/allow a motif within/across the same/different series. The primary change is to redefine the set of subsequences that are compared with the latest subsequence at the time of insertion. Thus, in the N-list and R-list, nodes can point to points in both of the sequences. Both of the observations of Section 4.3.2 hold for such N-lists, and the size of an N-list on average is still $O(w^{\frac{1}{2}})$.

The update cost is now $O(wd)$, where d is the number of simultaneous time series. The space needed for the whole data structure is $O(w^{\frac{3}{2}}d)$. The closest pair can be found as before by checking the heads of the N-lists in both of the data structure.

4.7 Applications of Online Motifs

Online motif discovery is appropriate for settings where real-valued numbers are generated at a high rate and there is a necessity for tracking a particular behavior that creates similar

subsequences in the stream. We have tested our algorithm on several datasets that fit this model, and use online motif discovery as a sub-routine. We note that these case studies are really demonstrations rather than experiments. In particular, space limitations prohibit us from providing pseudocode and some minor details. However, this is useful to motivate some applications made possible by online motif discovery. Note that, as before, all data and code is freely available at [2].

4.7.1 Online Summarization/Compression

Online summarization/compression of time series data is an important problem that has attracted considerable research. Existing approaches use various time series representations, including piecewise linear approximations (PLA) (as shown in Figure 4.13(*middle*)), piecewise constant approximations [**?**], Fourier approximations [80] and wavelets [16]. However, the obvious idea of summarizing a real-valued stream by dynamically finding reoccurring patterns in the data, placing one occurrence in a dictionary and assigning future occurrences to pointers to the dictionary entry, does not appear in the literature. We believe that this omission is due to the fact that until now there was no practical method to find the necessary reoccurring patterns in real time. Clearly the results in this chapter repair this omission.

Plugging motifs into virtually any online compression algorithm is very simple. Most of the algorithms keep a small buffer of raw data similar to our sliding window. Within that buffer they run a simple search algorithm, deciding, for example, whether to approximate a heartbeat with 6 or 5 linear segments (See Table 6 of [50] as a concrete example) All we have

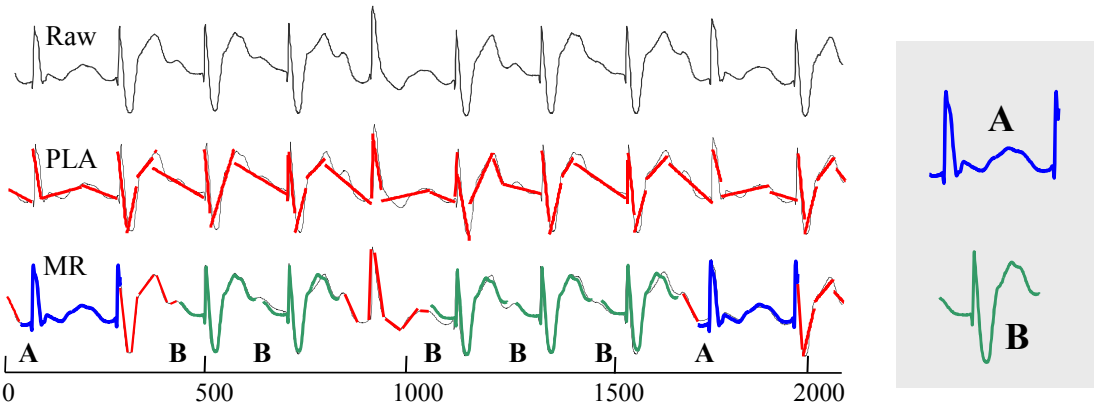

Figure 4.13: (*top*) An excerpt of record sddb/51 Sudden Cardiac Death Holter Data. (*middle*) A PLA of the data with an approximately 0.06 compression rate (*bottom*) A Motif-Representation approximation of the data at twice the compression rate still has a lower error

to do is add a new search operator that asks "*would it be better to approximate this section with linear segments, or one of the motifs in the current motif dictionary?*" Given this idea, all we need to do is set two parameters; how many motifs and of what length we should keep in the dictionary. In Figure 4.13 we show an excerpt where we chose (after seeing the first five minutes of the data) to maintain just two motifs, one of length 250 and one of length 200.

In this example we compare our approach to the most referenced method [81], which uses PLA. We found that even if we force the motif-representation based method to use half the space of PLA, it can still approximate the data with a residual error that is approximately one-ninth that of PLA. The approximations achieved are not only of a higher fidelity than other methods, but have the advantage of being highly interpretable in some circumstances. Note that the improvements achieved by the motif based algorithms are highly variable. On stock market data, with little or no repeated structure, there is no improvement; but on normal

heartbeats, which are of course highly repetitive, the reduction in size (for the same residual error as PLA) can be two or three orders of magnitude for larger datasets.

4.7.2 Acoustic Wildlife Management

Acoustic wildlife management is a useful tool for measuring the health of an eco-system, and several projects are currently monitoring the calls of various birds, frog and insects [103]. A key issue is that while sensors typically monitor twenty-four hours a day, memory limits for storage, or bandwidth limits for transmission, form a bottleneck on how much data can be retained in field-deployed sensors. For example, [103] reports that when using a simple thresholding algorithm, "*we have been able to reduce half an hour of raw recording to only 13 seconds of audio,*" however, they acknowledge that this data comes with some false positives. However, as [27] notes, "*Animals of many species sing or call in a repetitive and species specific fashion*" (our emphasis). We can exploit the repetitive nature of certain bird calls to reduce the amount of data retained while also reducing the false positive rate. For example, consider our efforts to monitor a sensor from woods in Monterrey, California. The sound data is converted into mel-frequency cepstrum coefficients **mfcc**, and only the first coefficient is examined. In this project, only Strigiformes (owls) are of interest, and domain experts have noted that most owls repeat their calls in a window of eight to ten seconds and that the calls last from one to three seconds [86]. Given this, we set w = 12 seconds, and m = 3 seconds, erring a little on the long side of those values. On a thirty second trace that we manually confirmed had only ambient noise, we found that the mean motif value was 42.3,

with standard deviation of 7.1. Given that we only record sounds that have corresponding motifs with a value less than 10.0, such a value is very unlikely to happen by chance. In Figure 4.14 we show an example of a detected motif with a value of 4.57, which corresponds to the call of a Great Horned Owl.

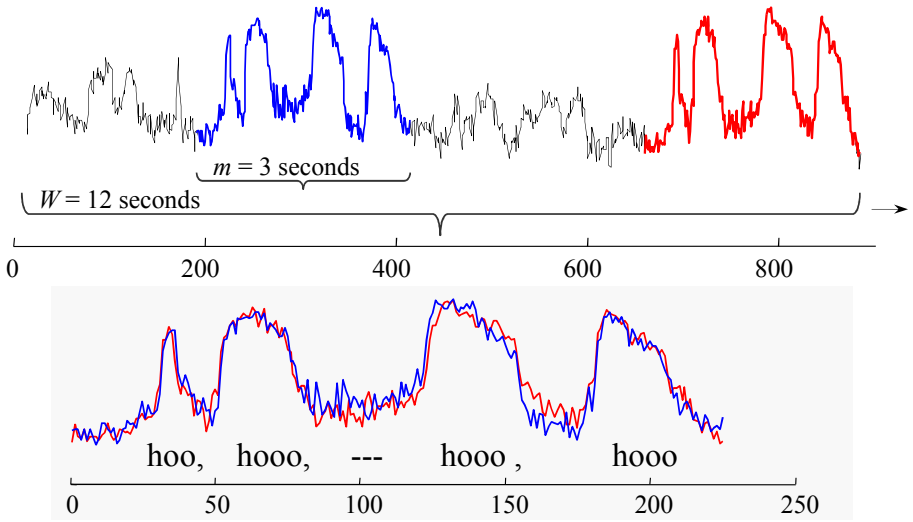

Figure 4.14: (*top*) A stream is examined for motifs 3 seconds long, with a history of 12 seconds. (*bottom*) The discovered motif is the cry of a Great Horned Owl, whose cry is phonetically written as hoo, hooo,—,hooo, hooo

4.7.3 Closing the Loop Problem

Closing the loop is a classic problem in robotics in which we task the robot with recognizing when it has returned to a previously visited place. The robot may sense its environment with a multitude of sensors, including cameras and ultrasonic transceivers, all of whose output can be represented as "time series." The problem is challenging in two aspects: first, the robot must be able to recognize that it has returned to a previously visited place. This is a significant

challenge, but assuming we can solve it, there is the second challenge of mitigating time and space complexity on resource limited robots. Naturally, we can see our algorithm as a tool for continuously maintaining the most likely candidate locations for loop closure.

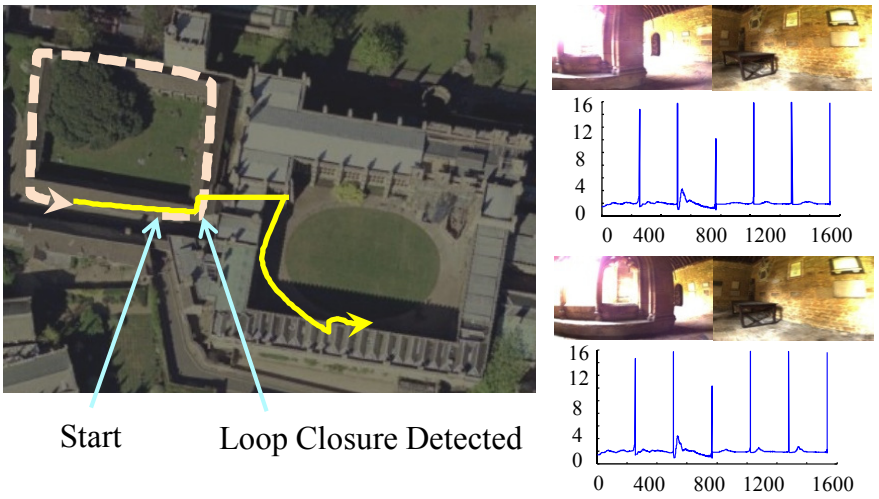

Figure 4.15: (*left*) The map of the "New College." A segment of robot motion is shown. (*right*) Motif: The most similar pair of image-pairs that are 90 samples apart and their color histograms. The image-pairs are from the same location and thus our algorithm detected the loop-closure

In an effort to verify this utility, we use the "New College" dataset [25], where a set of 2,146 images have been collected by a moving robot. The images are taken from both sides of the robot. We convert the images to "time series" by taking their color histogram and group the images from both sides to form a sequence of image-pairs. We feed our algorithm with this data and $w = 200$. We also provide a separation window of 90 images for excluding trivial similarities. Our algorithm found 89 unique motifs, 46 of them being loop-closures. One of the motifs and its location are shown in Figure 4.15. Although online MK catches about 50% of the loop closure, we argue that it essentially comes for free as online MK can

work at very high rate. The rest of the undetected closures can easily be confirmed by high level algorithms and thus online MK reduce the computational burden.

4.8 Conclusion

In this chapter we introduce the first practical algorithm for finding and maintaining time series motifs on fast moving streams. We showed applications of our ideas in robotics, online compression and wildlife management.

Chapter 5

Exact Discovery of Time Series Shapelets

Time series shapelet is a new primitive for classification of time series datasets [115]. In spite of being published for only a few years, shapelets are used in several work. For example, [112] uses shapelets on streaming data as "early predictors" of a class, [65] uses an extension of shapelets in an application in severe weather prediction, [45] uses shapelets to find representative prototypes in a dataset, and [44] uses shapelets to do gesture recognition from accelerometer data.

Inspired by the success of time series shapelets, we focus on improving the original algorithm to find them. The original algorithm requires massive computation for being exact and thus limiting its application on many real datasets potentially carrying important shapelets. We devise techniques to speedup the *exact* algorithm by efficient caching and admissible pruning.

We enrich time series shapelet demonstrating cases where multiple shapelets can identify complex concepts that were missed by single shapelet. We describe exact algorithm to discover such logical combinations of shapelets. It is important to note that there is essentially zero-cost for the expressiveness of logical-shapelets. If we apply them to a dataset that does not need their increased representational power, they simply degenerate to classic shapelets, which are a special case. Thus, our work complements and further enables the growing interest in shapelets.

5.1 Definition and Background

In this section, we define shapelets and the other notations used in this chapter. We start with redefining time series and subsequences for the ease of exposition.

Definition 5.1 [TIME SERIES] A *Time Series* T is a sequence of real numbers $t_1, t_2, \ldots, t_m$. A time series *subsequence* $S_{i,l} = t_i, t_{i+1}, \ldots, t_{i+l-1}$ is a continuous subsequence of T starting at position i and length l.

A time series of length m can have $\frac{m(m+1)}{2}$ subsequences of all possible lengths from one to m.

If we are given two time series X and Y of same length m, we can use the Euclidean norm of their difference (i.e. $X - Y$) as a distance measure. To achieve scale and offset invariance, we must normalize the individual time series using *z-normalization* before the actual distance is computed. This is a critical step; even tiny differences in scale and offset rapidly swamp

any similarity in shape (c.f. Section 2.3.4). In addition, we normalize the distances by dividing with the length of the time series. This allows comparability of distances for pairs of time series of different lengths. We call this *length-normalization*. In the rest of the chapter we use "normalized distance" to mean both length and z-normalization.

The normalized Euclidean distance is generally computed by the formula $\sqrt{\frac{1}{m}\sum_{i=1}^{m}(\hat{x}_i - \hat{y}_i)^2}$ (c.f. definition 2.8). Thus, computing a distance value requires time linear on the length of the time series. In contrast, we compute the normalized Eucledian distance between X and Y using five numbers derived from X and Y. These numbers are denoted as sufficient statistics in [90]. The numbers are $\sum x$, $\sum y$, $\sum x^2$,$\sum y^2$ and $\sum xy$. It may appear that we are doing more work than necessary; however, as we make clear in Section 5.2.1, computing the distance in this manner enables us to reuse computations and reduce the amortized time complexity from *linear* to *constant*.

The population mean and varaince can be computed from these statistics as $\mu_x = \frac{1}{m}\sum x$ and $\sigma_x^2 = \frac{1}{m}\sum x^2 - \mu_x^2$, respectively. The positive correlation and the normalized Euclidean distance between X and Y can then be expressed as below [72].

$$C(x,y) = \frac{\sum xy - m\mu_x\mu_y}{m\sigma_x\sigma_y} \tag{5.1}$$

$$dist(x,y) = \sqrt{2(1 - C(x,y))} \tag{5.2}$$

Many time series data mining algorithms such as 1-NN classification, clustering and density estimation require only the comparison of time series that are of equal lengths. In contrast,

time series shapelets require us to test if a short time series (the shapelet) is contained within a certain threshold somewhere inside a much longer time series. To achieve this, the shorter time series is slid against the longer one to find the best possible alignment between them.

Definition 5.2 [SUBSEQUENCE DISTANCE] The normalized *subsequence distance*, or in short $sdist$, between two time series x and y with lengths m and n, respectively, and for $m \leq n$ is

$$\begin{aligned} sdist(x,y) &= \sqrt{2(1 - C_s(x,y))} \\ C_s(x,y) &= \max_{0 \leq l \leq n-m} \frac{\sum_{i=1}^{m} x_i y_{i+l} - m\mu_x\mu_y}{m\sigma_x\sigma_y} \end{aligned} \tag{5.3}$$

In the above definition μ_y and σ_y denote the mean and standard deviation of m consecutive values from y starting at position $l + 1$. Note that, $sdist$ is not symmetric. Assume that we have a dataset D of N time series from C different classes. Let us also assume, every class i ($i = 1, 2, \ldots, C$) has n_i ($\sum_i n_i = N$) labeled instances in the dataset. An instance time series in D is also denoted by D_i for $i = 1, 2, \ldots, N$.

Definition 5.3 [ENTROPY] The *entropy* of a dataset D is defined as $E(D) = -\sum_{i=1}^{C} \frac{n_i}{N} log(\frac{n_i}{N})$.

If the smallest time series in D is of length m, there are at least $N\frac{m(m+1)}{2}$ subsequences in D that are shorter than every time series in D. We define a *split* as a tuple (s, τ) where

s is a subsequence and τ is a distance threshold. A split divides the dataset D into two disjoint subsets or partitions $D_{left} = \{x : x \in D, sdist(s,x) \leq \tau\}$ and $D_{right} = \{x : x \in D, sdist(s,x) > \tau\}$. We define the cardinality of each partition by $N_1 = |D_{left}|$ and $N_2 = |D_{right}|$. We use two quantities to measure the goodness of a split: *information gain* and *separation gap*.

Definition 5.4 [INFORMATION GAIN] The *information gain* of a split is $I(s,\tau) = E(D) - \frac{N_1}{N}E(D_{left}) - \frac{N_2}{N}E(D_{right})$.

Definition 5.5 [SEPARATION GAP] The *separation gap* of a split is $G(s,\tau) = \frac{1}{N_2}\sum_{x \in D_{right}} sdist(s,x) - \frac{1}{N_1}\sum_{x \in D_{left}} sdist(s,x)$.

Now we are in position to define time series shapelets.

Definition 5.6 [SHAPELET] The *shapelet* for a dataset D is a tuple of a subsequence of an instance within D and a distance threshold (i.e. a split) that has the maximum information gain while breaking ties by maximizing the separation gap.

5.1.1 Brute-force Algorithm

In order to properly frame our contributions, we begin by explaining the brute-force algorithm for finding shapelets in a dataset D (Algorithm 5.1). Dataset D contains multiple time series of two or more classes and possibly of different lengths.

Algorithm 5.1 $Shapelet\_Discovery(D)$

Require: A dataset D of time series
Ensure: Return the shapelet

1: $m \leftarrow$ minimum length of a time series in D
2: $maxGain \leftarrow 0, maxGap \leftarrow 0$
3: **for** $j \leftarrow 1$ to $|D|$ **do** every time series in D
4: $S \leftarrow D_j$
5: **for** $l \leftarrow 1$ to m **do** every possible length
6: **for** $i \leftarrow 1$ to $|S| - l + 1$ **do** every start position
7: **for** $k \leftarrow 1$ to $|D|$ **do** compute distances of every time series to the candidate shapelet $S_{i,l}$
8: $L_k \leftarrow sdist(S_{i,l}, D_k)$
9: $sort(L)$
10: $(\tau, updated) \leftarrow bestIG(L, maxGain, maxGap)$
11: **if** $updated$ **then** gain and/or gap are changed
12: $bestS \leftarrow S_{i,l}$,$best\tau \leftarrow \tau$,$bestL \leftarrow L$
13: **return** $(bestS, best\tau, bestL, maxGain, maxGap)$

Algorithm 5.2 $sdist(x, y)$

Require: Two time series x and y. Assume $|x| \leq |y|$.
Ensure: Return the normalized distance between x and y

1: $minSum \leftarrow \infty$
2: $x \leftarrow zNorm(x)$
3: **for** $j \leftarrow 1$ to $|y| - |x| + 1$ **do** every start position on y
4: $sum \leftarrow 0$
5: $z \leftarrow zNorm(y_{j,|x|})$
6: **for** $i \leftarrow 1$ to $|x|$ **do** compute the Eucledian distance
7: $sum \leftarrow sum + (z_i - x_i)^2$
8: $minSum \leftarrow min(minSum, sum)$
9: **return** $\sqrt{minSum/|x|}$

Since time series can be considered to be points in high dimensional space, we will denote D as a database of points. The algorithm generates a candidate subsequence $S_{i,l}$ in the three loops in lines 3, 5, and 6 of Algorithm 5.1, which essentially generates all possible subsequences of all possible lengths from D. In lines 7-9, an array L is created which holds the points in D in the sorted order of their distance from the shapelet candidate. A schematic view of this array is illustrated in Figure 5.1. We call this schematic line, as before, the

Algorithm 5.3 $bestIG(L, magGain, maxGap)$

Require: An order line L and current best gain and gap.
Ensure: Update the best information gain and separation gap and return the split point τ

```
for k ← 1 to |D| − 1 do
    τ ← (L_k + L_{k+1})/2
    Count n_{1,i} and n_{2,i} for i = 1, 2, ..., C.
    Count N_1 and N_2 for both the partitions.
    I ← Information gain computed by definition 5.4
    G ← Separation gap computed by definition 5.5
    if (I > maxGain or (I = maxGain ⋀ G > maxGap)) then
        maxGain ← I, maxGap ← G, maxτ ← τ
        updated ← true
return (maxτ, updated)
```

orderline. Intuitively, the ideal shapelet is the one that orders the data as such all instances of one class are near the origin, and all instances of the other classes are to the far right, with no interleaving of the classes.

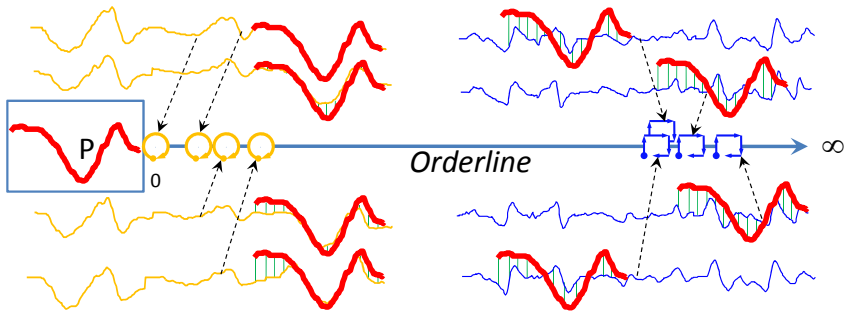

Figure 5.1: Orderline for the shapelet P. Each time series is placed on the orderline based on the $sdist$ from P. Note that, the time series that carries P is placed at position 0 on the orderline. Also note that, P aligns at different positions on different time series

Distance values are computed by Algorithm 5.2. Both of the normalizations – z-normalization and length-normalization (before and after the computation of Euclidean distance, respectively) – are performed here. Note that repeatedly performing normalization before comput-

ing distances is expensive because the same sequences are normalized multiple times. Our novel caching mechanism in Section 5.2.1 will remove this difficulty.

Given the orderline, the computation of information gain is linear in the dataset size. Algorithm 5.3 describes the method of finding the best split point paired with the current candidate sequence. At line 2, we selected the mid points between two consecutive points on the orderline as split (τ) points. Although there are infinite number of possible split points, there are at most $|D| - 1$ distinct splits that can result in different information gains. Therefore, it is sufficient to try only these $|D| - 1$ splits. Computing the information gain from the orderline requires $O(N)$ time, which includes counting statistics per class for the three sets (i.e. D, D_{left}, D_{right}) in line 3 and 4. The rest of the computation can be done in $O(C)$ time in line 5 and 6. Finally, the current best gain and gap are updated if appropriate.

The time complexity of the algorithm, assuming no additional memory usage (i.e. linear space cost), is clearly untenable. There are at least $N\frac{m(m+1)}{2}$ shapelet candidates (i.e. the number of iterations through loops in line 3, 5, and 6). For each of the candidates, we need to compute distances to each of the N time series in D (i.e. line 8). Every distance computation takes $O(m^2)$ time. In total, we need $O(N^2m^4)$ arithmetic operations. Such a huge computational cost makes the brute-force algorithm infeasible for real applications. Since N is usually small as being the size of a labeled dataset, and m is large to make the search for local features meaningful, we focus on reducing factors of m from the cost.

5.2 Speedup Techniques

The original shapelet work introduced by Ye, *et al.* [115] showed an admissible technique for abandoning some unfruitful entropy computations early. However, this does not improve the worst case complexity. In this work, we reduce the worst case complexity by caching distance computations for future use. In addition, we describe a very efficient pruning strategy resulting in an order of magnitude speedup over the method of Ye *et al.*

5.2.1 Efficient Distance Computation

In Algorithm 5.1, any subsequence of D_j with any length and starting at any position is a potential shapelet candidate. Such a candidate needs to be compared against every subsequence of D_k with the same length, starting at any position. The visual intuition of the situation is illustrated by Figure 5.2. For any two instances D_j and D_k in D, we need to consider all possible parallelograms like the ones shown in the figure. For each parallelogram, we need to scan the subsequences to sum up the squared errors while computing the distance. Clearly there is a huge redundancy of calculations between successive and overlapping parallelograms.

Euclidean distance computation for subsequences can be made faster by reusing overlapping computation. However, reusing computations of z-normalized distances for overlapping subsequences needs at least quadratic space and, therefore, is not tenable for most applications. When we need all possible pairwise distances among the subsequences, as we do in

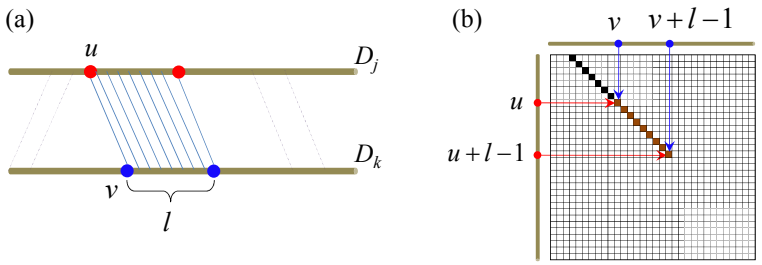

Figure 5.2: (a) Illustration of a distance computation required between a pair of subsequences starting at positions u and v, respectively, and of length l. Dashed lines show other possible distance computations. (b) The matrix $\mathbb{M}$ for computing the sum of products of the subsequences in (a)

finding shapelets, spending the quadratic space saves a whole order of magnitude of computation time.

For every pair of points (D_j, D_k) we compute five arrays. We represent these arrays as $Stats_{x,y}$ in the algorithms. Two of the arrays (i.e. $\mathbb{S}_x$ and $\mathbb{S}_y$) store the cumulative sum of the individual time series x and y. Another two (i.e. $\mathbb{S}_{x^2}$ and $\mathbb{S}_{y^2}$) store the cumulative sum of squared values. The final one (i.e. $\mathbb{M}$) is a 2D array that stores the sum of products for different subsequences of x and y. The arrays are defined mathematically below. All of the arrays have left margin of zero values indexed by 0. More precisely, $\mathbb{S}_x[0]$, $\mathbb{S}_y[0]$, $\mathbb{S}_{x^2}[0]$, $\mathbb{S}_{y^2}[0]$, $\mathbb{M}[0,0]$, $\mathbb{M}[u,0]$ for $u = 1, 2, \ldots, |x|$, and $\mathbb{M}[0,v]$ for $v = 1, 2, \ldots, |y|$ are all zeros.

$$\mathbb{S}_x[u] = \sum_{i=0}^{u} x_i\,, \qquad \mathbb{S}_y[v] = \sum_{i=0}^{v} y_i,$$

$$\mathbb{S}_{x^2}[u] = \sum_{i=0}^{u} x_i^2\,, \qquad \mathbb{S}_{y^2}[v] = \sum_{i=0}^{v} y_i^2$$

$$\mathbb{M}[u,v] = \begin{cases} \sum_{i=0}^{v} x_{i+t} y_i & \text{if } u > v, \\ \sum_{i=0}^{u} x_i y_{i+t} & \text{if } u \leq v \end{cases}$$

where $t = abs(u - v)$.

Algorithm 5.4 $sdist\_new(u, l, Stats_{x,y})$

Require: start position u and length l and the sufficient statistics for x and y.
Ensure: Return the subsequence distance between $x_{u,l}$ and y

```
1: minSum ← ∞
2: {M, S_x, S_y, S_x², S_y²} ← Stats_{x,y}
3: for v ← 1 to |y| − |x| + 1 do
4:     d ← distance computed by (5.1) and (5.2) in constant time
5:     if d < minSum then
6:         minSum ← d
7: return √minSum
```

Given that we have computed these arrays, the mean, variance, and the sum of products for any pair of subsequences of the same length can be computed as below.

$$\mu_x = \frac{\mathbb{S}_x[u+l-1]-\mathbb{S}_x[u-1]}{l}, \mu_y = \frac{\mathbb{S}_y[v+l-1]-\mathbb{S}_y[v-1]}{l}$$

$$\sigma_x^2 = \frac{\mathbb{S}_{x^2}[u+l-1]-\mathbb{S}_{x^2}[u-1]}{l} - \mu_x^2, \sigma_y^2 = \frac{\mathbb{S}_{y^2}[v+l-1]-\mathbb{S}_{y^2}[v-1]}{l} - \mu_y^2$$

$$\sum_{i=0}^{l-1} x_{u+i}y_{v+i} = \mathbb{M}[u+l-1, v+l-1] - \mathbb{M}[u-1, v-1].$$

This in turn means that the normalized Euclidean distance (the information we actually want) between any two subsequences $x_{u,l}$ and $y_{v,l}$ of any length l can now be computed using equations 5.1 and 5.2 in constant time. The Algorithm 5.4 describes the steps. The algorithm takes as input the starting position u and the length l of the subsequence of x. It also takes the precomputed $Stats_{x,y}$ carrying the sufficient statistics. The algorithm iterates for all possible start positions v of y and returns the minimum distance. Thus we can see that the procedure $sdist\_new$ saves at least an $O(m)$ inner loop computation from procedure $sditst$.

5.2.2 Candidate Pruning

The constant time approach for distance computation introduced in the previous section helps reduce the total cost of shapelet discovery by a factor of m. Our next goal is to reduce the number of iterations that occur in the **for** loop at line 6 in the Algorithm 5.1. The core idea behind our attack on this problem is the following observation: If we know (s, τ) is a poor shapelet (i.e. it has low information gain) then any similar subsequence to s must also result in a low information gain and, therefore, a costly evaluation (computing all the distances) can be safely skipped. Assume we have observed a good $best\text{-}so\text{-}far$ shapelet at some point in the algorithm. Let us denote this shapelet (S_p, τ_p) and its information gain I_p. Imagine we now test $(S_{i,l}, \tau)$, and it has very poor information gain $I_{i,l} < I_p$. Let us consider the next subsequence $S_{i+1,l}$. Here is our fundamental question. Is it possible to use the relationship (the Euclidean distance) between $S_{i,l}$ and $S_{i+1,l}$, to prune $S_{i+1,l}$?

To develop our intuition, let us first imagine a pathological case. Suppose that $dist(S_{i,l}, S_{i+1,l}) = 0$; in other words, $S_{i+1,l}$ is the exact same shape as $S_{i,l}$. In that case we can obviously prune $S_{i+1,l}$, since its information gain must also be $I_{i,l}$. Suppose, however, in the more realistic case, that $S_{i,l}$ and $S_{i+1,l}$ are similar, but not identical. We may still be able to prune $S_{i+1,l}$. The trick is to ask, "how good could $S_{i+1,l}$ be?" , or a little more formally, "What is an upper bound on the information gain of $S_{i+1,l}$." It turns out that it is simple to calculate this bound!

Let the distance between $S_{i,l}$ and $S_{i+1,l}$ be R. By triangular inequality, $sdist(S_{i+1,l}, D_k)$ can be as low as $sdist(S_{i,l}, D_k) - R$ and as high as $sdist(S_{i,l}, D_k) + R$ regardless of the alignment of $S_{i+1,l}$ on D_k. Thus, every point on the orderline of $S_{i,l}$ has a "*mobility*" in

the range $[-R, R]$ from its current position. Given this mobility, our goal is to find the best configuration of the points that gives maximum information gain when split into two partitions. The points that lie outside $[\tau - R, \tau + R]$ can have no effect on the information gain for candidate $(S_{i+1,l}, \tau)$. For the points inside $[\tau - R, \tau + R]$ we can shift them optimistically in either direction to increase the information gain.

Every class $c \in C$ has n_c instances in the database which are divided into two partitions by a split. Let $n_{c,1}$ and $n_{c,2}$ be the number of instances of class c in partition D_{left} and D_{right}, respectively. A class is weighted to the left (or simply called *left-major/right-minor*) if $\frac{n_{c,1}}{N_1} > \frac{n_{c,2}}{N_2}$. Similarly, a class is called *right-major/left-minor* if $\frac{n_{c,1}}{N_1} \leq \frac{n_{c,2}}{N_2}$. The following lemma describes the optimal choice for a single point.

Theorem 5.1 *Shifting a point from its minor partition to its major partition always increases information gain.*

Proof: Lets assume c is a left-major class. So $\frac{n_{c,1}}{N_1} > \frac{n_{c,2}}{N_2}$. If we move one point of class c from the right partition to the left partition, for $i = 1, 2, \ldots, C$ the change in information gain is ΔI.

$$\Delta I = \sum_i \frac{n_{i,1}}{N} \log \frac{n_{i,1}}{N_1} + \sum_i \frac{n_{i,2}}{N} \log \frac{n_{i,2}}{N_2} - \sum_{i \neq c} \frac{n_{i,1}}{N} \log \frac{n_{i,1}}{N_1+1}$$
$$- \sum_{i \neq c} \frac{n_{i,2}}{N} \log \frac{n_{i,2}}{N_2-1} - \frac{n_{c,1}+1}{N} \log \frac{n_{c,1}+1}{N_1+1} - \frac{n_{c,2}-1}{N} \log \frac{n_{c,2}-1}{N_2-1}$$

Using $\sum_{i \neq c} n_{i,1} = N_1 - n_{c,1}$ and $\sum_{i \neq c} n_{i,2} = N_2 - n_{c,2}$, we can simplify the above as below.

$$N.\Delta I = N_1 \log N_1 - n_{c,1} \log n_{c,1} + (n_{c,1} + 1) \log(n_{c,1} + 1)$$
$$+ N_2 \log N_2 - n_{c,2} \log n_{c,2} + (n_{c,2} - 1) \log(n_{c,2} - 1)$$
$$- (N_1 + 1) \log(N_1 + 1) - (N_2 - 1) \log(N_2 - 1)$$

Since, $\frac{n_{c,1}}{N_1} > \frac{n_{c,2}}{N_2}$ for $t \in [0,1]$ the following is also true $\frac{n_{c,1}+1-t}{N_1+1-t} > \frac{n_{c,2}-t}{N_2-t}$. In addition, $n_{c,1}, N_1, n_{c,2}, N_2$ are all positive integers; therefore, we can take log on both side and integrate from 0 to 1.

$\int_0^1 \log \frac{n_{c,1}+1-t}{N_1+1-t} \mathrm{d}t - \int_0^1 \log \frac{n_{c,2}-t}{N_2-t} \mathrm{d}t > 0$. If we evaluate the integral, it becomes the right-side part of the above equation for $N.\Delta I$. Therefore, $\Delta I > 0$. ∎

If we shift a point from the minor partition to the major partition, the major/minor partitions of the class remain the same as before shifting, simply because, if $\frac{n_{c,1}}{N_1} > \frac{n_{c,2}}{N_2}$ then $\frac{n_{c,1}+1}{N_1+1} > \frac{n_{c,2}-1}{N_2-1}$. Therefore, shifting all the minor points of a class to its major partition increases the information gain monotonically, and thus, this can be treated as an atomic operation. We denote such a sequence of shifts as a "*transfer*". Transferring a class may change the major-minor partitions of other classes and, consequently, the transfer direction for that class. Therefore, if we transfer every class based on the major-minor partitions in the initial orderline, it does not necessarily guarantee the maximum information gain.

In the case of two-class datasets, there is always one left-major and one-right major class. Therefore, shifting all of the points in $[\tau - R, \tau + R]$ to their major partition will not change the major-minor partitions of either of the classes; thus, this guarantees the optimization (i.e. the upper bound) even if the initial transfer directions are used.

For a case of more than two-classes, almost all of the time initial major-minor partitions hold after all of the transfers are done. Unfortunately, it is possible to *construct* counter examples, even if we rarely confront them on real data. To obtain the optimal bound we need to try all possible transfer directions for all of the classes resulting in 2^C trials. Fortunately,

many classification problems deal with a small number of classes. For example, 60% of the UCR time series datasets [3] have four or less classes. Given this fact, having the 2^C constant in the complexity expression will not be an issue for many problems.

Algorithm 5.5 describes the computation of the upper bound. The algorithm loops through all distinct split positions (line 2-3). For every split position, the algorithm transfers all of the classes to their major partition based on the initial orderline (line 5-6) and computes the information gain to find the upper bound. Note that line 4 is "commented out" in the algorithm which is a **for** loop that checks all of the 2^C combinations of transfer directions. For exact bound in the case of many-class datasets, this line should be uncommented.

To summarize, for the two-class case we have an exact and admissible pruning technique. In the multi-class case we have a powerful heuristic that empirically gives answers that are essentially indistinguishable from the optimal answer.

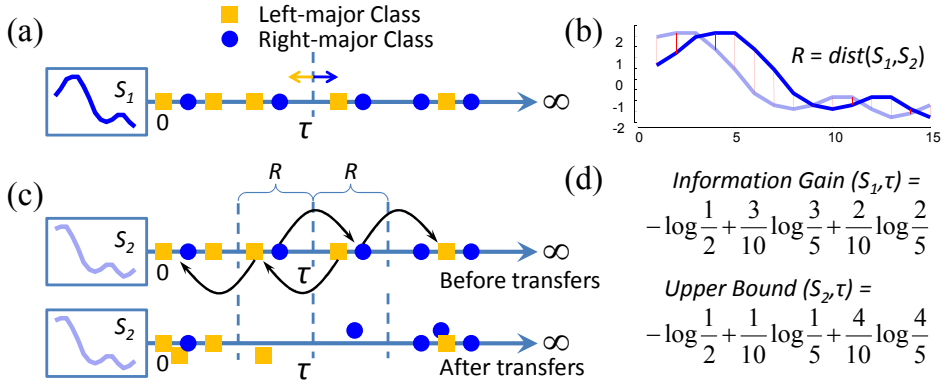

Figure 5.3: (a) A sequence S_1 and its orderline. (b) Distance between the sequences S_1 and S_2 is R. (c) The points on the orderline within $[\tau - R, \tau + R]$ are transferred to their majority partition. (d) The computation of the information gain for (S_1, τ) and upper bound for (S_2, τ)

Algorithm 5.5 $upperIG(L, R)$

Require: An order line L and the candidate distance R.
Ensure: Return an upper bound of information gain.

```
1:  maxI ← 0
2:  for k ← 1 to |D| − 1 except j do
3:      τ ← (L_k + L_{k+1})/2
4:      //for all 2^C combinations of transfer directions do
5:          for all points p ∈ [τ − R, τ + R]
6:              move p to its majority end.
7:          Count n_{1,i} and n_{2,i} for i = 1, 2, ..., C.
8:          Count N_1 and N_2 for both the partitions.
9:          I ← information gain computed by definition 5.4
10:         maxI ← max(maxI, I)
11:     return maxI
```

5.2.3 The Fast Shapelet Discovery Algorithm

With the speedup techniques described in the previous section, we are now ready to plug them into Algorithm 5.1 and build a faster version as shown in Algorithm 5.6.

In lines 5-7, the sufficient statistics are computed for the current time series D_j paired with every other time series D_k.

The algorithm maintains a set of orderlines in the history H. For every candidate $S_{i,l}$, before committing to the expensive computation of the orderline, the algorithm quickly computes upper bounds using the orderlines in the history (line 14). If *any* of the upper bounds is smaller than the maximum gain achieved so far we can safely abandon the candidate (line 15).

Since the upper bound computation is based upon the triangular inequality, we are only allowed to use the previous orderlines computed for sequences of the same length as the

current candidate [1]. Therefore, once the search moves on to the next length the algorithm clears the history H and starts building it up for the new length (line 9).

The size of the history H is a user-defined value and the algorithm is insensitive to this value once it is set to at least five. To prevent our history cache in line 22 growing without bound, we need to have a replacement policy. The oldest-first (LIFO) policy is the most suitable for this algorithm. This is because the recent subsequences tend to be correlated with the current candidate and, therefore, have small distances from the candidate. Note that, we do not add all orderlines to the history. We only add the orderlines that have less information gain (i.e. orderlines for poor shapelet candidate) than the current $maxGain$. Because only poor candidates have the power of pruning similar candidates by predicting their low information gain.

5.3 Logical-Shapelet

A shapelet is a tuple consisting of a subsequence and a split point (threshold) that attempts to separate the classes in exactly two different groups. However, it is easy to imagine situations where it may not be sufficient to use only one shapelet to achieve separation of classes, but a *combination* of shapelets. To demonstrate this, we show a simple example. In Figure 5.4(a), we have a two-class problem where each class has two time series. The square class contains two sinusoidal patterns with both positive and negative peaks, while the circle class has only

[1]The reader may wonder why we cannot create a bound between a sequence and a shorter sequence that is its prefix. Such bounds cannot be created because we are normalizing *all* sequences, and after normalizing the distances may increase *or* decrease.

Algorithm 5.6 $Fast\_Shapelet\_Discovery(D)$

Require: A dataset D of time series
Ensure: Return the shapelet

```
1: m ← minimum length of a time series in D
2: maxGain ← 0, maxGap ← 0
3: for j ← 1 to |D| do every time series in D
4:    S ← jth time series of D
5:    for k ← 1 to |D| do compute statistics for S and D_k
6:       x ← S, y ← D_k
7:       Stats_{x,y} ← {M, S_x, S_y, S_{x^2}, S_{y^2}}
8:    for l ← 1 to m do every possible length
9:       clear H
10:      for i ← 1 to |S| do every start position
11:         for w ← 1 to |H| do every candidate in H
12:            (L', S') ← H[w]
13:            R ← sdist(S_{i,l}, S')
14:            if upperIG(L', R) < maxGain then prune this candidate
15:               continue with next i
16:         for k ← 1 to |D| do since not pruned; compute distances of every time series to
            the candidate S_{i,l}
17:            L_k ← sdist_new(i, l, Stats_{x,y})
18:         sort(L)
19:         (τ, updated) ← bestIG(L, maxGain, maxGap)
20:         if updated then gain and/or gap are changed
21:            bestS ← S_{i,l},bestτ ← τ,bestL ← L
22:         add (L, S_{i,l}) to H if maxGain is not changed
23: return (bestS, bestτ, bestL, maxGain, maxGap)
```

one positive *or* one negative peak. If we attempt to use the classic shapelet definition to separate these classes, we find there is no way to do so. Classic shapelets simply do not have the expressiveness to represent this concept. The reason is that every single distinct pattern appears in both of the classes, or in only one of the time series of one of the classes. For example, in 5.4(b) three different unary shapelets and their orderlines are illustrated, and none of them achieved a separation between the classes.

To overcome this problem, we propose using multiple shapelets which allow distinctions based on logical combinations of the shapelets. For example, if we use the first two shapelets in Figure 5.4(b) then we can say that (S_1, τ_1) **and** (S_2, τ_2) separate the classes best. From now on, we use standard logic symbols $\bigwedge$ and $\bigvee$ for **and** and **or** operations. For a new instance x, if $sdist(S_1, x) < \tau_1 \bigwedge sdist(S_2, x) < \tau_2$ is true, then we can classify x as a member of the square class, or otherwise the circle class. When using multiple shapelets in such a way, chaining of logical combinations among the shapelets is possible; for example, $(S_1, \tau_1) \bigwedge (S_2, \tau_2) \bigvee (S_3, \tau_3)$. However, for the sake of simplicity and to guard against over fitting with too complex a model [31], we only consider two cases, only $\bigwedge$, and only $\bigvee$ combinations. We further guard against overfitting with too flexible a model by allowing only just a single threshold for both shapelets.

We have already seen how to create an orderline for classic shapelets, how do we define an orderline for conjunctive or disjunctive shapelets? To combine the orderlines for such cases, we adopt a very simple approach. For $\bigwedge$ operation, the *maximum* of the distances from the literal shapelets is considered as the distance on the new orderline, and for $\bigvee$ the *minimum* is considered. Apart from these minor changes, the computation of the entropy and information gain are unchanged.

For example, in Figure 5.4(d) the combined orderline for $(S_1 \bigwedge S_2, \tau)$ is shown. The two classes are now separable because both the shapelets occur *together* in the square class and do not both occur together in individual instances of the circle class.

Given these minor changes to allow logical-shapelets, we can still avail ourselves of the speedup techniques proposed in Section 5.2. The original shapelet algorithm just needs to be modified to run multiple times, and some minor additional bookkeeping must be done. When the first shapelet is found by the Algorithm 5.6, we now test to see if there is any class whose instances are in both the partitions of the optimal split. We call such a class "broken" (We could say "non-linearly separable," however, our situation has a slightly different interpretation). If there is a broken class, we continue to search for a new shapelet on the same dataset that can merge the parts of the broken class. However, this time we *must* make sure that new shapelet does not have a match with a subsequence in D_k that overlaps with the matching subsequence of an already discovered shapelet. After finding the new shapelet, we combine the orderlines and the threshold based on the appropriate logic operation. Finally, we are in position to measure the information gain to see if it is better than our $best\text{-}so\text{-}far$. If the gain improves, we test for a broken class again and continue as before. Once there is no broken class or the gain does not increase, we recursively find shapelet(s) in the left and right partitions.

The above approach is susceptible to overfitting. It can degenerate into $(S_1, \tau_1) \bigvee (S_2, \tau_2) \bigvee \ldots \bigvee (S_{n_i}, \tau_n)$ where n_i is the number of instances in the class i and each S_j for $j = 1, 2, \ldots, n_i$ is a subsequence of different instances of class i in the dataset D. In this case, all of the instances of class i would be in the left side with zero distances. To avoid overfitting, we can have a hard bound on the number of literals in the logic expression. In this work, the bound is hard coded to four, however, for a very large dataset we could slowly increase

this number, if we carefully check to guard against overfitting. Note that the situation here is almost perfectly analogous to the problem of overfitting in decision trees, and we expect that similar solutions could work here.

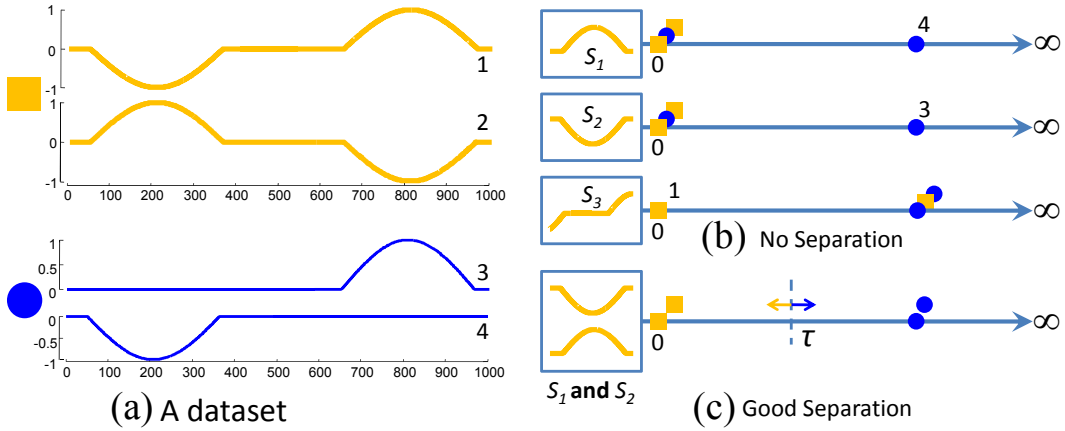

Figure 5.4: (a) Two classes of synthetic time series. (b) Examples of single shapelets that cannot separate the classes. Any other single shapelet would fail similarly. (c) Two shapelets connected by an *and* operation can separate the classes

5.4 Evaluation

We begin by noting that we have taken extraordinary lengths to make all our experiments reproducible. As such, all code and data are available at [2], and will be maintained there in perpetuity. Moreover, while our proposed algorithm is not sensitive to the few parameters required as inputs, we explicitly state all parameters for every dataset at [2].

We wish to demonstrate two major points with our experiments. First, our novel speedup techniques can be used to find both classic and logical shapelets significantly faster. Second,

there exist real-world problems for which logical-shapelets are significantly more accurate than classic shapelets or other state-of-the-art techniques (see Section 5.5).

To demonstrate the speedup, we have taken 24 datasets from the UCR time series archive [51]. For brevity the names and properties of these datasets and *tables* of time taken for running *the* shapelet algorithms on these datasets can be found at [2]. Here we content ourselves with a visual summary. In Figure 5.5*(left)*, we show the speedups over the original algorithm. Our algorithm obtained *some* speedup for all of the datasets with a maximum of 27.2.

The two speedup methods described in Section 5.2 are not independent of each other. Therefore, we also measure the *individual* speedups for each of the techniques while deactivating the other. The individual speedup factors for both the techniques are plotted in Figure 5.5*(right)*. There is no speedup for two of the datasets (shown by stars) when only the candidate pruning method is active. The reason is that the amount of pruning achieved for these datasets cannot surpass the overhead costs of computing the upper bounds for every candidate. However, when the technique of efficient distance computation is active, speedups are achieved for *all* of the datasets including these two.

It is critical to note that our improvements in speed are not due to trivial differences in hardware or to simple implementation optimizations, but reflect the utility of the two original techniques introduced in Section 5.2. In particular, all experiments were done on exactly the same environment and on the same input data files. The code for the original shapelet discovery algorithm was graciously donated by Dr. Ye who also confirmed that we are using

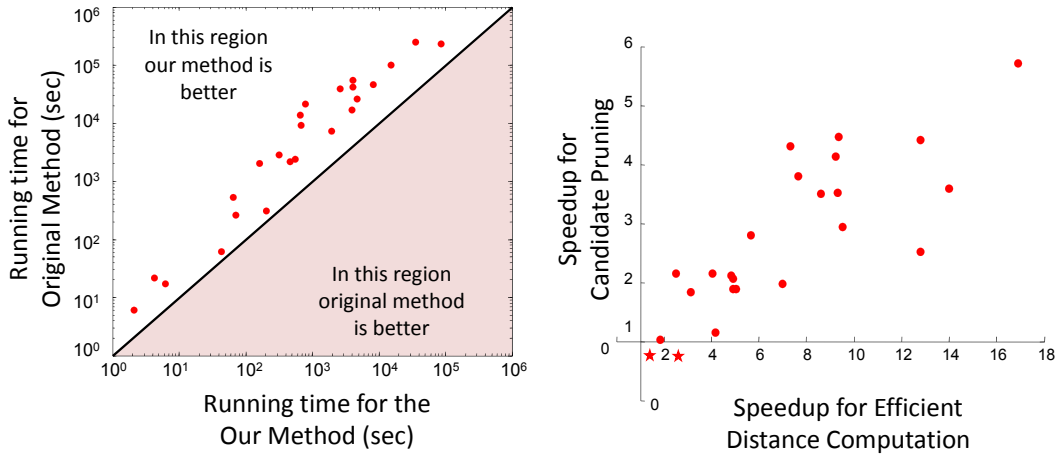

Figure 5.5: *(left)* Comparison of running times between our method and the original shapelet algorithm. Note the log scale on both axes. *(right)* The individual speedup factors for both of our proposed techniques: Candidate Pruning and Efficient Distance Computation

her code in the best possible manner. Since our technique reduced the worst case complexity by a factor of m and has an admissible pruning technique which is not present in the original code, we can be sure that the speedup is valid.

As described in Section 5.2.2, our linear time upper bound is not admissible for the many-class cases. Among the 24 datasets we used for scalability experiments, 13 datasets have more than two classes. For these 13 datasets, the average rate of computing false upper bound is just 1.56% with a standard deviation of 2.86%. In reality, the impact of false bounds on the final result is inconsequential because of the massive search space for shapelet. Our algorithm rarely misses the optimal information gain in that space and has not missed in *any* of the above 13 many-class datasets.

5.5 Case Studies

In this section we show three case studies in three different domains. In all the case studies we demonstrate that logical combinations of shapelets can describe the difference between the classes very robustly. We compare our algorithm to the 1-NN classifier using Dynamic Time Warping (DTW) because a recent extensive comparison of dozens of time series classification algorithms, on dozens of datasets, strongly suggests that 1-NN DTW is exceptionally difficult to beat [29]. Note that the 1-NN classifier using DTW is less amenable for realtime classification of time series since it requires an $O(m^2)$ distance measure to be computed for every instance in the dataset. In contrast, classification with shapelets requires just a single $O(n(m-n))$ calculation (n is the length of the shapelet). Thus, classification with time series shapelets is typically thousands of times faster than 1-NN DTW. We do not experimentally show this here, since it was amply demonstrated in the original shapelet paper [115].

5.5.1 Cricket: Automatic Scorer

In the game of cricket (a very popular game in British Commonwealth countries), an umpire signals different events in the game to a distant scorer/book-keeper. Typically, the signals involve particular motions of the hands. For example, the umpire stretches up both the hands above the head to signal a score of "six." A complete list of signals can be found in [1]. Researchers have recently begun to attempt to classify these signals automatically to ease/remove the task of a scorer [55].

In [55], a dataset was collected in which two accelerometers have been placed on both wrists of four different umpires. The umpires performed twelve different signals used in the game of cricket at least ten times. For simplicity of presentation, we select only two classes from the dataset that has a unique possibility of confusion. The two classes are "No Ball" and "Wide Ball." To signal "No Ball," the umpire must raise *one* of his hands to the shoulder-height and keep his hand horizontal until the scorer confirms the record. To signal "Wide Ball," the umpire stretches both of the hands horizontally at shoulder-height (see Figure 5.6).

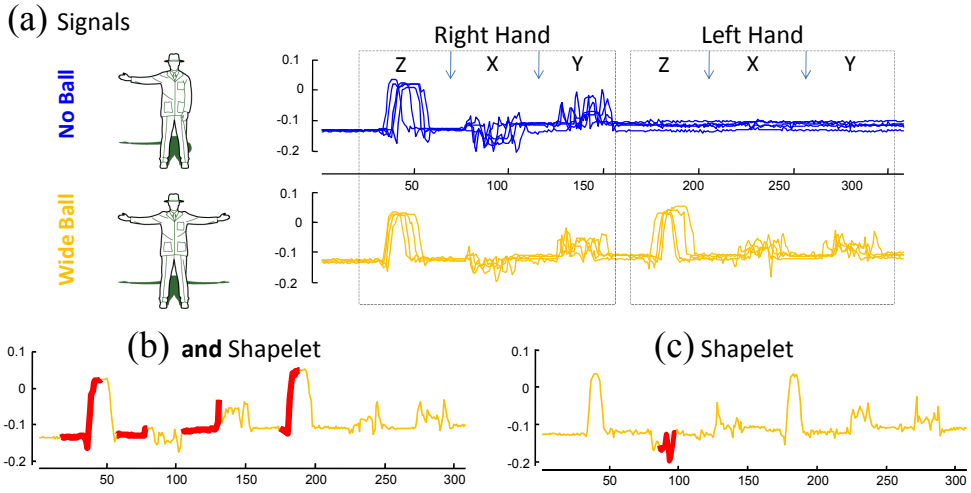

Figure 5.6: (a)The training set of the cricket dataset by concatenating signals from every axis of the accelerometer. (b) The two signs an umpire performs to declare two types of illegal delivery. (c) Shapelets found by our algorithm and the original algorithm

Each accelerometer has three synchronous and independent measurements for three axes (X,Y, and Z). For every signal performed, the six channels are concatenated to form one time series. We append low variance white noise to each example in the dataset to make them of length 308. The training set has nine examples as shown in Figure 5.6. The test set has 98 examples. Note that the examples for "No Ball" are only right hand signals. This

Algorithms	Original Test set	New Test set
1-NN Euclidean distance	94.89%	56.25%
1-NN Dynamic Time Warping	**98.98**%	87.50%
1-NN DTW-Sakoe-Chiba	94.89%	75.00%
Shapelet	44.89%	48.43%
Logical Shapelet	95.91%	**89.06**%

Table 5.1: The accuracies of different algorithms on the two test sets.

is true for the test set also. To cover the left handed case and also to include examples of different lengths, we have generated another test set using the accelerometer on a standard smart phone. This test set contains 64 examples of length 600. Each class has an equal number of examples in both of the training sets.

On this dataset, we performed 1-NN classification using Euclidean distance and Dynamic Time Warping (DTW) as distance measures. We also considered DTW with the Sakoe-Chiba band [89], as it has been shown to outperform classic DTW on many datasets [52]. The results are shown in table 5.1. It is interesting to consider why Logical Shapelets generalize the new data the best. We conjecture that it is the ability of Logical Shapelets to extract just the meaningful part of the signal. In contrast, DTW and Euclidean distance must account for the entire time series, including sections that may be unique to individual umpires idiosyncratic motion, but not predictive of the concept.

The Computationally expensive 1-NN DTW performs well in both of the cases, but not suitable for real-time applications. The original shapelet algorithm fails to capture the fact that the inherent difference between the classes is in the number of occurrences of the shapelet. Our logical-shapelet algorithm captures the sharp rises in the Z-axes for “Wide

Ball" from the original training set. No such "No Ball" signal can have two such rises in the Z-axes, and therefore, classification accuracy does not decrease significantly for the new test set.

5.5.2 Sony AIBO Robot: Surface Detection

The SONY AIBO Robot is a small, dog-shaped, quadruped robot that comes equipped with multiple sensors, including a tri-axial accelerometer. We consider a dataset created by [106] where measurements of the accelerometer are collected. In the experimental setting, the robot walked on two different surfaces: carpet and cement. For simplicity we consider only the X-axis readings. A snap shot of the data is shown in Figure 5.7(a). Each time series represents one walk cycle. Cemented floors are hard compared to carpets and, therefore, offer more reactive force than carpets. As such, there are clear and sharp changes in the acceleration on the cemented floor. In addition, there is a much larger variability when the robot walks on cement.

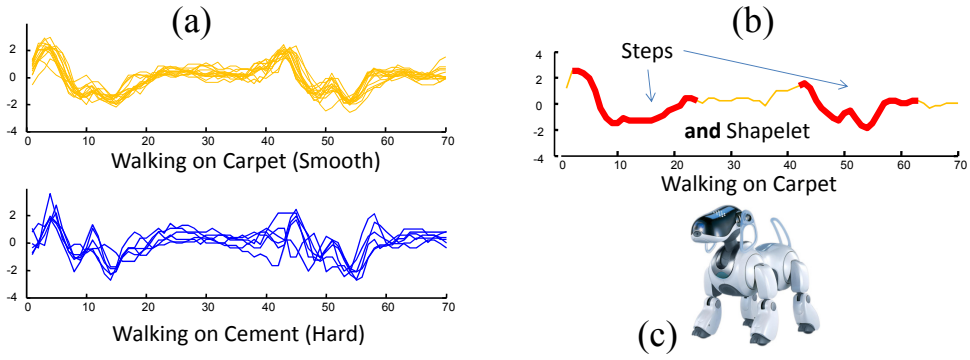

Figure 5.7: (a) Two classes of time series from the SONY AIBO accelerometer. (b) The *and*-shapelets from the walk cycle on carpet. (c) The Sony AIBO Robot

Algorithms	Surface Detection	Passgraphs
1-NN Euclidean distance	69.55%	63.36%
1-NN Dynamic Time Warping	72.55%	71.76%
1-NN DTW-Sakoe-Chiba	69.55%	**74.05**%
Shapelet	93.34%	60.31%
Logical Shapelet	**96.34**%	70.23%

Table 5.2: The accuracies of different algorithms on the passgraph trajectories and accelerometer signals from SONY AIBO robot.

The test set has 20 instances of walk cycles on the two types of floors. The training set has 601 instances. A walk cycle is of length 70 at 125 hertz. We have experimented on this dataset and found a pair of shapelets shown in Figure 5.7(b). The shapelets are connected by $\bigwedge$ and come from the two different shifts-of-weight in the walk cycle on the carpet floor. The pair of shapelets has a significantly higher classification accuracy compared to classic nearest neighbor algorithms (see table 5.2).

5.5.3 Passgraphs: Preventing Shoulder-Surfers

Passgraphs are a recently proposed biometric system used to authenticate a person and allow her access to some resource. A grid of dots is presented to the user and she is tasked to connect some of the dots in the grid in some specific order as a password. In contrast to text-based passwords where the user can shield the the entered text (at an ATM for example), Passgraphs are vulnerable to "shoulder-surfing" as it is easy for a miscreant to see and memorize the connection sequence over the shoulder of the user. To prevent the shoulder-surfing attack, [63] has proposed methods involving pen-pressure. There are also methods based on other pen properties such as friction and acceleration.

In this case study, we use the data from [63] to see if logical-shapelets can classify between the same pen sequences performed by different users. We selected x-axis trajectories of two different users drawing the same passgraph. The logical-shapelet shown in Figure 5.8 consists of tiny fragments of the three turns (peaks in the time series) in the passgraph connected by $\bigvee$ operations. These shapelets have better accuracies than 1-NN classifier and are promising enough to use in a real authentication system. This is because the shapelets can be learned at training time when the user sets the password, and it is not possible for the shoulder surfer to mimic the idiosyncratic pen path of the real user when attempting to impersonate her. Note that, $\bigwedge$ operations in this case would have imposed a harder rule for the user to produce *all* instead of *some* of the shapelets exactly to get authenticated.

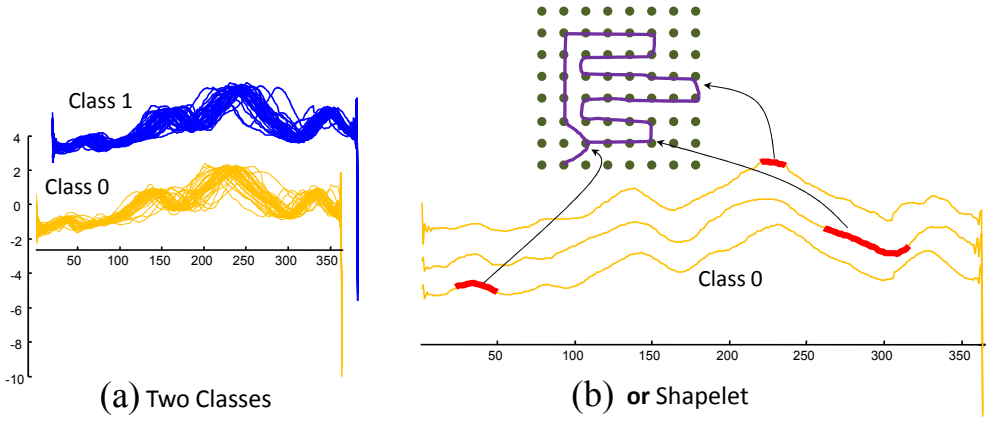

Figure 5.8: (a) Two classes of X-axis trajectories drawn by different users. (b) The *or*-shapelets from three different examples of class 0 showing three turns in the passgraphs

5.6 Conclusion

In this chapter, we introduce logical-shapelets: a new time series classification primitive with more expressiveness than classic shapelets. We have demonstrated the existence of logical concepts in time series datasets, and the utility of logical-shapelets in domains as diverse as gesture recognition, robotics and user authentication. We further describe novel techniques to efficiently find both classic and logical-shapelets. Our approach is significantly faster for every one of the twenty-four datasets we tested.

Chapter 6

Conclusion

Time series data is growing rapidly. We are close to if not already in a time when data is generated faster than the resources we have to process them. Not all of this massive corpus of data are equally useful. Automated method to organize and extract knowledge from this data is of prime importance now. We believe that such automated intelligent systems should be incrementally built in a hierarchical fashion. Time series data mining system is no different and should be built upon a set of very efficient primitives. Time series motifs and shapelets are two such primitives. Other well known primitives include bursts, periods, outliers and so on.

In this context, the contributions of this thesis are as follows.

- We show efficient exact algorithms for two primitives for time series data mining. The algorithms are guaranteed to output the optimal answers and very efficient compared to the available alternatives. The algorithms are based on the metric properties of the

similarity measure (i.e. the Euclidean distance) and use several computational tricks to reduce and reuse computation.

- We show applications of our algorithms as "primitives" to classification, online compression, and summarization. Our primitives increase both accuracy and efficiency of these algorithms. We show successful applications on time series data generated from diverse sensors such as motion capture, accelerometer, EEG, EOG, ECG and EPG.

- We extend our algorithms for various environments. We show efficient solutions for large scale motif discovery on disk resident data. We show online algorithm for motif discovery in streaming time series. We show exact algorithm for finding logical-shapelets to represent complex concepts that single shapelets cannot do.

This thesis have had a strong impact in the research community. There have been numerous downloads of our software packages and over hundred citations to our published articles. Our core technique (i.e. the *order line* and its application in search-space pruning) has been adopted by [13][14][83]. The parallel version of the MK algorithm is proposed by [75]. Online MK algorithm has been extended to top-K motif discovery in [57]. [83] and [77] extend the MK algorithm to discover motifs of variable lengths. The MK algorithm is also used to initialize the K-means clustering for time series data [87].

Mining time series data will continue to be an important area of research in coming years because of the growing ubiquity of time series. We can classify the future development of time series data mining in the following major areas.

- Developing new primitives with solid evidences of versatility and efficiency. Examples can be pattern association/linkage discovery, active (i.e. user assisted) pattern discovery etc.

- Defining new problems that use existing primitives. Examples include rule discovery, dictionary building, pattern visualization etc.

- Expanding to new domains e.g. historical manuscripts mining, social media mining, sports data mining, geographic data mining etc.

- Improving the efficiency of existing primitives, possibly using modern hardwares such as multicore processors, GPU (Graphic Processing Units) [91], FPGA (Field Programmable Gate Arrays) etc.

We expect this thesis will continue to play important roles in the future development of time series data mining and serve the practitioners with valuable insights into this fascinating area of research.

Bibliography

[1] Bbc sport — cricket — laws & equipment. http://news.bbc.co.uk/sport2/hi/cricket/rules_and_equipment.

[2] Supporting webpage containing code, data, excel sheet and slides. http://www.cs.ucr.edu/~mueen.

[3] The ucr time series classification/clustering homepage. www.cs.ucr.edu/~eamonn/time_series_data/.

[4] H Abe and T Yamaguchi. Implementing an integrated time-series data mining environment - a case study of medical kdd on chronic hepatitis. In *1st international conference on complex medical engineering CME2005*. 2005.

[5] Hidenao Abe, Miho Ohsaki, Hideto Yokoi, and Takahira Yamaguchi. Implementing an integrated time-series data mining environment based on temporal pattern extraction methods: A case study of an interferon therapy risk mining for chronic hepatitis. In *New Frontiers in Artificial Intelligence*, volume 4012, pages 425–435. 2006.

[6] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. *Efficient similarity search in sequence databases*, volume 8958546, pages 69–84. 1993.

[7] D. Arita, H. Yoshimatsu, and R. Taniguchi. Frequent motion pattern extraction for motion recognition in real-time human proxy. In *Proceedings of JSAI Workshop on Conversational Informatics*, pages 25–30, 2005.

[8] Christian Böhm and Florian Krebs. High performance data mining using the nearest neighbor join. In *Proceedings of ICDM*, pages 43–50, 2002.

[9] Philippe Beaudoin, Stelian Coros, Michiel van de Panne, and Pierre Poulin. Motion-motif graphs. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08, pages 117–126, 2008.

[10] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19:322–331, 1990.

[11] Jon Louis Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23:214–229, 1980.

[12] Sergei N. Bespamyatnikh. An optimal algorithm for closest pair maintenance (extended abstract). In *Proceedings of the eleventh annual symposium on Computational geometry*, SCG '95, pages 152–161, 1995.

[13] K. Bhaduri, Qiang Zhu, N.C. Oza, and A.N. Srivastava. Fast and flexible multivariate time series subsequence search. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 48 –57, 2010.

[14] Kanishka Bhaduri, Bryan L. Matthews, and Chris R. Giannella. Algorithms for speeding up distance-based outlier detection. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, pages 859–867, 2011.

[15] N. Bigdely-Shamlo, A. Vankov, R.R. Ramirez, and S. Makeig. Brain activity-based image classification from rapid serial visual presentation. *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, 16(5):432 –441, 2008.

[16] A. Bulut and A.K. Singh. Swat: hierarchical stream summarization in large networks. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 303 – 314, 2003.

[17] Jean Cardinal and David Eppstein. Lazy algorithms for dynamic closest pair with arbitrary distance measures. In *Algorithm Engineering and Experiments Workshop*, 2004.

[18] B. Celly and V. Zordan. Animated people textures. In *17th International Conference on Computer Animation and Social Agents (CASA)*, 2004.

[19] Varun Chandola. *Anomaly Detection for Symbolic Sequences and Time Series Data*. PhD thesis, University of Minnesota, MN, USA, 2009.

[20] Edgar Chavez Gonzalez, Karina Figueroa, and Gonzalo Navarro. Effective proximity retrieval by ordering permutations. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30:1647–1658, 2008.

[21] S.S. Cheung and T.P. Nguyen. Mining arbitrary-length repeated patterns in television broadcast. In *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, volume 3, pages III – 181–4, 2005.

[22] Bill Chiu, Eamonn Keogh, and Stefano Lonardi. Probabilistic discovery of time series motifs. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, pages 493–498, 2003.

[23] Richard Cole, Dennis Shasha, and Xiaojian Zhao. Fast window correlations over uncooperative time series. In *KDD*, pages 743–749, 2005.

[24] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Closest pair queries in spatial databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 189–200, 2000.

[25] Mark Cummins and Paul Newman. Fab-map: Probabilistic localization and mapping in the space of appearance. *The International Journal of Robotics Research*, 27(6):647–665, 2008.

[26] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31:1794–1813, 2002.

[27] Deanna K Dawson and Murray G Efford. Bird population density estimated from acoustic signals. *Journal of Applied Ecology*, 46(6):1201–1209, 2009.

[28] A. Delorme and S. Makeig. Eeg changes accompanying learned regulation of 12-hz eeg activity. *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, 11(2):133 –137, 2003.

[29] Hui Ding, Goce Trajcevski, Xiaoyue Wang, and Eamonn Keogh. Querying and mining of time series data: Experimental comparison of representations and distance measures. In *In Proc of the 34 th VLDB*, pages 1542–1552, 2008.

[30] Vlastislav Dohnal, Claudio Gennaro, and Pavel Zezula. Similarity join in metric spaces using ed-index. In *Database and Expert Systems Applications*, volume 2736, pages 484–493. 2003.

[31] Pedro Domingos. Process-oriented estimation of generalization error. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 714–719, 1999.

[32] Florence Duchêne, Catherine Garbay, and Vincent Rialle. Learning recurrent behaviors from heterogeneous multivariate time-series. *Artificial Intelligence in Medicine*, 39(1):25–47, 2007.

[33] M.G. Elfeky, W.G. Aref, and A.K. Elmagarmid. Periodicity detection in time series databases. *Knowledge and Data Engineering, IEEE Transactions on*, 17(7):875 – 887, 2005.

[34] David Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *J. Exp. Algorithmics*, 5, 2000.

[35] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. *SIGMOD Rec.*, 23:419–429, 1994.

[36] Pedro G. Ferreira, Paulo J. Azevedo, Cndida G. Silva, and Rui M. M. Brito. Mining approximate motifs in time series. In *In proceedings of the 9th International Conference on Discovery Science*, pages 7–10, 2006.

[37] Erich Fuchs, Thiemo Gruber, Jiri Nitschke, and Bernhard Sick. On-line motif detection in time series with swiftmotif. *Pattern Recogn.*, 42:3015–3031, 2009.

[38] Ary L. Goldberger, Luis A. N. Amaral, Leon Glass, Jeffrey M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H. Eugene Stanley. Physiobank, physiotoolkit, and physionet : Components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000.

[39] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14:47–57, 1984.

[40] Thomas Guyet, Catherine Garbay, and Michel Dojat. Knowledge construction from time series data using a collaborative exploration system. *J. of Biomedical Informatics*, 40:672–687, 2007.

[41] James Hafner, Harpreet S. Sawhney, Will Equitz, Myron Flickner, and Wayne Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 17:729–736, 1995.

[42] Raffay Hamid, Siddhartha Maddi, Amos Johnson, Aaron Bobick, Irfan Essa, and Charles Isbell. Unsupervised activity discovery and characterization from event-streams. In *In Proc. of the 21st Conference on Uncertainty in Artificial Intelligence (UAI05*, 2005.

[43] Jiawei Han, Guozhu Dong, and Yiwen Yin. Efficient mining of partial periodic patterns in time series database. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 106–115, 1999.

[44] B. Hartmann and N. Link. Gesture recognition with inertial sensors and optimized dtw prototypes. In *IEEE International Conference on Systems Man and Cybernetics (SMC)*, pages 2102–2109, 2010.

[45] Bastian Hartmann, Ingo Schwab, and Norbert Link. Prototype optimization for temporarily and spatially distorted time series. In *the AAAI Spring Symposia*, 2010.

[46] Magnus Lie Hetland. Ptolemaic indexing. *CoRR*, abs/0911.4384, 2009.

[47] J. Vitolo I. Androulakis, J. Wu and C. Roth. Selecting maximally informative genes to enable temporal expression profiling analysis. In *Proceedings of Foundations of Systems Biology in Engineering*, 2005.

[48] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30:364–397, 2005.

[49] S. Kaffka, B. Wintermantel, M. Burk, and G. Peterson. *Protecting high-yielding sugarbeet varieties from loss to curly top*, volume 1. November 2000.

[50] E. Keogh, S. Chu, D. Hart, and M. Pazzani. An online algorithm for segmenting time series. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 289–296, 2001.

[51] E. Keogh, X. Xi, Wei L., and C. A. Ratanamahatana. *The UCR Time Series Classification/Clustering Homepage:*. www.cs.ucr.edu/∼eamonn/time_series_data/.

[52] Eamonn Keogh. Exact indexing of dynamic time warping. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 406–417, 2002.

[53] Eamonn Keogh, Li Wei, Xiaopeng Xi, Sang-Hee Lee, and Michail Vlachos. Lb_keogh supports exact indexing of shapes under rotation invariance with arbitrary representations and distance measures. In *Proceedings of the 32nd international conference on Very large data bases*, pages 882–893, 2006.

[54] Jon Kleinberg. Bursty and hierarchical structure in streams. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 91–101, 2002.

[55] Ming Hsiao Ko, G. West, S. Venkatesh, and M. Kumar. Online context recognition in multisensor systems using dynamic time warping. In *Intelligent Sensors, Sensor Networks and Information Processing Conference, 2005.*, pages 283 – 288, 2005.

[56] Nick Koudas and Kenneth C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. *IEEE Trans. on Knowl. and Data Eng.*, 12:3–18, 2000.

[57] H T Lam, N D Pham, and Toon Calders. Online discovery of top-k similar motifs in time series data. *SIAM Conference on Data Mining, SDM '11*, 2011.

[58] Te-Won Lee, Mark Girolami, and Terrence J. Sejnowski. Independent component analysis using an extended infomax algorithm for mixed subgaussian and supergaussian sources. *Neural Computation*, 11:3–18, 1999.

[59] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Pranav Patel. Finding motifs in time series. In *Proc. of 2nd Workshop on Temporal Data Mining at KDD*, pages 53–68, 2002.

[60] Jiayang Liu, Lin Zhong, Jehan Wickramasuriya, and Venu Vasudevan. uwave: Accelerometer-based personalized gesture recognition and its applications. *Pervasive and Mobile Computing*, 5(6):657–675, 2009.

[61] Zheng Liu, Jeffrey Yu, Xuemin Lin, Hongjun Lu, and Wei Wang. Locating motifs in time-series data. In *Advances in Knowledge Discovery and Data Mining*, volume 3518, pages 1–5. 2005.

[62] A.L. Loomis, E. Harvey, and G. Hobart. Disturbance patterns in sleep. *J. Neurophysiology*, 2:413–430, 1938.

[63] Behzad Malek, Mauricio Orozco, and Abdulmotaleb El. Saddik. Novel shoulder-surfing resistant haptic-based graphical password. In *In the Proceedings of the Euro-Haptics conference*, 2006.

[64] Holger Holst Mb, Mattias Ohlsson, Carsten Peterson, Lars Edenbrandt, and Holger Holst. A confident decision support system for interpreting electrocardiograms. *Clin Physiol.*, 19:410–418, 1999.

[65] Amy McGovern, Derek Rosendahl, Rodger Brown, and Kelvin Droegemeier. Identifying predictive multi-dimensional time series motifs: an application to severe weather prediction. *Data Mining and Knowledge Discovery*, 22:232–258, 2011.

[66] Jingjing Meng, Junsong Yuan, Mat Hans, and Ying Wu. Mining motifs from human motion. In *EUROGRAPHICS*, 2008.

[67] D. Minnen, T. Starner, M. Essa, and C. Isbell. Discovering characteristic actions from on-body sensor data. In *Wearable Computers, 2006 10th IEEE International Symposium on*, pages 11–18, 2006.

[68] David Minnen, Charles L. Isbell, Irfan Essa, and Thad Starner. Discovering multivariate motifs using subsequence density estimation. In *AAAI Conf. on Artificial Intelligence*, 2007.

[69] Dalia Motzkin and Christina L. Hansen. An efficient external sorting with minimal space requirement. *International Journal of Parallel Programming*, 11:381–396, 1982.

[70] Abdullah Mueen, Eamonn J. Keogh, Qiang Zhu 0002, Sydney Cash, and M. Brandon Westover. Exact discovery of time series motifs. In *SDM*, pages 473–484, 2009.

[71] Abdullah Mueen, Eamonn J. Keogh, and Nima Bigdely Shamlo. Finding time series motifs in disk-resident data. In *ICDM*, pages 367–376, 2009.

[72] Abdullah Mueen, Suman Nath, and Jie Liu. Fast approximate correlation for massive time-series data. In *SIGMOD Conference*, pages 171–182, 2010.

[73] K. Murakami, S. Doki, S. Okuma, and Y. Yano. A study of extraction method of motion patterns observed frequently from time-series posture data. In *Systems, Man and Cybernetics, 2005 IEEE International Conference on*, volume 4, pages 3610–3615, 2005.

[74] Alexandros Nanopoulos, Yannis Theodoridis, and Yannis Manolopoulos. C2p: Clustering based on closest pairs. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 331–340, 2001.

[75] Ankur Narang and Souvik Bhattacherjee. Parallel exact time series motif discovery. In *Euro-Par 2010 - Parallel Processing*, volume 6272, pages 304–315. Springer Berlin / Heidelberg, 2010.

[76] Ernst Niedermeyer and Fernando L. da Silva. *Electroencephalography: Basic Principles, Clinical Applications, and Related Fields*. Lippincott Williams & Wilkins, 5th edition, 2004.

[77] P. Nunthanid, V. Niennattrakul, and C.A. Ratanamahatana. Discovery of variable length time series motif. In *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2011 8th International Conference on*, pages 472 –475, 2011.

[78] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. Alphasort: a cache-sensitive parallel external sort. *The VLDB Journal*, 4:603–628, 1995.

[79] A. M. Odlyzko and E. M. Rains. On longest increasing subsequences in random permutations. In *Amer. Math. Soc., Contemporary Math.*, volume 251, pages 439–451, 2000.

[80] Y. Ogras and Hakan Ferhatosmanoglu. Online summarization of dynamic time series data. *The VLDB Journal*, 15:84–98, 2006.

[81] Themistoklis Palpanas, Michail Vlachos, Eamonn Keogh, Dimitrios Gunopulos, and Wagner Truppel. Online amnesic approximation of streaming time series. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, 2004.

[82] Spiros Papadimitriou and Philip Yu. Optimal multi-scale patterns in time series streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 647–658, 2006.

[83] Dhaval Patel, Wynne Hsu, Mong Lee, and Srinivasan Parthasarathy. Lag patterns in time series databases. In *Database and Expert Systems Applications*, volume 6262, pages 209–224. 2010.

[84] Pranav Patel, Eamonn Keogh, Jessica Lin, and Stefano Lonardi. Mining motifs in massive time series databases. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, 2002.

[85] Debprakash Patnaik, Manish Marwah, Ratnesh Sharma, and Naren Ramakrishnan. Sustainable operation and management of data center chillers using temporal data mining. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pages 1305–1314, 2009.

[86] Vincenzo Penteriani. Variation in the function of eagle owl vocal behaviour: territorial defence and intra-pair communication? *Ethology Ecology Evolution*, 14(1988):275–281, 2002.

[87] Le Phu and Duong Anh. Motif-based method for initialization the k-means clustering for time series data. In *AI 2011: Advances in Artificial Intelligence*, volume 7106, pages 11–20. 2011.

[88] Simona Rombo and Giorgio Terracina. Discovering representative models in large time series databases. In *Flexible Query Answering Systems*, volume 3055, pages 84–97. 2004.

[89] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 26(1):43–49, 1978.

[90] Yasushi Sakurai, Spiros Papadimitriou, and Christos Faloutsos. Braid: Stream mining through group lag correlations. In *SIGMOD Conference*, pages 599–610, 2005.

[91] Doruk Sart, Abdullah Mueen, Walid Najjar, Vit Niennattrakul, and Eamonn J. Keogh. Accelerating dynamic time warping subsequence search with gpus and fpgas. In *ICDM*, pages 1001–1006, 2010.

[92] Jin Shieh and Eamonn Keogh. *i*sax: indexing and mining terabyte sized time series. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 623–631, 2008.

[93] Jin Shieh and Eamonn J. Keogh. Polishing the right apple: Anytime classification also benefits data streams with constant arrival times. In *ICDM*, pages 461–470, 2010.

[94] C. A. Stafford and G. P. Walker. Characterization and correlation of dc electrical penetration graph waveforms with feeding behavior of beet leafhopper, circulifer tenellus. *Entomologia Experimentalis et Applicata*, 130(2):113–129, 2009.

[95] Bojana Stefanovic, Wolfram Schwindt, Mathias Hoehn, and Afonso C Silva. Functional uncoupling of hemodynamic from neuronal response by inhibition of neuronal nitric oxide synthase. *J Cereb Blood Flow Metab*, 27(4):741–754, 2006.

[96] John M. Stern and Jerome Engel Jr. *Atlas of EEG patterns*. Lippincott Williams & Wilkins, 2004.

[97] R. L. Rivest T. H. Cormen, C. E. Leiserson and C. Stein. *Introduction to Algorithms, 2nd Edition*. The MIT Press, McGraw Hill Book Company, 2001.

[98] Yoshiki Tanaka, Kazuhisa Iwamoto, and Kuniaki Uehara. Discovery of time-series motif from multi-dimensional data based on mdl principle. *Mach. Learn.*, 58:269–300, 2005.

[99] Heng Tang and Stephen Shaoyi Liao. Discovering original motifs with different lengths from time series. *Know.-Based Syst.*, 21:666–671, 2008.

[100] Sandeep Tata. *Declarative Querying For Biological Sequences*. The University of Michigan, 2007.

[101] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '2003, pages 309–320, 2003.

[102] A. Torralba, R. Fergus, and W.T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(11):1958 –1970, 2008.

[103] Vladimir Trifa, Lewis Girod, Travis Collier, Daniel T Blumstein, and Charles E. Taylor. Automated wildlife monitoring using self-configuring sensor networks deployed in natural habitats. In *International Symposium on Artificial Life and Robotics (AROB07)*, 2007.

[104] K. Ueno, Xiaopeng Xi, E. Keogh, and Dah-Jye Lee. Anytime classification using the nearest neighbor algorithm with applications to stream mining. In *Data Mining, 2006. ICDM '06. Sixth International Conference on*, pages 623–632, 2006.

[105] Alireza Vahdatpour, Navid Amini, and Majid Sarrafzadeh. Toward unsupervised activity discovery using multi-dimensional motif detection in time series. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 1261–1266, 2009.

[106] Douglas Vail and Manuela Veloso. Learning from accelerometer data on a legged robot. In *In Proceedings of the 5th IFAC/EURON Symposium on Intelligent Autonomous Vehicles*, 2004.

[107] Michail Vlachos, Suleyman Serdar Kozat, and Philip S. Yu. Optimal distance bounds on time-series data. In *SDM*, pages 109–120, 2009.

[108] Michail Vlachos, Christopher Meek, Zografoula Vagena, and Dimitrios Gunopulos. Identifying similarities, periodicities and bursts for online search queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 131–142, 2004.

[109] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 194–205, 1998.

[110] Li Wei, Nitin Kumar, Venkata Nishanth Lolla, Eamonn J. Keogh, Stefano Lonardi, and Chotirat (Ann) Ratanamahatana. Assumption-free anomaly detection in time series. In *SSDBM*, pages 237–240, 2005.

[111] D. Randall Wilson and Tony R. Martinez. Reduction techniques for instance-basedlearning algorithms. *Mach. Learn.*, 38:257–286, 2000.

[112] Zhengzheng Xing, Jian Pei, Philip Yu, and Ke Wang. Extracting interpretable features for early classification on time series. In *the Proceedings of SDM*, 2011.

[113] Dragomir Yankov, Eamonn Keogh, Jose Medina, Bill Chiu, and Victor Zordan. Detecting time series motifs under uniform scaling. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, pages 844–853, 2007.

[114] Dragomir Yankov, Eamonn J. Keogh, and Umaa Rebbapragada. Disk aware discord discovery: Finding unusual time series in terabyte sized datasets. In *ICDM*, pages 381–390, 2007.

[115] Lexiang Ye and Eamonn Keogh. Time series shapelets: a new primitive for data mining. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD, pages 947–956, 2009.

[116] Cui Yu, Bin Cui, Shuguang Wang, and Jianwen Su. Efficient index-based knn join processing for high-dimensional data. *Inf. Softw. Technol.*, 49:332–344, 2007.

[117] Jesin Zakaria, Sarah Rotschafer, Abdullah Mueen, Khaleel Razak, and Eamonn Keogh. Mining massive archive of mice sounds with symbolized representations. SDM, 2012.

[118] Xin Zhang. *Fast Algorithms for Burst Detection*. PhD thesis, Courant Institute of Mathematical Sciences, New York University, USA, 2006.