Online Discovery and Maintenance of Time Series Motifs

Abdullah Mueen
University of California, Riverside.
mueen@cs.ucr.edu

Eamonn Keogh
University of California, Riverside.
eamonn@cs.ucr.edu

ABSTRACT

The detection of repeated subsequences, *time series motifs*, is a problem which has been shown to have great utility for several higher-level data mining algorithms, including classification, clustering, segmentation, forecasting, and rule discovery. In recent years there has been significant research effort spent on efficiently discovering these motifs in static *offline* databases. However, for many domains, the inherent streaming nature of time series demands *online* discovery and maintenance of time series motifs.

In this paper, we develop the first online motif discovery algorithm which monitors and maintains motifs *exactly* in real time over the most recent history of a stream. Our algorithm has a worst-case update time which is linear to the window size and is extendible to maintain more complex pattern structures. In contrast, the current offline algorithms either need significant update time or require very costly pre-processing steps which online algorithms simply cannot afford.

Our core ideas allow useful extensions of our algorithm to deal with arbitrary data rates and discovering multidimensional motifs. We demonstrate the utility of our algorithms with a variety of case studies in the domains of robotics, acoustic monitoring and online compression.

Categories and Subject Descriptors

H.3.3 [**Information Systems**]: Information Search and Retrieval

General Terms

Algorithms, Performance

Keywords

Time Series, Motifs, Online Algorithms

1. INTRODUCTION

Time series motifs are approximately repeated subsequences of a longer time series stream. Figure 1 shows an example of a ten-minute long motif discovered in telemetry from a shuttle mission.

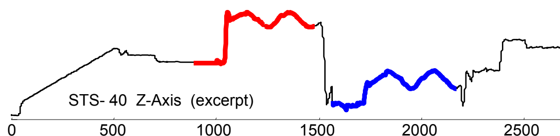

Figure 1: Forty-five minutes of Space Shuttle telemetry from an accelerometer. The two occurrences of the best ten-minute long motif are highlighted.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
KDD'10, July 25–28, 2010, Washington, DC, USA.
Copyright 2010 ACM 978-1-4503-0055-1/10/07...$10.00.

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. The utility of automatic algorithms for finding such motifs has been demonstrated in many domains. For example, [24] 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 also a core subroutine in at least a dozen research projects on *activity discovery* for humans and animals, with applications in elder care [27], surveillance and sports training. In addition, there has been a recent explosion of interest in motifs from the graphics and animation communities, where they are used for a variety of tasks, including finding transition sequences to allow just a few motion capture sequences to be stitched together in an endless cycle [2].

Given the ubiquity of time series motifs it is hardly surprising that many researchers have introduced techniques to find them efficiently. Until recently, all scalable algorithms were *approximate* [6] [27][2][16], but in [17] a scalable *exact* algorithm was introduced, and it was shown that exact motif discovery is tenable for a database with tens of millions of time series objects. 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 work 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 2 gives a visual intuition of the problem[1].

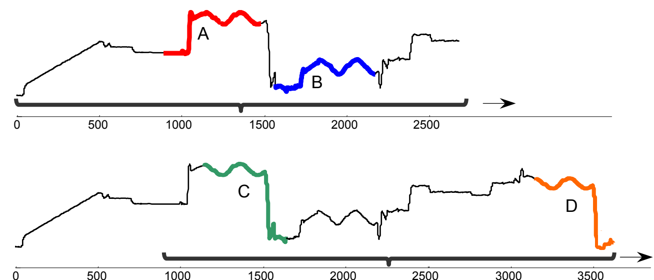

Figure 2: Maintaining motifs on a forty-five minute long sliding window. *top*) Initially A and B are the motif pair. *bottom*) but at time 790, two new subsequences C and D become the new motif pair of subsequences.

[1] As this is an inherently visual and dynamic problem, we have created a video version of Figure 2 at [31].

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 paper 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.

2. 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 time series:

Definition 1: *A time series* is a continuous sequence $\mathbf{x}=(x_1,x_2,...x_t)$ 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 2: A *sliding window* (W) is the latest w numbers $(x_{t-w+1},x_{t-w+2},...x_t)$ in the sequence $\mathbf{x}$.

Within a sliding window, we are interested in motifs, which informally are repeated subsequences:

Definition 3: A *subsequence* of length m of a time series $\mathbf{x}=(x_1,x_2,...x_t)$ is a time series $\mathbf{x}_{i,m} = (x_i,x_{i+1},...,x_{i+m-1})$ for $1 \le i \le t-m+1$.

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

Definition 4: The *online motif* of length m of a time series $\mathbf{x}=(x_1,x_2,...x_t)$ is a pair of subsequences $(\mathbf{x}_{i,m}, \mathbf{x}_{j,m})$ for $1 \le i < i+m \le j \le t-m+1$ such that $distance(\mathbf{x}_{i,m}, \mathbf{x}_{j,m})$ 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 [6]. Glancing back at Figure 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 5: The *exact search* for the online motif of length m of a time series $\mathbf{x}=(x_1,x_2,...x_t)$ finds the pair of subsequences $(\mathbf{x}_{i,m}, \mathbf{x}_{j,m})$ for $1 \le i < i+m \le j \le t-m+1$ such that *Euclidean distance* between $\mathbf{x}_{i,m}$ and $\mathbf{x}_{j,m}$ 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. Moreover, motifs from different windows can be collectively useful in high level data mining for collaborative structuring [29]. We think these high level modifications are out of this paper's scope.

To measure the distance between subsequences we use the ubiquitous Euclidean distance in Definition 5. Recent work has shown that in terms of time series classification accuracy, the Euclidean distance is surprisingly competitive [10]. Furthermore, Euclidean distance is a *metric* and allows the classic *early abandoning* optimization [1] and we exploit both facts in this work. To make the algorithm invariant to baseline and amplitude scaling, we z-*normalize* every subsequence and store it to avoid renormalizing every time it is compared. It is known that z-normalization improves the accuracy of time series classification for virtually every problem (i.e. on 37 out of 39 problems considered in [10]), but if appropriate we can work with non-normalized data. Therefore, we are now obliged to think of a subsequence as an independent object or a point in a high dimensional space unrelated to other objects/points. So the motif becomes the *closest pair of points* in this space.

At every time tick, a new subsequence $\mathbf{x}_{t-m+2,m}$ of length m is generated in W and the oldest subsequence $\mathbf{x}_{t-w,m}$ 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 like individual images, video frames, transactions, motion poses, etc.

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* $(1/bmw)$ at which our algorithm guarantees to operate. We assume that λ is below this maximum rate until section 6.1, where we remove the restriction.

2.1 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 [12], 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 a 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.

3. RELATED WORK

Eppstein describes an algorithm for maintaining the closest pair of points under any distance measure [12]. 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 [12][19] 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 [12] 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.

Another approach in dealing with dynamic closest pair maintenance is commonly known as the "lazy approach" [5]. 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 [3] 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 [28]. 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 [12], 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* $O(w^{3/2})$ *space* complexity.

In [8], 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 [8] 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. 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 3(a) (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, etc. We will begin with a naive version and revise it to define our algorithm.

4.1 The First Solution

First note that the closest pair in Figure 3(a) can be changed by one or both of the following two events (see Figure 3(b)):

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.

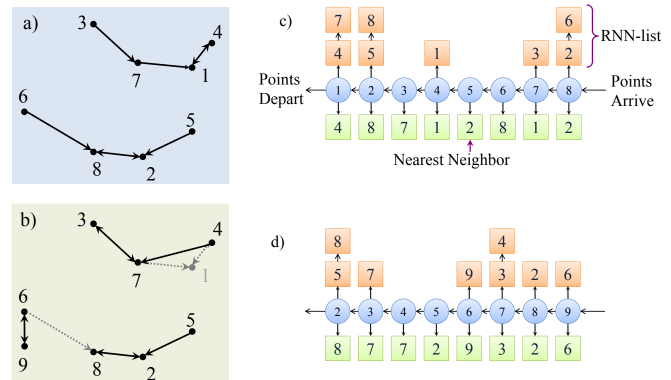

Figure 3: a) A set of 8 points. b) At a certain time tick 1 is deleted and 9 is inserted. c) The data structure of points. d) The data structure after the update.

Now, the arrows connecting the points in Figure 3(a,b) represent the nearest neighbor relation. For example, the arrow from 5 to 2 denotes that 2 is the nearest neighbor of 5. To 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 3(c) for this purpose. Here the horizontal arrows show the direction of insertion and deletion of points representing normalized subsequences. 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 (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 3(b)), 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 3(d)). The update upon insertion is $O(wm)$, as we have no way to avoid those 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 3(d)). 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 4(a), 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 4(a)). 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 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 4(a)), 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.

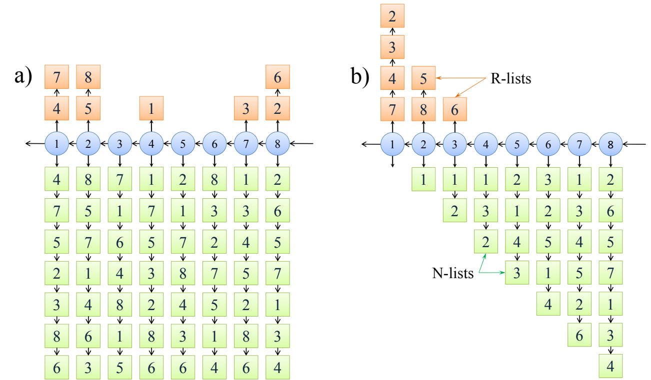

Figure 4: a) The squared space structure. Each point has one RNN-list (upper part) and one N-list (lower part). N-lists are in order of the distances from the owner. b) The reduction of space using observation 1.

4.2 Reducing Space and Time Complexity

We use two observations stated below for further refinement.

> *Observation 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 N-list after inserting all of the old objects. Figure 4(b) shows the data structure after applying observation 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 2:* A point x can never make a motif pair (x,y) with a later point y if there is a point $x < z < y$ such that $d(x,y) \geq d(z,y)$.

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 4(b)) and we can safely skip (6,4) when the N-list of 6 is created in the newer version (Figure 5(a)). 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 5(a).

After building the N-list, we can use observation 2 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($w^{3/2}$) 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,… ,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($n^{1/2}$) [20]. Therefore, the expected space needed for the data structure is O($w^{3/2}$).

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* [17] 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 [11]. 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 $y > x$.

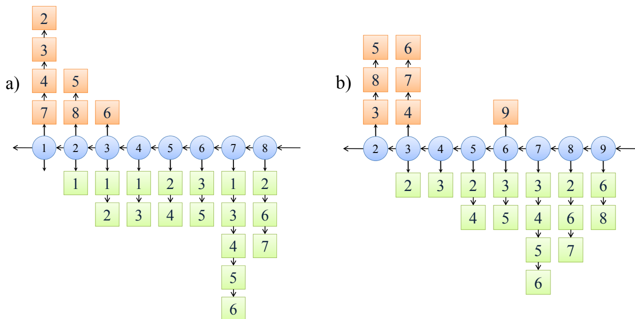

Figure 5: a) The space reduction using the temporal ordering of the neighbors. b) In the next time tick 1 is deleted from all of the lists and 9 is inserted.

To illustrate this idea, in Figure 6 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 6. First of all, 3 is inserted as $UL(3,6)=\infty$. Now, 2 has a lower bound $LB(2,6)=1$, 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.

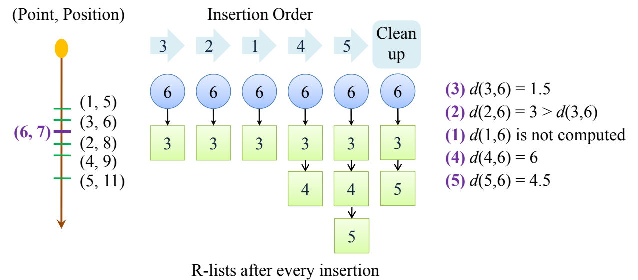

Figure 6: Building the Neighbor list of point 6. (*left*) The order line while 6 is being inserted. (*middle*) The states of the N-lists after each insertion. (*right*) The distance values assumed in this example.

4.3 The 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. Table 1 through Table 3 show the pseudocode of our algorithm. There are two subroutines for insertion and deletion made to the sliding window. Each of them takes in a point as the argument and performs the necessary operations on the data structure. At every time tick, *insertPoint(latest point)* and *deletePoint(oldest point)* 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)* (Table 1) is called with the new point, p, p is compared with the reference point (randomly generated or chosen from the database [17]). 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)* (Table 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 from p on the order line (line 2 in Table 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.

Table 1: Algorithm for Insertion.

Procedure	*insertPoint(p)*
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`
7	`Update motif pair with (p,p.N-list.head)`

When *buildNeighborList(p)* returns, the N-list is sorted according to the distances from p (line 3 of Table 1) and all the points that meet observation 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 of reduction in distance computation. We leave it as a design choice for the practitioners for brevity and lack of space.

Table 2: Algorithm for creating an N-list.

Procedure	*buildNeighborList(p)*
1	`while true`
2	`n ←next point from p on the order line`
3	`u ←UL(n, p.N-list)`
4	`if LB(n, p)< u`
5	`then if d(n,p)<u`
6	`insert n in p.N-list at the head`
7	`else if LB(n, p) ≥ d(prev_time(p),p)`
8	`break`

Table 3: Algorithm for Deletion.

Procedure	*deletePoint(p)*
1	`for all points q in R-list of p`
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`
6	`Find x for which d(x,x.N-list.head) is minimum`
7	`Update motif pair with (x,x.N-list.head)`

When the *deletePoint(p)* (Table 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 the new 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).

5. PERFORMANCE EVALUATION

We have used four very different datasets in our experiments, EEG trace [17], EOG trace, insect behavior trace [17] 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 [31]. 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[2], 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 [12] for this purpose. To be fair to the author of [12], 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 [12]. Figure 7(*top*) 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) 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 7(*bottom*) 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).

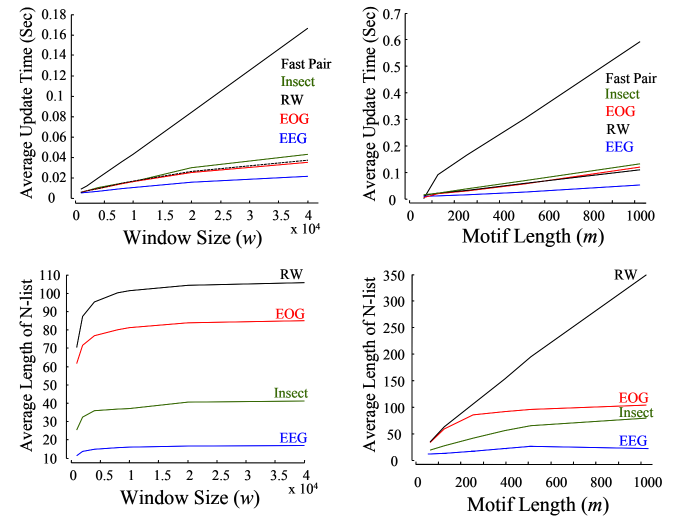

Figure 7: Empirical demonstration of the slow growth of (*top*) avg. update time and (*bottom*) avg. length of N-list. The parameters varied are (*left*) the window size with m=256 and

[2] Based just the title, the reader may imagine that *On-line motif detection in time series with SwiftMotif* [13] discovers time series motifs online. However this work finds approximate motifs *offline* then approximately filters them *online*.

(*right*) the motif length with *w*=40,000. Labels are in order of the heights of the right-most points of the curves.

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.

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 8(*left*) we show the average update time per point for every combination of two sets of possible values of w and m (Figure 8). Although the figure shows values for the EEG dataset, other datasets exhibit a similar shape. Figure 8(*right*) 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.

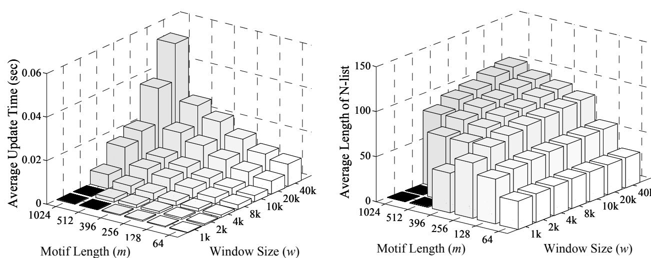

Figure 8: (*left*) Time usage per point in EEG dataset with varying *w* and *m*. (*left*) Space usage per point in EOG dataset with varying *w* and *m*.

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 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 d_m in our algorithm, only line 8 in Table 2 needs to be changed, to test if $LB(n, p) \geq \mathbf{min}(\ d(prev_{time}(p), p), d_m\ )$. If we can obtain a reasonable d_m 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 9(*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 9(*right*).

6. EXTENSIONS

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 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.

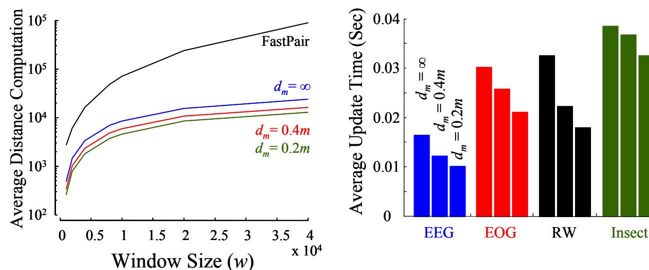

Figure 9: (*left*) The average amount of distance computation is much less in our algorithm than FastPair for EEG and further decreases with decreasing d_m. (*right*) Speedup is consistent over all of the datasets for m=256 and w = 40,000.

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 (cf. Section 3). 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. 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 [18], 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 $\mathbf{x}_{i,m}$ takes 30 ms then our algorithm would skip two immediate points ($\mathbf{x}_{i+1,m}$ and $\mathbf{x}_{i+2,m}$) and would start updating from the third point ($\mathbf{x}_{i+3,m}$) on. However, if an update takes less than 10 ms then we would not skip the following point. Therefore, for a smaller average update time (i.e. 6 ms 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 6 ms 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, etc. Figure 10(*left*) shows the fraction of the stream that is not skipped for different data rates with of w=32,000 and m=256.. 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 10(*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.

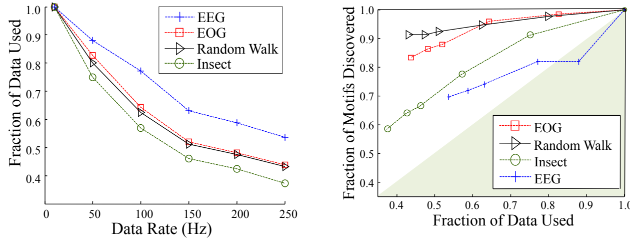

Figure 10: (*left*) Fraction of Data Used (the amount of subsequences considered) plotted against the varying data rate for *w*=32,000 and *m*=256. Our algorithm can operate at 200Hz while skipping roughly every other point. (*right*) The fraction of the motifs discovered drops more slowly than the fraction 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*" [6]). 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.

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 5(a). 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 4.2 hold for such N-lists, and the size of an N-list on average is still $O(w^{1/2})$.

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 $O(w^{3/2}d)$. The closest pair can be found as before by checking the heads of the N-lists in both of the data structure.

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* (recall our classic experiments are in Section 5). In particular, space limitations prohibit us from providing pseudocode and some minor details. However, this section is useful to motivate some applications made possible by online motif discovery. Note that, as before, all data and code is freely available at [31].

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 11.*middle*), piecewise constant approximations [15], Fourier approximations [15] and wavelets [4]. 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 paper repair this omission.

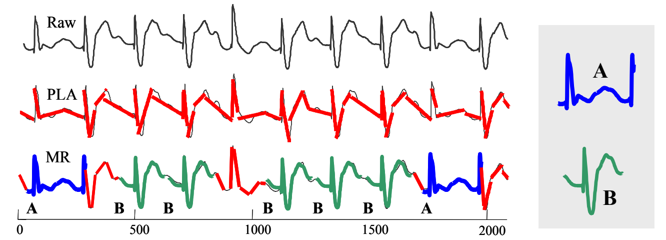

Figure 11: *top*) An excerpt of record sddb/51 Sudden Cardiac Death Holter Data. *middle*) A PLA of the data with an approximately 0.06 compression rate *bottom*) A Motif-Representation approximation of the data at twice the compression rate still has a lower error.

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 (c.f. Section 3). 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 [14] as a concrete example) All we have 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 11 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 [22], 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.

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 [26]. 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, [26] 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 [9] 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, 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 [25]. 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 an STD 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 12 we show an example of a detected motif with a value of 4.57, which corresponds to the call of a Great Horned Owl.

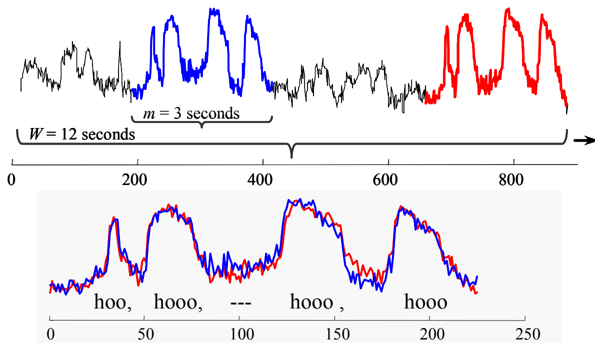

Figure 12: *top*) A stream is examined for motifs 3 seconds long, with a history of 12 seconds. *bottom*) The discovered motif is the cry of a Great Horned Owl, whose cry is phonetically written as *hoo, hooo,---,hooo, hooo.* Audio is available at [31].

7.3 Closing the Loop

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.

In an effort to verify this utility, we use the "New College" dataset [7], 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 13.

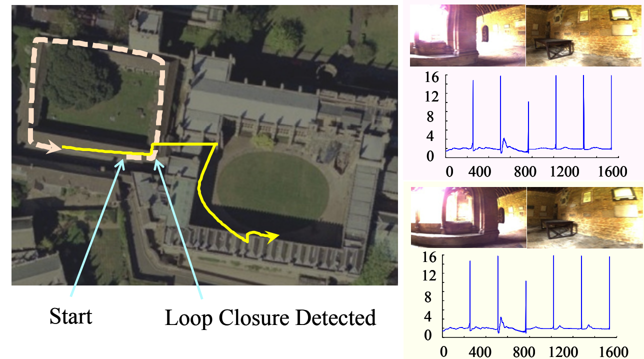

Figure 13: (*left*) The map of the "New College." A segment of robot motion is shown. (*right*) Motif: The most similar pair of image-pairs that are 90 samples apart and their color histograms. The image-pairs are from the same location and thus our algorithm detected the loop-closure.

8. CONCLUSION

In this work we introduced the first practical algorithm for finding and maintaining time series motifs on fast moving streams. Our algorithm performs updates in O(w) time and O($w^{3/2}$) amortized space where w is the size of the most recent window. We showed applications of our ideas in robotics, online compression and wildlife management. Future work includes reducing the worst case space complexity and an extensive field testing of the wildlife monitoring scenario.

9. REFERENCES

[1] Agrawal, R., Faloutsos, C. and Swami, A.N. Efficient Similarity Search in Sequence Databases. FODO 1993: 69-84.

[2] Beaudoin, P., Panne, M., Poulin, P. and Coros, S. Motion-Motif Graphs. ACM/EG Symposium on Computer Animation 2008.

[3] Bespamyatnikh, S. N. An Optimal Algorithm for Closest Pair Maintenance. ACM SCG '95, 152-161.

[4] Bulut, A. and Singh, A.: SWAT: Hierarchical Stream Summarization in Large Networks. In Proceedings of the ICDE (2003).

[5] Cardinal, J. and Eppstein, D. Lazy Algorithms for Dynamic Closest Pair with Arbitrary Distance Measures. ALENEX/ANALC 2004.

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

[7] Cummins, M. and Newman, P. *FAB-MAP: Probabilistic Localization and Mapping in the Space of Appearance.* The International Journal of Robotics Research, 27(6), 647-665, 2008.

[8] Datar, M., Gionis, A., Indyk, P., and Motwani, R. *Maintaining Stream Statistics over Sliding Windows.* SIAM J. Comput. 31, 6 (Jun. 2002), 1794-1813.

[9] Dawson, D. K. and Efford, M. G. *Bird Population Density Estimated from Acoustic Signals.* Journal of Applied Ecology. Volume 46 Issue 6, Pages 1201–1209.

[10] Ding, H., Trajcevski, G., Scheuermann, P., Wang, X. and Keogh, E. *Querying and Mining of Time Series Data: Experimental Comparison of Representations and Distance Measures*. VLDB 2008.

[11] Dohnal, V., Gennaro C. and Zezula, P. *Similarity Join in Metric Spaces Using eD-Index.* Database and Expert Systems Applications, Volume 2736, pp. 484-493, 2003.

[12] Eppstein, D. *Fast Hierarchical Clustering and Other Applications of Dynamic Closest Pairs.* ACM Journal of Experimental Algorithmics 5:1 (2000).

[13] Fuchs, E., Gruber, T., Nitschke, J. and Sick, B. *On-line Motif Detection in Time Series with SwiftMotif.* In: Pattern Recognition 42(11):3015-3031, 2009.

[14] Keogh, E., Chu, S., Hart, D. and Pazzani, M. *An Online Algorithm for Segmenting Time Series.* ICDM, pp. 289–296, 2001.

[15] Lazaridis, I. and Mehrotra, S. *Capturing Sensor-Generated Time Series with Quality Guarantees.* ICDE 2003.

[16] Lin, J., Keogh, E., Lonardi, S. and Patel, P. *Finding Motifs in Time Series*, Workshop on Temporal Data Mining (KDD'02), 2002.

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

[18] N. Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M. and Stonebraker. M. *Load Shedding in a Data Stream Manager.* VLDB 2003, pp. 309-320.

[19] Nanopoulos A., Theodoridis Y. and Manolopoulos, Y. *C2P: Clustering Based on Closest Pairs.* VLDB, pp. 331–340, 2001.

[20] Odlyzko, A.M. and Rains, E.M. *On Longest Increasing Subsequences in Random Permutation,* Analysis, Geometry, Number Theory: the Mathematics of Leon Ehrenpreis.439–451, Contemp. Math., 251, Amer. Math. Soc., Providence, RI, 2000.

[21] Ogras, Y. and Ferhatosmanoglu, H. *Online Summarization of Dynamic Time Series Data.* The VLDB Journal, 15(1):84–98, 2006.

[22] Palpanas, T., Vlachos, M., Keogh, E., Gunopulos, D. and Truppel, W. *Online Amnesic Approximation of Streaming Time Series.* In ICDE 2004.

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

[24] Patnaik, D., Marwah, M., Sharma, R.K. and Ramakrishnan, N. *Sustainable Operation and Management of Data Center Chillers using Temporal Data Mining.* KDD 2009: 1305-1314.

[25] Penteriani, V. *Variation in the Function of Eagle Owl Vocal Behaviour: Territorial Defence and Intra-Pair Communication?* Ethol. Ecol. Evol. 14: 275–281.

[26] Trifa, V.M., Girod, L., Collier, T., Blumstein, D.T. and Taylor, C.E. *Automated Wildlife Monitoring Using Self-Configuring Sensor Networks Deployed in Natural Habitats.* AROB 2007.

[27] Vahdatpour, A., Amini, N. and Sarrafzadeh, M. *Towards Unsupervised Activity Discovery using Multi Dimensional Motif Detection in Time Series.* 21st International Joint Conference on Artificial Intelligence (IJCAI) 2009, Pasadena, California.

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

[29] Wurst, M., Morik, K. and Mierswa, I. *Localized Alternative Cluster Ensembles for Collaborative Structuring*. ECML 2006. pp. 485-496.

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

[31] Supporting webpage containing Data, Code, Videos, Excel sheet and Presentation slides.
Link: http://www.cs.ucr.edu/~mueen/OnlineMotif/index.html