Atomic Wedgie: Efficient Query Filtering for Streaming Time Series

Li Wei Eamonn Keogh
Computer Science & Engineering Dept.
University of California – Riverside
Riverside, CA 92521
{wli, eamonn}@cs.ucr.edu

Helga Van Herle
David Geffen School of Medicine
University of California – Los Angeles
Los Angeles, CA 90095
hvanherle@mednet.ucla.edu

Agenor Mafra-Neto
ISCA Technologies
Riverside, CA 92517
isca@iscatech.com

Abstract

In many applications it is desirable to monitor a streaming time series for predefined patterns. In domains as diverse as the monitoring of space telemetry, patient intensive care data, and insect populations, where data streams at a high rate and the number of predefined patterns is large, it may be impossible for the comparison algorithm to keep up. We propose a novel technique that exploits the commonality among the predefined patterns to allow monitoring at higher bandwidths, while maintaining a guarantee of no false dismissals. Our approach is based on the widely used envelope-based lower bounding technique. Extensive experiments demonstrate that our approach achieves tremendous improvements in performance in the offline case, and significant improvements in the fastest possible arrival rate of the data stream that can be processed with guaranteed no false dismissal.

1. Introduction

In many applications it is desirable to monitor a streaming time series for a set of predefined patterns. Note that this problem is very different to the classic (and much studied) time series indexing problem [7][10]. It is however a very close analogue to the problem of *Query Filtering* for discrete valued data (e.g. XML) [3]. As noted in [3], "*filtering is the inverse problem of querying a database: In a traditional database system, a large set of data is stored persistently. Queries, coming one at a time, search the data for results. In a filtering system, a large set of queries is persistently stored. (new data), coming one at a time, drive the matching of the queries.*" While the need for filtering is well established in discrete domains (XML, Bioinformatics etc), to the best of our knowledge it has not been addressed for time series before. We will therefore take the time to motivate the need for time series filtering in several domains.

Electrocardiogram Monitoring: Cardiologists often encounter new interesting ECG patterns. These patterns may be unannotated or explicitly/implicitly annotated (eg. a pattern shows up older patients that were given the drug Terbutaline [1], or a pattern shows up when the Holter electrodes have gotten wet). In either case, once seeing an interesting pattern, a cardiologist will attempt to remember it so that future encounters with similar patterns can benefit from his experience. In our framework, all new interesting patterns are simply saved in the cardiologists "profile" and any future occurrences of similar patterns will be automatically flagged.

Audio Sensor Monitoring: The damage done by agricultural insect pests costs more than US$300 billion annually [6]. The best way known to mitigate this cost is to monitor insect populations and target harmful species before they can become a major problem. Technological advances and falling prices of hardware have created an explosion of interest in continuous, real-time monitoring of critical pest data by automated ("smart") traps in recent years [12]. While it has been shown in the lab that insects can be identified (species and sex) from audio of their wing beats [11], these successes are hard to reproduce in the field because field stations typically have low powered CPUs and the greater variety of possible insects (i.e patterns) encountered.

As shown above, time series filtering is applied in diverse domains. With continuously arriving data and large number of patterns, it may be impossible for the comparison algorithms to keep up. However, in real world, there is likely to be significant commonality among the predefined patterns. Based on this (empirically validated) assumption, we propose a hierarchical wedge-based comparison approach, which merges large number of patterns into a small set of wedges (with similar patterns being merged together) and then compares this set of wedges against the subsequence in the coming data stream. The experimental results show that our approach provides tremendous improvements in performance.

1.1 Related Work

To the best of our knowledge, this problem has not been addressed in the literature before. The most similar work is by Gao and Wang [4] , where the authors consider the problem of continuously finding the nearest neighbor to a streaming time series. They assume that the database of predefined patterns is in secondary memory. Thus the problem in question is disk-bound. In contrast, our problem is CPU-bound. We can easily fit our relatively small database of predefined patterns in main memory (indeed, in the insect monitoring problem, there is no secondary storage on the sensors). Furthermore, in Gao and Wang's problem definition, there is always *some* nearest neighbor to any streaming query. In contrast, we are only interested in finding *when* a streaming query is within r of anything in our database and we generally expect this to very rarely be the case.

The problem is very similar to the widely studied dictionary matching problem with errors and don't cares [2]. This problem is defined in [2] as "*preprocess(ing) a text or collection of strings, so that given a query string p, all matches of p with the text can be reported quickly*". The crucial difference is that this problem deals with discrete data, and researchers are therefore able to tackle it with an arsenal of tools that are defined only for discrete data, such as suffix trees and lexicographic sorting.

The rest of the paper is organized as follows. In Section 2 we review background material. We introduce our algorithms and representations in Section 3. Section 4 sees a comprehensive empirical evaluation and we offer some conclusions in Section 5.

2. Background Material

In this section, we review some background material. We begin with a definition of our data type of interest, time series.

Definition 1. *Time Series*: A time series $T = t_1,\ldots,t_m$ is an ordered set of m real-valued variables.

We are typically not interested in the global properties of a time series; rather, data miners confine their interests to subsequences of the time series.

Definition 2. *Subsequence*: Given a time series T of length m, a subsequence C_p of T is a sampling of length $w < m$ contiguous positions from T, that is, $C = t_p,\ldots,t_{p+w-1}$ for $1 \le p \le m - w + 1$.

In this work, we extract all the subsequences from a time series and compare them to the target time series. The extraction is achieved by use of a sliding window.

Definition 3. *Sliding Window*: Given a time series T of length m, and a user-defined subsequence of length w, all possible subsequences can be extracted by "sliding a window" across T and extracting subsequence C_p.

Definition 4. *Euclidean Distance*: Given two time series (or time series subsequences) both of length n, the *Euclidean Distance* between them is the square root of the sum of the squared differences between each pair of corresponding data points: $D(Q,C) \equiv \sqrt{\sum_{i=1}^{n}(q_i - c_i)^2}$, as shown in Figure 1.

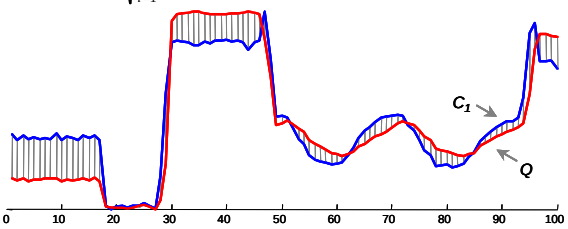

Figure 1: Illustration of Euclidean distance

If we are comparing two time series, in an attempt to discover if they are within a given distance r from each other, we can potentially speed up the calculation by doing *early abandoning*.

Definition 5. *Early Abandon*: During the computation of the Euclidean distance, if we note that the current sum of the squared differences between each pair of corresponding data points exceeds r^2, we can stop the calculation, secure in the knowledge that the exact Euclidean distance had we calculated it, would exceed r.

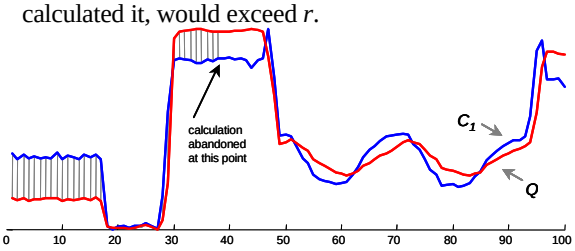

Figure 2: Illustration of early abandoning

While the idea of early abandoning is fairly obvious and intuitive [7], it is so critical to our work we illustrate it in Figure 2 and provide pseudocode in Table 1. We call the distance computation of each pair of corresponding data points a *step*, and we use num_steps to measure the utility of early abandonment.

Table 1: Euclidean distance with early abandonment

```
1   Function [dist, num_steps] = Optimized_Euclidean_Dist(Q, C, r)
2   accumulator = 0
3
4   For i = 1 to length(Q)                    // Loop over time series
5     accumulator += (qi - ci)^2              // Accumulate error contribution
6     If accumulator > r^2                    // Can we abandon?
7       disp('doing an early abandon')
8       num_steps = i
9       Return [ infinity, num_steps ]        // Terminate and return an
10    End                                     // infinite error to signal the
11  End                                       // early abandonment.
12  Return [ sqrt(accumulator), length(Q) ]   // Return true dist
```

3. Time Series Filtering

We are now in the position to give a formal statement of the problem. Assume we are given a set of time series **C** = {C_1, C_2,…, C_k} all of length w and a range r by a user. We want to either:

- Search a long batch time series for any subsequences that are within r of any time series in the set **C**, or
- Monitor a time series stream for any subsequences that are within r of any time series in the set **C**.

We make two realistic assumptions: 1) for the streaming case we only have a small O(**C**) memory buffer; 2) once we are given **C** and r, we have some reasonable amount of time (say O($\mathbf{C}^2$)) to prepare.

We begin by defining a representation of a set of time series. We use the set of C_1,..,C_k to form two new sequences U and L:

$$U_i = \max(C_{1i},..,C_{ki})$$
$$L_i = \min(C_{1i},..,C_{ki})$$

U and L stand for *Upper* and *Lower* respectively. We can see why in Figure 3. They form the smallest possible bounding envelope that encloses all members of the set C_1,..,C_k from above and below. More formally:

$$\forall_i \quad U_i \geq C_{1i},..,C_{ki} \geq L_i$$

We call the combination of U and L a *wedge*, and denote it as $W = \{U, L\}$. Now we define a lower bounding measure between an arbitrary query Q and the entire set of candidate sequences contained in a wedge W:

$$LB\_Keogh(Q,W) = \sqrt{\sum_{i=1}^{n} \begin{cases} (q_i - U_i)^2 & \text{if } q_i > U_i \\ (q_i - L_i)^2 & \text{if } q_i < L_i \\ 0 & \text{otherwise} \end{cases}}$$

The lower bounding property is proved in [10] using a different notation.

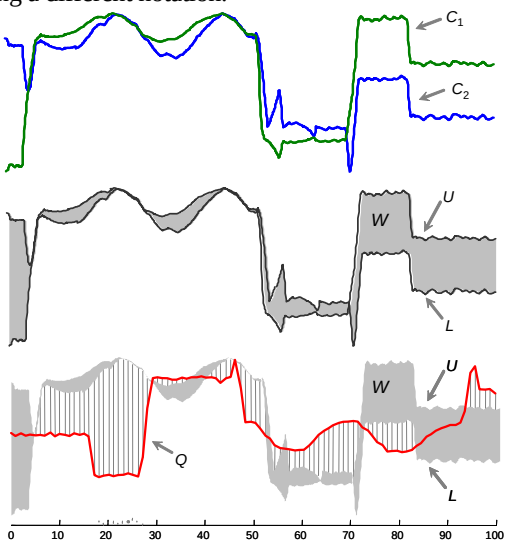

Figure 3: *Top*) Two time series C_1 and C_2 *Middle*) A time series *wedge W*, created from C_1 and C_2 *Bottom*) An illustration of LB_Keogh

In the special case where W is created from a single candidate sequence, LB_Keogh degenerates to the Euclidean distance. More importantly, we can do *early abandoning* with LB_Keogh, as shown in Table 2.

Table 2: LB_Keogh with early abandonment

```
1  Function [dist, num_steps] = EA_LB_Keogh(Q, W, r)
2  accumulator = 0
3
4  For i = 1 to length(Q)                    // Loop over time series
5    If q_i > W.U_i                          // Accumulate error contribution
6      accumulator += (c_i - W.U_i)^2
7    Elseif q_i < W.L
8      accumulator += (c_i - W.L_i)^2
9    End
10   If accumulator > r^2                    // Can we abandon?
11     disp('doing an early abandon')
12     num_steps = i
13     Return [ infinity, num_steps ]        // Terminate and return an
14   End                                     // infinite error to signal the
15 End                                       // early abandonment.
16 Return [ sqrt(accumulator), length(Q) ]   // Return true dist
```

Suppose we have two candidate sequences C_1 and C_2 of length n, and we are given a query sequence Q and asked if one (or both) of the candidate sequences are within r of the query, we naturally wish to minimize the number of steps we must perform ("step" was defined in Section 2). We are now in a position to outline two possible approaches to this problem.

- We can simply compare the two candidate sequences, C_1 and C_2 (in either order) to the query using the early abandon algorithm. We call this algorithm, *classic*.
- We can combine the two candidate sequences into a wedge, and compare the query to the wedge using LB_Keogh. If the LB_Keogh function early abandons, we are done. Otherwise, we need to individually compare the two candidate sequences, C_1 and C_2 (in either order) to the query. We call this algorithm, *wedgie*.

Let us consider the best and worst cases for each approach. For *classic* the worst case is if both candidate sequences are within r of the query, which will require $2n$ steps. In the best case, the first point in the query may be radically different to the first point in either of the candidates, allowing immediate early abandonment and giving a total cost of 2 steps. For *wedgie*, the worst case is also if both candidate sequences are within r of the query. We will waste n steps in the lower bounding test between the query and the wedge, and then $1n$ steps for each individual candidate, for a total of $3n$. However the best case, also if the first point in the query is radically different, would allow us to abandon with a total cost of 1 step.

Which of the two approaches is better depends on:

- The shape of the candidate sequences. If they are similar, this greatly favors *wedgie*.

- The shape of the query. If the query is truly similar to one (or both) of the candidate sequences, this would greatly favor *classic*.
- The matching distance r. Here the effect is non monotonic and dependent on the two factors above.

We generalize the notion of wedges by hierarchally nesting them. For example, in Figure 4 we have three sequences C_1, C_2, and C_3. A wedge is built from C_1 and C_2, and we denote it as $W_{(1,2)}$. Again, we can combine $W_{(1,2)}$ and W_3 into a single wedge by finding maximum and minimum values for each i^{th} location, from *either* wedge. More concretely:

$$U_i = \max(W_{(1,2)i}, W_{3i})$$
$$L_i = \min(W_{(1,2)i}, W_{3i})$$
$$W_{((1,2),3)} = \{U, L\}$$

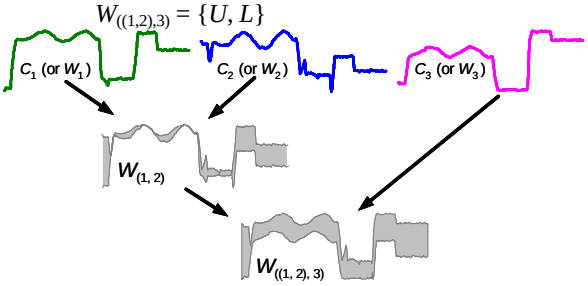

Figure 4: An example of hierarchally nested wedges

Having the generalization to hierarchal wedges, now we generalize the *wedgie* approach. Given a query Q and a wedge $W_{((1,2),3)}$, we compare the query to the wedge using LB_Keogh. If it early abandons, we are done - none of the three candidate sequences is within r of the query. Otherwise, we need to recursively compare the two child wedges, $W_{(1,2)}$ and W_3 to the query using LB_Keogh. The procedure continues until we early abandon or reach individual candidate sequence. Because our algorithm works by examining nested wedges until (if necessary) only atomic wedges are left, we call it *Atomic Wedgie*.

To demonstrate the utility of *Atomic Wedgie*, we compared it to *classic*, using the 3 time series shown in Figure 4. We measured the utility by the number of steps needed by each approach. We found that for reasonable values of r, the type of data we compared it to made little difference: *Atomic Wedgie* was almost always 3 times faster on average.

While this result is promising, we cannot expect the linear speedup to hold for all possible collections of candidate sequences. This is because the utility of a wedge is strongly correlated with its area. We can get some intuition as to why by visually comparing LB_Keogh(Q, $W_{(1,2)}$) with LB_Keogh(Q, $W_{((1,2),3)}$) in Figure 5. Note that the area of $W_{((1,2),3)}$ is much greater than that of $W_{(1,2)}$, and that this reduces the value returned by the lower bound function and thus the possibility to early abandon.

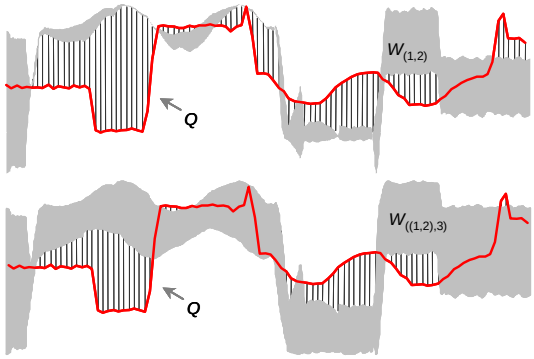

Figure 5: *Top*) Illustration of LB_Keogh(Q, $W_{(1,2)}$). *Bottom*) Illustration of LB_Keogh(Q, $W_{((1,2),3)}$)

At this point we can see that the efficiency of *Atomic Wedgie* is dependent on the candidate sequences and the data stream itself. In general, merging similar sequences into a hierarchal wedge is a good idea, but merging dissimilar sequences is a bad idea. Since the meaning of *similar/dissimilar* is relative to a data stream that by definition we cannot see in advance, it is difficult to predict if *Atomic Wedgie* will be useful.

These observations motivate a further generalization of *Atomic Wedgie*. Given a set of k sequences, we can merge them into K hierarchal wedges, where $1 \le K \le k$. This merging forms a partitioning of the data, with each sequence belonging to exactly one wedge. We use $\mathbf{W}$ to denote a set of hierarchal wedges:

$$\mathbf{W} = \{W_{set(1)}, W_{set(2)}, ..., W_{set(K)}\}, \quad 1 \le K \le k$$

where $W_{set(i)}$ is a (hierarchally nested) subset of the k candidate sequences. Note that we have

$$W_{set(i)} \cap W_{set(j)} = \varnothing \text{ if } i \neq j, \text{ and}$$
$$| W_{set(1)} \cup W_{set(2)} \cup .. \cup W_{set(K)} | = k$$

We can then compare this set of wedges against our query. Table 3 formalizes the algorithm.

Table 3: Algorithm *Atomic Wedgie*

```
1  Function [] = Atomic_Wedgie(Q, W, K, r)
2  S = {empty}                          // Initialize a stack.
3  For i = 1 to K                       // Place all the wedges into the stack.
4     enqueue(W_set(i), S)
5  End
6
7  While Not empty(S)
8     T = dequeue(S)
9     dist = EA_LB_Keogh(Q, T, r)       // This is early abandon version.
10    If isfinite(dist)                 // We did not early abandon.
11       If cardinality(T) = 1          // T was an individual sequence.
12          disp('The sequence ', T, 'is ', dist, ' units from the query')
13       Else                           // T was a wedge, find its children
14          enqueue(children(T), S)     // and push them onto the stack.
15       End
16    End
17 End
```

As we shall see in the experiments, *Atomic Wedgie* can produce impressive speedup if we make judicious

choices in the set of hierarchal wedges that make up **W.** However, the number of possible ways to arrange the hierarchal wedges is greater than K^K, and the vast majority of these arrangements will generally be worse than *classic*. So specifying a good arrangement of **W** is critical.

Note that hierarchal clustering algorithms have very similar goals to an ideal wedge-producing algorithm. Hierarchal clustering algorithms attempt to minimize the distances between objects in each subtree, while a wedge-producing algorithm attempts to minimize the area of each wedge. However the area of a wedge is simply the maximum Euclidean distance between any sequences contained therein (i.e Newton-Cotes rule from elementary calculus). This motivates us to derive wedge sets based on the result of a hierarchal clustering algorithm. Figure 7 shows wedge sets **W**, of every size from 1 to 5, derived from the dendrogram in Figure 6.

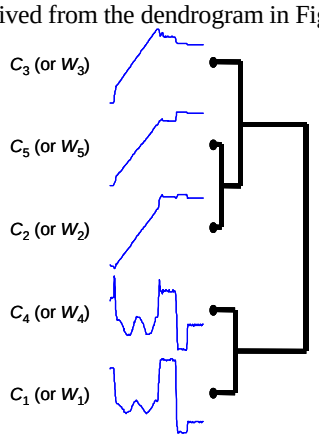

Figure 6: A dendrogram of five sequences

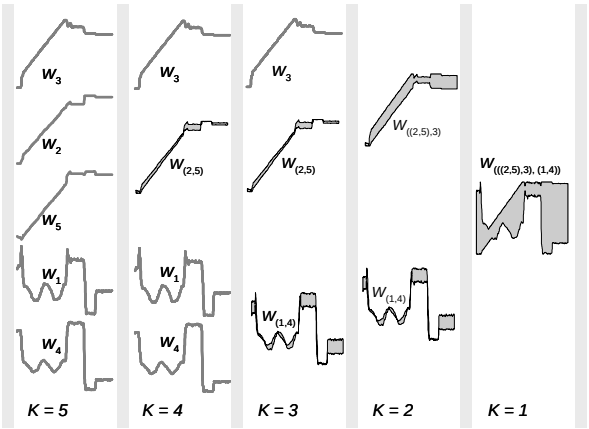

Figure 7: Wedge sets W, of size 1 to 5, derived from the dendrogram in Figure 6

Given that the clustering algorithm produces *k* wedge sets, all we need to do is to choose the best one. We could attempt to do this by eye, for example in Figure 7 it is clear that any sequence that early abandons on W_3, will almost certainly also early abandon on both W_2 and W_5; similar remarks apply to W_1 and W_4. At the other extreme, the wedge at $K = 1$ is so "fat" that it is very likely to have poor pruning power. The set **W** = $\{W_{((2,5),3)}, W_{(1,4)}\}$ is probably the best compromise. However because the set of time series might be very large, visual inspection is not scalable. More generally, we choose the wedge set based on empirical tests. We test all *k* wedge sets on a sample of data that we believe to be representative of future data and choose the most efficient one.

3.1 A Bound on *Atomic Wedgie*

As it stands, *Atomic Wedgie* is an efficient tool for comparing a set of time series to a large batch dataset. However so far it does not make any contribution to the problem of streaming time series. The reason is that while it is efficient on *average*, streaming algorithms are limited by their *worst* case. The worst case is easy to see. Imagine that we might have chosen **W** with size of $K = 1$, and the query Q is within r of *all* k candidates. This means that we would do EA_LB_Keogh $2k$-1 times, without early abandoning. This is actually worse than *classic*, which only requires k complete invocations of EA_LB_Keogh in the worst case.

Fortunately, we can generally derive much tighter bounds for the worst case of *Atomic Wedgie*. The intuition is that for realistic values of r and realistic sets of time series, no query Q will be within r of all members of the pattern set. For example, consider the five time series in Figure 7. Clearly any sequence Q that is close to C_1 or C_4 cannot also be close to C_3, C_5 or C_2. A more formal explanation is given below.

Given two wedges $W_1 = \{U_1, L_1\}$ and $W_2 = \{U_2, L_2\}$, we define the distance between them as:

$$d(W_1, W_2) \equiv \sqrt{\sum_{i=1}^{n} \begin{cases} (L_{1i} - U_{2i})^2 & \text{if } L_{1i} > U_{2i} \\ (L_{2i} - U_{1i})^2 & \text{if } L_{2i} > U_{1i} \\ 0 & \text{otherwise} \end{cases}}$$

An example is given in Figure 8.

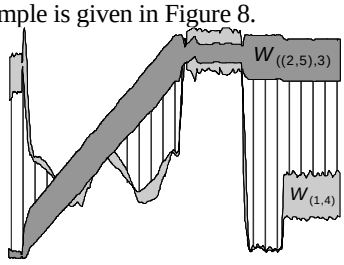

Figure 8: Illustration of distance between wedges

We claim that, for any two time series (one from W_1 and one from W_2), the distance between them is at least $d(W_1, W_2)$. This is easy to see. For unoverlapped portion, we sum up the distance between the closest edges of the two wedges. Recall that wedge forms the smallest possible bounding envelope that encloses all its members, which means any pair of the time series from

W_1 and W_2 cannot be closer than the closest edge pair. For overlapped area, we count the distance as zero.

Now we are ready to give the upper bound of the cost of *Atomic Wedgie*. Say for the five time series shown in Figure 7, we start from the biggest wedge $W_{(((2,5),3),(1,4))}$ and fail, then we need to test on wedge $W_{(1,4)}$ and $W_{((2,5),3)}$, respectively. If $d(W_{(1,4)}, W_{((2,5),3)}) \geq 2 \cdot r$, it is guaranteed that we would not fail both on $W_{(1,4)}$ and $W_{((2,5),3)}$. Without loss of generality, we can assume that the test fails on $W_{(1,4)}$, which means $d(Q, W_{(1,4)}) < r$. According to triangle inequality,

$$\begin{aligned} d(Q, W_{((2,5),3)}) &> d(W_{((2,5),3)}, W_{(1,4)}) - d(Q, W_{(1,4)}) \\ &> 2 \cdot r - r \\ &> r \end{aligned}$$

which means testing on $W_{((2,5),3)}$ would prune the query and we can safely stop that branch there. However, if $d(W_{(1,4)}, W_{((2,5),3)}) < 2 \cdot r$, we have to test on both wedges recursively. We illustrate the computation of the cost upper bound in Table 4.

Table 4: Compute cost upper bound of *Atomic Wedgie*

```
1  Function upperBound = Compute_ub( W, r, len )
2  cost = len                          // Current test fails
3
4  If W is atomic                      // Contains no child wedges
5    Return cost
6  Else                                // Contains child wedges
7   [W1, W2] = Get_children(W)         // Get child wedges
8   If d(W1, W2) >= 2r                 // Could not fail on both wedges
9     Return cost + max{Compute_ub(W1, r, len), Compute_ub(W2, r, len)}
10  Else                               // May fail on both wedges
11    Return cost + Compute_ub(W1, r, len) + Compute_ub(W2, r, len)}
12  End
13 End
```

Recall that r is the threshold to define the similarity between subsequence and interesting patterns, so usually r is a relatively small value, which increases the possibility for two wedges having distance larger than $2 \cdot r$. As the result, the *Atomic Wedgie* algorithm can skip a lot of computations based on the proof we gave above. As we shall see in Section 4, with reasonable value of r, in the worst case *Atomic Wedgie* is still three to four times faster than the brute force approach.

3.2 A Final Optimization

There is one simple additional optimization that we can do to speed up *Atomic Wedgie*. Recall that in both Table 1 and Table 2 when we explained early abandoning we assumed that the distance calculation proceeded from left to right (cf. Figure 2). When comparing individual sequences we have no reason to suppose that left to right, right to left, or any other of the $w!$ possible orders in which we accumulate the error will allow an earlier abandonment. However this order *can* make a huge difference. It is simply that we cannot know this order in advance.

The situation is somewhat different when we are comparing a query to a wedge. In this case we do have an a priori reason to suspect that some orders are better than others. Consider the wedge shown in Figure 3. The left side of this wedge is so "fat", that most query sequences will pass through this part of the wedge, thus contributing nothing to the accumulated error. In contrast, consider the section of the wedge from 10 to 50. Here the wedge is very thin, and there is a much greater chance that we can accumulate error here.

The optimization then, is to simply modify Table 2 such that the loop variable is sorted in ascending order by the value of U_i - L_i (the local thickness of the wedge). This sorting takes O($w$log(w)) for each wedge (w is the length of the wedge), but it only needs to be done once. As we shall see, this simple optimization speeds up the calculations by an order of magnitude.

4. Experimental Results

In this section, we test our proposed approach with a comprehensive set of experiments. For each experiment, we compared *Atomic Wedgie* to three other approaches, *brute force*, *classic*, and *Atomic Wedgie Random* (*AWR*). Among them, *brute force* is the approach that compares each pattern to the query without early abandoning. *AWR* is similar to *Atomic Wedgie*, except that instead of using the wedge sets resulted from the hierarchical clustering algorithm (in this paper we use complete linkage clustering), we randomly merge time series. This modification is essentially a lesion study which helps us separate the effectiveness of *Atomic Wedgie* from our particular wedge merging strategy.

Throughout this work we have referred to "*reasonable* values of r". As the reader may already appreciate, the value of r can make a huge difference to the utility of our work. We want to know the performance of *Atomic Wedgie* at the values of r which we are likely to encounter in the real world. The two domain experts (cardiology and entomology) that are co-authors of this work independently suggested the following policy.

Assume that our dataset contains typical examples of the patterns of interest. A logical value for r would be the average distance from a pattern to its nearest neighbor. The intuition is that if the patterns seen before tended to be about r apart, then a future query Q that is actually a member of this class will probably also be within r of one (or more) pattern(s) of our dataset.

Note that all datasets used in this work are freely available at the following URL [8].

4.1 ECG Dataset

We examined one dataset from the MIT-BIH Arrhythmia Database [5], which contains half an hour's excerpts of two-channel ambulatory ECG recordings. The recordings were digitized at 360 samples per second per channel with 11-bit resolution over a 10 mV range. We use signals from one channel as our batch time series, which has 650,000 data points in total. Our pattern set consists of 200 time series, each of length 40. According to the cardiologists' annotation, they are representative patterns of *left bundle branch block beat*, *right bundle branch block beat*, *atrial premature beat*, and *ventricular escape beat*. For *Atomic Wedgie* and *AWR*, we tested all 200 wedge sets on first 2,000 data points, and chose the most efficient one to use.

Using the policy described above, we set *r* to 0.5 and illustrate the number of steps needed by each approach in Figure 9 (the precise numbers are recorded in the Appendix). The result shows that our approach is faster than *brute force* by three orders of magnitude, and faster than *classic* by two orders of magnitude. Note that *AWR* does not achieve the same speedup as *Atomic Wedgie*, suggesting that our wedge building algorithm is effective. We also computed the upper bound of the cost of *Atomic Wedgie* for ECG dataset, which is 2,120 steps. This is about 4 times faster than the *brute force* approach, which in the worst case will need to compare the subsequence to all patterns, resulting in 200 * 40 = 8,000 steps.

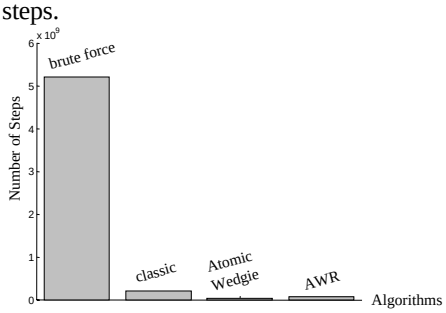

Figure 9: Speedup for ECG dataset

4.2 Stock Dataset

Our second experiment considered the problem of finding interesting patterns in a stock dataset. We tested on a stock time series with 2,119,415 data points. There are 337 time series of length 128 in the pattern set. They represent three types of patterns which where annotated by a technical analyst, with 140 for *head and shoulders*, 127 for *reverse head and shoulders*, and 70 for *cup and handle*. Again, for *Atomic Wedgie* and *AWR*, we tested all 337 wedge sets on first 2,000 data points, and used the most efficient one for the rest of the data.

This time *r* is set to 4.3, and the number of steps needed by each approach is illustrated in Figure 10. The result again indicates impressive speedup of *Atomic Wedgie*. *Atomic Wedgie* is faster than *brute force* by two orders of magnitude, and faster than *classic* by one order of magnitude. For stock dataset, the cost upper bound of *Atomic Wedgie* is 18,048, which is about one third to that of the *brute force* approach (337*128 = 43,136).

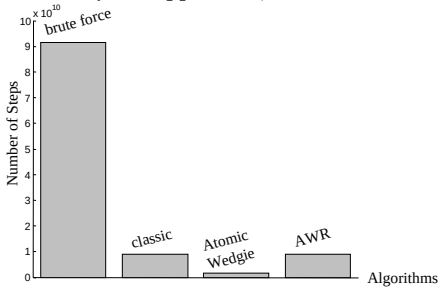

Figure 10: Speedup for Stock dataset

4.3 Audio Dataset

In this experiment, we tested a one-hour wave file to monitor the occurrences of some harmful mosquito species. The wave file, at sample rate 11,025HZ, was converted to a 46,143,488 data points' time series. Here we used a sliding window of size 11,025 data points (1 second's sound) and slid it by 5,512 points (0.5 second) each time. Because insect detection is based on the frequency of wing beat, we applied Fourier transformation on each subsequence and then resampled the time series we got (note that the FFT was performed by specialized hardware directly on the sensor [9] and that the time taken for this is inconsequential compared to the algorithms considered here). We have 68 candidate time series of length 101, which are obtained through the same procedure (FFT plus resampling) from three different species of harmful mosquitoes, *Culex quinquefasciatus*, *Aedes aegypti*, and *Culiseta spp*. For *Atomic Wedgie* and *AWR*, we used first three minutes' sound to decide which wedge set to use.

The number of steps needed by each approach is shown in Figure 11. Here the parameter *r* equals to 4.14. *Atomic Wedgie* is faster than *brute force* by two orders of magnitude. Note that here *AWR* is worse than *classic*. For audio dataset, the cost upper bound of *Atomic Wedgie* is 2,929, which is about one third to that of the *brute force* approach (68*101 = 6,868).

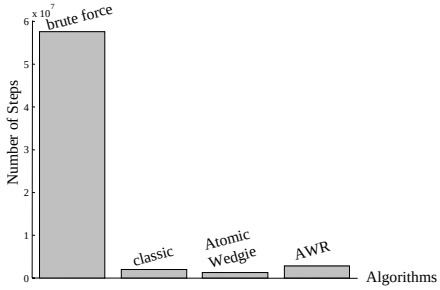

Figure 11: Speedup for Audio dataset

4.4 Speedup by Sorting

In Section 3.2, we described an optimization on *Atomic Wedgie*, where the distance calculation proceeds in the ascending order of the local thickness of the wedge. To demonstrate the effect of this optimization, we compare the wedge in Figure 3 to 64,000 random walk time series, and recorded the total number of steps required with and without sorting. The result, shown in Figure 12, demonstrates that the simple optimization can speed up the calculations by an order of magnitude.

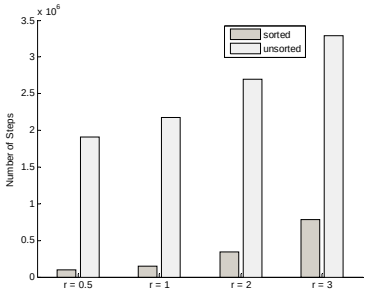

Figure 12: Speedup by sorting

5. Conclusions

In this paper, we introduce the problem of time series filtering: fast, on-the-fly subsequence matching of streaming time series to a set of predefined patterns. Given the continuously arriving data and the large number of patterns we wish to support, a brute force strategy of comparing each pattern with every time series subsequence does not scale well. We propose a novel filtering approach, which exploits commonality among patterns by merging similar patterns into a wedge such that they can be compared to the time subsequence together. The resulting shared processing provides tremendous improvements in performance.

The approach currently chooses the wedge set based on the empirical test. For data changing over time (i.e concept drift), dynamically choosing the wedge set will be more useful. We leave such considerations for future work.

Acknowledgments: We gratefully acknowledge Dr. Reginald Coler for technical assistance with the insect audio dataset.

References

[1] Carson, M. P., Fisher, A. J., & Scorza W. E. (2002). Atrial Fibrillation in Pregnancy Associated With Oral Terbutaline. *Obstet. Gynecol*, 100(5): pp. 1096-1097, 2002.

[2] Cole R., Gottlieb L., & Lewenstein M. (2004). Dictionary Matching and Indexing with Errors and Don't Cares. *In Proceedings of the 36th annual ACM Symposium on Theory of Computing*, pp. 91-100, Chicago, Illinois, June 13-15, 2004.

[3] Diao, Y., Altinel, M., Franklin, M. J., Zhang, H., & Fischer, P. (2003). Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Transactions on Database Systems*, 28(4): pp. 467-516, 2003.

[4] Gao L. & Wang X. (2002). Continually Evaluating Similarity-based Pattern Queries on a Streaming Time Series. *In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pp. 370-381, Madison, Wisconsin, June 3-6, 2002.

[5] Goldberger A., Amaral L., Glass L., Hausdorff J., Ivanov P., Mark R., Mietus J., Moody G., Peng C., & He S. (2000). PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals. *Circulation* 101(23): pp. 215-220, 2000.

[6] http://www.whitehouse.gov/omb/budget/fy2005/agriculture.html (US Dept of Agriculture, Office of Budget and Management Website)

[7] Keogh, E. & Kasetty, S. (2002). On the Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration. *In Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 102-111, Edmonton, Alberta, Canada, July 23-26, 2002.

[8] Keogh, E. http://www.cs.ucr.edu/~wli/ICDM05/

[9] Kuo, J.-C., Wen, C.-H. & Wu, A.-Y. (2003). Implementation of a Programmable 64~2048-point FFT/IFFT Processor for OFDM-Based Communication Systems. *In Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 121-124, Bangkok, Thailand, May 25-28, 2003.

[10] Li Q., López I., & Moon B. (2004). Skyline Index for Time Series Data. *IEEE Transactions on Knowledge and Data Engineering*. 16(6): pp. 669-684, 2004.

[11] Moore A. & Miller, R. H. (2002). Automated Identification of Optically Sensed Aphid (Homoptera: Aphidae) Wingbeat Waveforms. *Annals of the Entomological Society of America*, 95(1): pp. 1-8, 2002.

[12] Pheromone Pest Management Moritor Technologies, Inc. http://www.moritor.com/web/

Appendix

Table A: Number of steps for each algorithm

Algorithm	Number of Steps		
	ECG	Stock	Audio
brute force	5,199,688,000	91,417,607,168	57,485,160
classic	210,190,006	13,028,000,000	1,844,997
Atomic Wedgie	8,853,008	3,204,100,000	1,144,778
AWR	29,480,264	10,064,000,000	2,655,816

This table contains the numbers graphed in Figure 9, Figure 10, and Figure 11.

Table B: Number of steps w/ and w/o sorting

	r = 0.5	r = 1	r = 2	r = 3
Sorted	95,025	151,723	345,226	778,367
Unsorted	1,906,244	2,174,994	2,699,885	3,286,213

This table contains the numbers graphed in Figure 12.