Online Amnesic Approximation of Streaming Time Series

Themistoklis Palpanas
University of California, Riverside
themis@cs.ucr.edu

Michail Vlachos
University of California, Riverside
mvlachos@cs.ucr.edu

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

Dimitrios Gunopulos
University of California, Riverside
dg@cs.ucr.edu

Wagner Truppel
University of California, Riverside
wagner@cs.ucr.edu

Abstract

The past decade has seen a wealth of research on time series representations, because the manipulation, storage, and indexing of large volumes of raw time series data is impractical. The vast majority of research has concentrated on representations that are calculated in batch mode and represent each value with approximately equal fidelity. However, the increasing deployment of mobile devices and real time sensors has brought home the need for representations that can be incrementally updated, and can approximate the data with fidelity proportional to its age. The latter property allows us to answer queries about the recent past with greater precision, since in many domains recent information is more useful than older information. We call such representations amnesic.

While there has been previous work on amnesic representations, the class of amnesic functions possible was dictated by the representation itself. In this work, we introduce a novel representation of time series that can represent arbitrary, user-specified amnesic functions. For example, a meteorologist may decide that data that is twice as old can tolerate twice as much error, and thus, specify a linear amnesic function. In contrast, an econometrist might opt for an exponential amnesic function. We propose online algorithms for our representation, and discuss their properties. Finally, we perform an extensive empirical evaluation on 40 datasets, and show that our approach can efficiently maintain a high quality amnesic approximation.

1 Introduction

Time series are one of the most frequently encountered forms of data. Many applications in diverse domains, produce voluminous amounts of time series [29, 26]. Examples of such applications exist in finance [29], medicine [21], meteorology, oceanography [26], manufacturing, network management [17], sensor networks [11], and other domains.

The sheer number and size of the time series we need to manipulate in many of the real-world applications mentioned above dictates the need for a more compact representation of time series than the raw data itself. A plethora of representations have been proposed for time series approximation [19].

The problem of approximating time series becomes more interesting and challenging in the context of streaming time series, where data values are continuously generated, potentially forever. In this situation, we cannot apply approximation techniques that require knowledge of the entire series, such as singular value decomposition [6] and most symbolic approaches [2]. Furthermore, all current time series representations treat every point of the time series equally. This means that, when computing the approximation, the time position of a point does not make a difference in the fidelity of its approximation. This may be desirable for some applications, such as archiving, however, there exist many real world situations where we would like to take into account the time dimension in the approximation of the time series. The intuition behind this requirement may be stated as follows. While we are willing to accept some margin of error in the approximation, we would like the most recent data to have low error, and we would be more forgiving of error in older data. We call this kind of time series approximation *amnesic*, since the fidelity of approximation decreases with time, and it therefore requires less memory for the events further in the past.

The potential utility of such a representation has been documented in many domains. Consider the following motivating examples.

- The Environmental Observation and Forecasting System [26] is a large-scale distributed system designed to monitor, model, and forecast wide-area physical processes such as river systems. They note that in their current model, the loss of a repeater station results in the loss of real time information. Allowing the stations to record some data to a buffer can mitigate this problem. However, since the station does not know how long it will be offline and has a finite buffer, amnesic approximation is the only logical way to record the data.

- NASA is developing robots to be used in an urban setting [15]. Typical applications include search and rescue, and inspection of hazardous environments. In many situations, information about the path traversed must be known if the robot is to back up to a more promising avenue of exploration after reaching a dead end. Power
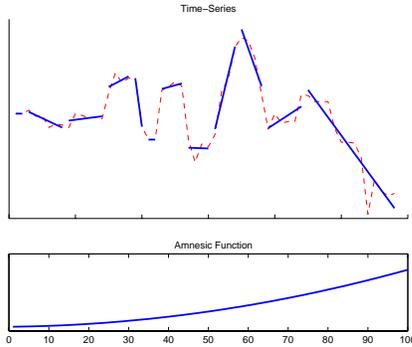
Figure 1. Depiction of an amnesic approximation, using the piecewise linear approximation technique.

and size constraints prohibit the robot from storing all the data with perfect fidelity, so the utility of an amnesic approximation has been noted for this domain [15].

- Hussain et al. [17] propose a framework for classifying denial of service attacks using (among other things) temporal information. They explicitly note that the utility of information is in proportion to its age.

Although this work suggests that the usefulness of data can diminish with age, we note that the rate at which its utility decays depends on the application. The function that determines the amount of error we can tolerate at each point in the time series is called an *amnesic* function. Ideally, we would like to allow arbitrary amnesic functions, so that we can match the requirements of a wide variety of applications. For example, a meteorologist may decide that data that is twice as old can tolerate twice as much error, and thus, specify a linear amnesic function. In contrast, an econometrist using classic models might well specify an exponential amnesic function. Figure 1 depicts an amnesic approximation of a static time series, and the amnesic function that was used. Note that as we get to older points (to the right) the approximation gets coarser.

In this paper, we describe a framework for online amnesic approximation of streaming time series. We characterize the different classes of amnesic functions, and present corresponding algorithms for performing amnesic approximation. We study two variations of the problem. First, the case when we are interested in approximating the entire time series seen so far. We refer to this case as the *unrestricted window*. Second, the *sliding window* case, where at any point in time, we are only interested in a fixed number of the last values of the time series. In Figure 2 we show how the approximation of a time series changes as a function of time, for five different timestamps. In this example, we use an unrestricted window to approximate the Space Shuttle STS-57 dataset, using piecewise linear approximation with ten linear segments. The time progresses form right to left (i.e., the most recent point is the left-most point). We observe that the approximation of the most recent points always remains accurate, while it gracefully degrades at each time step for the older points.

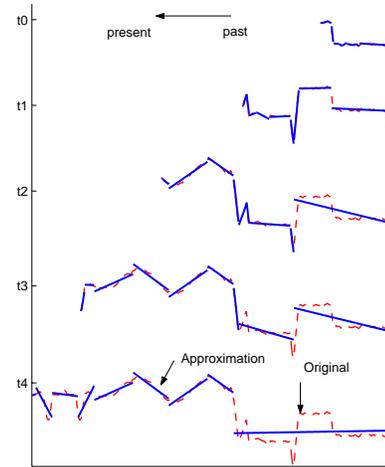While some recent work [8, 5] has proposed tools and tech-

Figure 2. Example of online amnesic approximation.

niques for computing special cases of amnesic approximations of time series, as we discuss in Section 6, these solutions are specific and rather restrictive in the variety of applications they can accommodate. In particular, the types of amnesic functions they can use are dictated by the *representation* of the time series. In contrast, our framework is general and able to operate with a wide class of amnesic functions, which are defined by the *user*.

Our contributions can be summarized as follows.

- We introduce the notion of general amnesic functions. We present a taxonomy of these functions, discuss their properties, and describe how they affect the solution of the problem of online amnesic approximation.

- We formulate the above problem as an optimization problem, where we wish to minimize the reconstruction error given the available amount of memory for the approximation. We study two important variations of it, namely, the unrestricted and the sliding window cases.

- We propose efficient algorithms for solving the above optimization problems. The time complexity of the algorithms we propose is independent of the size of the time series. The time to process each new point is essentially constant (logarithmic on the number of segments used in the approximation). These are the first algorithms proposed for solving the general case of the problem.

- We present an extensive experimental evaluation of our techniques, using more than 40 synthetic and real datasets. The experiments show the applicability of our approach, and the quality of solutions of our algorithms.

The rest of the paper is organized as follows. In Section 2 we give the necessary background. In Section 3 we introduce some new terminology and formally define the problems we study. The algorithms we propose are presented in Section 4, and Section 5 discusses the experimental evaluation. Section 6 reviews related work, and Section 7 concludes the paper.

2 Time Series Approximation

A time series, $T[i]$, is a series of data points, each one arriving at a distinct time instance t_i. $T[i..j]$ defines a range of data points. When the total number of data points in the time series, N, is known in advance, we call the time series *static*, and we say that is has length N. When data points are arriving continuously, in a streaming fashion, the value of N represents the number of data points seen in the time series so far, and we call the time series *streaming*. The focus of our work is on streaming time series.

Several techniques have been proposed in the literature for the approximation of time series, including *Discrete Fourier Transform (DFT)* [25, 10], *Discrete Cosine Transform (DCT)*, *Piecewise Aggregate Approximation (PAA)* [28], *Discrete Wavelet Transform (DWT)* [23, 7], *Adaptive Piecewise Constant Approximation (APCA)* [6, 22], *Piecewise Linear Approximation (PLA)* [20], *Piecewise Quadratic Approximation (PQA)*, and others. Before we consider which of these representations is best suited for the task at hand, it is natural to ask which is best, simply in terms of reconstruction accuracy. In order to answer this question, we experimentally compare the above approaches using many real-world datasets. We conducted such an experiment on 40 diverse time series from the UCR Time Series Data Mining Archive [1].

For our experiment, we randomly extracted a subsequence of length 512 from each time series, and approximated it with each of the representations under consideration, using a 16 to 1 compression ratio. This was a fair comparison, using the same amount of memory for each representation, and applying all possible optimizations for all representations. However, for the piecewise polynomial approaches, the optimal representation requires quadratic time to produce, and we used a well known near linear-time algorithm instead [18, 20]. We measured the quality of the approximation using the root mean squared error. We repeated this procedure 100 times, averaged the results, and normalized the performance of each representation by dividing by the best performing approach. Finally we averaged all 40 scores as shown in Table 1.

The results may appear surprising, because there is little difference between all the approaches. In fact, similar results have been documented elsewhere as well [19, 6]. The overall conclusion from this experiment is the following. If we want to choose a representation for the task of approximating time series, then we should not choose the representation based on approximation fidelity, but rather on other features.

One important feature may be the visual appearance of the representation, since in many application domains we are interested in visualizing the time series. In Figure 3 we visually compare all representations[1] on an important and familiar example, an electrocardiogram. We show just one example for brevity. For a fair comparison, we use an equal number of bytes for each approach (as discussed above). Although the
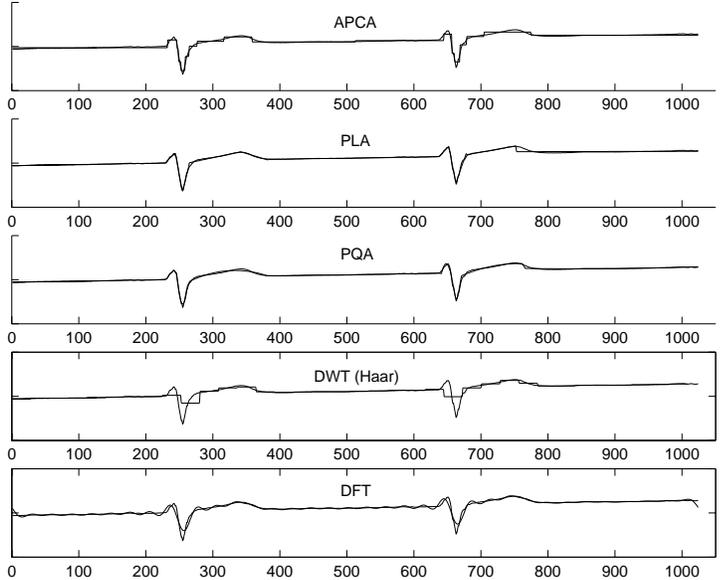
---

[1]We omit the result for *DCT* since it is indistinguishable from *DFT*.

Figure 3. Comparison of (top to bottom) *APCA*, *PLA*, *PQA*, *DWT* (Haar), and *DFT* ("koski_ecg.dat" dataset [1]).

quality of visualization for a representation is subjective, we feel that the *PLA* approach (second from top) is the best of the approaches.

When considering the alternative representations in the context of amnesic approximation, it is not obvious how some of them can accommodate the requirements of this new environment. The *DWT* representation is intrinsically coupled with approximating sequences whose length is a power of two, which severely restricts the choices of amnesic functions. Using wavelets with sequences that have other lengths requires ad-hoc measures that reduce the fidelity of the approximation, and increase the complexity of the implementation. While *DFT* has been successfully adapted to incremental computation [29], it is not clear that it can be adapted to perform amnesic approximation, since each *DFT* coefficient corresponds to a global contribution to the entire time series. The same is true for *DCT* as well.

In contrast to the above, the piecewise polynomial methods offer several desirable properties for the task at hand. Much is already known about their incremental calculation, and because each segment is independent of each other, we can reduce the fidelity of "older" segments simply by merging them with their neighbors, without affecting "newer" segments. The only question remaining is which piecewise polynomial technique to use. We decide on *PLA* for the following reasons. Piecewise linear approximations are already widely used and accepted in the medical and financial domains [16, 21]. There are many useful distance measures defined on *PLA*, including weighed measures [20], time warping [27], Markov model based measures [12] and lower bounding approximations to the Euclidean distance.

DFT	DCT	PAA	DWT (Haar)	DWT (Daub12)	APCA	PLA	PQA
0.951	0.923	0.948	0.948	0.902	0.893	0.940	0.927

Table 1. Comparison among various techniques for time series approximation.

2.1 Properties of *PLA* Approximation

In *PLA*, we approximate the data points in a time series using a number of linear segments whose ends need not be contiguous [20]. The *PLA* approximation scheme has some desirable properties that allow incremental computation of the solution. These properties are necessary in order for the algorithm to be able to operate efficiently on large datasets. In the following paragraphs we present these properties in the form of theorems, and we discuss their applications in Section 4.

Assume we have N data points of a time series, $T[i]$, $1 \leq i \leq N$, and we use them to fit two line segments (using least squares). Let the first line, s_1, approximate points 1 to n, $n < N$, and the second line, s_2, approximate points $n+1$ to N. In addition, suppose we use a single line segment to approximate all the points 1 to N, call it $s_{1,2}$. The above three lines are depicted in the top graph of Figure 4. Related to these three lines are the errors $E(s_1)$, $E(s_2)$, and $E(s_{1,2})$. The error of a segment s is computed according to the formula $E(s) = \sum_{j \in s} (T[j] - s[j])^2$, where j ranges over all the points in segment s, $T[j]$ is the value of point j in the time series, and $s[j]$ is the estimate for point j given by segment s.

Now imagine that we keep s_1 and s_2, and throw away the original N points, and that we want to use a single line segment to approximate all the original points. The construction of this new line, $\overline{s_{1,2}}$, can be based only on the information in s_1 and s_2, and we prove that $\overline{s_{1,2}}$ is the same as $s_{1,2}$. Since we no longer have the original points, we assume that all N points lie *on* line segments s_1 and s_2, and we build $\overline{s_{1,2}}$ based on this assumption. This situation is depicted in the bottom graph of Figure 4. The residual error of this new line is $E(\overline{s_{1,2}})$. Unlike the previous cases, this is the error between the points on line $\overline{s_{1,2}}$ and the points on lines s_1 and s_2. (Remember that line $\overline{s_{1,2}}$ is not calculated based on the original points of the time series.) It turns out that we can also calculate $E(s_{1,2})$ without the need to refer to the original N points.
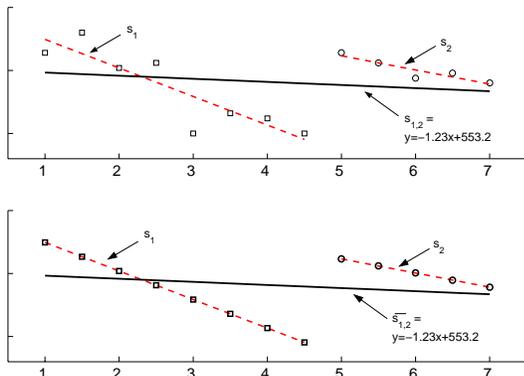

Figure 4. Combining two regression lines.

We can now prove the following theorems regarding the process of merging two line segments into one.

Theorem 1 [Computing the New Line Segment.] *The line segment $\overline{s_{1,2}}$, built from the two line segments s_1 and s_2, is the same as the line segment $s_{1,2}$, built from the original points of the time series[2]. That is, $s_{1,2} = \overline{s_{1,2}}$.*

Theorem 2 [Computing the New Error.] *The error of the line segment approximating all the original data points can be computed as the sum of the errors of the two individual line segments, and the error between those two line segments and the line calculated based on those two. That is, $E(s_{1,2}) = E(s_1) + E(s_2) + E(\overline{s_{1,2}})$.*

Another interesting property of *PLA* is that for the computation of the error $E(\overline{s_{1,2}})$ we do not need to process individually all the points corresponding to line segment $\overline{s_{1,2}}$. We can instead avoid the linear complexity of this procedure and compute the value of $E(\overline{s_{1,2}})$ in constant time, according to the following lemma.

Lemma 1 [Computing the Error Between Two Segments.] *The error, $E(\overline{s_{1,2}})$, of a line segment, $\overline{s_{1,2}}$, which was constructed from two line segments, s_1 and s_2, can be computed with a closed form formula[3] in time $O(1)$, regardless of the length of the line segments.*

The intuition behind Lemma 1 is that we can compute the error between two lines as a summation over the corresponding discrete points, by taking into account the offsets and slopes of the two lines. The above formulation leads to a closed form formula for the computation of the error.

The properties of *PLA*, presented in Theorems 1 and 2 and Lemma 1, form the basis for the design of the online algorithms we propose. These properties enable our algorithms to merge two line segments, and calculate exactly the resulting line segment along with its residual error in constant time.

3 Problem Formulation

In the following paragraphs we establish some additional terminology necessary for the rest of the paper. Then, we formally define the problems that we address with this work.

3.1 Amnesic Functions

As we mentioned earlier, we need a way to specify for each point in time the amount of error allowed for the approximation of the time series. In order to achieve this goal, we use the

---

[2]A similar result has also appeared elsewhere [8].

[3]The formula requires the introduction of additional notation, and we omit it due to lack of space.

amnesic function $A(x)$, which returns the acceptable approximation error for point $x = t_N - t_i$, where t_N is the current time, and t_i is the time that point $T[i]$ arrived. The time t_N refers to the time when the last data point arrived, and corresponds to position $x = 0$ of the amnesic function. Note that the function $A(x)$ is only defined for $x \geq 0$, since $t_i \leq t_N$.

A key property that an amnesic function has to satisfy is the *monotonicity* property.

Definition 1 [Monotonic Amnesic Functions.] *An amnesic function, $A(x)$, is called monotonic if and only if $A(x) \leq A(x+1)$, for every value of x in its domain.*

The approximation of a time series is a lossy compression technique, which by definition is irreversible. Thus, the monotonicity property poses a natural restriction in our setting. It ensures that if at time t we can tolerate some error $E^t(T[i])$ in the approximation of point $T[i]$, then we will not request an approximation of the same point $T[i]$ with error $E^{t'}(T[i]) < E^t(T[i])$, at any time $t' > t$.

We now define a taxonomy of amnesic functions (refer to Figure 5). As we discuss in the next section, each class in the taxonomy has its own special characteristics, which have to be taken into account when designing an efficient algorithm for the amnesic approximation of time series.

Piecewise Constant: The *piecewise constant function* is the simplest form that an amnesic function can assume (with the exception of the constant function, which is a trivial case, and we do not discuss it here). It has the following general form.

$$A(x) = \begin{cases} c_1 & , 0 \leq x < d_1; \\ \dots & \\ c_L & , d_{L-1} \leq x, \end{cases}$$

where $c_1, \dots, c_L$ are constants, such that $c_1 < \dots < c_L$. We refer to each step of the function as a *section*, to distinguish it from the segments used in the approximation.

Linear: A *linear function* has the general form: $A(x) = \alpha x + \beta, \alpha, \beta > 0$.

Piecewise Linear: The general form of a *piecewise linear function* with L sections is as follows.

$$A(x) = \begin{cases} \alpha_1 x + \beta_1 & , 0 \leq x < d_1; \\ \dots & \\ \alpha_L x + \beta_L & , d_{L-1} \leq x, \end{cases}$$

where $\alpha_j \in [0, \pi/2)$, $1 \leq j \leq L$, $\beta_1 > 0$, and $\beta_2 = \alpha_1 d_1 + \beta_1, \dots, \beta_L = \alpha_{L-1} d_{L-1} + \beta_{L-1}$.

Continuous: The amnesic functions of this class can take any form not subsumed by the previous classes. The only restriction is that the function is monotonic (according to Definition 1). We do not require that these functions have a closed form formula.

We also define two forms of amnesic functions, namely, the *relative*, $RA(x)$, and the *absolute*, $AA(x)$, amnesic functions.
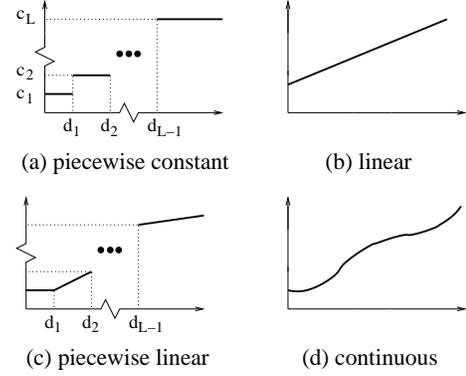

(a) piecewise constant   (b) linear

(c) piecewise linear   (d) continuous

Figure 5. The different classes of amnesic functions.

Relative: A relative amnesic function determines the relative approximation error we can tolerate for every point in the time series. When we use a relative amnesic function, we essentially weight the error of some data point by the inverse of the amnesic function corresponding to that point, so that $E(x) = E(x)/RA(x)$. For example, the relative amnesic function $RA(x) = x$, specifies that when we approximate a point that is twice as old, we will accept twice as much error.

Absolute: An absolute amnesic function specifies, for every point in the time series, the *maximum* allowable error for the approximation. The error $E(x)$, at point x, should satisfy the inequality $E(x) \leq AA(x)$.

When we have to apply an amnesic function to a segment s, we pick a single point from the segment, on which we apply the amnesic function. Nevertheless, this computation refers to the entire segment. Without loss of generality, for the rest of this paper we assume that segment s is represented by its most recent point, $T[i_s]$. Then, when we want to apply an amnesic function to s, we simply consider the point of the amnesic function corresponding to point $T[i_s]$. We can also apply more elaborate schemes. For example, we could consider taking the average value of the amnesic function corresponding to the first, middle, and last points of s.

3.2 Problems for Amnesic Approximation

Under the assumptions discussed above, we want to maintain a *PLA* model Q with K segments for a streaming time series with an unrestricted window. More formally, we define the following two problems.

Problem 1 [Unrestricted Window with Relative Amnesic (URA)] *Given the number of segments K and a relative amnesic function $RA(x)$, find an approximation Q using K segments that at each time step minimizes the error $E(T[1..N]) = \sum_{j=1}^{K} \left(E(s_j)/RA(t_N - t_{s_j}) \right)$.*

Problem 2 [Unrestricted Window with Absolute Amnesic (UAA)] *Given an absolute amnesic function $AA(x)$, construct*

a model Q with the minimum number of segments K, subject to the constraints $E(s_j) \le AA(t_N - t_{s_j})$, $1 \le j \le K$.

We are looking for online algorithms that, when a new point arrives, they update the approximation model in sub-linear time on the number of segments. Note that in the *URA* and *UAA* problems the optimization objective is different. In the *URA* problem we seek to minimize the approximation error given the memory space used by *PLA*, while in the *UAA* problem we want to minimize the space used in the approximation given the maximum error allowed.

Following the definition of the problems for the unrestricted window, we now define the corresponding problems for the case where we consider the sliding window model.

Problem 3 [Sliding window with Relative Amnesic (SRA)] *Given a sliding window of length W, the number of segments K and a relative amnesic function $RA(x)$, find an approximation Q using K segments that at each time step minimizes the error $E(T[t_{N-W+1}..t_N]) = \sum_{j=1}^{K} \left(E(s_j)/RA(t_N - t_{s_j}) \right)$.*

Problem 4 [Sliding window with Absolute Amnesic (SAA)] *Given a sliding window of length W, and an absolute amnesic function $AA(x)$, construct a model Q with the minimum number of segments K, subject to the constraints $E(s_j) \le AA(t_N - t_{s_j})$, $1 \le j \le K$.*

4 Algorithms for Amnesic Approximation

We now describe algorithms for the *URA* and *SRA* problems. In the experimental evaluation we show that our algorithms perform very close to optimal. At the end of the section, we briefly discuss solutions for *UAA* and *SAA*.

4.1 Unrestricted Window with Relative Amnesic

4.1.1 Optimal Solution

The optimal solution for the *URA* problem can be obtained using dynamic programming [4]. Note that in order to get the optimal solution in a streaming environment, we have to run the dynamic programming algorithm every time that a new data point arrives. The reason is that we cannot reuse the computations made during the previous step, because the amnesic function causes the approximation error of each point, and their interrelationships, to change at every time step. The time complexity for the dynamic programming algorithm is $O(N^2 K)$, which renders this approach inapplicable for the online version of the problem. Nevertheless, in the experimental section we show that our algorithms always find a solution that is very close to optimal.

4.1.2 The *GrAp-R* Algorithm

In this section we present the skeleton of our algorithm, *GrAp-R*, for solving the *URA* problem.

At each time step, the algorithm merges the consecutive pair of segments whose merge will result in the least approximation error, among all possible merges. The pair of segments that should be merged, s_m and s_{m+1}, is given by the heap structure H. We merge those in one segment, $s_{m,m+1}$, according to Theorems 1 and 2. Then we compute the approximation error that would result by merging the new segment with each one of its two neighbors, s_{m-1} and s_{m+2}, according to Lemma 1. We use these values for the errors to update the heap H, in order to reflect the new set of possible merges. This merge results in a spare segment, which we assign to the newly arrived point of the time series. Once again we have to compute the approximation error when merging this segment with its neighbor, and update the heap H. A high-level description of the algorithm is depicted in Figure 6.

```
1 let H be a min-priority queue on the approximation errors resulting from
     merging each pair of consecutive segments;
2 let EQ = ∅ be a time-event queue;
3 procedure GrAp-R()
4    when a new point, T[i], of the time series arrives at time t_N
5       pick the minimum element from H, and merge the corresponding
            segments, s_m and s_{m+1}, into a new segment s_{m,m+1};
6       update H with the errors of merging s_{m,m+1} with its two
            neighboring segments;
7       assign a new segment, s_{T[i]}, to the newly arrived point, T[i];
8       update H with the error of merging s_{T[i]} with its neighboring
            segment;
9       ManageEvents(EQ, t_N, s_m, s_{m+1}, s_{m,m+1});
10   return;
```

Figure 6. The skeleton of the *GrAp-R* algorithm

The *GrAp-R* algorithm also makes use of a time-event queue EQ. This structure keeps track of the way that the dependencies among the segments used for the approximation change as a result of the amnesic function. The procedure that manages these dependencies is *ManageEvents()*, and we describe it in more detail in the next paragraphs.

In the following subsections we elaborate on the way the framework of the *GrAp-R* algorithm described above changes when we consider the different classes of amnesic functions. We discuss the specific details of each case, and present the time and space complexities of the solutions we propose.

4.1.3 Piecewise Constant Amnesic Functions

When the amnesic function belongs to the class of piecewise constant functions, a change to the relative ordering of the pair of segments that should be merged during the next step of the algorithm only happens when a segment crosses a discontinuity between two sections of the amnesic function.

Example 1 *Assume we have the amnesic function $RA(x) = 1, 0 \le x < 10$ and $RA(x) = 4, x \ge 10$. Let $s_{1,2}$ and $s_{3,4}$ be two pairs of segments, candidates for merging, that, at the current time, are at positions $x = 7$ and $x = 2$, and have errors $E(s_{1,2}) = 4$ and $E(s_{3,4}) = 2$, respectively (Figure 7(a)). Then, their relative errors*

are $E(s_{1,2})/RA(7) = 4$ and $E(s_{3,4})/RA(2) = 2$, which means that $s_{3,4}$ is the first candidate for merging. However, after three time instances, when $s_{1,2}$ first gets to the point $x = 10$, its error becomes $E(s_{1,2})/RA(10) = 1 < E(s_{3,4})/RA(5) = 2$ (Figure 7(b)). Thus, $s_{1,2}$ is now the candidate pair for merging.
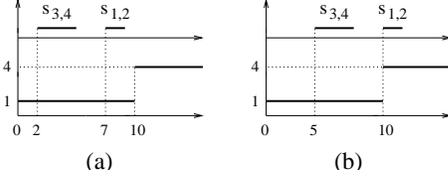

Figure 7. **Event example for piecewise constant.**

In order to keep track of these changes, we need to maintain the heap H, and, in addition, a time-event queue EQ. The heap H determines the next pair of segments that should be merged. The queue EQ flags the times at which the segments cross a discontinuity in the amnesic function (remember that during these computations we assume that each segment is represented by its most recent point). When this happens, we update the position of the segment in the heap, and we compute the next time that it will cross a discontinuity. Figure 8 shows the *ManageEvents()* procedure for the case of piecewise constant amnesic functions. The *GrAp-R* algorithm remains as discussed earlier.

---

1 proc **ManageEvents**(queue EQ, time t, segments $s_m, s_{m+1}, s_{m,m+1}$)
2   remove from EQ any events corresponding to segments s_m and s_{m+1};
3   if (next event e in EQ is scheduled for time $t < t_e \leq t+1$)
4   remove e, related to segments $s_{e,1}$ and $s_{e,2}$, from EQ;
5   update in H the position of the pair $s_{e,1}$ and $s_{e,2}$;
6   compute the new time when the pair $s_{e,1}$ and $s_{e,2}$ will cross a discontinuity;
7   insert in EQ the new event (if any);
8   insert in EQ any new dependencies identified concerning $s_{m,m+1}$;
9   return;

Figure 8. **The *ManageEvents()* procedure for piecewise constant amnesic functions.**

The following theorem states the space and time complexity of the algorithm.

Theorem 3 *The space complexity of* GrAp-R *with a piecewise constant amnesic function is $O(K)$, and the time complexity to process each new point is $O(\log K)$.*

Proof: The algorithm needs $O(K)$ space to store the K segments used in the approximation. A heap structure is used to determine the pair of segments that will be merged at each step of the algorithm. The heap requires $O(K)$ space to store the $K-1$ adjacent pairs of segments. Finally, we must keep track of the times when segments cross a discontinuity of the amnesic step function. At each point in time we only need to maintain in the time-event queue one such event for every segment. Therefore, the queue has a worst space complexity

of $O(K)$, and $O(K)$ is the overall space complexity of the algorithm as well.

At each time unit, the algorithm can pick from the heap the pair of segments to merge, and identify in the time-event queue the segments that cross a discontinuity, in $O(1)$ time. The time to merge two segments is constant, because of the Theorems 1 and 2, and Lemma 1. The time to update the heap is $O(\log K)$, and, since the size of the time-event queue is $O(K)$, the time to insert or delete an event from the queue is $O(\log K)$ (when the queue is implemented using skiplists [24], or any other equivalent data structure that offers logarithmic search times). Thus, the overall time complexity for each iteration is $O(\log K)$. □

Both the procedure *ManageEvents()* and Theorem 3 assume that only a single segment is crossing a discontinuity at each time step. The extension of the algorithm to handle multiple segments is straightforward. However, note that the above situation does not arise often, especially when more than a few segments are involved. Hence, its impact on the performance is very small. The same arguments hold for all the algorithms described in the rest of this section.

4.1.4 Linear Amnesic Functions

In the case of linear amnesic functions, each event in EQ specifies the time at which the relative ordering of the merging error of two pairs of segments changes. It turns out that, if we know the approximation error of each segment and the closed formula of the amnesic function, we can compute the times at which these changes will occur. We refer to those times as the *crosspoints*.

Example 2 *Assume we have the amnesic function $RA(x) = x + 1, x \geq 0$. Let $s_{1,2}$ and $s_{3,4}$ be two pairs of segments, candidates for merging, that were created at the current time, at positions $x = 6$ and $x = 2$, and have errors $E(s_{1,2}) = 24$ and $E(s_{3,4}) = 12$, respectively (Figure 9(a)). Then, their relative errors are $E(s_{1,2})/RA(6) = 3.4$ and $E(s_{3,4})/RA(2) = 4$, which means that $s_{1,2}$ is the first candidate for merging. However, after four time instances, when $s_{1,2}$ first gets to the point $x = 10$, its error becomes $E(s_{1,2})/RA(10) = 2.2 > E(s_{3,4})/RA(6) = 1.7$ (Figure 9(b)). Thus, $s_{3,4}$ is now the candidate pair for merging.*
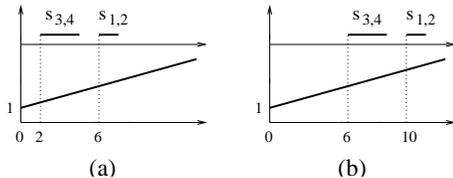

Figure 9. **Event example for piecewise constant.**

Consider the general case, where we have a linear relative amnesic function, $RA(x) = \alpha x + \beta$, and we want to compute the time when the relative ordering of segments s_1 and s_2 will change. (In fact, each one of s_1 and s_2 represents the merge of a pair of segments.) Let $E(s_1)$ and $E(s_2)$ be the approximation errors for s_1 and s_2, respectively. Finally, assume that t_{s_1} is the time when s_1 was created. This time is defined as the time when the most recent point of s_1 arrived. We define

t_{s_2} in a similar way. Then, their crosspoint, t_c, is given by the following equation.

$$\frac{E(s_1)}{\alpha \cdot (t_c - t_{s_1}) + \beta} = \frac{E(s_2)}{\alpha \cdot (t_c - t_{s_2}) + \beta}, \text{ or}$$

$$t_c = \frac{(\alpha \cdot t_{s_2} - \beta) \cdot E(s_1) - (\alpha \cdot t_{s_1} - \beta) \cdot E(s_2)}{(E(s_1) - E(s_2)) \cdot \alpha}. \quad (1)$$

We only consider the positive solutions of this equation. Note that it may be the case that their relative ordering never changes, that is, there is no positive solution. Furthermore, we do not need to compute the crosspoint of each segment with all the others. It suffices to consider only the segments stored in neighboring nodes in the heap H, and maintain these dependencies up to date as the heap changes. We now give upper bounds on the number of crosspoint computations that we have to perform as a result of changes in H. All these computations can be performed in constant time according to Equation 1.

Lemma 2 *Assume we have processed a crosspoint, and the heap has been updated. The possible crosspoints we have to compute are no more than 4.*

Lemma 3 *Assume two segments have merged, and the heap has been updated. The possible crosspoints we have to compute are no more than $\lceil \log K \rceil + 1$.*

The above lemmata guarantee that the work we have to do every time the heap changes is minimal. The *ManageEvents()* procedure for the case of linear amnesic functions is depicted in Figure 10.

```
1 proc ManageEvents(queue EQ, time t, segments s_m, s_{m+1}, s_{m,m+1})
2      remove from EQ any events corresponding to segments s_m and s_{m+1};
3      if (next event e in EQ is scheduled for time t < t_e ≤ t + 1)
4          remove e, related to segments s_{e,1} and s_{e,2}, from E;
5          swap in H the positions of s_{e,1} and s_{e,2};
6          compute crosspoints between s_{e,1} and s_{e,2} and all their
                new neighbors (i.e., parent and children nodes) in H;
7          insert in EQ events for any new crosspoints identified;
8      insert in EQ any new crosspoints identified concerning s_{m,m+1};
9      return;
```

Figure 10. The *ManageEvents()* procedure for linear amnesic functions.

The problem of keeping track of the crosspoints is reminiscent of the work in the area of *kinetic data structures* [3], where bounds are given on the number of crosspoints that need to be considered. However, the above work examines only linear motion, and does not apply to our problem. In practice, the size of EQ, $|EQ|$, remains small, and does not affect the performance of our algorithms.

The complexity of the algorithm is as follows.

Theorem 4 *The space complexity of* GrAp-R *with a linear amnesic function is $O(K + |EQ|)$, and the time complexity to process each new point is $O(\log K + \log |EQ|)$.*

Proof: The algorithm requires $O(K)$ space to store the K segments and the heap, and $O(|EQ|)$ space for the time-event queue.

At each iteration, the time to find the pair of segments to merge, and the segment that has reached a crosspoint, is $O(1)$. We need $O(\log K)$ time to update the heap after those changes, and Lemmata 3 and 2 state that there is only a small constant number of computations that we have to perform. We also need to update the queue, which takes $O(\log |EQ|)$ time. Therefore, the overall time complexity for each iteration is $O(\log K + \log |EQ|)$. $\square$

4.1.5 Piecewise Linear Amnesic Functions

Assume that the amnesic function is comprised of L sections. Then, we treat each section separately, as in the case of linear amnesic functions discussed above. We maintain L heaps, one for each section, and a single time event queue. The time-event queue, in addition to keeping track of all the crosspoints, also maintains the times at which a segment moves from one section to another. The above L heaps carry local information, as to which is the best pair of segments to merge within each section. Then, at each iteration of the algorithm, it is easy to determine the overall best pair of segments to merge, either by performing a linear scan of the top element of the L heaps, or by maintaining a heap of those L elements. For all practical purposes, L is relatively small, in the order of a few dozens. Therefore, a linear scan is sufficiently fast, and avoids the need for maintaining the extra heap structure, which in the worst case has time complexity $O(L \log L)$. For the rest of this work, we only consider the linear scan approach.

The following theorem gives the space and time complexity of the algorithm.

Theorem 5 *The space complexity of* GrAp-R *with a piecewise linear amnesic function is $O(K + |EQ|)$, and the time complexity to process each new point is $O(L + L \log \frac{K}{L} + \log |EQ|)$.*

Proof: We assume that an equal number of segments corresponds to each section of the amnesic function[4]. The algorithm requires $O(K)$ space for storing the K segments and the L heaps (since all the heaps combined store $O(K)$ values). The space required by the time-event queue accounts for the second term in the complexity function. This space is equal to the number of crosspoints and the number of events related to segments moving from one section to the next. Therefore, we need $O(K + |EQ|)$ space in total.

In terms of time, the algorithm at each iteration needs $O(\log |EQ|)$ time to update the time-event queue, $O(L \log \frac{K}{L})$ time to update the L heaps, and $O(L)$ time to pick the best pair of segments to merge. $\square$

---

[4]This assumption is realistic because of the following observation. The sections of the amnesic function that refer to the newer values of the time series will tend to be of finer granularity and encompass a smaller portion of the time series than the sections referring to the older values. Yet, they will require a higher ratio of segments per data point, since the requirements for accuracy in the newer data points is higher than that for the older ones.

4.1.6   Continuous Amnesic Functions

When the amnesic function is continuous, we identify two cases. First, the amnesic function has a closed form formula. In this case, we can compute the crosspoints of the segments, and we proceed as with the linear amnesic functions. Second, when the amnesic function does not have a closed form formula, we replace the continuous function with a piecewise linear approximation using L sections. Then, we proceed as with the piecewise linear amnesic functions. We construct L heaps, and search in those for the best pair of segment to merge. Since the resulting amnesic function is an approximation of the original function, instead of examining only the top element from each heap, we consider the top-q elements. We calculate the exact error (i.e., based on the continuous amnesic function) of those elements, and pick the best pair of segments among them. This technique proves to work very well, even for small q. We defer further discussion to the full version of this paper.

The following theorem gives the space and time complexity of the algorithm (the proof is similar to the case of piecewise linear amnesic functions).

Theorem 6 *Assume we approximate a continuous amnesic function with L piecewise linear sections. Further, assume that we consider the top-q elements of each heap in order to identify the best pair of segments to merge. Then, the space complexity of* GrAp-R *with a continuous amnesic function is $O(K + |EQ|)$, and the time complexity to process each new point is $O(qL + L \log \frac{K}{L} + \log |EQ|)$.*

4.2   Sliding Windows With Relative Amnesic

In this section we discuss algorithms that solve the online amnesic approximation problem for a sliding window of a streaming time series. Assume a sliding window of size W, and that we use *PLA* to build a build an approximation model Q with K segments. We refer to the side of the sliding window from which new points enter the window as the *start* of the sliding window. We call *end* of the sliding window the side from where points exit, and *last* segment, the segment of Q that approximates the points of the series at the end of the sliding window.

The skeleton of the algorithms for the sliding windows case is the same as the one presented in the previous section, for the amnesic approximation of time series in an unrestricted window. The only difference is that we now have to adjust the approximation such that there is no segment that refers to data points beyond the end of the sliding window. In order to achieve this goal, we simply discard the last segment as soon as it gets entirely out of the sliding window, and we reuse it at the start of the window. Observe though, that the amnesic function is more tolerable to the approximation error towards the end of the sliding window. Then, a question that arises naturally is whether it is possible for the last segment to continue growing by merging with the second to last segment,

and consequently never fall out of the boundaries of the sliding window. The following lemma addresses this question.

Lemma 4 *The last segment of model Q will never grow to represent the entire set of points beyond the end of the sliding window.*

The above lemma guarantees that a sliding window amnesic approximation will never degenerate to an unrestricted window approximation of the time series, but does not give us a bound on the size of the last segment. In Section 5 we experimentally show that the size of the last segment is always relatively small.

4.3   Algorithms for Absolute Amnesic Functions

When we use absolute amnesic functions, there is no restriction in the number of segments that we may use for the approximation. Furthermore, we can calculate the time when a neighboring pair of segments will be eligible to merge. Hence, in this case we do not have to keep track of the segments whose merge will result in the least additional error, and subsequently, there is no need to maintain a heap structure on the adjacent pairs of segments, as we did for the case of the relative amnesic functions.

The algorithms for the *UAA* and *SAA* problems are based on the corresponding algorithms presented for the relative amnesic functions. The only difference, as discussed above, is that there is no need for a priority queue structure. We do not discuss these algorithms any further, due to lack of space.

5   Experimental Evaluation

We implemented our algorithms and conducted a series of experiments to evaluate their efficiency. We also implemented the optimal algorithm, using dynamic programming, and the traditional *BottomUp* algorithm for *PLA* [18], to compare against our techniques.

In order to evaluate our algorithms, we used an extensive set of real-world datasets. These are 40 datasets coming from diverse fields, including finance, medicine, biometrics, chemistry, astronomy, robotics, networking and industry, and covering the complete spectrum of stationary/non-stationary, noisy/smooth, cyclical/non-cyclical, symmetric/asymmetric, etc. [1]. When not explicitly mentioned, the results reported are averages over all 40 datasets. Some of the datasets used in our experiments are illustrated in Figure 11. For all the experiments shown here, we employed a piecewise linear amnesic function. The results for other amnesic functions are similar.

5.1   Comparison to *BottomUp*

In the first set of experiments, we compare the performance of *GrAp-R* to *BottomUp*, which is essentially a comparison between an online and the corresponding offline algorithm.
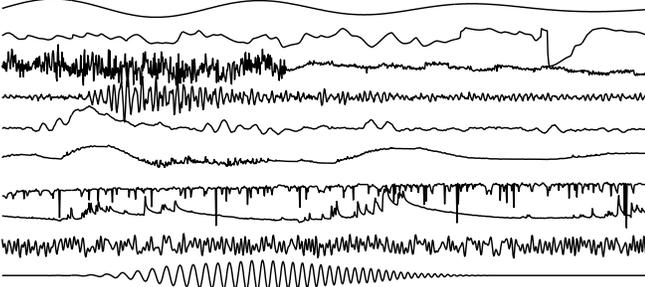
Figure 11. **Some of the datasets used in our experiments.**

Figure 12 depicts the approximation error and computation time for *GrAp-R* and *BottomUp*, for a single dataset. (We get similar graphs for all the datasets we used in our experiments.) We use the unrestricted window model and 10 segments, and we report the error and time as a function of the window size. Our online algorithm consistently provides approximations that are very close to those found by the offline algorithm. At the same time our algorithm is much faster, requiring only constant time for processing every new point (actually, as we discussed in Section 4, the time is logarithmic to K, but independent of N). On the other hand, *BottomUp* has time complexity $O(N \log N)$.
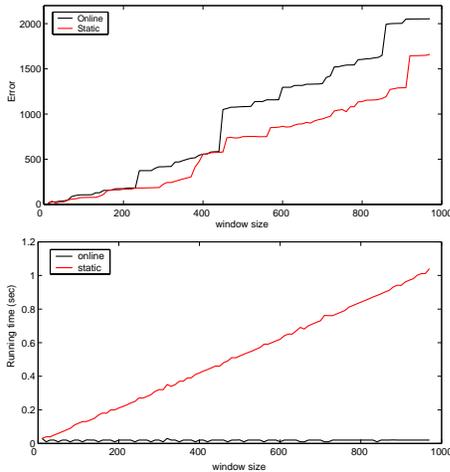

Figure 12. **Typical progression of error (top) and time (bottom) for *GrAp-R* and *BottomUp*, for a single dataset (unrestricted window).**

In the next set of experiments, we quantify the differences in the performance of the two algorithms. We report the cumulative relative error, CRE, which measures the relative increase in the cumulative error when using *GrAp-R*.

$$CRE = 100 \cdot \frac{\sum_{j=1}^{N}(E_{GrAp-R}(T[1..j]) - E_{BottomUp}(T[1..j]))}{\sum_{j=1}^{N} E_{BottomUp}(T[1..j])}$$

The second measure of interest is the speedup, which measures hom many times faster *GrAp-R* is when compared to *BottomUp*.

$$Speedup = \frac{\sum_{j=1}^{N} Time_{BottomUp}(T[1..j])}{\sum_{j=1}^{N} Time_{GrAp-R}(T[1..j])}$$

In Figure 13, we depict CRE as a function of K and N, for the unrestricted window model. Using 50 segments, our algorithm performs within $3\% - 11\%$ of the offline algorithm, for streams of length $1000 - 3000$ points (Figure 13(a)). Though, for increasing N we observe a very slow build-up of the relative error. In the experiment of Figure 13(b), the number of segments we use is 1%, 3%, and 5% of N. In this case, where the ratio N/K remains fixed, CRE remains relatively stable as we increase N. In both cases, our algorithm performs better as the number of segments increases.
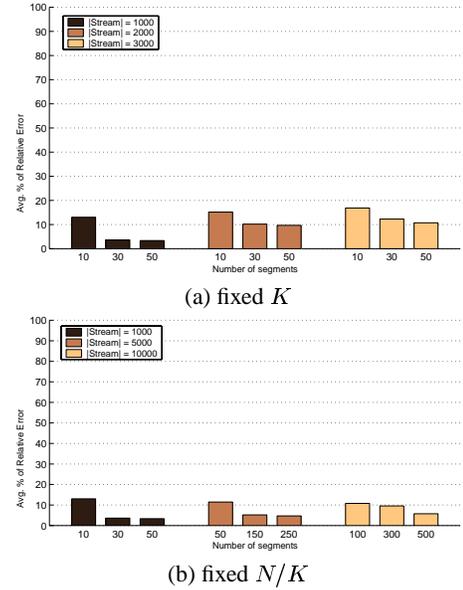

(a) fixed K

(b) fixed N/K

Figure 13. **Comparison of the approximation error between *GrAp-R* and *BottomUp* (unrestricted window).**

Figure 14 shows the speedup that our algorithm achieves, which translates to one or two orders of magnitude faster execution than the offline algorithm (for the experiments we ran). We observe that the speedup increases significantly for decreasing K. This is because the amount of work that *GrAp-R* does remains almost constant (depends on $\log K$), while *BottomUp* requires lots of extra effort for smaller values of K. As expected, the speedup gets larger when we increase N.

We also run the same experiments for the sliding window model. Figure 15 illustrates the results for the speedup, which in this case is mainly a function of the window size (K does not seem to affect the speedup in this case, because of the particular choices of K and the window size). The *GrAp-R* algorithm is $10 - 30$ times faster than *BottomUp*. The results for the error are similar to those for the unrestricted window model, and are omitted due to space restrictions.

The trends for the error and time remain the same as we increase K and N. All the above results show that the online algorithm achieves considerable benefits in terms of speed while losing little in approximation accuracy, when compared to the
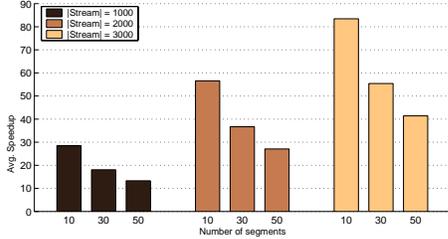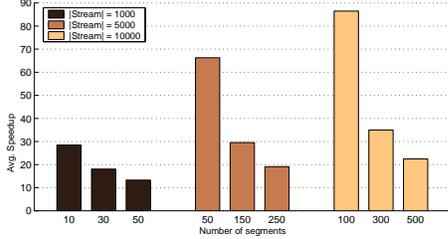
(a) fixed K

(b) fixed N/K

Figure 14. Speedup of *GrAp-R* against *BottomUp* (unrestricted window).
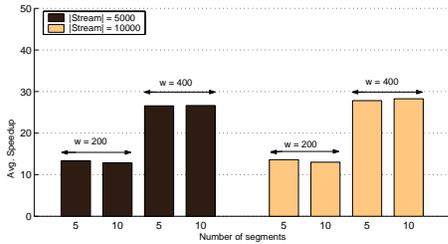

Figure 15. Speedup of *GrAp-R* against *BottomUp* (sliding window).

offline algorithm.

With the next experiment, we address a question that was raised in light of Lemma 4. In the sliding window model, we temporarily allow the last segment of the approximation model to grow beyond the end of the window, until it completely falls out of the boundaries of the window and we discard it. Figure 16 depicts the average number of points outside the sliding window that are represented by the last segment, as a percentage of the window size. In all the cases we tested, this number ranges between $10\% - 15\%$, and therefore, is not a restricting factor for our representation.
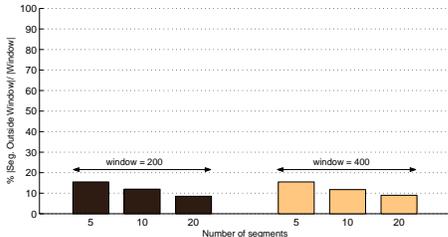

Figure 16. Number of excess points represented by *GrAp-R* (sliding window).

5.2 Comparison to Optimal

In this section we investigate how our techniques compare to the optimal algorithm, *Opt*, implemented with dynamic programming. Unfortunately, due to the high time complexity of the optimal algorithm, this experiment is only possible for relatively small datasets.

We use the same set of 40 datasets and perform the experiment as follows. From each dataset, we randomly extract a subsequence of length 512, and segment it into 16, 32, and 64 segments, using both algorithms under consideration. We measure the relative increase in error for the *BottomUp* algorithm, defined as $(E_{BottomUp} - E_{Opt})/E_{Opt}$. A zero value for the relative error means that *BottomUp* has found the optimal solution. For each dataset, and each number of segments, we average the results over 10 randomly extracted subsequences, and then average the relative error over all 40 datasets. The results are shown in Table 2. They clearly suggest that we lose little by using *BottomUp* as opposed to *Opt*, since *BottomUp* finds solutions very close to optimal. Consequently, based on the experiments we presented in the previous paragraphs, we can safely conclude that *GrAp-R* performs close to optimal, as well. Finally, the last column of the table reports how much slower *Opt* executes (more than two orders of magnitude), and illustrates the inapplicability of the optimal algorithm for an online environment.

K	$(E_{BottomUp} - E_{Opt})/E_{Opt}$	$Time_{Opt}/Time_{BottomUp}$
16	0.058	112
32	0.051	137
64	0.042	173

Table 2. Comparison between *BottomUp* and optimal.

6 Related Work

There exists an extensive literature in the area of time series approximation [19]. Some of the representations that have been proposed include the Fourier transform [10, 25], many different wavelets [23, 7], piecewise polynomials [28, 6], singular value decomposition [6] and symbolic approximations [2]. Many of the above approximation techniques have been adapted to work in an online fashion. For example, piecewise constant approximation can be created online with little loss of accuracy [22], as well as *DFT* [29]. Most of other time series representations have been, or could trivially be, calculated in an incremental fashion [18]. There has also appeared work on data stream summarization, using wavelets [13] and histograms [14]. Cohen and Strauss [9] present a framework for maintaining time-decaying stream aggregates, such as sum and average.

Chen et al. [8] describe a framework for multi-dimensional regression analysis of time series with a tilt time frame. Yet, they do not explicitly tailor their representations to match different amnesic functions. Bulut and Singh proposed using wavelets to represent "data streams which are biased towards the more recent values" [5], and successfully implemented

their method. Although the bias to more recent values can be seen as a special case of an amnesic function, the particular function is dictated by the hierarchical nature of the wavelet transform. Our work removes all the restrictions inherent in the above approaches. The framework we propose takes into account the form of the amnesic function as an integral part of the problem, and provides an effective and efficient solution for a much more general class of amnesic functions.

7 Conclusions

We have introduced the first method to allow the online approximation of streaming time series, which allows arbitrary, user-defined reduction of quality with time. This kind of approximation is of increasing importance in many diverse application domains, such as mobile and real-time devices. We justified our choice of representation with extensive comparisons to competing techniques, and described how we can adapt to allow arbitrary amnesic functions for streaming data. We empirically evaluated our algorithms with extensive experiments on 40 different datasets. The results show that our algorithms offer significant performance improvements over the direct computational approach, while maintaining the quality of the approximation close to optimal. Possible directions for future work include supporting indexed similarity search and other queries on our representation.

Acknowledgements

References

[1] The UCR Time Series Data Mining Archive. University of California, Riverside, Computer Science and Engineering Department. http://www.cs.ucr.edu/~eamonn/TSDMA/, 2002.

[2] H. André-Jönsson and D. Badal. Using Signature Files for Querying Time-Series Data. In *Principles of Data Mining and Knowledge Discovery*, pages 211–220, Trondheim, Norway, June 1997.

[3] Julien Basch. Kinetic Data Structures. Stanford University, Department of Computer Science. PhD Thesis, 1999.

[4] Richard Bellman. On the Approximation of Curves by Line Segments Using Dynamic Programming. *Communications of the ACM*, 4(6):284, 1961.

[5] Ahmet Bulut and Ambuj K. Singh. SWAT: Hierarchical Stream Summarization in Large Networks. In *International Conference on Data Engineering*, pages 303–314, Bangalore, India, March 2003.

[6] Kaushik Chakrabarti, Eamonn J. Keogh, Sharad Mehrotra, and Michael J. Pazzani. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. *ACM Transactions on Database Systems*, 27(2):188–228, 2002.

[7] K. Chan and W. Fu. Efficient Time Series Matching by Wavelets. In *International Conference on Data Engineering*, pages 126–133, Sydney, Australia, March 1999.

[8] Yixin Chen, Guozhu Dong, Jiawei Han, Benjamin W. Wah, and Jianyong Wang. Multi-Dimensional Regression Analysis of Time-Series Data Streams. In *VLDB International Conference*, pages 323–334, Hong Kong, China, August 2002.

[9] Edith Cohen and Martin Strauss. Maintaining Time-Decaying Stream Aggregates. In *ACM PODS International Conference*, pages 223–233, San Diego, CA, USA, jun 2003.

[10] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast Subsequence Matching in Time-Series Databases. In *ACM SIGMOD International Conference*, pages 419–429, Minneapolis, MI, USA, May 1994.

[11] D. Ganesan, B. Greenstein, D. Perelyubskiy, D. Estrin, and J. Heidemann. An Evaluation of Multi-Resolution Search and Storage in Resource-Constrained Sensor Networks. Technical Report CENS 0010, April 2003.

[12] Xianping Ge and Padhraic Smyth. Segmental Semi-Markov Models for Endpoint Detection in Plasma Etching. In *AEC/APC Symposium*, Lake Tahoe, NV, USA, September 2000.

[13] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. In *VLDB International Conference*, pages 79–88, Rome, Italy, sep 2001.

[14] Sudipto Guha and Nick Koudas. Approximating a Data Stream for Querying and Estimation: Algorithms and Performance Evaluation. In *International Conference on Data Engineering*, pages 567–576, San Jose, CA, USA, March 2002.

[15] R. Hogg, A. Rankin, M. McHenry, D. Helmick, C. Bergh, S. Roumeliotis, and L. Matthies. Sensors and Algorithms for Small Robot Leader/Follower Behavior. In *SPIE AeroSense Symposium*, Orlando, FL, USA, April 2001.

[16] Jim Hunter and Neil McIntosh. Knowledge-Based Event Detection in Complex Time Series Data. In *Artificial Intelligence in Medicine and Medical Decision Making*, pages 271–280, Aalborg, Denmark, June 1999.

[17] Alefiya Hussain, John Heidemann, and Christos Papadopoulos. A Framework for Classifying Denial of Service Attacks. In *ACM SIGCOMM Conference*, page To appear, Karlsruhe, Germany, August 2003.

[18] E. Keogh, S. Chu, D. Hart, and M. Pazzani. An Online Algorithm for Segmenting Time Series. In *IEEE International Conference on Data Mining*, pages 289–296, San Jose, CA, USA, November 2001.

[19] Eamonn J. Keogh and Shruti Kasetty. On the Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration. In *International Conference on Knowledge Discovery and Data Mining*, pages 102–111, Edmonton, Canada, July 2002.

[20] Eamonn J. Keogh and Michael J. Pazzani. An Enhanced Representation of Time Series Which Allows Fast and Accurate Classification, Clustering and Relevance Feedback. In *International Conference on Knowledge Discovery and Data Mining*, pages 239–243, New York, NY, USA, August 1998.

[21] A. Koski, M. Juhola, and M. Meriste. Syntactic Recognition of ECG Signals By Attributed Finite Automata. *Pattern Recognition*, 28(12):1927–1940, 1995.

[22] Iosif Lazaridis and Sharad Mehrotra. Capturing Sensor-Generated Time Series with Quality Guarantees. In *International Conference on Data Engineering*, pages 429–440, Bangalore, India, March 2003.

[23] Ivan Popivanov and Renée J. Miller. Similarity Search Over Time Series Data Using Wavelets. In *International Conference on Data Engineering*, pages 802–813, San Jose, CA, USA, February 2002.

[24] William Pugh. Skiplists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, 1990.

[25] Davood Rafiei. On Similarity-Based Queries for Time Series Data. In *International Conference on Data Engineering*, Sydney, Australia, March 1999.

[26] David Steere, Antonio Baptista, Dylan McNamee, Calton Pu, and Jonathan Walpole. Research Challenges in Environmental Observation and Forecasting Systems. In *Mobile Computing and Networking*, Boston, MA, USA, August 2000.

[27] H. J. L. M. Vullings, M. H. G. Verhaegen, and H. B. Verbruggen. ECG Segmentation Using Time-Warping. In *International Symposium on Intelligent Data Analysis*, pages 275–285, London, England, August 1997.

[28] B. Yi and C. Faloutsos. Fast Time Sequence Indexing for Arbitrary LP-Norms. In *VLDB International Conference*, pages 385–394, Cairo, Egypt, September 2000.

[29] Yunyue Zhu and Dennis Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *VLDB International Conference*, pages 358–369, Hong Kong, China, August 2002.