Matrix Profile XXIV: Scaling Time Series Anomaly Detection to Trillions of Datapoints and Ultra-fast Arriving Data Streams

Yue Lu
University of California, Riverside
Riverside, California, USA
ylu175@ucr.edu

Renjie Wu
University of California, Riverside
Riverside, California, USA
rwu034@ucr.edu

Abdullah Mueen
University of New Mexico
Albuquerque, New Mexico, USA
mueen@unm.edu

Maria A. Zuluaga
EURECOM
Sophia Antipolis, France
maria.zuluaga@eurecom.fr

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

ABSTRACT

Time series anomaly detection remains one of the most active areas of research in data mining. In spite of the dozens of creative solutions proposed for this problem, recent empirical evidence suggests that *time series discords*, a relatively simple twenty-year old distance-based technique, remains among the state-of-art techniques. While there are many algorithms for computing the time series discords, they all have limitations. First, they are limited to the batch case, whereas the online case is more actionable. Second, these algorithms exhibit poor scalability beyond tens of thousands of datapoints. In this work we introduce DAMP, a novel algorithm that addresses both these issues. DAMP computes exact left-discords on fast arriving streams, at up to 300,000 Hz using a commodity desktop. This allows us to find time series discords in datasets with trillions of datapoints for the first time. We will demonstrate the utility of our algorithm with the most ambitious set of time series anomaly detection experiments ever conducted.

CCS CONCEPTS

• Computing methodologies → Anomaly detection.

KEYWORDS

Time Series, Anomaly Detection, Streaming Data.

ACM Reference format:
Yue Lu, Renjie Wu, Abdullan Mueen, Maria A. Zuluaga, Eamonn Keogh. 2022. Matrix Profile XXIV: Scaling Time Series Anomaly Detection to Trillions of Datapoints and Ultra-fast Arriving Data Streams. *In Proceedings of ACM Conference on Knowledge Discovery and Data Mining (KDD '22), August 14–18, 2022, Washington, DC, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3534678.3539271

KDD '22, August 14–18, 2022, Washington, DC, USA

ACM ISBN 978-1-4503-9385-0/22/08. https://doi.org/10.1145/3534678.3539271

1. INTRODUCTION

Time series anomaly detection is one of the most useful tools investigated by the data mining community [2][8][11]. It can be used offline to investigate archival data, or online, to monitor critical situations that offer a potential to intervene. For example, summoning a doctor or shutting down a machine that may be about to damage itself. Given its importance, it is unsurprising that this area attracts a lot of attention from the community, with dozens of algorithms proposed each year. However, in spite of the plethora of algorithms in the literature, there is increasing evidence that a twenty-year-old distance-based method called *time series discords* is still competitive [11]. Discords are competitive with deep learning methods in spite (or perhaps *because*) of their great simplicity. A time series discord is simply the subsequence of a time series that is maximally far from its nearest neighbor.

At least one-hundred papers have reported using discords to solve problems in diverse domains, and discords seem to be the only time series anomaly detection technique to produce "superhuman" results (see discussion in Section 2). However, discords have three important limitations that have limited their broader adoption:

- If an anomalous pattern appears at least twice in the time series, then each occurrence will be the other nearest neighbor, and thus fail to optimize the discord definition. This is informally called the *twin-freak* problem.
- Discords are only defined for the *batch* case, but anomaly detection is most actionable in *online* settings.
- In spite of extensive progress in speeding up discord discovery, datasets with millions of datapoints remain intractable.

In this paper we introduce DAMP (Discord Aware Matrix Profile), a novel algorithm which solves all the above problems.

- DAMP is not confused by repeated anomalies (twin-freaks), it simply flags the first occurrence (if desired, other occurrences can then be found by simple similarity search).
- DAMP is defined for both online and offline cases. Moreover, DAMP has an extraordinary fast throughput, exceeding 300,000 Hz on standard hardware.
- As the previous bullet point suggests, DAMP is extraordinarily scalable. For the first time, this allows us to consider datasets with millions, billions and even trillions of datapoints.

The rest of this paper is organized as follows. In Section 2 we motivate the use of *discords* as the time series anomaly definition most worthy of acceleration and generalization. We also concretely

define a new term, *effectively online*, that allows DAMP to tackle ultra-fast real-time data sources found in industry and science. Section 3 contains the necessary definition and notation require, and Section 4 discusses related work, before we introduce our algorithm in Section 5. In Section 6 we conduct the most ambitious empirical evaluation of time series anomaly detection ever attempted.

2. MOTIVATION

Before continuing, we need to answer the following question. Why make the effort to fix discord's scalability issues, rather than invent a new algorithm, or make one of the many dozens of proposed approaches more scalable?

The reason is that there is increasing evidence that discords remain competitive with the state-of-the-art[1] [11]. Among the hundreds of time series anomaly detection algorithms proposed in the last two decades, only time series discords could claim to have been adopted by more than one hundred independent teams to actually solve a real-world problem (a partial bibliography is here [6]).

In addition, time series discords seem to be the only anomaly detection algorithm that has been demonstrated to perform at superhuman levels [11]. All other algorithms that we are aware of have shown to discover anomalies that are also readily apparent to the human eye. For example, a recent paper proposed an LSTMs network for anomaly detection and evaluated it on data retrieved from Mars [8]. However, the only anomaly shown in the paper shows a visually obvious anomaly where a repeated periodic pattern suddenly transitions to a literal flatline. Of course, this does not mean that such algorithms have no value, as human attention is very expensive. However, the literature also offers some examples where discords have found anomalies that are very subtle, defying the possibility of human discovery. Consider Figure 1, which shows the vibration of an industrial motor [4][12].

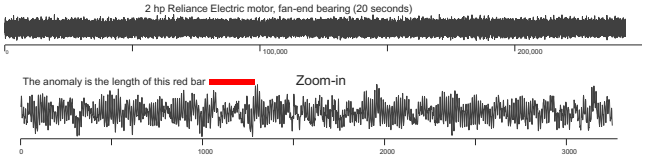

Figure 1: *top*) A 20-second run of an industrial motor. *bottom*) a zoom-in of the region known to contain an anomaly, which is the length of (but not necessarily at the location of) the red bar.

The data comes for a motor running under no load, however for a brief instant a load was applied and immediately removed, creating an anomaly. It is clearly fruitless to visually search for the anomaly in the *full* dataset, however, even if we zoom into a local region containing the anomaly, it is not clear where it is. In Figure 2 we task time series discords with detecting the anomaly.

[1] Note that some papers misattribute the success of their anomaly detection to the Matrix Profile or to HOTSAX, but these are simple different algorithms to compute time series discords.

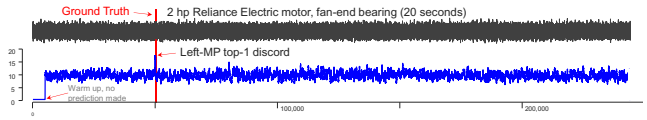

Figure 2: *top*) A 20-second run of an industrial motor. *bottom*) The time series discord discovered by the Left-MP correctly locates the anomaly.

Beyond the accuracy of discords prediction here, note that this dataset contains 244,189 datapoints, representing about 20 seconds of wall clock time recorded at 12,000 Hz. We are not aware of any anomaly detection algorithm in the literature that could process this dataset in real-time, however, as we will show, DAMP *can*.

We also consider a dataset that is dramatically different to the bearing data. In Figure 3 we show the Left-MP for an ECG which we know contains a single anomaly beat, a *ventricular contraction*.

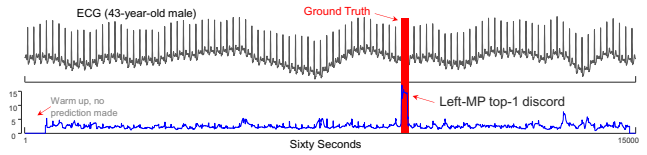

Figure 3: *top*) A sixty-second snippet of an ECG. *bottom*) The top-1 time series discord correctly locates the anomaly.

This dataset has a wandering baseline which is diagnostically meaningless, but which distracts the human eye (and many algorithms). However, once again time series discords have no problem detecting the anomaly, which cardiologist Dr. Greg Mason says is on the cusp of his ability to detect by eye.

Finally, in Figure 4 we consider a dataset that was explicitly created with the sole purpose of having anomalies that are "*difficult to spot for the human eye*" [13]. Here again discords are superhuman.

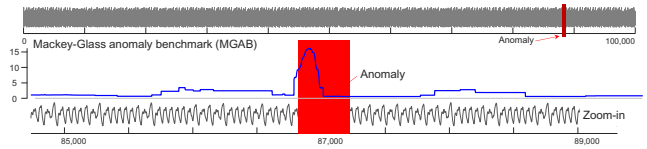

Figure 4: *top*) The MGAB dataset was built to defy visual discovery of anomalies. *bottom*) The Top-1 time series discord correctly locates the anomaly.

In summary, both the recent literature and our experiments suggest that time series discords are *at least* competitive with recently proposed algorithms, and thus worthy of accelerating to allow discords to be discovered in settings that are currently infeasible.

2.1 Effectively Online Anomaly Detection

The meaning of the terms *batch* and *online* are obvious, however in order to make our claims clearer, it is useful to introduce a new term, *effectively online*. A true online algorithm reports the instant it detects a monitored condition. However, let us imagine the following scenario: After a difficult cardiac surgery, a doctor decides she wants to monitor her patient for anomalous heartbeats, which may be an indication of postoperative Cardiac Tamponade (CT). If the patient does have an ECG suggestive of CT symptoms, the doctor has perhaps eight to ten minutes to confirm CT with an

ultrasound and perform pericardiocentesis, a procedure done to remove fluid that has built up in the sac around the heart [9]. Because the doctor is nervous about the possibility of CT, she arranges the rest of her day such that she can be in the ICU within two minutes, for example eating her lunch in a hospital cafeteria rather than her favorite restaurant across town. Clearly in this situation an algorithm that reported an anomalous heartbeat ten minutes after its appearance would be unacceptable. However, an algorithm that reported an anomalous heartbeat at most two seconds after it appears would be just as good as a true online algorithm. As such we propose the following:

Definition 1: An algorithm is said to be *effectively online*, if the lag in reporting a condition has little or no impact on the actionability of the reported information.

Note that the scale of the permissible lag is problem dependent. In the above scenario, two seconds made sense to the cardiologists we consulted. In an ultrafast arriving data stream, the permissible lag may be as little as 0.1 seconds, and for telemetry arriving from devices with a slow cycle rate, say the daily periodicity of pedestrian traffic, the permission lag may be minutes to hours.

We suspect that many algorithms that are referred to as online in the literature, are really effectively online. The above discussion allows us to frame our contribution. Our proposed algorithm DAMP is parameterized by a single variable called *lookahead*.

- If *lookahead* is zero, DAMP is a fast *true* online algorithm.
- If *lookahead* is allowed to be arbitrarily large, DAMP is an ultrafast batch algorithm. We should not be surprised that a batch algorithm can be much faster, as it has access to all the information at once.

And now the *raison d'etre* for our digression:

- Even if *lookahead* is a small (but non-zero) number, DAMP is effectively online algorithm, yet it retains most or all the speedup of the arbitrarily large *lookahead* algorithm.

As we will show, DAMP allows for the discovery of time series discords in ultra-fast-moving streams for the first time.

3. DEFINITIONS AND BACKGROUND

We begin by defining the key terms used in this work. The data we work with is a *time series*.

Definition 2: A *time series* T is a sequence of real-valued numbers t_i: $T = [t_1, t_2, \ldots, t_n]$ where n is the length of T.

Typically, we consider only local *subsequences* of the times series.

Definition 3: A *subsequence* $T_{i,m}$ of a time series T is a continuous subset of data points from T of length m starting at position i. $T_{i,m} = [t_i, t_{i+1}, \ldots, t_{i+m-1}], 1 \leq i \leq n - m + 1$.

The length of the subsequence is typically set by the user based on domain knowledge. For example, for most human actions, ½ second may be appropriate, but for classifying transient stars, three days may be appropriate.

If we take any subsequence $T_{i,m}$ as a query, calculate its distance from all subsequences in the time series T and store the distances in an array in order, we get a *distance profile*.

Definition 4: *Distance profile* D_i for time series T refers to an ordered array of Euclidean distances between the query subsequence $T_{i,m}$ and all subsequences in time series T. Formally, $D_i = d_{i,1}, d_{i,2}, \ldots, d_{i,n-m+1}$, where $d_{i,j}$ $(1 \leq i, j \leq n - m + 1)$ is the Euclidean distance between $T_{i,m}$ and $T_{j,m}$.

For distance profile D_i of query $T_{i,m}$, the i^{th} position represents the distance between the query and itself, so the value must be 0. The values before and after position i are also close to 0, because the corresponding subsequences have overlap with query. Our algorithm neglects these matches of the query and itself, and instead focuses on *non-self match*.

Definition 5: *Non-Self Match*: Given a time series T containing a subsequence $T_{p,m}$ of length m starting at position p and a matching subsequence $T_{q,m}$ starting at q, $T_{p,m}$ is a *non-self match* to $T_{q,m}$ with distance $d_{p,q}$ if $|p - q| \geq m$.

With the definition of non-self match, we can define *time series discords*.

Definition 6: *Time Series Discord*: Given a time series T, the subsequence $T_{d,m}$ of length m beginning at position d is said to be a discord of T if the distance between $T_{d,m}$ and its nearest non-self match is maximum. That is, $\forall$ subsequences $T_{c,m}$ of T, non-self matching set M_D of $T_{d,m}$, and non-self matching set M_C of $T_{c,m}$, $min(d_{d,M_D}) > min(d_{c,M_C})$.

Although there are many ways to locate time series discord, the most effective one recently is the *matrix profile* [18].

Definition 7: A *matrix profile* P of a time series T is a vector storing the z-normalized Euclidean distance between each subsequence and its nearest non-self match. Formally, $P = [min(D_1), min(D_2), \ldots, min(D_{n-m+1})]$, where D_i $(1 \leq i \leq n - m + 1)$ is the distance profile of query $T_{i,m}$ in time series T. It is easy to see that the highest value of the matrix profile is the time series discord.

As we will explain below, we can compute a special matrix profile which only looks to the past. We call it the *left matrix profile*.

Definition 8: A *left matrix profile* P^L of a time series T is a vector that stores the z-normalized Euclidean distance between each subsequence and the nearest non-self match appearing before that subsequence. Formally, given a query subsequence $T_{i,m}$, let $D_i^L = d_{i,1}, d_{i,2}, \ldots, d_{i,i-m+1}$ be a special distance profile that records only the distance between the query subsequence and all subsequences that occur before the query, then we have $P^L = [min(D_1^L), min(D_2^L), \ldots, min(D_{n-m+1}^L)]$.

Note that the term discord in this paper refers to the highest value on the left matrix profile P^L, i.e., left-discord. For the sake of simplicity, we will refer to left-discord as discord where there is no ambiguity. It is clear that in the *online* case, we must use the Left-MP. However, here we argue that even in the *offline* case we should use it. To see why, consider the example shown in Figure 5.

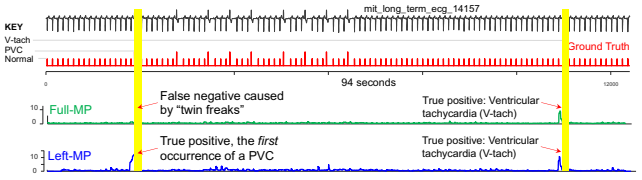

Figure 5: *top to bottom*) A snippet of ECG with two types of anomalous heartbeats indicated by a ground truth vector. A full Matrix Profile can find the sole occurrence of V-tach, but is confused by the multiple occurrences of PVCs (twin-freaks) and cannot find them. In contrast, the Left-MP flags the first occurrence of a PVC and the first (and only) V-tach.

Here left-discords solve the twin-freak problem by reporting the first occurrence of the anomaly (later occurrences, if of interest, can be trivially found with subsequence search/monitoring).

4. RELATED WORK

The topic of anomaly detection in time series has seen a dramatic explosion of interest in recent years, it is a difficult area to survey in limited space. We refer the interested reader to [1][2][3][8][11][13] and the references therein. We have also compiled a longer annotated biography that explicitly discusses discords at [6].

There are two important points that we have gathered from our survey of the literature. The first is due mostly to a single paper [16], that forcefully suggests some of the apparent success of recently proposed algorithms may be questionable, due to severe problems with the commonly used benchmarks in this area. Beyond the issues that [16] notes, we wish to add another issue. Most of these benchmarks are minuscule. For example, the individual time series problems in Yahoo S5 dataset are all about 1,200 data points long. We suspect that the small datasets that the community has focused on is at least partly due to the poor scalability of current approaches. For example, a recent paper examines time series of length 140,256 and "*Given the length of the dataset, we sub-sample it by a factor 10.*" [1]. This paper is by researcher group at Amazon, who presumably do not lack for computational resources. For reference, DAMP takes 0.9 seconds of the *full*-sized version of this dataset [6].

5. DAMP

5.1 Intuitive Overview of DAMP

Before we give a formal explanation of our algorithm, we will first give the intuition of how it works. We will discuss the batch case, and then later explain the (minor) step needed to generalize to the online case. As shown in Figure 6, it is helpful to explain the algorithm mid execution, as it is processing the subsequence T_i.

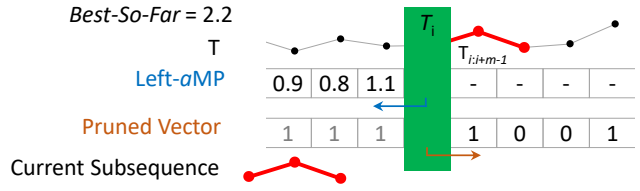

Figure 6: A sketch of the DAMP algorithm in progress, processing the current subsequence. *top*) The time series T. *center*) The Left-*a*MP, its values between 1 and *i* are computed, its values after *i* have yet to be computed. *bottom*) the Pruned Vector indicates subsequences that can be ignored without effecting the final result.

Figure 6.*top* shows the time series T being processed, the green bar indicating the current subsequence being processed at location i. Note that we have created two parallel vectors to accompany T. The Left-*a*MP is the vector we are computing. It is an approximation to the true Left-MP, with the following properties:

- If location j is the true left-discord for the time series $T_{1:j}$, then the discord value at $a\text{MP}_j$ is not an approximation, but the true left-discord value.
- Otherwise, the approximation at $a\text{MP}_j$ is strictly bounded: $\text{MP}_j \le a\text{MP}_j \le \max(\text{MP}_{1:j})$

These properties tell us that we can take any prefix of T (inducing the special case of the entire length of T), and the left-discord reported by the Left-*a*MP will be the same as that reported by the Left-MP.

In Figure 6.*bottom* we show the other parallel vector that accompanies T and the Left-$a\text{MP}_j$. The Pruned Vector tells us which subsequences could not be the left-discord, and hence do not need to be processed. At initialization time, this vector is set to all '1's, indicating that all subsequences must be processed. However, as we process the data, we may be able to "peek into the future" and cheaply determine locations that could not be a discord, and flip their corresponding bits to '0'.

At the i^{th} location, the processing can be divided into two independent steps, *backward* processing and *forward* processing.

5.1.1 *Backward Processing*

Here we must discover if the current subsequence $T_{i:i+m-1}$ is the discord. The naïve way to do this would be to compute its nearest neighbor distance to any subsequence in $T_{1:i}$.

However, note that in general we may not need to find the nearest neighbor, any neighbor whose distance is less than the *Best-So-Far* will disqualify the current subsequence from being the discord. This suggests an early abandoning scheme that we can optimize with the two following observations:

- Instead of incrementally searching from the beginning, we should expect to be able to abandon earlier if we search *backwards* from the i^{th} location. The reason this is true is because the patterns can drift over time. In other words, the pattern most likely to be similar to the current subsequences is generally the subsequence *just* before the current subsequence.
- The MASS algorithm is optimized for queries with powers of two length. For example, using the machine that performed all the experiments in this paper, we find that a MASS search with a query of length 512, takes 0.025 seconds for a time series of length 524,288 (i.e. 2^{19}). But if we delete a single point to get a 524,287, it takes 0.177 seconds. This suggests we should attempt to construct a backward search algorithm that is comprised mostly or solely of such p^{integer} length queries.

These two observations suggest an algorithm. We should look backwards at the prefix that is the next power-of-two longer than m. If that yields a neighbor that is less than the *Best-So-Far (BSF)* we are done, we simply place that value in $a\text{MP}_i$ as our approximation. If that was not the case, we double the length of the prefix to *two* times the next power-of-two longer than m, and try again. We continue to iteratively double until we find a nearest neighbor distance that is less than the *Best-So-Far*, or until our prefix includes the full span back to the beginning of T. In that latter case, we use the nearest neighbor distance to update both the *Best-So-Far* and $a\text{MP}_i$.

5.1.2 *Forward Processing*

Here we will attempt to cheaply discover and prune subsequences that could not be the left-discord. If we take the current subsequence and compare it to the suffix of T, that is, to $T_{i+m:n}$ (the search must start at $i+m$ to avoid self-match), any subsequence that is less than the *Best-So-Far* distance to current subsequence can be pruned (have its corresponding bit in the Pruned Vector set to '0').

In principle, we could do this search from $i+m$ to the end. However, the two observations in the previous section still apply. While the next few cycles may be similar and yield a good pruning rate, over time the patterns tend to drift and the pruning rate falls. The combination of a long expensive similarity search and the lower

pruning rate means that the forward step may not "pay" for itself. So instead, we can look forward a limited amount, say *four* times the next power-of-two longer than m. After completing both the backward and forward processing, the algorithm increments the current pointer from i to the next index which has a '1' in the Pruned Vector, and repeats the two processing steps.

5.2 Formal Pseudocode for DAMP

Here we formalize the intuition of the previous sections with the pseudocode shown in Table 1. For ease of explanation, we first consider only the batch case. In lines 1 and 2 we initialize two vectors that are essentially the same length as the time series T, but are actually of length n-m+1. These are *PV* (Pruned Vector), a Boolean vector that indicates which indices can be dismissed without evaluation, and *a*MP, which is the approximate Matrix Profile we wish to compute. The current highest discord score encountered during execution is stored in the *BSF*, initialized to zero in line 3.

Table 1: The Main DAMP Algorithm

```
Function: DAMP(T, m, splndex)
Input:    T:       Time series
          m:       Subsequence length
          splndex: Location of split point between training and test data
Output:   aMP:     Left approximate Matrix Profile
1  PV = ones(1,length(T)-m+1)
2  aMP = zeros(1,length(T)-m+1)
3  BSF = 0          // The current best discord score
4  // Scan all subsequences in the test data
5  For i = splndex to length(T) − m + 1
6      If NOT PV_i      // Skip the pruned subsequence
7          aMP_i = aMP_i-1
8      Else
9          [aMP_i, BSF] = BackwardProcessing(T, m, i, BSF)
10         PV = ForwardProcessing(T, m, i, BSF, PV)
11 return aMP
```

In lines 5 to 10, we iterate through all subsequences of length m in the test data. In each iteration, we first determine whether the current subsequence was pruned, i.e., whether it is marked as 0 in the *PV* (line 6). If yes, we assign the discord score of the previous subsequence to the current subsequence and then skip to the next subsequence (line 7). If the current subsequence was not pruned, we must process it. In line 9 we call BackwardProcessing to calculate the discord score of the current subsequence. In particular, if the backward search finds a value higher than the current highest discord score (*BSF*), BackwardProcessing returns the *exact* score of the current subsequence and updates the *BSF*; otherwise, BackwardProcessing returns an approximate score of the current subsequence and does not update the *BSF*. Note that while this score is approximate, it is bounded between the true score and the current *BSF*.

At this point we have completely processed the current location. However, before we increment our loop index to process the next location, we take a brief digression. We will use the current subsequence to look "forward", finding any subsequences ahead of it that have a distance to it that is less than the current *BSF*. It is easy to see that any such subsequences could not be a better discord than the current *BSF*, as when they do BackwardProcessing, they would find the current subsequences to be close enough to disqualify them. This observation allows us to prune these "near-enough" neighbors of the current subsequence. Concretely, line 10 invokes ForwardProcessing to find out the subsequences that can be pruned within a specific range in the future (if any), and their corresponding vectors are marked as 0 and recorded in the Pruned Vector *PV*. Finally in line 11 we return the left approximate Matrix Profile computed by the DAMP algorithm.

Table 1 provides a high-level overview of how the DAMP algorithm works. Let us now "zoom in" and look at the two core subroutines of DAMP, BackwardProcessing and ForwardProcessing. We begin with Table 2 to explain backward processing, whose intuition we laid out in Section 5.1.1.

Table 2: DAMP Backward Processing Algorithm

```
Function: [aMP_i, BSF] = BackwardProcessing(T, m, i, BSF)
Input:    T:       Time series
          m:       Subsequence length
          i:       Index of current query
          BSF:     Highest discord score so far
Output:   aMP_i:   Discord value at position i
          BSF:     Updated highest discord score so far
1  aMP_i = inf
2  prefix = 2^nextpow2(m)  // Initial length of prefix
3  While aMP_i ≥ BSF
4      If the search reaches the beginning of the time series
5          aMP_i = min(MASS(T_1:i, T_i:i+m-1))
6          If aMP_i > BSF  // Update the current best discord score
7              BSF = aMP_i
8          break
9      Else
10         aMP_i = min(MASS(T_i-prefix+1:i, T_i:i+m-1))
11         If aMP_i < BSF
12             break     // Stop searching
13         Else         // Double the length of prefix
14             prefix =2*prefix
15 return aMP_i, BSF
```

In line 1 we begin by initializing the discord score of the current query at position i to positive infinity. Then in line 2 we specify the initial length of the backward processing and stores it in the variable *prefix*. We employ 2^nextpow2(m) to define this initial length. Specifically, when we fed the subsequence length m into 2^nextpow2(m), it will return the smallest power of 2 greater than m. Recall that we are doing this because MASS is significantly faster when the length of the time series is a power of two. Since we are going to do a "piecewise" search of the time series that precedes the subsequence being processed, it makes sense to make these pieces be a power of two in length.

The loop in lines 3-14 evaluates the exact or approximate discord score of the current query. Here we adopt the idea of "iterative doubling". At the beginning, we find the nearest neighbor of the current query in the initial length *prefix* and save the distance between the current query and the nearest neighbor into $a\text{MP}_i$ (line 10). If this distance is lower than the current highest discord score, this means that we find a nearest neighbor for the current query within *prefix* that is more similar than the current discord and its nearest neighbor, so it cannot be a discord, and the iteration terminates (lines 11-12). However, if the distance between the query and its nearest neighbor $a\text{MP}_i$ is higher than the current highest discord score *BSF*, we double the length of the backward processing and continue the search in the next iteration (lines 13-14). This idea is visualized in Figure 7. We keep iteratively doubling until we compute a score smaller than the *BSF* within the range *prefix*, or search to the beginning of the time series T. If the search gets to the beginning of the time series, we first find the nearest neighbor of the query from position 1 to i and store the distance to the nearest neighbor in $a\text{MP}_i$ (lines 4-5).

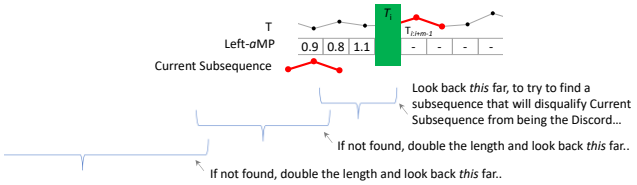

Figure 7: A visualization of the *iterative doubling* search policy used in lines 10-14 of Table 2. See also Figure 6.

After that, we will check whether $a\text{MP}_i$ is still larger than *BSF* (line 6). If yes, this means that we cannot find a nearest neighbor that is similar enough to the current query, and clearly, the current query is the new discord. In this case, we will update the highest discord score and break out of the loop (lines 7-8). Finally, line 15 returns the result of backward processing, the score of the current query $a\text{MP}_i$, and the current highest discord value *BSF*.

Note that if the search reaches the very beginning of the time series, our computation is performed in the global region (from 1 to *i*), not in the local region *prefix*, in which case the discord score of the current query $a\text{MP}_i$ is an *exact* value; whereas if our score is computed in the local region *prefix*, $a\text{MP}_i$ is an approximate value, but bounded between the true score and the current *BSF*.

If we *just* use the backward processing step (line 9 of Table 1), then we have a fast online algorithm to compute the *a*MP. However, the use of forward processing as outlined in Table 3 can speed up the processing by at least a further order of magnitude. This is the algorithm whose intuition was laid out in Section 5.1.2.

Table 3: DAMP Forward Processing Algorithm

Function:	*PV* = ForwardProcessing(*T*, *m*, *i*, *BSF*, *PV*)
Input:	*T*: Time series m: Subsequence length *i*: Index of current query *BSF*: Highest discord score so far *PV*: Pruned Vector
Output:	*PV*: Updated Pruned Vector
1	*lookahead* = 2^nextpow2(*m*) // Length to peek ahead
2	**If** the search does not reach the end of the time series
3	*start* = *i* + *m*
4	*end* = min(*start* + *lookahead* − 1,length(T))
5	$D_i^{'}$ = MASS($T_{start:end}, T_{i:i+m-1}$) // Definition 4
6	*indices* = all indices in $D_i^{'}$ with values less than *BSF*
7	*indices* = *indices* + *start* − 1 // Convert indices on distance profile to indices on time series
8	$PV_{indices}$ = 0 // Update the Pruned Vector
9	**return** *PV*

The purpose of forward processing is admissible pruning. That is, if there is evidence that some future subsequences cannot be a discord, we will ignore these subsequences and no longer perform expensive processing on them. To achieve this in line 1 we need to define *lookahead*, the range of how many subsequences to peek ahead. Here we also use 2^nextpow2(*m*), i.e., the smallest power of 2 larger than the subsequence length *m*. After that, we need to determine whether the forward search exceeds the range of *T* to ensure that our processing is safe and there is no out-of-bounds problem (line 2). Line 3 defines the start position of the forward search, namely *start*. To avoid self-matching, we set the *start* to the position after the end of the query, that is, *i+m*. Line 4 explicitly defines the end position of the forward search, and since the length of our forward search is *lookahead*, or *n*. we can easily conclude that *end* is *start* + *lookahead* - 1. In line 5, we calculate the distance profile $D_i^{'}$ by calling MASS.

The distance profile $D_i^{'}$ here is slightly different from the one described in Definition 4 because it is computed under a specific range. That is, $D_i^{'}$ stores the distance between the current query and all subsequences in the range of *lookahead* (from *start* to *end*) instead of the distance between the current query and all subsequences of *T*. Once the distance profile $D_i^{'}$ is constructed, we can use it for pruning. Suppose there exist subsequences in the future that are more similar to the current query than the discord to its nearest neighbor. In that case, these subsequences cannot be a discord, so we can prune them. Therefore, we can use the current highest discord score *BSF* as a criterion to find all the indices in the distance profile with values lower than the *BSF* (line 6). Since the indices on the distance profile start at 1 and are not aligned with the true indices of the time series, we need an additional step in line 7 to convert the indices on the distance profile to the true indices of the subsequence. After line 7 we get a list of indices for the subsequences that can be pruned out. The Pruned Vector values at the corresponding positions specified in the list *indices* are set to 0 (line 9), indicating that when later iterations process the subsequences listed in *indices* we can simply skip them. At last, line 10 returns the updated Pruned Vector *PV*.

The forward processing algorithm has exactly one parameter, the *lookahead* length. How should we set this? In Figure 8.*left* we sketch out the tradeoffs involved. A longer *lookahead* can prune more subsequences, but this comes at the cost of more expensive similarity searches. As Figure 8.*right* shows, this intuition is borne out by experiment. The good news is that the speedup is dramatic, that the sweet spot is early (given us effectively online detection), and that the exact value of the *lookahead* parameter is not too critical. All datasets we examined exhibit this "U-shaped" behavior, although the height of the base of the "U" can be lower (smooth and highly periodic data) or higher.

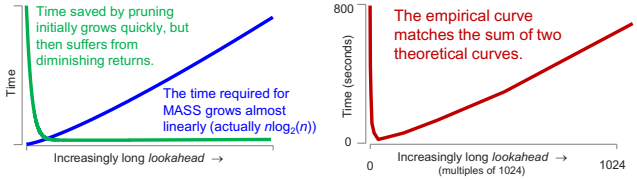

Figure 8: *left*) The *lookahead* tradeoff is based on two factors. As the *lookahead* grows, the pruning rate becomes greater, but the cost of the similarity search increases. *right*) The empirically measured effectiveness of forward processing (on random walks of length 2^{20}) is indeed the sum of the two factors.

5.2.1 The Time and Space Complexity of DAMP

The space complexity of DAMP is just the size of the original data, O(*n*). The worst-case time complexity is O(*n*log*n*) per datapoint ingested, the time required to do a full similarity search with MASS [10]. However, empirically, on diverse real-world datasets, more than 99.999% of the times we enter the loop in line 3 of Table 2 we will break out in the first iteration (line 12), making the algorithm effectively O(*m*log*m*) per datapoint ingested, and linear in the time series length.

5.2.2 DAMP Variants

The basic DAMP algorithm can be easily modified to handle more general cases, for example:

- The algorithm as explained in Table 1 is a *batch* algorithm. To make it an *online* algorithm, we simply must reduce the size of the

lookahead (Table 3, line 1) to the largest delay we are willing to accept (including possibly *zero* delay).

- The algorithm as explained in Table 1 computes the Left-*a*MP, however we can modify it to compute the classic Full-*a*MP. If the backward processing step reaches the beginning of the time series, instead of updating the *BSF*, we do the same type of iterative doubling search, but *forward* from the current index (not to be confused with forwards pruning search in Table 3). We have made this code available at [6], but we do not consider it further here, due to page limits.
- It may be useful to limit how far back the backward processing can look, essentially redefining anomalies as "*the subsequence with the maximum distance to any of the X subsequences before it*". We call this variant the *X-Lag-Amnesic DAMP*.
- Instead of searching an ever-growing amount of previously seen data in the BackwardProcessing step, we can search a fixed pool of explicit training data. For example, an engineer could curate a dataset that contains all the allowable behaviors for a manufacturing process (i.e., the "*golden batch*").

Other useful variants are possible [6].

6. EMPIRICAL EVALUATION

To make certain that our experiments are reproducible, we have built a website [6] that contains all the data/code used in this work. All experiments were conducted on an Intel® Core i7-9700CPU at 3.00GHz with 32 GB of main memory, unless otherwise stated.

There are two things one normally needs to establish to validate an anomaly detection algorithm.

- **Effectiveness**: Here we feel less of an obligation. As we noted in Section 2, there are at least one hundred independent papers that have used discords to solve a real-world problem and that discords are the only technique that seem to be able to discover anomalies that are not visually obvious (Figure 2, Figure 3 and Figure 4). Nevertheless, for completeness we will show examples in Sections 6.1 and 6.2 that further demonstrate the excellent effectiveness of discords in diverse domains, and in Section 6.3 offer a comparison to several deep learning-based methods.
- **Efficiency**: As this is the main contribution of the paper, here we will attempt the most ambitious set of anomaly detection experiments in terms of both throughput and scale.

6.1 Energy Grid Dataset

A consortium from Texas A&M and USC recently released a large dataset from decarbonized energy grids [17]. The dataset contains files representing three years of measurements of various metrics in sixty-six electrical zones in the continental USA. As Figure 9 suggests, each file represents eleven measurements, ten of which are *measured* (temperature, wind speed etc.), but one *computed* from the first principles of astronomy, the Solar Zenith Angle. The total size of this dataset is 12 GB, representing 2,174 years of data with 1,142,668,098 datapoints. As such, we believe that it is the largest real dataset ever searched for anomalies. This complete search took only 2.06 days.

As Figure 9 shows, most of the anomalies discovered do have a semantic meaning that can be traced.

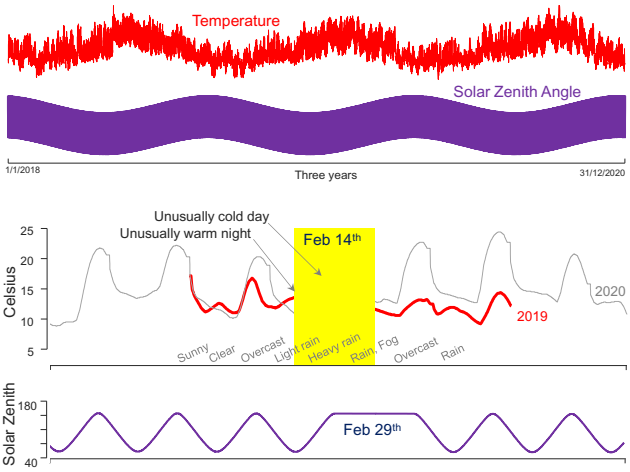

Figure 9: *top*) Two examples of time series from [17]. Most, like temperature are *measured*, but Solar Zenith Angle is *computed*. *bottom*) The two corresponding top discords in these datasets.

For example, a temperature trace from California had a discord that reflected "*Valentine's Day Storm Slams California*" [14]. Even the *computed* time series reveals a strange anomaly echoing a biblical event. Joshua persuades God to stop the sun from moving for a day "*There has never been a day like it before or since* (Joshua 10:14)". In our dataset there is a similarly unique day in which the sun apparently does not move! The reader will appreciate the cause of this anomaly, after noting it occurs on the 29th of February [15].

6.2 Machining Dataset

The previous example shows the utility of anomaly detection in data exploration. However, in some cases if we can do anomaly detection in real-time, we may be able to perform an intervention to improve an outcome. For example, consider the process of making parts using a CNC milling machine. Occasionally a problem arises where an item being machined is not held correctly and it moves. This can cause a milling machine to "crash" [5]. High-end CNC mills can cost over one million dollars, and crashes resulting in more than $20,000 in damage are known. Many (but not all) machining processes can be paused by an operator, so in principle it may be possible to stop a machine before it crashes. However, with the speed at which these machines operate, it is unlikely that the operators' reflexes would be fast enough.

This suggests the question, could we monitor the process with telemetry, and pause the process if we detected an anomaly? To test this, we recreated a common scenario in Figure 10.

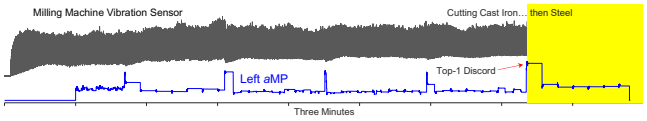

Figure 10: *top*) Vibration telemetry from a milling machine that was cutting cast iron, but then overshot to start cutting the steel jaws of the vice. *bottom*) The Left-*a*MP discovers the transition.

A common CNC programming error is to give the wrong coordinates for a cutting pass, and have the cutter overshoot the intended material to be machined, and inadvertently attempt to remove material from the jaws of the vice. Because the jaws are

typically harder than the material they hold, and more resistant to cutting, two things can happen:

- The milling cutter itself will break. This is a $20 to $200 error.
- A much worse possibility is that the cutter will move the vice. If it happens to push it into the path of later traversal, this could cause a head crash, which is a $2,000 to $20,000 error.

As Figure 10 shows, the *a*MP can detect the change of material, and this could be used to sound an alarm, or pause the machining process until the operator can inspect this.

Note that before the true anomaly there are other areas with high discord scores. They are when the milling cutter changes direction (from *Climb* milling to *Conventional* milling). Under our proposed scheme these would have a small cost, the process would pause until the operator visually confirms all is well, and hits *continue*.

6.3 Comparisons to Deep Learning

There are now dozens of competing deep learning anomaly detection (DLAD) algorithms. However, it is impossible to say which is the state-of-the-art. This is because, as Wu and Keogh have demonstrated, the amount of mislabeling in the benchmark datasets dwarfs the reported differences between algorithms [16]. It makes no sense to say that algorithm **A** is 5% better than algorithm **B**, when up to 30% of the ground truth labels are suspect.

To bypass this issue, here we will compare to just *Telemanom*. It is the most cited anomaly detection paper of the last five years [8], and several independent papers have also found it to be effective. The general idea of this work is to use LSTM to predict future values, then detect anomalies based on the difference between predictions and actual data. Can *Telemanom* detect the anomalies we consider in this work?

- **ECG** (Figure 3) **No**. Given the same 500 datapoint prefix as training data, it fails to find the anomaly. If we give it ten times as much training data (the first 5,000 datapoints), it *still* fails.
- **Bearing** (Figure 2): **Yes**. However, *Telemanom* took a total of (517.6 training + 700.4 testing) 1,218 seconds. This is two orders of magnitude slower than DAMP, which took 16.1 seconds. More importantly, *Telemanom* is an order of magnitude slower than real-time, precluding any possibility of online monitoring.
- **Energy Grid** (Section 6.1) **Maybe**. There are only *objective* labels for Solar Zenith Angle (this anomaly was discovered with DAMP but *confirmed* with the data creators). If *Telemanom* sees only the first week as training data (as DAMP did), then it only learns that the Solar Zenith Angle can decrease over time, and it will flag as anomalous anything that happens after the summer solstice. A solution to this problem is to allow *Telemanom* to train on the full first year, then test on the subsequent years. Then it *may* find the "Joshua" anomaly. However, this will take 59.1 hours, over 1,300 times slower than DAMP.
- **Milling Data** (Figure 10) **No**. Actually, *Telemanom* can detect the same anomaly as DAMP. But recall it can only start training when the first 5,000 datapoints arrive, and it takes 411 seconds to train the model. However, 127 seconds after it begins training, we encounter the anomaly, and about 21 seconds after that, the endmill snaps off. *Telemanom* is just too slow to be useful here.

These comparisons suggest that DLAD is not as accurate as DAMP, requires more training data, and is much slower.

To further see the limitations of deep learning anomaly detection, we can compare DAMP to DLAD algorithms on publicly available benchmarks. Wu and Keogh have shown that most benchmarks in this space are too trivial to be interesting, and in any case are plagued by mislabeling and other problems [16]. Instead, we consider the KDD Cup 2021 dataset consisting of 250 univariate time series. This archive was designed to be diverse, have a spectrum of difficulties ranging from easy to essentially impossible, and has a detailed provenance for each of the 250 datasets, giving us some confidence that the ground truth is correct. Table 4 shows the results.

Table 4: Accuracy and Time for Six AD Methods

Method	Accuracy	Train and Test Time
USAD [2]	0.276	8.05 hours
LSTM-VAE (ref at [6])	0.198	23.6 hours
AE (ref at [6])	0.236	6.11 hours
Telemanom [8]	*Out of memory error on longer examples*	
SCRIMP (Full-MP)	0.416	24.5 minutes
DAMP (Left-MP) out-of-the-box	0.512	4.26 hours
DAMP (Left-MP) sharpened data	0.632	4.26 hours

Once again, these results show that DAMP is more accurate and faster than deep learning-based methods. It is important to note that the results for DAMP are completely free of *any* human intervention or tuning. We use four hardcoded lines of Matlab (see Reproducibility Section) to find the approximate period in each training dataset, and used that as the value of m. Likewise, we simply hardcoded a *single lookahead* value for all 250 datasets. Further optimizing the former would improve accuracy and personalizing the latter for each individual problem would improve the speed. However, we wanted to show that even the most naïve out-of-the-box use of DAMP is highly competitive. As an example of a small intervention that can further improve accuracy, if we run DAMP on *sharpened data* (one line of code) the accuracy improves to 0.632.

The left-discords of DAMP are significantly more accurate than the full-discords computed by SCRIMP, because some anomalies have near "twin-freaks" that suppress the distance of the anomaly to its nearest neighbor. Note that the time for SCRIMP here is relatively good, as there are 250 *short* time series. In Figure 12 we will see that for longer time series this advantage of SCRIMP rapidly fails.

6.4 Threshold Learning for DAMP

Up to this point we have shown that DAMP can *locate* the most anomalous subsequence. However, we have not shown how to then make the binary decision to flag the subsequence as anomalous or not. To do so we simply need to learn a *threshold*. To demonstrate, consider the following experiment. We created 200 random walk time series of length one million. As shown in Figure 11.*top*, into half of them we randomly inserted a subtle anomaly, a low amplitude random section of length 950. In Figure 11.*left*, we show the top-1 discord score (for $m = 1{,}024$) for all 200 time series, divided into the two cases. This plot suggests that a threshold of 36.0 is the optimal value to maximize the accuracy on future occurrences. To test this, we created and tested an additional million examples, all of which are also of length one million, classifying an actual anomaly as a true positive if the correct location of the anomaly was discovered *and* the top-1 discord score was above the threshold. Figure 11.*right* shows the confusion matrix. We note in passing that this experiment (which took several days distributed across commodity laptops and desktops), trained on time series with a total length of 200 million, and tested on time series with a total length of 128 billion. To the best of our knowledge, this is the largest scale time series anomaly detection experiment ever conducted. Could deep learning do this? We estimate that *Telemanom* [8] would take

about twelve years to do this, although in practice it gives *out-of-memory* errors.

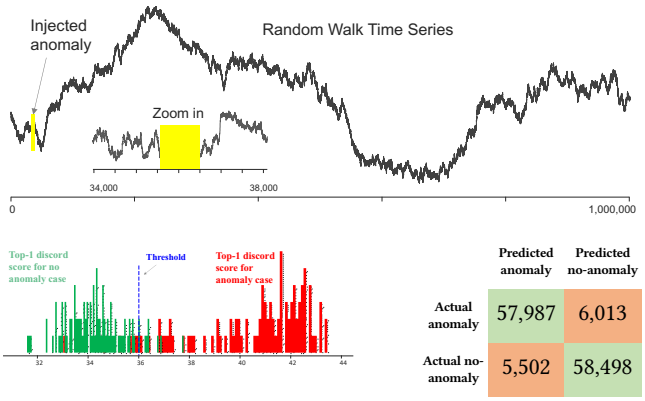

Figure 11: *top*) A sample random walk with an anomaly embedded. *left*) The distribution of top-1 discord scores for the two cases of interest. *right*) The confusion matrix for this task.

6.5 Scalability Comparisons

In order to understand which elements of our proposed approach are responsible for its efficiency, we have performed an ablation study in which in Figure 12.

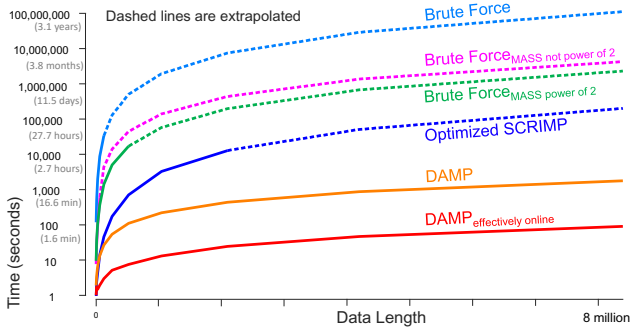

Figure 12: The CPU time vs time series length for various discord discovery algorithms. Note the Y-axis is in log scale.

It is clear that each element we proposed does actually contribute to speed up, and that DAMP is effectively linear in n. A recent paper pushed that envelope by considering a two million length ECG dataset [3]. In fact, these authors gave us the *exact* dataset they used, and helped us create a perfectly commensurate experiment, as shown in Figure 13. A real-time video trace of this experiment is at [6].

Figure 13: (Most of this figure is taken from [3], only the green elements are new). The scalability of various algorithms on increasing large subsets of a long ECG trace. All algorithms except DAMP are limited the first 2M data points by [3]. Note that the Y-axis is logarithmic.

Note that of the many approaches considered, some time out (i.e., are not finished in a 4-hour cutoff) at length 500K. In contrast, DAMP can handle 8-million points in just 22.3 seconds, this is 358,000 Hz.

6.6 Scalability and Stability of DAMP

Wu and Keogh have criticized the common benchmarks for time series anomaly detection [16]. We add one more criticism, the datasets considered are tiny. We conducted an experiment that required performing anomaly detection on time series with a total length of 1.648 trillion datapoints, using off-the-shelf hardware [6].

7. CONCLUSIONS AND FUTURE WORK

We introduce the left-discord anomaly detection framework, generalizing classic time series discords to the online case, and in the process solving the twin-freak problem. We believe that the throughput and scalability of DAMP will allow the community to address datasets and applications that are currently out of reach, and that this will open new challenges and research problems.

8. ACKNOWLEDGMENTS

This research was supported by NSF OIA-1757207, CNS-2008910 and RI-2104537, the French National Research Agency (ANR-19-P3IA-0002), and NSF 2103976, Mitsubishi, Visa and Toyota.

9. REFERENCES

[1] Aubet, F., Zügner, D. and Gasthaus, J. Monte Carlo EM for Deep Time Series Anomaly Detection. *arXiv preprint arXiv:2112.14436*.
[2] Audibert, J., Marti, S., Guyard, F. and Zuluaga, M.A., From Univariate to Multivariate Time Series Anomaly Detection. in *Advanced Analytics and Learning on Temporal Data*, (2021), Springer, 186-194.
[3] Boniol, P., et. al. Unsupervised and scalable subsequence anomaly detection in large data series. *The VLDB Journal*. 1-23.
[4] Case Western Reserve University Bearing Data Center. Accessed: Nov. 15, 2021. [Online]. csegroups.case.edu/ bearingdatacenter/home
[5] CNC Crashes. Video. (15 Feb 2018). Retrieved December 20, 2021 from https://youtu.be/t2tBtZCa7j4?t=205
[6] DAMP (2022). sites.google.com/view/discord-aware-matrix-profile
[7] Higham, Nicholas (2002). Accuracy and Stability of Numerical Algorithms (2 ed). ISBN: 978-0-89871-521-7
[8] Hundman, K., et al. Detecting Spacecraft Anomalies Using LSTMs and Nonparametric Dynamic Thresholding. in, (2018), SIGKDD, 387-395.
[9] Kirti, R. and Karadi, R. Cardiac tamponade: atypical presentations after cardiac surgery. *Acute medicine*, *11* (2). 93-96.
[10] Mueen, A., et. al. The fastest similarity search algorithm for time series under Euclidean distance. Retrieved January, 2022 from www.cs.unm.edu/~mueen/FastestSimilaritySearch.html
[11] Nakamura, T., Imamura, M., Mercer, R. and Keogh, E., MERLIN: Parameter-Free Discovery of Arbitrary Length Anomalies in Massive Time Series Archives. in, (2020), IEEE, 1190-1195.
[12] Neupane, D. and Seok, J. Bearing Fault Detection and Diagnosis Using Case Western Reserve University Dataset With Deep Learning Approaches: A Review. *IEEE Access*, *8*. 93155-93178.
[13] Thill, M., Konen, W. and Bäck, T., Time Series Encodings with Temporal Convolutional Networks. in *Bioinspired Methods and Their Applications*, (2020), Springer, 161-173.
[14] Wastewater News. *Valentine's Day Storm Slams California, Pushing Water Agencies to the Edge*. Retrieved Dec 1 2021 from www.news.cornell.edu/Chronicle/00/5.18.00/wireless_class.html
[15] Wikipedia. Leap year problem. Retrieved December 1, 2021 from https://en.wikipedia.org/wiki/Leap_year_problem
[16] Wu, R. and Keogh, E. Current Time Series Anomaly Detection Benchmarks are Flawed and are Creating the Illusion of Progress. *IEEE TKDE* (2021) 1.
[17] Zheng, X., et al. PSML: A Multi-scale Dataset for Machine Learning in Decarbonized Energy Grids. *arXiv:2110.06324*.
[18] Zhu, Y., et.al. Matrix profile XI: SCRIMP++: time series motif discovery at interactive speeds. in *2018 IEEE ICDM*, IEEE, 837-846.

Appendix A: Reproducibility

As we noted in the main text, to make certain that our experiments are reproducible, we have built a website [6] that contains all the data/code used in this work. Here we take advantage of the two pages available to highlight some of our reproducibility steps.

Some experiments (Section 6.4 and 6.6) make use of random numbers. We have provided the seeds and code to all users to reproduce all such data bit-for-bit.

In some places we omitted a discussion of the parameter m. This was done to enhance the flow of the paper. The reader will recall that m is the only parameter that affects the output of the algorithm (*lookahead* affects only the speed). We repair this omission here:

- Figure 2 (**Bearing**) m was 300. However, the results would be near identical for m in the range 100 to 500 [6].
- Figure 3 (**ECG$_{\text{Wandering Baseline}}$**) m was 150. However, the results would be near identical for m in the range 100 to 300.
- Figure 4 (**Mackey-Glass**) m was 40. However, the results would be near identical for m in the range 20 to 200 [6].
- Figure 5 (**ECG**), m was 150. However, the results would be near identical for m in the range 100 to 300 [6].
- Figure 9 (**Energy Grid Dataset**) m was 5760 (equivalent to four days of wall clock time). However, we can easily find the "Joshua" anomaly with m in the range 100 to 10,000 [6].
- Figure 10 (**Machining**) m was 16. However, the results would be near identical for m in the range 8 to 64 [6].
- Figure 11 (**Random Walks**) m was 1,024. Here we had to carefully tune the length on the embedded anomaly so that we did *not* get a perfect result each time.
- Figure 12 (**Long ECG**) m was 94 [6].
- Figure 13 (**Long ECG**) m was 94 [6].

Notes on Section 6.3: In Section 6.3 we suggested how long it would take deep learning to solve the task we considered in that section. We found that for *Telemanom* [3] training time is linear to the time series length with $R^2 = 0.9933$. Unfortunately, it runs out of memory on this task, but we trained it to n = 80,000 which took 3656.5 seconds. This suggests one million datapoints would take 12.7 hours to train, but recall we trained on 200 such examples, so the total training time would be about 105.8 days.

For *testing Telemanom* is also linear. We found that when processing the bearing dataset, which is of length 244,189, testing took 700.4 seconds, suggest a throughput of about 348.6 Hz. This suggests it would take about 11.6 years to process the 128 billion datapoints (of course, this could be done in parallel). The timing experiments for *Telemanom* can be found at [6].

The training time here is the biggest hurdle. The is a qualitative difference between a model you can train during a coffee break, and one that requires three months.

Notes on Section 6.4: In Section 6.4, we noted that we "*inserted a subtle anomaly, a low amplitude random section of length 950.*" We choose the odd length of 950, because we found that if we made the anomaly the same length at m (1024), the accuracy on the training set was 100%. We wanted to stress test our algorithm *and* have an experiment that others could improve upon.

Notes on Table 4: As we noted in the main text, the results of DAMP shown in Table 4 do not require any human effort. We use the following four lines of Matlab code to automatically learn the period for each data set and use it as the parameter m for DAMP.

```
[autocor,lags] = xcorr(T,'coeff');
[~,m] = findpeaks(autocor(length(T)+10:length(T)+1000),...
lags(length(T)+10:length(T)+1000),'SortStr','descend','NPeaks',1);
m(isempty(m))=1000;
m = floor(m);
```

The period is obtained by finding the peak of autocorrelation in the range of 10 to 1000 (the value of parameter m is limited to the range of 10 to 1000). To avoid the 'findpeaks' function returning a null value, we set the default value of m to 1000.

Although off-the-shelf DAMP has achieved an accuracy of 51.2%, significantly better than the best of the deep learning approaches, we noted that there are some simple "tricks" that can further improve its performance. We showed just one example if we use the following line of code to "sharpen" each dataset.

```
T=[normalize(T,'range')*( max(1,mean(std(T))))+1].^10;
```

The accuracy of DAMP will increase to 63.2%. The high-level idea of this approach is to apply a mathematical model with the same growth (but faster) to the original time series to "highlight" the anomaly in the time series.

Important General Note on using DAMP

There is an important thing to remember when viewing an *a*MP, as in the blue line in Figure 10.*bottom*. Failure this understand this may lead a user to think the *a*MP is indicating an anomaly where there is none.

When you search for the top-k left-discords, the k highest peaks *do* correctly show the location and strength (the height of the peaks) of the top-k left-discords. However, the remaining peaks in the *a*MP should not be assumed to indicate slightly smaller anomalies. They may indicate slightly smaller anomalies, but they also simply indicate regions that were pruned by encountering a matching subsequence that was just below the current *Best-So-Far*.