Data Mining and Knowledge Discovery (2023) 37:627–669
https://doi.org/10.1007/s10618-022-00911-7

DAMP: accurate time series anomaly detection on trillions of datapoints and ultra-fast arriving data streams

Yue Lu[1] · Renjie Wu[1] · Abdullah Mueen[2] · Maria A. Zuluaga[3] · Eamonn Keogh[1]

Received: 18 June 2022 / Accepted: 15 December 2022 / Published online: 11 January 2023

Abstract

Time series anomaly detection is one of the most active areas of research in data mining, with dozens of new approaches been suggested each year. In spite of all these creative solutions proposed for this problem, recent empirical evidence suggests that the *time series discord*, 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. We will further show that our speedup improvements can be applied in the multidimensional case.

Communicated by Johannes Fürnkranz.

✉ Yue Lu
ylu175@ucr.edu

Renjie Wu
rwu034@ucr.edu

Abdullah Mueen
mueen@cs.unm.edu

Maria A. Zuluaga
maria.zuluaga@eurecom.fr

Eamonn Keogh
eamonn@cs.ucr.edu

1 University of California, Riverside, Riverside, USA

2 Department of Computer Science, University of New Mexico, Albuquerque, USA

3 Data Science Department, EURECOM, Sophia Antipolis, France

Keywords Time series · Anomaly detection · Streaming data

1 Introduction

Time series anomaly detection is one of the most important and widely used tools investigated by the data mining community (Audibert et al. 2021; Hundman et al. 2018; Nakamura et al. 2020). It can be applied offline to investigate archival data, or online, to monitor critical situations where real-time human intervention is possible. For example, by 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 (Nakamura et al. 2020). 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 Sect. 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 searches. These other occurrences can be clustered, average, or otherwise summarized as appropriate.
- 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 Sect. 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 required, and Sect. 4 discusses related

work, before we introduce our algorithm in Sect. 5. In Sect. 6 we conduct the most ambitious empirical evaluation of time series anomaly detection ever attempted.

2 Motivation

Before we continue, it is necessary to answer the following question. Why do we attempt to fix discord's scalability issues instead of inventing a new algorithm, or making one of the many dozens of more recently proposed methods more scalable?

The reason is that there is increasing evidence that discords remain competitive with the state-of-the-art[1] (Nakamura et al. 2020). 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. For example, a group of climatologists at France's UMR Espace-Dev laboratory uses discords to find anomalies in climate data (Khansa et al. 2012). A team of researchers at NASA's JLP lab have applied discord discovery to planetary data, noting that "(discords) *detect Saturn bow shock transitions well*" (Daigavane et al. 2022). A group based in Halmstad University created a tool called IUSE for applying discord discovery to industrial datasets. One of their first applications was to a City Bus Fleet dataset, where they noted that the discords discovered did indeed have an objective meaning "*The discords in this case primarily consisted of significant drops of pressure … likely correspond to the drainage of the wet tank*." (Nilsson 2022). Finally, a team of researchers at the National Renewable Energy Laboratory, in Golden, Colorado, have used discords to find anomalies in a large building portfolio, showing that they could discover anomalies with diverse causes caused by both "*internal (occupant behavior) and external factors (weather conditions)*." (Park et al. 2020). There are several other time series anomaly detection algorithms that are well cited (Hundman et al. 2018; Su et al. 2019), but most of the citations are from rival methods comparing these algorithms on a handful of benchmarks (Wu and Keogh 2021). It is not clear that anyone actually uses these algorithms to solve real-world problems, as a detailed literature search does not produce any examples.

In addition, time series discords seem to be the only anomaly detection algorithm that has been demonstrated to perform at superhuman levels (Nakamura et al. 2020). 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 a LSTMs network for anomaly detection and evaluated it on data retrieved from Mars (Hundman et al. 2018). 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. For example, in Nakamura et al. (2020), their Figs. 8 and 9 both seem to meet that criterion. For completeness, we will show some additional examples. Consider Fig. 1,

[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.

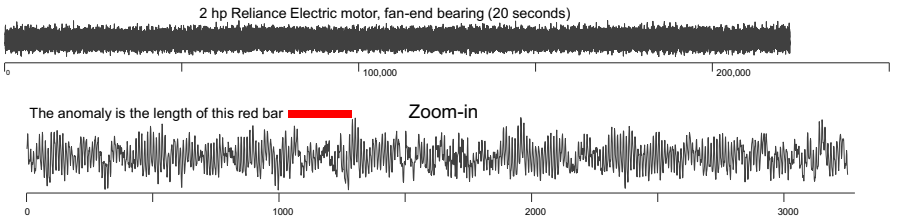

Fig. 1 *Top* A 20-s run of an industrial motor. *Bottom* A zoom-in of the region known to contain an anomaly, which is the length of (but not necessarily at the location of) the red bar. Here $m = 300$

which shows the vibration of an industrial motor (Case Western Reserve University Bearing Data Center 2021; Neupane and Seok 2020).

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 Fig. 2 we task time series discords with detecting the anomaly.

Beyond the accuracy of discords prediction here, note that this dataset contains 244,189 datapoints, representing about 20 s 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 Fig. 3 we show the Left-MP for an ECG which we know contains a single anomaly beat, a *ventricular contraction*.

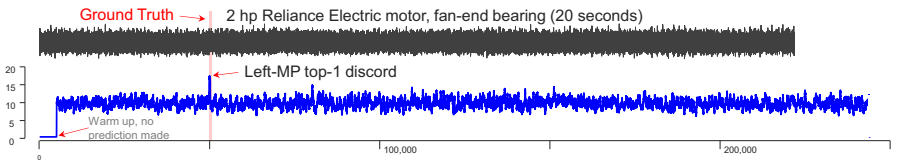

Fig. 2 *Top* A 20-s run of an industrial motor. *Bottom* The time series discord discovered by the Left-MP correctly locates the anomaly. Note that higher values are more anomalous. Here $m = 300$

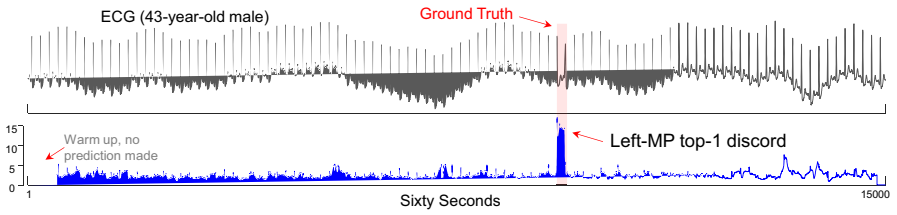

Fig. 3 *Top* A sixty-second snippet of an ECG. *Bottom* The top-1 time series discord correctly locates the anomaly. Here $m = 150$

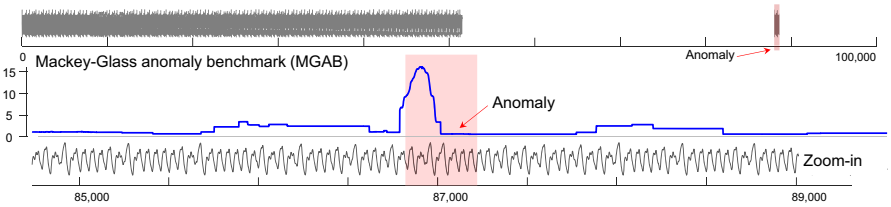

Fig. 4 *Top* The MGAB dataset was built to defy visual discovery of anomalies. *Bottom* The Top-1 time series discord correctly locates the anomaly. Here $m = 40$

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 noted cardiologist Dr. Gregory Mason says is on the cusp of his ability to detect by eye.

Finally, in Fig. 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*" (Thill et al. 2020). Here again discords are superhuman.

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

Let us take a moment to make clear what the terms *batch* and *online* mean. If we are tasked with finding the top-k anomalies in a batch setting, we have random access to all data. For example, we could initially define April 1st as an anomaly, but when we later see data from say the summer months, we can change our mind, revisit April 1st, and reduce its anomaly score. For that matter, we could revisit April 1st, and *increase* its anomaly score. In contrast, in the *online* case we see the data incrementally arrive and must make an irrevocable decision as to the appropriate anomaly score. When recording this score, we do have access to all the data previously seen, but clearly we cannot see any future data. For some time series anomaly detection algorithms this distinction is important, and the algorithm can give different answers in the two settings. However, as we will show, the algorithm we propose in this work will produce the exact same answers in either setting.

Now that the terms *batch* and *online* are clear, it is helpful 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 (Kirti and Karadi 2012). 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:

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 s, and for telemetry arriving from devices with a slow cycle rate, say the daily periodicity of pedestrian traffic, the permissible 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 ultrafast-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: $T = [t_1, t_2, ..., t_n]$ 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. $T_{i,m} = \left[t_i, t_{i+1}, \ldots, t_{i+m-1}\right]$, $1 \leq i \leq n-m+1$.

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 ith 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* (Zhu et al. 2018).

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 Fig. 5.

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).

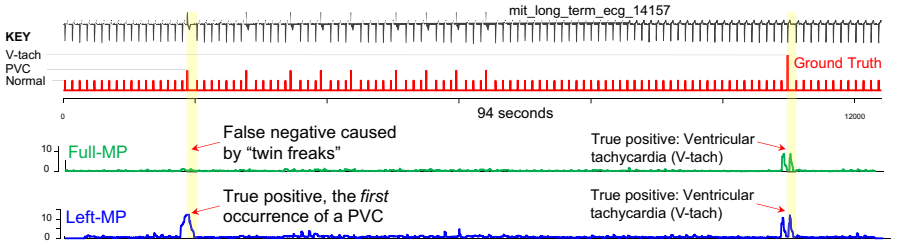

Fig. 5 *Top to bottom* A snippet of ECG with two types of anomalous heartbeats indicated by a ground truth vector. A full Matrix Profile can find the sole occurrence of V-tach, but is confused by the multiple occurrences of Premature Ventricular Contractions (PVCs), i.e., twin-freaks, and cannot find them. In contrast, the Left-MP flags the first occurrence of a PVC and the first (and only) V-tach. Here $m = 150$

4 Related work

In recent years, there has been a surge of research interest in the topic of time series anomaly detection. For a detailed review, we refer the interested reader to Aubet et al. (2021), Audibert et al. (2021), Boniol et al. (2021a), Hundman et al. (2018), Nakamura et al. (2020), Thill et al. (2020) and the references therein. In addition to the work listed in Sect. 2, we have also compiled a longer annotated biography at (DAMP 2022) that explicitly discusses discords.

There are two important points that we have gathered from our survey of the literature. The first is due mostly to a single paper (Wu and Keogh 2021), 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 four issues that (Wu and Keogh 2021) notes with benchmarks datasets, we wish to add another issue. Most of these benchmarks are minuscule. We suspect that the small datasets that the community has focused on are at least partly due to the poor scalability of current approaches. For example, a recent paper examines time series of length 140,256 and notes "*Given the length of the dataset, we sub-sample it by a factor 10*." (Aubet et al. 2021). This paper is by a research group at Amazon, who presumably does not lack for computational resources. For reference, it takes our proposed algorithm 0.9 s on the *full*-sized version of this dataset (DAMP 2022) on a commodity desktop.

In addition to the problems caused by using poor quality benchmarks, a recent paper suggests yet another compelling reason why much of the recent apparent success of recent research efforts should be viewed with caution. Paper (Doshi et al. 2022) notes that "*most recent approaches employ an inadequate evaluation criterion leading to an inflated F1 score. (however) a rudimentary Random Guess method can outperform state-of-the-art detectors in terms of this popular but faulty evaluation criterion*.".

A recent SIGKDD workshop keynote makes a related point about evaluation (Keogh 2021). Suppose you have a year of data monitoring an industrial boiler, and it happens that on Christmas, the boiler leaks all day, causing an anomaly. One might imagine the best way to evaluate an algorithm on the task of discovering this anomaly would be a binary score, success/failure. However, many papers essentially consider each

datapoint as if it was an independent event. Suppose they predicted all of Xmas day, and the first minute of the next day was an anomaly. They would report an F1 score of 0.9997. The four significant decimal digits imply some extraordinarily careful and significant measurement was made. However, with a little introspection will allow the diligent reader to see that this precision is unwarranted and misleading. The Time Series Anomaly Detection (TSAD) literature is replete with impressively large tables of numbers with four (and sometimes, five or six!) digits, that simply give the illusion of progress and rigor.

It is somewhat surprising that so few papers in the literature discuss time complexity. This can possibly also be attributed to issues with the benchmark datasets. For example, by far the two most discussed datasets in the literature are Yahoo and NY-Taxi (NAB), with lengths of 1200 and 10,321 respectively. Even the most sluggish of algorithms are unlikely to be taxed by such tiny datasets. If building a particular highly-quality anomaly detection algorithm had a high *one-time* cost, then we might be willing to throw whatever computational resources are needed at the task, and then deploy the model in perpetuity. However, the situation is worse than that. In virtually any domain, the model will become stale due to concept drift, and need to be periodically retrained, either on a regular schedule (say once a week), or when the model detects that it has drifted from the newly arriving data.

Recently a handful of papers have recognized that the slow training times for deep learning anomaly detectors can be an issue. For example, (Truong et al. 2022) notes that "*fast training times* (are needed) *to cope with the requirement of frequently re-updating the learning model*". These authors then went on to introduce a "light-weight" anomaly detection system that can complete training in as little as twenty minutes (using GPUs) in a dataset of size 274,627. This kind of time frame may work for some domains, for example the three-year-long energy grid/weather data we consider in Sect. 6.1. We surely could afford a few hours to build the model, and perhaps a few hours at the end of each month to retrain it. However, consider the machining dataset we examine in Sect. 6.2. Here we see the first thirty seconds of data, and then must *instantly* have a working model. While DAMP *can* do this, it is not clear that any other anomaly detector in the literature can. One might imagine that other methods could potentially look only at say, the first twenty seconds of data, and use the remaining ten seconds to build their model. However, this would require most of the algorithms in the literature to be accelerated by several orders of magnitude.

Finally, the reader may wonder why we do not test on the large collection of datasets during (Paparrizos et al. 2022) in our empirical section. There are two reasons. First, the data collection includes datasets that (Wu and Keogh 2021) notes are deeply flawed, including mislabeled ground truth. If a significant fraction of the datasets have mislabeled ground truth, as Wu and Keogh point out (Wu and Keogh 2021), and which the authors of Paparrizos et al. (2022) have acknowledged (Palpanas 2022), it is hard to have any faith in evaluation on the overall data collection. For at least some of the datasets in this collection, including NAB-NYTaxi, NASA-MSL (trace G-1), YAHOO (A1-real46), it is known that at least 50% of the ground truth labels are incorrect (Wu and Keogh 2021). With that amount of mislabeling, it would be fruitless to claim that one algorithm is superior to another because it was say 6.3% better than another. In

any case, testing on small synthetic or unrealistic datasets seems pointless when we can test on large real datasets, as we do in this work.

In Table 9 we will compare to several rival methods. We refer the interested reader to the original papers for more detailed descriptions, but below we present a terse description of these rival methods.

An Auto-Encoder (AE) is a neural network architecture consisting of a combined encoder and decoder (Audibert et al. 2021). The encoder maps the input windows into a set of latent variables, while the decoder maps the latent variables back into the input space as a reconstruction. The difference between the input window and its reconstruction is the reconstruction error. The AE learns to minimize this error. The anomaly score of a window is the corresponding reconstruction error. A window with a high score is considered abnormal.

The Unsupervised Anomaly Detection (USAD) (Audibert et al. 2021) extends the AE concept and constructs two AEs sharing the same encoder. The architecture is driven in two phases. In the first phase, the two AEs learn to reproduce the normal windows. In the second phase, an adversarial training teaches the first AE to fool the second one, while the second one learns to recognize the data coming from the input or the reconstructed by the first AE. The anomaly score is the difference between the input data and the data reconstructed by the concatenated AEs.

Long Short-Term Memory Variational Auto-Encoders (LSTM-VAE) uses an LSTM to model temporal dependency (Park et al. 2018), whereas the VAE projects the input data and its temporal dependencies into a latent space. During decoding, the latent space representation allows to estimate the output distribution. An anomaly is detected when the log-likelihood of the current data is below a threshold. LSTM-VAE's have the capacity to identify anomalies that span over multiple time scales (Park et al. 2018).

Telemanom is a Long Short-Term Memory (LSTMs) networks, a type of Recurrent Neural Network (RNN) (Hundman et al. 2018). Once model predictions are generated, we offer a nonparametric, dynamic, and unsupervised thresholding approach for evaluating residuals.

NORMA can be thought of as a variant of a Golden Batch Matrix Profile, which uses a clustering preprocessing step to compact the training data into a small, therefore quickly searched, reference dataset (Boniol et al. 2021a).

SCRIMP is a fast method to compute the classic Matrix Profile.

5 DAMP

5.1 Intuitive overview of DAMP

Before giving a formal explanation of our algorithm, we will first provide an intuitive description of how it works. We will start with discussing the batch case and then further generalize to the (minor) steps required for the online case. As shown in Fig. 6, it will be helpful to explain the algorithm mid-execution, as it is processing the subsequence T_i.

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

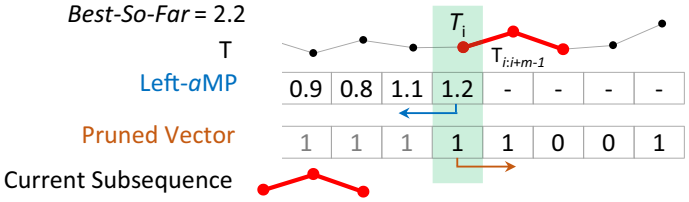

Fig. 6 A sketch of the DAMP algorithm in progress, processing the current subsequence. *top*) The time series T. *center*) The Left-*a*MP, its values between 1 and i are computed, its values after i have yet to be computed. *bottom*) the Pruned Vector indicates subsequences that can be ignored without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

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: $\text{MP}_j \leq aMPj \leq \max(\text{MP}_{1:j})$

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 Fig. 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 ith location, the processing can be divided into two independent steps, *backward* processing and *forward* processing.

5.1.1 Backward processing

The main task of backward processing is to discover whether the current subsequence $T_{i:i+m-1}$ is the left-discord, for which the naïve way would be to compute its nearest neighbor distance to any subsequences 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 ith location. The reason this is true is that 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.[2]

- 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 s 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 s. 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

In the forward processing step, we attempt to discover and prune subsequences that cannot be 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 the 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 give the pseudocode shown in Table 1 to formalize the intuition of the previous sections. 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,

[2] This observation is true for heartbeats, gaits, machine cycles etc. One exception is for events tied to a cultural calendar. For example, for taxi demand or electrical power demand, the most similar day to any given day, is not the previous day, but the same day one week earlier.

Table 1 The main DAMP algorithm

```
Function:  DAMP(T, m, spIndex)
Input: T:   Time series
       m:   Subsequence length
       spIndex: 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 = spIndex to length(T) - m + 1
6      If NOT PVi          // Skip the pruned subsequence
7         aMPi = aMPi-1
8      Else
9         [aMPi, BSF] = BackwardProcessing(T, m, i, BSF)
10        PV = ForwardProcessing(T, m, i, BSF, PV)
11  return aMP
```

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.

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 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
```

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 Sect. 5.1.1.

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 store it in the variable *prefix*. We employ 2^nextpow2(m) to define this initial length. Specifically, when we feed 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

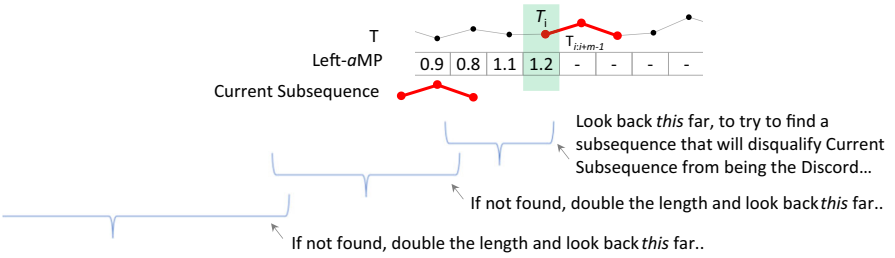

Fig. 7 A visualization of the *iterative doubling* search policy used in lines 10–14 of Table 2. See also Fig. 6

length of the backward processing and continue the search in the next iteration (lines 13–14). This idea is visualized in Fig. 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). 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 Sect. 5.1.2.

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 the MASS function.

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
8                                       //profile to indices on time series
9  PV_indices = 0                    // Update the Pruned Vector
10 return PV
```

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 Fig. 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. 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 similarity searches. As Fig. 8 *right* shows, this intuition is borne out by experiment. The height of the base of the "U" can be lower (smooth and highly periodic data) or higher.

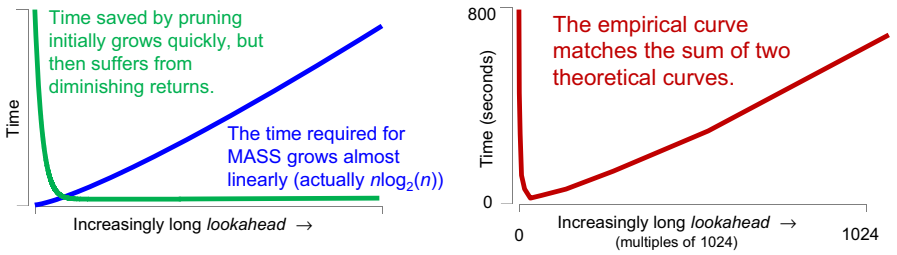

Fig. 8 *Left* The theoretical *lookahead* tradeoff is based on two factors. As the *lookahead* grows, the pruning rate becomes greater, but the cost of the similarity search increases. *Right* The empirically measured effectiveness of forward processing (on random walks of length 2^{20}) is indeed the sum of the two factors. Here $m = 1{,}024$

Finally, this is a good place to mention an important caveat about interpreting a Left-*a*MP that is computed using forward processing. Failure to understand this caveat may lead a user to think the *a*MP is indicating an anomaly where there is none. Consider Fig. 9 which compares the results of Brute Force and DAMP with and without forward processing for a dataset from the KDD Cup 2021.

As can be seen from Fig. 9, the discord scores calculated by different approaches look different for the same data set. So how should we interpret these results?

First, the Brute Force illustrated by the blue curve is identical to the one shown in Fig. 22. It does not have forward and backward processing, and for each iteration, it searches from the current position to the beginning of the time series. Therefore, each value on Left-MP generated by Brute Force is an exact value. That is, for any of the peaks on Left-MP, there could be physical meaning that we can interpret.

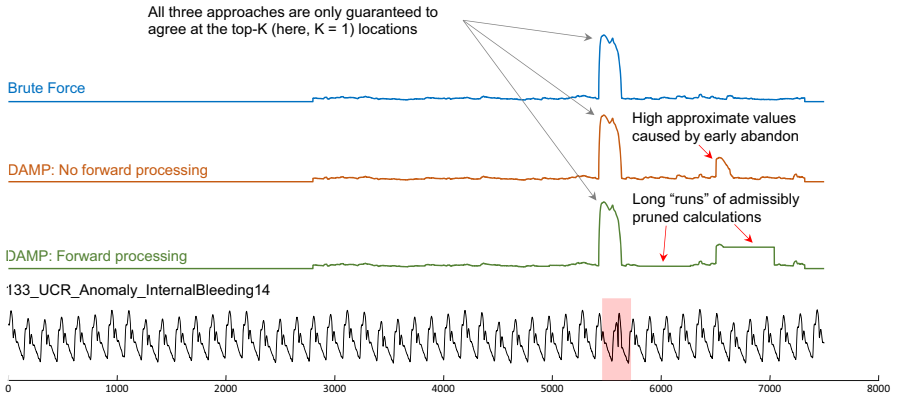

Fig. 9 *Bottom-to-top* top-1 Left-*a*MP computed with forward processing. The top-1 Left-*a*MP computed without forward processing. The Left-MP computed without backward and forward processing. Both top-1 Left-*a*MPs have a secondary peak at around 6500, which seems to indicate an anomaly, whereas in fact they are caused by early abandon and are not meaningful. The top-1 Left-*a*MP computed with forward processing produces long constant runs that indicate that the algorithm admissibly skipped those regions. Here $m = 180$

By contrast, when using both DAMP algorithms to 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 (in Fig. 9, $k = 1$). However, the remaining $k + 1$ peaks should not be assumed to indicate slightly smaller anomalies. This is because both DAMP methods perform the iterative doubling backward search, yielding either approximate or exact discord scores. Whether the score is exact or approximate depends on two cases, which we detailed in Sect. 5.2 and will not repeat here. In brief, for most iterations of DAMP, the backward processing is terminated before it reaches the beginning of the time series, thus producing approximate scores on Left-aMP that are larger than the exact scores. Therefore, most peaks on Left-aMP except for the top-k ones are probably "false positives" due to this early abandoning scheme. For example, the Left-aMPs represented by the orange and green curves in the figure both have a secondary peak at around 6,500, which seems to indicate an anomaly at that location; however, by comparing it to the Left-MP results shown by the blue curve, it is clear that the scores at around 6,500 are actually below average. Thus, these secondary peaks cannot be interpreted as anomalies.

Further, we can observe a lot of piecewise constant regions on the Left-aMP generated by DAMP using forward processing, i.e., the green curve. They simply indicate regions that were pruned by encountering a matching subsequence that was below the current *Best-So-Far* and had no practical meaning. For example, at the end of the green curve, there are two long constant plateaus, one of which has a relatively high value. As we can see by comparing that region to the corresponding region in the topmost blue curve, we should not assume that there are any anomalies in that region.

Again, to summarize: The top-k peaks of the top-k Left-aMP should be interpreted as having the correct values of top-k discords of T, but the remaining values of the top-k Left-aMP have no meaningful interpretation.

5.2.1 The time and space complexity of DAMP

Since all computation results are stored in a one-dimensional vector of size n, 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 (Mueen et al. 2017). 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. Figure 24 shows this linear assumption strongly holds up to at least $n = 2^{30}$.

5.3 DAMP variants

There are more general cases that can be easily handled by modifying the basic DAMP, for example:

- The algorithm as explained in Table 1 is a *batch* algorithm. To make it an *effectively 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 forward pruning search in Table 3). We have made this code available at (DAMP 2022), 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*").

There are several other useful variants that we have considered, and we suspect the community will quickly exploit the scalability of the basic DAMP algorithm to invent further variants.

Below we give more details about the two useful variants of DAMP, *X*-Lag-Amnesic DAMP and Golden DAMP mentioned above. To help the reader better understand how these two variants work, let us start with the most basic variant, namely, *Classic DAMP*.

The Classic DAMP algorithm illustrated in Fig. 10 *top* was already discussed in Sects. 5.1 and 5.2. It is worth noting here that for Classic DAMP, all data collected before the current time $T_{1:i-1}$ are our training data by default, and our backward search is executed on this progressively growing training data. This means that to calculate the discord score of the current subsequence $T_{i:i+m-1}$, Classic DAMP searches all the way forward from position i by the iterative doubling process, and, in the worst case, all the way to the beginning of the time series, i.e., $T_{1:i}$. Therefore, as we process more

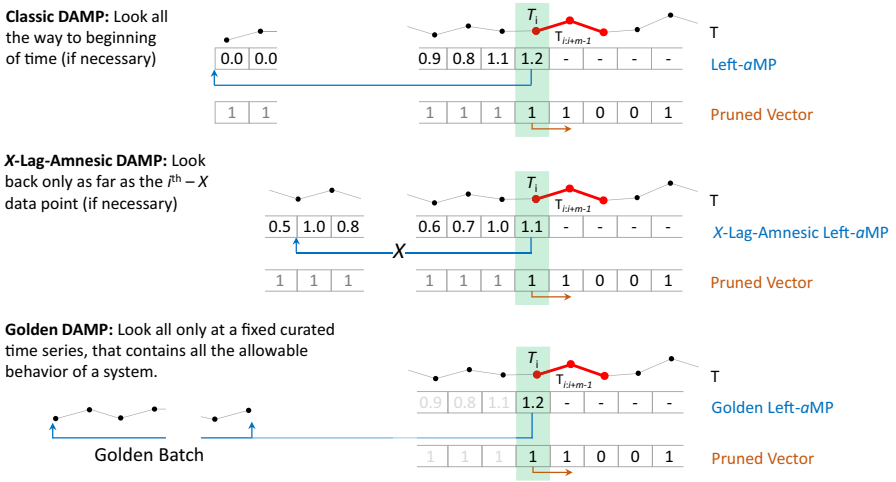

Fig. 10 Three variants of DAMP. *Top* Classic DAMP *Middle X*-Lag Amnesic DAMP *bottom*) Golden DAMP

and more data points over time, our backward search may also require more and more time.

As we shall see in our experimental section, empirically this is not a problem on the dozens of datasets we consider. Nevertheless, X-Lag-Amnesic DAMP and Golden DAMP allow us to provide a strict bound on the worst-case behavior, in addition to possessing other useful properties.

5.3.1 X-lag-amnesic DAMP

In some settings we may require an algorithm that can show us the most unusual behavior in *just* the last few minutes, days, months, or years. In that case, a DAMP variant that constrains how far back the backward search can look is required. Formally, we refer to such a DAMP variant as *X-Lag-Amnesic DAMP*.

Compared with Classic DAMP, the time overhead of X-Lag-Amnesic DAMP is bounded and controllable. This is because it only cares about what happened in a fixed unit of time before the present, and its calculation is based on fixed-size and real-time updated training data. For example, if we only need to find anomalies that occurred in the most recent month, X-Lag-Amnesic DAMP will perform an iterative doubling search in the most recent month's data rather than searching through all past data. Consequently, the time cost of X-Lag-Amnesic DAMP is bounded by the length of X as opposed to increasing gradually.

In addition, X-Lag-Amnesic DAMP can better deal with concept drift. For time series in some domains, their patterns change over time and the dependence between their data weakens as the distance increases, at which point it makes no sense to consider data that is too far from the present. For example, for many batch processes in the food and beverage industry the time series patterns are known to drift over each day, due to changes in ambient temperature and humidity. A pattern that happens during the nightshift may be anomalous because the process is "running hot". That is to say, it is exhibiting behaviors that would be normal if it was in the middle of a hot day, but these behaviors are anomalous given that they are observed in the cool of the night. It might be obvious if we compare only to the patterns in the previous hour or so, but it will not be obvious if we allow comparisons back to the previous midday. Obviously, since X-Lag-Amnesic DAMP focuses only on what happened recently, it can avoid such issues caused by concept drift. By contrast, Classic DAMP is more vulnerable to this, as its backward search may cover all data that occurred before the present, and all these data have the same weight for the discord score calculation regardless of their proximity to the current subsequence.

Figure 10 *middle* describes how the X-Lag-Amnesic DAMP works. Here we introduce a new parameter X, the maximum length that the backward processing algorithm can look back, specified by the user as needed. The framework of the X-Lag-Amnesic DAMP algorithm is the same as Classic DAMP; it retains the forward and backward processing steps, in which the forward processing is identical to Classic DAMP. The only difference between X-Lag-Amnesic DAMP and Classic DAMP is that for the current subsequence being processed $T_{i:i+m-1}$, we only perform a backward search on the X data points before it, not on all the previous data. However, the search is still iteratively doubled: it terminates either when it finds the nearest neighbor with a

Table 4 Pseudo code snippet for X-Lag-Amnesic DAMP

```
1   If Starting position of the search < max(i-X,1) Or X < prefix
2       If i - X < 1
3           aMP_i = min(MASS(T_1:i, T_i:i+m-1))
4       Else
5           aMP_i = min(MASS(T_i-X:i, T_i:i+m-1))
```

distance smaller than the *BSF* or when it reaches the beginning of X. Therefore, to make X-Lag-Amnesic DAMP work, we simply need to change lines 4–5 of Table 2 for Classic DAMP to the five lines shown in Table 4.

In line 1 we added two new criteria for search termination, i.e., reaching the beginning of the time series $T_{i-X:i}$, or the maximum length of looking back X is less than the initial length of the iterative doubling search *prefix*. In both cases, we do not iteratively double our search anymore. We have reached the limit of the history we think should inform our decision. Instead, we only search for the nearest neighbor of the current subsequence in the range $i-X$ to i (lines 4–5). Moreover, there is a special case where the number of data points that arrived has not yet reached X ($i < X + 1$). In this case, we can only conduct the backward search in all available data $T_{1:i}$ as shown in lines 2–3.

Others works have noted the utility of amnesic anomaly detection (although not using that phrase), including the SAND algorithm of Boniol et al. (2021b). However, SAND requires significant effort to build a reference dataset, and the setting of several unintuitive parameters.

5.3.2 Golden DAMP

Recall that Classic DAMP has a parameter called *spIndex*, which sets the location of the split point between the training and test data in the initial state. When Classic DAMP processes a time series, it assumes that the data before *spIndex*, $T_{1:spIndex-1}$ are normal, which may lead to three issues. First, this causes the algorithm to ignore the potential anomalous behavior present in $T_{1:spIndex-1}$, resulting in certain false-negative results. Second, this approach may have the algorithm wasting time searching redundant data. It is possible that the patterns in $T_{1:spIndex-1}$ are highly redundant, such as 1000 heartbeats that are essentially identical. If the heartbeats all have the same pattern, it would suffice for the algorithm to take just one of them to learn[3]; there is no need to consider the same pattern 1000 times, which will waste a lot of time. Further, it may be difficult for $T_{1:spIndex-1}$ to contain every normal pattern, which can cause the algorithm to incorrectly identify normal behavior that does not appear in $T_{1:spIndex-1}$ as an anomaly. For example, if $T_{1:spIndex-1}$ only contains data on the solar zenith angle during the day, the algorithm may incorrectly identify normal solar zenith

[3] Actually, using exactly one heartbeat (or *pattern* more generally), may make the downstream algorithms brittle to the choice of the starting point of the heartbeat. To bypass this issue, we always extract two consecutive beats.

angles at night as anomalies. These potential problems can undermine the accuracy and efficiency of the algorithm.

Golden DAMP is our proposed solution to the above three problems. It processes each subsequence not by referring to information that occurred before the current time, but to user-defined, curated, out-of-band information, denoted as *Golden Batch*. The Golden Batch implicitly defines every possible legal behavior, such as every possible dance move, every normal heartbeat, etc. It includes all the things the user expects to happen in the system. With this correct and comprehensive priori knowledge, the algorithm will be able to make more accurate and efficient decisions.

This idea of creating a curated collection of data that spans the space of all possible acceptable behaviors is well known in the process industry (Yeh et al. 2019). For example, food/beverage engineers will often set aside one day to create a recipe under all combinations of conditions encountered: under cool conditions, under hot conditions, with carbonated infeed, with flat infeed etc. However, the use of these batch profiles is typically human comparison of the evolving process to the Golden Batch(es) (Yeh et al. 2019). Here we are interested in automatic anomaly detection. In addition, note that while the Golden Batch data can be hand curated, it can also be created automatically by various numerosity reduction algorithms (Imani et al. 2020; Yeh et al. 2021).

Further note that the execution time of Golden DAMP is also bounded because its training data is the Golden Batch with a fixed size. Therefore, as we explained in Sect. 5.3.1, the cost of Golden DAMP's backward search is proportional to the size of Golden Batch.

Figure 10 *bottom* illustrates the idea of Golden DAMP. When processing the current subsequence $T_{i:i-m+1,}$ Golden DAMP no longer looks backward in the time series T but toward the Golden Batch, a vector containing all acceptable patterns. We still use the iterative doubling search policy shown in Fig. 7 for Golden Batch. The search keeps iteratively doubling until it finds the nearest neighbor within the *prefix* whose distance from $T_{i:i-m+1}$ is less than the BSF, or it gets to the beginning of the Golden Batch. After computing the approximate or exact discord score for position i, we invoke the same forward processing procedure as in Classic DAMP to disqualify future subsequences that are unlikely to become a discord.

The implementation details of Golden DAMP are given in Tables 5 and 6. Since most of them are the same as Tables 1 and 2, we will highlight the parts that we changed.

The main framework of Golden DAMP is shown in Table 5. Golden DAMP has a new input, *GoldenBatch*, a long vector that joins all normal patterns together. As with Table 1, the algorithm starts with initialization in lines 1–3. Since we already have the training data *GoldenBatch*, we no longer need to use the first *spIndex*-1 data of the time series T. As a result, in line 5 we adjust the processing range of Golden DAMP from $T_{spIndex:n-m+1}$ to $T_{1:n-m+1}$. After that, within the loop, lines 6–7 decide whether to process the current subsequence $T_{i:i-m+1}$ according to the value in the pruned vector PV. If the subsequence at position i needs to be processed, we first invoke *BackwardProcessing* in line 9 to calculate the discord score for position i and update the current highest discord value, and then call *ForwardProcessing* in line 10 to determine the subsequences to be pruned in the future. Finally, lines 5–10 iterate

Table 5 The main golden DAMP algorithm

```
Function:  Golden_DAMP(T, m, GoldenBatch)
Input:  T:   Time series
        m:   Subsequence length
        GoldenBatch:     A long time series with all possible normal
        patterns
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 = 1 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, GoldenBatch)
10       PV = ForwardProcessing(T, m, i, BSF, PV)
11 return aMP
```

Table 6 Golden DAMP backward processing algorithm

```
Function:  [aMP_i, BSF] = BackwardProcessing(T, m, i, BSF, GoldenBatch)
Input:  T:   Time series
        m:   Subsequence length
        i:   Index of current query
        BSF:             Highest discord score so far
        GoldenBatch:     A long time series with all possible normal
        patterns
Output: aMP_i:Discord value at position i
        BSF:       Updated highest discord score so far
```

```
1  aMP_i = inf
2  prefix = min(2^nextpow2(m),length(GoldenBatch))
3  While aMP_i ≥ BSF
4     If the search reaches the beginning of the Golden Batch
5        aMP_i = min(MASS(GoldenBatch_1:end,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(GoldenBatch_end-prefix+1:end,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
```

through each subsequence in $T_{1:n-m+1}$ and line 11 returns the Golden Left-*a*MP. In particular, the *ForwardProcessing* here is identical to that of Classic DAMP, so we do not repeat it below. However, we partially changed *BackwardProcessing* from Table 2 of Classic DAMP, so we give Table 6 detailing the backward processing for Golden DAMP.

Table 6 illustrates the backward processing algorithm of Golden DAMP. As the backward search is performed on top of Golden Batch, we need to enter *GoldenBatch* into the algorithm. The first two lines of Table 6 are still the initialization phase. Line 1 is the same as in Table 2, initializing the discord score of the current subsequence to positive infinity. In line 2 we define the initial length of the iterative doubling search *prefix*. Here we set it as the lower bound of $2^{\text{nextpow2}(m)}$ and Golden Batch size to prevent possible array out-of-bounds problem at line 10. Then in the loop in lines 3–14, we perform the iterative doubling search, which starts from the end of Golden Batch and goes backwards. We keep searching in $GoldenBatch_{end\text{-}prefix+1:end}$ until we find the nearest neighbor whose distance from the current subsequence is less than *BSF* (line 11) or reach the beginning of Golden Batch (line 4). Specifically, if we find the nearest neighbor within the range *prefix*, we assign the approximate discord score of the current subsequence to $a\text{MP}_i$ and stop the search (lines 10–12); if not, in lines 13 and 14 we double the length of *prefix* and continue the search in $GoldenBatch_{end\text{-}prefix+1:end}$. If the search finally reaches the beginning of Golden Batch (line 4), we first calculate the exact discord score of the current subsequence using all the data in *GoldenBatch* (line 5), and then determine whether the current highest discord score *BSF* needs to be updated (line 6). If the discord score of the current subsequence is still greater than *BSF*, it means that the subsequence at position *i* does not have a nearest neighbor similar enough to it in the Golden Batch and it is a discord, at which point we should update the current highest discord score *BSF* in line 7.

This discussion of Golden DAMP is a good place to highlight an interesting and important property of the general DAMP algorithm. Virtually all other TSAD algorithms, including USAD (Audibert et al. 2021), AE (Audibert et al. 2021), *Telemanom* (Hundman et al. 2018), NORMA (Boniol et al. 2021a) and LSTM-VAE (Park et al. 2018), exist *only* as the implicit equivalent of Golden Batch algorithm. Here the training data given to the algorithm acts as the Golden Batch. This can be a problem if the period of the data changes, and we wish to be invariant of that. For example, a healthy human heartrate can vary between about 40 to 120 beats per minute (bpm). If a batch algorithm is trained on one heartrate, it may have difficulty generalizing to a different heartrate. In contrast, classic DAMP will be unaffected, because at every time step it is using *all* previously seen data as training data. Thus, so long as the heartrate change is not instantaneous, it can adjust to the new periodicity.

To illustrate this, in Fig. 11 we perform an experiment comparing classic DAMP with *Telemanom* (Hundman et al. 2018), on a dataset that has a changing periodicity.

In a sense, the news is even worse for *Telemanom* than Fig. 11 suggests. The algorithm has a stochastic element. We ran it three times, and this is the *best* of the three runs. In addition, note that this is an offline experiment. However, as we discuss in Sect. 6.3, all algorithms except DAMP have a period between the time they are given the training data, and the time they are ready to begin monitoring (we call this

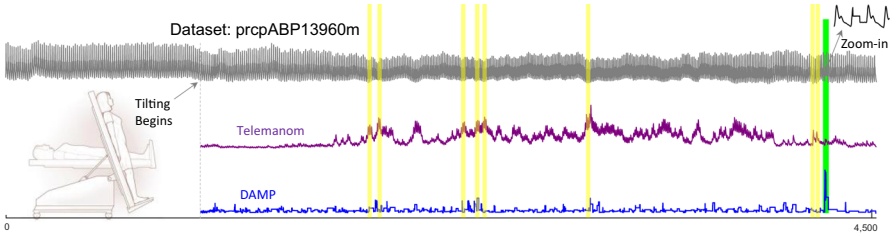

Fig. 11 *Top* A snippet of arterial blood pressure (ABP) data from a healthy patient undergoing a tilt-table test. There are no biological anomalies in the dataset, but near the end there is a short disconnection artifact (highlighted in green). *Middle* If we train *Telemanom* on the prefix of the snippet, before the table was tilted, it has a hard time adjusting to the post-tilt increased heartrate. It flags eight anomalies (highlighted in yellow), all false positives, and fails to discover the single true anomaly. *Bottom* In contrast, because DAMP is using *all* previously seen data, it can adjust to the changing heartrate, and it strongly peaks at the location of the true anomaly. Here $m = 33$

"linger"). Thus, in a real-time situation, there would be a period of a few tens of seconds, for which *Telemanom* would be undefined.

These observations do open an interesting issue, should we be invariant to changes in periodicity? This is a domain dependent question. Most biological signals can vary innocuously within a certain range. For example, heartbeats, respiration, gait cycles etc. In contrast, cycles guided by the circadian progression of the Earth's rotation, traffic patterns, electrical power demand, web traffic etc., will not be expected to have a change of periodicity, and any apparent change of periodicity probably warrants flagging as an anomaly. The Golden Batch implementation of DAMP allows the user to create a curated dataset that reflects the domain constraints. For example, suppose a user is given normal heartbeats at say 60 bpm. If she wants to be invariant to the heartrate varying between say 50 and 70 bpm, she can just create such rescaled time series and add them to her Golden Batch.

5.4 Multidimensional DAMP

The previous sections have shown how to find anomalies in a one-dimensional time series. We believe that in many cases, anomaly detection of *all* the one-dimensional data is sufficient for user demands. For example, in a hospital setting, a doctor may monitor a patient's ECG, blood pressure, and respiration. Most life-threatening situations will show up in at least one of the above. For example, a myocardial infarction, will first show up in the patient's ECG, septicemia will first show up in the patient's blood pressure, and tracheomalacia will first show up in the patient's blood respiration.

However, there are also special cases where anomalies occur in only two or more dimensions. For example, in the low-latitude Pacific West Coast region, typhoons accompanied by heavy precipitation occasionally make landfall in summer. In order to identify such unusual weather events, it is insufficient to monitor *only* precipitation or wind speed. This is because these areas may have strong winds but sunny weather, or extreme rainfall but still air. As a result, we need to combine wind speed and precipitation as two-dimensional data to find out which day has both precipitation and

wind speed anomalies. If such anomalies can be identified in two dimensions, there is a high chance of typhoon weather on that day. Therefore, it is necessary to generalize our DAMP algorithm to support searching in high-dimensional spaces. We refer to the DAMP algorithm for multidimensional data anomaly detection as *multidimensional DAMP*. We note that there are several ways in which the information from multiple time series can be combined. This issue is perhaps worthy of a detailed investigation. Here we show one simple and obvious method and demonstrate that DAMP can easily support it.

The basic idea of multidimensional DAMP is the same as the one-dimensional DAMP we introduced in Sect. 5.1, which retains the procedure of backward iterative doubling and forward pruning. The difference between them is reflected solely in the calculation of the discord score.

Figure 12 illustrates how the multidimensional DAMP calculates the discord score for position i. Let T^A be the time series of dimension A in a two-dimensional time series, while T^B corresponds to dimension B, and the length and frequency of T^A and T^B are equal. For position i, we first compute the distances between the current subsequence of T^A and T^B and the subsequences before position i in their respective dimensions, forming two distance vectors $D^A{}_i$ and $D^B{}_i$ (see Definition 4). After that, we add the elements of the two distance vectors two by two according to their positions to produce a new vector MD_i, which contains the distance information in both dimensions A and B. Finally, the minimum value on MD_i is the discord score at position i. As the algorithm progresses, the BSF continuously tracks the current highest discord score that combines information from both dimensions.

Tables 7 and 8 give the implementation details of multidimensional DAMP. Here we only demonstrate the two-dimensional version, however the reader can easily modify it to work with higher dimensional data. Since the basic steps of multidimensional DAMP and one-dimensional DAMP are the same, the framework of multidimensional DAMP is identical to Table 1.

Table 7 presents the multidimensional backward processing algorithm. As it is primarily similar to Table 2, we refer the reader to Sect. 5.2 for more details on the iterative doubling backward algorithm. Here we only highlight the parts that have changed. Compared to Table 2, we add two new inputs T^A and T^B, the time series in dimensions A and B. In lines 5 and 10, we change the calculation of the discord score at

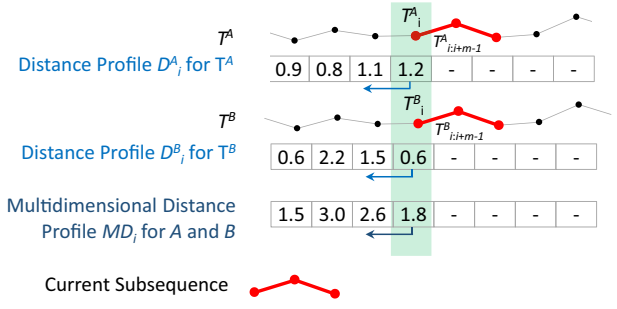

Fig. 12 Multidimensional distance profile for position i

Table 7 Multidimensional DAMP backward processing algorithm

```
Function:  [aMP_i, BSF] = BackwardProcessing(T^A, T^B, m, i, BSF)
Input:  T^A:  Dimension A of the multidimensional time series
        T^B:  Dimension B of the multidimensional 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^A_{1:i}, T^A_{i:i+m-1}) + MASS(T^B_{1:i}, T^B_{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^A_{i-prefix+1:i}, T^A_{i:i+m-1}) + MASS(T^B_{i-prefix+1:i}, T^B_{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
```

Table 8 Multidimensional DAMP forward processing algorithm

```
Function:  PV = ForwardProcessing(T^A, T^B, m, i, BSF, PV)
Input:  T^A:  Dimension A of the multidimensional time series
        T^B:  Dimension B of the multidimensional 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     MD'_i = MASS(T^A_{start:end}, T^A_{i:i+m-1}) + MASS(T^B_{start:end}, T^B_{i:i+m-1})
6     indices = all indices in MD'_i with values less than BSF
7     indices = indices + start - 1      // Convert indices on distance
8     profile to indices on time series
9  PV_indices = 0                  // Update the Pruned Vector
10 return PV
```

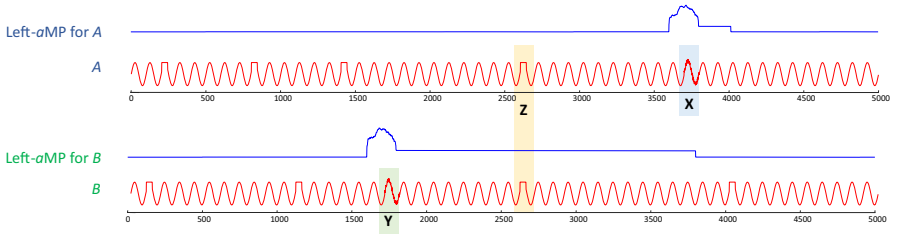

Fig. 13 Synthetic time series A and B. *Top* Synthetic dataset A and its corresponding one-dimensional Left-aMP. *Bottom* Synthetic dataset B and its corresponding one-dimensional Left-aMP. Here we set the window size to be the minimum positive period of the sine wave, i.e., $m = 100$

position i $a\text{MP}_i$. In line 5, to obtain $a\text{MP}_i$, we call MASS twice to calculate the distance between the current subsequence of T^A and T^B and all subsequences before position i respectively. Next, we add the elements in the two distance vectors returned by MASS two by two according to the positions to obtain the multidimensional distance profile. Finally, the minimum value of the multidimensional distance vector is taken as the exact discord score of position i. Line 10 is similar to line 5. The only difference is that line 10 only finds the nearest neighbor in the prefixes of T^A and T^B before position i and $a\text{MP}_i$ is the approximate discord score for position i.

Multidimensional DAMP also has a similar forward pruning process to that of one-dimensional DAMP, as shown in Table 8. Compared with Table 3, we need to only change line 5. In the range of *lookahead*, the distances between the current and future subsequences of T^A and T^B are calculated separately. Then the distance vectors of A and B dimensions are summed to yield a distance vector MD'_i containing two-dimensional information. Our pruning decisions are made based on this two-dimensional distance vector.

Let us start with a toy data set to understand the difference between multidimensional DAMP and one-dimensional DAMP. The red curves in Fig. 13 illustrate two synthetic time series A and B. These two time series consist mainly of sine waves. Specifically, for time series A, the data at positions 3700–3799 (**X**) are noisier than the other parts, while for time series B, the data at positions 1700–1799 (**Y**) are noisier. If you look closely, you will find that the two time series will have a square wave at random positions from time to time. It so happens that at positions 2605–2644, both time series show a square wave simultaneously, which is where our real anomaly lies. We denote it as **Z**. We tested the time series A and B with one-dimensional DAMP and two-dimensional DAMP respectively to see if they could find the true anomaly **Z**. Figure 13 also gives the results of performing a one-dimensional DAMP on time series A and B. It is easy to see by the highest point of the blue curve in Fig. 13 *top* that one-dimensional DAMP is attracted to the noisy sine wave in A and does not notice the anomaly at position **Z**. Similarly, as illustrated in Fig. 13 *bottom*, one-dimensional DAMP on B also fails to detect the anomaly at **Z**, instead considers the nosier **Y** as the anomaly. Missing information in another dimension, the one-dimensional DAMPs mistakenly believe that the presence of the square wave at **Z** is justified because they observe similar patterns before **Z**.

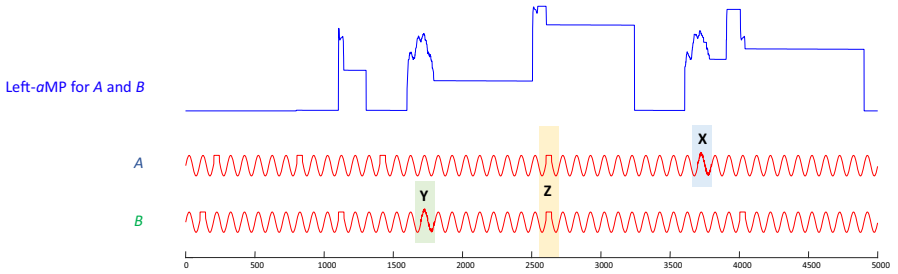

Fig. 14 Left-*a*MP generated by two-dimensional DAMP. Here we set the window size to be the minimum positive period of the sine wave, i.e., $m = 100$

Next, we combine A and B into a two-dimensional time series and feed it into the two-dimensional DAMP to see if the results will be different. The Left-*a*MP generated by two-dimensional DAMP is shown in Fig. 14. Note that compared with the Left-*a*MP generated by the one-dimensional DAMP in Fig. 14, the two-dimensional Left-*a*MP captures more anomalies with more "bumps" on its curve. All these bumps can be interpreted intuitively. For example, when both square and sine waves are present, or when one of the sine waves is noisier, they are recognized by the algorithm as a potential anomaly and correspond to a bump in the Left-*a*MP. What is more, the position of the highest point of Left-*a*MP in Fig. 14 corresponds to the coincidence of two square waves, that is, **Z**. This is because if you look at the entire time series of A and B, you will see that the square wave only appears at **Z** in both dimensions simultaneously, which cannot be observed at other locations.

We have seen that we can create a synthetic dataset that has an anomaly that can be discovered only by considering two time series simultaneously. However, can we discover two-dimensional anomalies in real data? Surprisingly, we are not aware of any such benchmark dataset. Most datasets in the space are synthetic, or are multidimensional, but have anomalies that are so obvious that it suffices to examine any *single* dimension (Audibert et al. 2021; Hundman et al. 2018). However, we can explore energy grid data published by a consortium of Texas A&M and USC in 2021 (Zheng et al. 2021), and use out-of-band data to evaluate the returned anomalies. Figure 15 *top* shows three years of wind speed and relative humidity data from the New York area between 2018 to 2020 (Zheng et al. 2021).

Figure 15 *bottom* shows the results of our search on the one-dimensional data of wind speed and relative humidity, respectively, and the anomalies identified by one-dimensional DAMP are marked in red. First, for wind speed, the one-dimensional DAMP reports the constant interval occurring on January 24, 2019, as an anomaly; however, we do not find any reported climate anomaly in New York State on that date. That is to say, although the algorithm finds an anomaly with a pattern that is different from its context, it does not seem to noticeably affect people's lives. As a result, we can conclude that the wind speed anomaly is trivial. Second, for relative humidity, the one-dimensional DAMP identifies the continuous peak occurring on November 23, 2018, as an anomaly. Through a Google search, we found reports of heavy rainfall and flooding that occurred in New York State on that day (National Weather Service 2019),

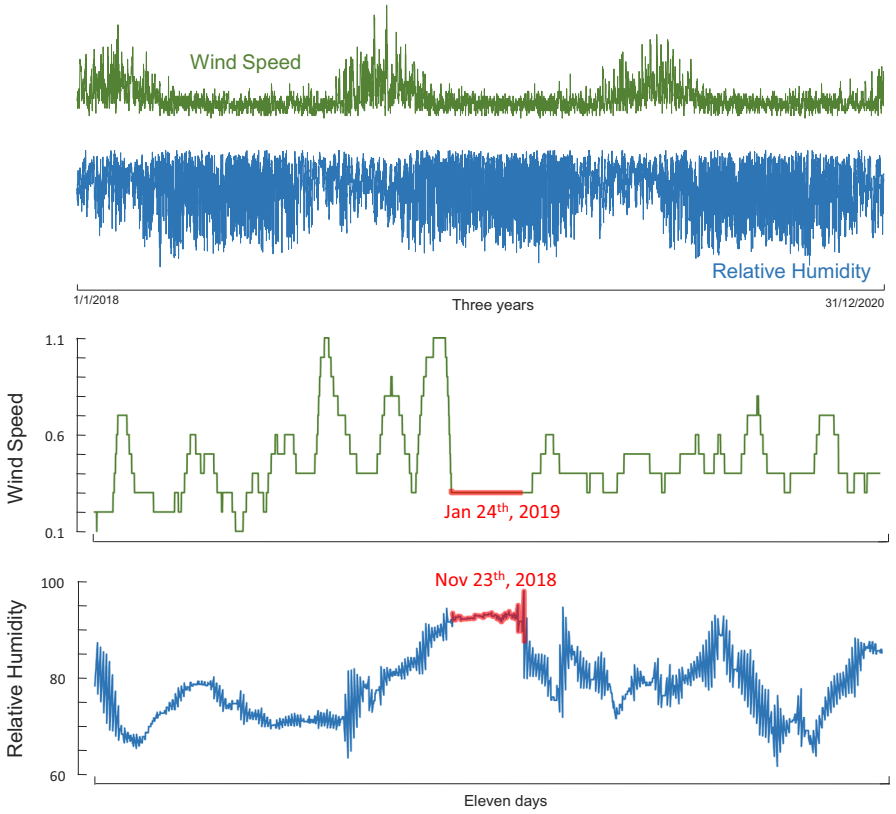

Fig. 15 *Top* Three years of wind speed and relative humidity data for the New York area from (Zheng et al. 2021). *Bottom* The two corresponding top 2D discord in this dataset. Here $m = 1440$ (one day in minutes)

which confirms that the anomalies identified in the dimension of relative humidity are informative and that the one-dimensional DAMP is effective.

However, if we combine wind speed and humidity and search in two dimensions, will the algorithm give us more interesting results? To investigate this, we took wind speed as dimension A and relative humidity as dimension B and re-executed this two-dimensional data using multidimensional DAMP. The results are presented in Fig. 16. Note that the two-dimensional DAMP reports a different date to either of the one-dimensional DAMP runs, May 28, 2019. This means that *both* humidity and wind speed in New York City showed anomalous patterns on this date. This anomaly is confirmed by the news "*A powerful thunderstorm slammed Staten Island Tuesday night, pounding the borough with large hail, heavy rain and the threat of a tornado.*" (Silive.com 2022).

We have demonstrated the utility of multidimensional DAMP. However, readers may wonder if it will pay a large time overhead for it. To investigate this, we used the data shown in Fig. 15 *bottom* (wind speed) and Fig. 16 and recorded the time cost of the one-dimensional and two-dimensional algorithms for increasingly long subsets.

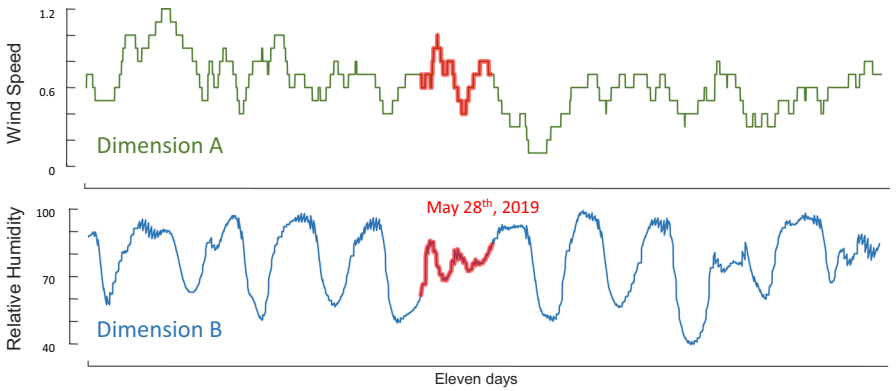

Fig. 16 Top discord for two-dimensional DAMP. Here $m = 1440$ (one day in minutes)

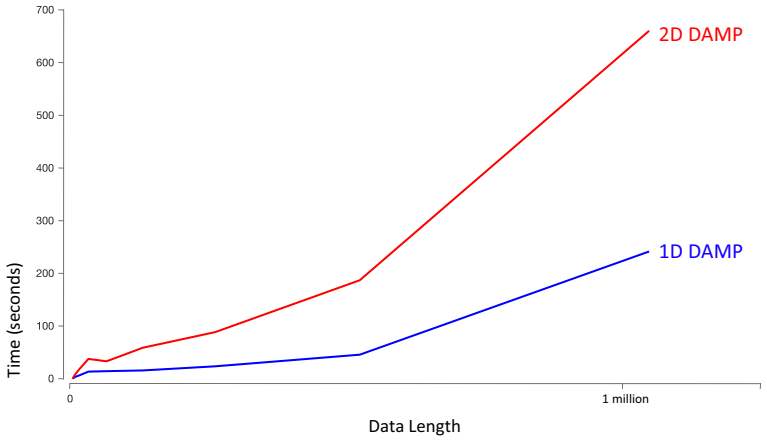

Fig. 17 The scalability of 1D and 2D DAMP over increasingly large datasets. The cost to double the number of dimensions considered is only slightly worse than double the time, suggesting that multidimensional DAMP search inherits the efficiency of the 1D version. Here $m = 1440$ (one day in minutes)

The experimental results are shown in Fig. 17. It can be seen that the time cost of a two-dimensional DAMP is only a small constant ratio of approximately 3.0 slower than the cost of a one-dimensional DAMP, which suggests the good scalability for multidimensional DAMP.

6 Empirical evaluation

To ensure the reproducibility of our experiments, we have built a website (DAMP 2022) containing all the data/code used in this work. All experiments were conducted on an Intel® Core i7-9700CPU at 3.00 GHz 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 Sect. 2, there are at least one hundred independent papers that have used discords to solve a real-world problem and that have shown that discords are the only technique that seems to be able to discover anomalies that are not visually obvious (Fig. 2, Fig. 3 and Fig. 4). Nevertheless, for completeness we will show examples in Sects. 6.1 and 6.2 that further demonstrate the excellent effectiveness of discords in diverse domains, and Sects. 6.3 and 6.4 offer comparisons to several deep learning-based methods.
- *Efficiency* As this is the main contribution of the paper, here we will attempt an ambitious set of anomaly detection experiments in terms of both throughput and scale.

6.1 Energy grid dataset

Recently, a consortium from Texas A&M and USC released a large dataset on decarbonized energy grids (Zheng et al. 2021). The dataset contains files representing three years of measurements of various metrics in sixty-six electrical zones in the continental USA. As Fig. 18 suggests, each file represents eleven measurements, ten of which are *measured* (temperature, wind speed etc.), but one is *computed* from the first principles of astronomy, the Solar Zenith Angle.

The total size of this dataset is 12 GB, representing 2174 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 Fig. 18 shows, most of the anomalies discovered do have a semantic meaning that can be traced. For example, a temperature trace from California had a discord that reflected "*Valentine's Day Storm Slams California*" (Wastewater News 2021). 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 readily appreciate the cause of this anomaly, after noting it occurs on the 29th of February (Wikipedia 2021). It is a classic leap year bug. Note that we informed the Texas A&M and USC team of this bug, so presumably it will be fixed in upcoming releases.

6.2 Machining dataset

The example in Sect. 6.1 demonstrates the utility of anomaly detection in batch 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" (CNC Crashes 2018). High-end CNC mills can cost over one million dollars, and crashes resulting in more than $20,000 in damage are known.

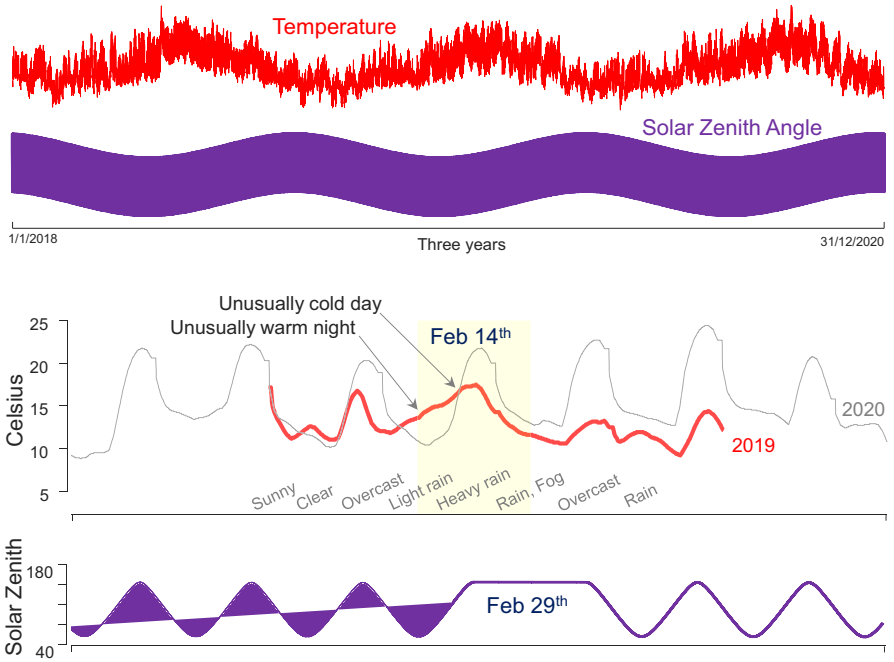

Fig. 18 *Top* Two examples of time series from (Zheng et al. 2021). Most, like temperature are *measured*, but Solar Zenith Angle is *computed*. *Bottom* The two corresponding top discords in these datasets. Here $m = 5760$ (four days in minutes)

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? In order to test this, we recreated a common scenario in Fig. 19.

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

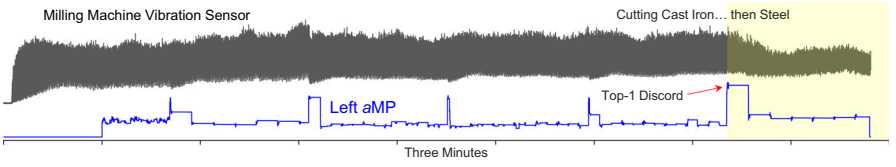

Fig. 19 *Top* Vibration telemetry from a milling machine that was cutting cast iron, but then overshot to start cutting the steel jaws of the vice. *Bottom* The Left-*a*MP discovers the transition. Here $m = 16$

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 Fig. 19 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 Comparison to LSTM deep learning

Although dozens of competing deep learning anomaly detection (DLAD) algorithms now exist, 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 (Wu and Keogh 2021). 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 (Hundman et al. 2018), 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* (Fig. 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* (Fig. 2): *Yes*. However, *Telemanom* took a total of (517.6 training + 700.4 testing) 1,218 s. This is two orders of magnitude slower than DAMP, which took 16.1 s. More importantly, *Telemanom* is an order of magnitude slower than real-time, precluding any possibility of online monitoring.
- *Energy Grid* (Sect. 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 h, over 1300 times slower than DAMP.
- *Milling Data* (Fig. 19) *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 s to train the model. However, 127 s after it begins training, we encounter the anomaly, and about 21 s after that, the endmill snaps off. *Telemanom* is just too slow to be useful here.

These comparisons suggest that the most cited deep learning anomaly detection algorithm is not as accurate as DAMP, requires more training data, and is much slower.

6.4 Comparison on the KDD Cup 2021 datasets

To further see the limitations of deep learning time series 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 (Wu and Keogh 2021). Instead, we consider the KDD Cup 2021 dataset consisting of 250 univariate time series (Dau et al. 2019). 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. Moreover, the datasets include a wide range of domains, including cardiology, industry, medicine, zoology, weather, human behavior, etc. We use the accuracy metric that was suggested by the dataset's creators. In brief, each of the 250 datasets has a single anomaly. Each algorithm is tasked with predicting the location of that anomaly. Let the length of the anomaly be L. If the prediction is within plus or minus L data points of the anomaly's true location, it is judged correct. If L is less than 100, then it will be set to 100. The scores in Table 9 show the ratio of correct predictions for the 250 datasets.

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 DAMP 2022) 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

Table 9 Accuracy and time for eight TSAD methods

Method	Accuracy	Train and test time
USAD (Audibert et al. 2021)	0.276	8.05 h
LSTM-VAE (Park et al. 2018)	0.198	23.6 h
AE (Audibert et al. 2021)	0.236	6.11 h
Telemanom (Hundman et al. 2018)	*Out of memory error on longer examples*	
NORMA (Boniol et al. 2021a)	0.474	17.8 min
SCRIMP (Full-MP)	0.416	24.5 min
DAMP (Left-MP) out-of-the-box	0.512	4.26 h
DAMP (Left-MP) sharpened data	0.632	4.26 h

competitive. As an example of a small intervention that can further improve accuracy, if we run DAMP on *sharpened data* (a single extra line of code, see (DAMP 2022) for details) 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 and NORMA here is relatively good, as there are 250 *short* time series. In Fig. 22 we will see that for longer time series this advantage of SCRIMP/NORMA rapidly inverts.

We included a comparison to the recently published NORMA (Boniol et al. 2021a), which can be seen as a sort of Matrix Profile that uses an automatically discovered subset of the training data as the reference data. Here we used the original authors' tools and suggestions to set the parameters (we were able to make the results *slightly* better with our own parameter settings (DAMP 2022). The time for NORMA is good, but it is important to note the following:

- These datasets have tiny training data splits (they were deliberately made that way, to allow the deep learning community to consider them in a tenable fashion (Dau et al. 2019). But as Fig. 23 shows the NORMA algorithm scales poorly for large datasets.
- On these datasets, we can easily close all of the time gap by using either X-Lag-Amnesic DAMP (Sect. 5.3.1) or Golden DAMP (Sect. 5.3.2), with only a minimal decrease in accuracy. Indeed, the Golden DAMP algorithm essentially subsumes NORMA as a special case.
- The results in Table 9 mask a unique timing advantage that DAMP has over not only NORMA, but all other non-trivial anomaly detectors.[4] We believe that DAMP is the only *instantaneous* TSAD in the literature. To see this, consider the situation in Fig. 20.

The figure shows a dataset from the KDD Cup 2021. The first forty seconds of wall-clock time pass, and then we are invited to monitor for anomalies in the remainder of the data. We define "linger" as the time a TSAD algorithm requires to ingest the training data, build its model, and be ready to start monitoring. As shown in Fig. 20, the linger for NORMA on this problem is thirteen seconds. This means that any anomaly that occurs in the first thirteen seconds will not be detected (or will only be detected post-mortem). Note that DAMP appears to be unique among TSAD algorithms in having zero linger. In this example, the linger of NORMA may not be too consequential (although it grows rapidly with more training data, see Fig. 23). Perhaps the attending physician can wait with the patient while the model is being built. However, recall our machining example in Sect. 6.2. Here, if the linger is more than 127 seconds, the TSAD algorithm would not be able to avoid the expensive head-crash.

Recall that Table 9 notes "*Out of memory error on longer examples*" for *Telemanom* (CNC Crashes 2018). There does not seem to be any simple way to fix this issue, so

[4] Here we explain "non-trivial anomaly detector". Simple rule-based conditionals such as: "**if** the time series ever reports a value that is higher than any value you have seen before, **then** flag anomaly" could be used as an anomaly detector, and could be instantaneously instantiated. By *non-trivial* we mean any TSAD algorithm that examines each subsequence for any information about shape, autocorrelation, Markov properties etc., and compares this information (in the most general sense), to a model gleaned from training data. The reader will appreciate that this includes essentially all proposed anomaly detectors in the literature.

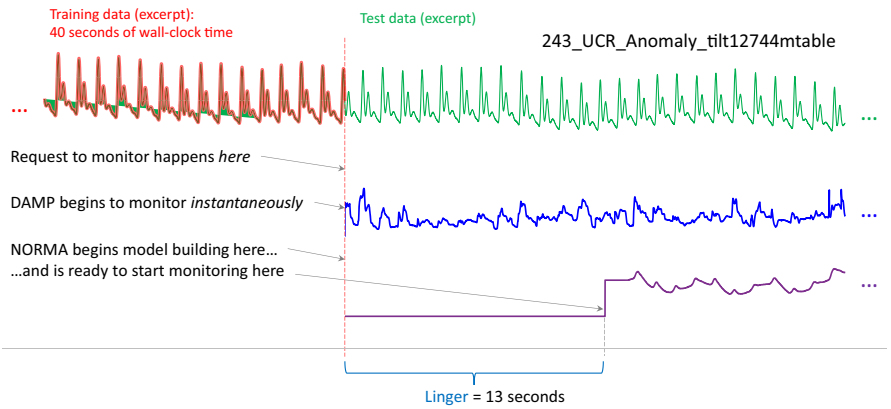

Fig. 20 An excerpt from the 243_UCR_Anomaly_tilt12744mtable dataset. The task is to exploit information in the training split, to detect the most significant anomaly in the test split. When requested, DAMP can instantaneously begin to monitor. However, NORMA (and all other TSAD algorithm), must have a period of inaction or "linger" while they build their models. Here $m = 276$

we did the following. We sorted all the datasets from smallest to largest, and kept evaluating increasingly longer datasets until the first failure. *Telemanom* failed at the 63rd smallest dataset (114_UCR_Anomaly_CIMIS44AirTemperature2). On the first 62 datasets it correctly found the anomaly on 29, giving an accuracy of 0.468. This took *Telemanom* 3.4 h. When we run DAMP on just these 62 shorter datasets, it takes 64.9 s. In general, the 62 shorter test cases are the easier ones (they certainly have a much higher default rate), yet both flavors of DAMP are still significantly more accurate.

Finally note that Table 9 does not include any comparisons to the algorithms that entered the KDD Cup in 2021 (Dau et al. 2019). The best performing algorithms scored an impressive 88.4%. However, note that none of the top performers have made code publicly available. Moreover, all the top performers use *meta-algorithms*. For example, the top place algorithm, DeepBlueAI, used a meta-algorithm that included at least four different algorithms ("Fourier Transformation based methods", Matrix Profile, LightGBM and Dilated CNN). In all cases, the logic used to switch between or combine the atomic algorithms is not clear (We hope that in at least some cases, the participants with publishing a publication will make that clear). In contrast, Table 9 compares the leading *single* algorithms, which have usable public implementations. Combining them in a meta-algorithm or ensemble would be an interesting project but is beyond the scope of this paper.

6.5 Threshold learning for DAMP

Up to this point, we have experimentally demonstrated that DAMP can *locate* the most anomalous subsequence. However, we have not shown how the algorithm makes a binary decision thereafter to flag the subsequence as anomalous or not. For this purpose, 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 Fig. 21 *top*, into half of them we randomly inserted a subtle anomaly, a low amplitude

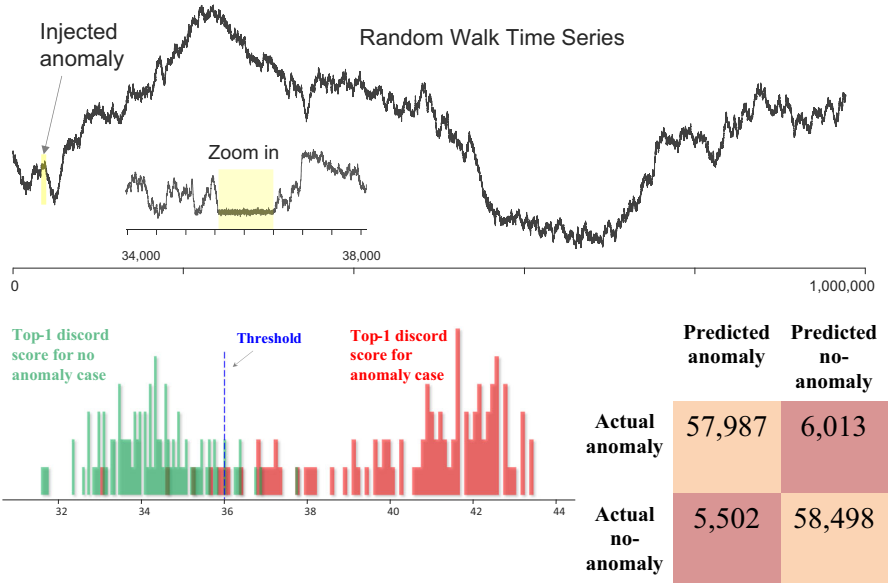

Fig. 21 *Top* A sample random walk with an anomaly embedded. *Left* The distribution of top-1 discord scores for the two cases of interest. *Right* The confusion matrix for this task. Here $m = 1024$

random section of length 950 (Why length 950? We found that if we used length 1,000 we got perfect accuracy, which is uninteresting for this experiment. So, we tuned the value to give an error rate of about 10%). In Fig. 21 *left*, we show the top-1 discord score (for $m = 1024$) 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 21 *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* (Hundman et al. 2018) would take about twelve years to do this, although in practice it gives *out-of-memory* errors.

6.6 Scalability comparisons

To find out which elements of our proposed method contribute most to its efficiency, we have performed an ablation study, in which various elements of DAMP were progressively crippled. As a baseline, we also compare to SCRIMP (Zhu et al. 2018). This comparison to SCRIMP is a little unfair, as it discovers motifs as well as discords.

However, it seems to be the most used discord discovery algorithm in recent years. Figure 22 summarizes our findings.

It is clear that each element we proposed does actually contribute to speed up, and that DAMP is effectively linear in n.

As we earlier noted, most of the benchmark datasets are only hundreds to thousands of datapoints long (Wu and Keogh 2021), and that seems to have set the limit of the ambition of most of the community when it comes to scalability. However, a recent paper pushed that envelope by considering a two million length ECG dataset (Boniol et al. 2021a). In fact, these authors graciously gave us the *exact* dataset they used, (which was in fact even longer than they considered in Boniol et al. 2021a), and helped us create a perfectly commensurate experiment, as shown in Fig. 23. A real-time video trace of this experiment is at (DAMP 2022).

Note that of the many approaches considered, some time out (i.e., are not finished in a four-hour cutoff) at length 500 K. In contrast, DAMP can handle eight million datapoints in just 22.3 s, this is over 358,000 Hz. In fact, DAMP is so fast, that the

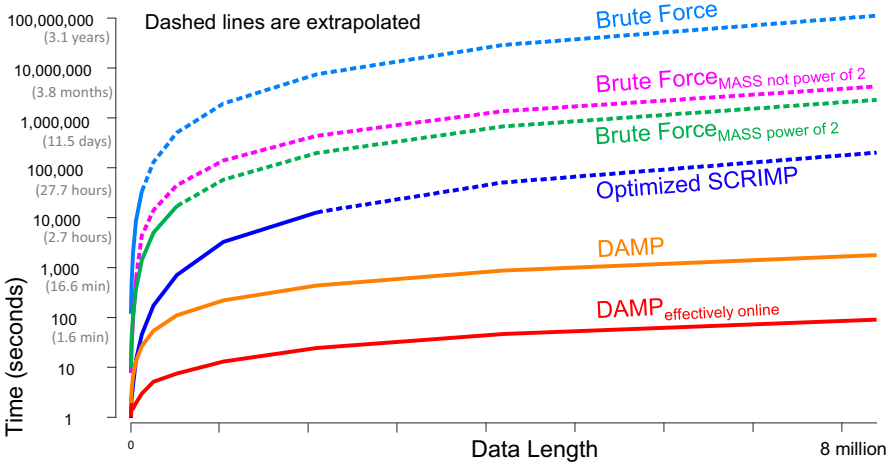

Fig. 22 The CPU time vs time series length for various discord discovery algorithms. Note the Y-axis is in log scale. Note that $\text{DAMP}_{\text{effectively online}}$ means that the forward processing algorithm introduced in Table 3 was used. Here $m = 94$

Fig. 23 (Most of this figure is taken from (Boniol et al. 2021a) with permission, 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 to the first 2 M data points by (Boniol et al. 2021a). Note that the Y-axis is logarithmic. Here $m = 94$

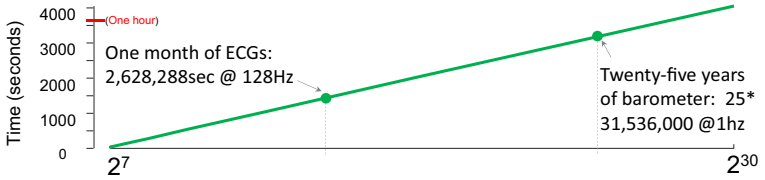

Fig. 24 The time taken for DAMP to process a random walk time series of length 2^{30} (just over one billion). For context, we have labeled the size of two concrete tasks, processing a month of ECGs and twenty-five years of sensor data. Here $m = 128$

time it reports for the 50 K length trial is literally off the original chart, taking less than one second.

As eight million datapoints are about the longest publicly available ECG, in Fig. 24 we conclude this section by searching a single random walk time series of length 2^{30}.

6.7 Scalability and stability of DAMP

One of Wu and Keogh's criticisms of common benchmarks is *unrealistic anomaly density* (Wu and Keogh 2021). They noted that over 20% of the data is labeled anomalous in many benchmarks, which poses a real problem for the evaluation. Suppose that an algorithm has near perfect sensitivity, but it will randomly give out a false positive once in every million datapoints (perhaps due to the numerical instability of streaming algorithms (Higham 2002). Note that because most benchmarks in the literature only have a few thousand datapoints, this issue would almost certainly not be observed during testing. However, it clearly would be a problem for any real-world deployment. For example, for a continuous processing system with telemetry reporting every second, this would give us about thirty-one false positives a year.

To demonstrate DAMP does not have this issue, we did the following test. Recall the subtle anomaly shown in the 100,000 datapoint MGAB dataset in Fig. 4. We can append anomaly-free data from the same Mackey–Glass model (but free of the embedded anomalies (Thill et al. 2020) to make it one thousand times longer, i.e., a total length of 100 million. When we search this with DAMP ($m = 40$), we count a trial successful if the top-1 discord is found in the first 100,000 datapoints (created by Thill et al. 2020), rather than from the appended ninety-nine million nine hundred thousand datapoints. Each of the coauthors of this work ran this experiment multiple times in the background of their desktops over a week, and in total conducted over 16,000 such trials, finding a total of zero false positives.

Note that this experiment required performing anomaly detection on time series with a total length of 1.648 trillion datapoints, using off-the-shelf hardware. This is something that would be inconceivable with any other anomaly detection method.

7 Conclusions and future work

In this paper, we created the left-discord anomaly detection framework, generalizing classic time series discords that previously only handled the batch case, to the online

and *effectively online* case, and solving the twin-freak problem in the process. Further, we have introduced DAMP, a fast and scalable algorithm to discover such discords. Experimental results have demonstrated that our proposed left-discords outperform the current SOTA methods, including the most cited deep learning methods in terms of accuracy. Moreover, we have further demonstrated that DAMP is orders of magnitude faster and more scalable than any method in the literature.

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. Finally, we have made all our code and data available to the community to confirm and build upon our work.

In future work, we plan to address the limitations of DAMP. For example, DAMP uses the Z-normalized Euclidean distance, but you cannot Z-normalize constant regions of the time series (as you get a divide-by-zero error). Another type of anomaly that DAMP cannot detect is a sudden *decrease* in the noise level of a time series, as smooth time series tend to have a relatively low distance from all other time series. As of now, we can catch these two special cases with ad-hoc rules, but a more principled and elegant solution is desirable.

Acknowledgements We sincerely thank the authors of Boniol et al. (2021a) for their help in creating Fig. 23.

Funding 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.

Declarations

Conflict of interest The authors have no conflict of interest.

Ethical approval This research complies with all ethical guidelines at the three institutions represented.

Human or animal rights No human subjects were used by the current authors. The ECG data was collected by others over a decade ago, under strict human subject protocols.

Appendix: tool to set window size

For most of the experiments in this paper, we use human intuition to manually set the value of the parameter m. We followed the well-known "folk wisdom" that was explained and tested in Nakamura et al. (2020). The idea is simply this. Plot a subset of the data, estimate the length of a single period "by-eye" and set m to a value a little less than a period. Here we used $m = 90\%$ of estimated period length. In (Nakamura et al. 2020) the authors show that in most cases, the anomalies returned doing this, would not change even if you set m to twice or half this value (They were using the full MP, not the left MP, but we found similar results with DAMP).

However, for the experiment in Table 9, we want to completely exclude any human intervention or tuning. Therefore, we used the following four lines of Matlab code to automatically calculate the window size for the 250 KDD Cup 2021 datasets.

```
[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 window size is obtained by finding the peak of autocorrelation in the range of 10–1000 (the value of parameter m is limited to the range of 10–1000). To avoid the 'findpeaks' function returning a null value (very unlikely, but possible), we set the default value of m to 1000.

References

Aubet F-X, Zügner D, Gasthaus J (2021) Monte Carlo EM for deep time series anomaly detection. arXiv: 2112.14436 [cs, stat]

Audibert J, Marti S, Guyard F, Zuluaga MA (2021) From univariate to multivariate time series anomaly detection with non-local information. In: Lemaire V, Malinowski S, Bagnall A et al (eds) Advanced analytics and learning on temporal data. Springer International Publishing, Cham, pp 186–194

Boniol P, Linardi M, Roncallo F et al (2021a) Unsupervised and scalable subsequence anomaly detection in large data series. VLDB J 30:909–931. https://doi.org/10.1007/s00778-021-00655-8

Boniol P, Paparrizos J, Palpanas T, Franklin MJ (2021b) SAND: streaming subsequence anomaly detection. Proc VLDB Endow 14:1717–1729

Case Western Reserve University Bearing Data Center (2021) Available: https://csegroups.case.edu/bearingdatacenter/home. Accessed: Nov. 15, 2021

CNC Crashes. Video. (15 Feb 2018). from https://youtu.be/t2tBtZCa7j4?t=205. Retrieved December 20, 2021

Daigavane A, Wagstaff KL, Doran G et al (2022) Unsupervised detection of Saturn magnetic field boundary crossings from plasma spectrometer data. Comput Geosci 161:105040

DAMP (2022) https://sites.google.com/view/discord-aware-matrix-profile

Dau HA, Bagnall A, Kamgar K et al (2019) The UCR time series archive. IEEE/CAA J Autom Sin 6:1293–1305. https://doi.org/10.1109/JAS.2019.1911747

Doshi K, Abudalou S, Yilmaz Y (2022) TiSAT: time series anomaly transformer. arXiv:2203.05167 [cs, eess, stat]

Higham NJ (2002) Accuracy and stability of numerical algorithms, 2 edn. ISBN: 978-0-89871-521-7

Hundman K, Constantinou V, Laporte C et al (2018) Detecting spacecraft anomalies using LSTMs and non-parametric dynamic thresholding. In: Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining. ACM, London United Kingdom, pp 387–395

Imani S, Madrid F, Ding W et al (2020) Introducing time series snippets: a new primitive for summarizing long time series. Data Min Knowl Disc 34:1713–1743. https://doi.org/10.1007/s10618-020-00702-y

Keogh E (2021) Irrational exuberance why we should not believe 95% of papers on time series anomaly detection. In: 7th SIGKDD workshop on mining and learning from time series at SIGKDD 2021. Workshop Keynote https://www.youtube.com/watch?v=Vg1p3DouX8w&t=324s

Khansa HE, Gervet C and Brouillet A (2012) Prominent discord discovery with matrix profile: application to climate data insight. In: 10th international conference of advanced computer science & information technology (ACSIT 2022) May 21~22, 2022, Zurich, Switzerland

Kirti R, Karadi R (2012) Cardiac tamponade: atypical presentations after cardiac surgery. Acute Med 11:93–96

Mueen A, Zhu Y, Yeh M et al (2017) The fastest similarity search algorithm for time series subsequences under euclidean distance. http://www.cs.unm.edu/~mueen/FastestSimilaritySearch.htmlAccessed 24 Janurary, 2022

Nakamura T, Imamura M, Mercer R, Keogh E (2020) Merlin: parameter-free discovery of arbitrary length anomalies in massive time series archives. In: 2020 IEEE international conference on data mining (ICDM). IEEE, Sorrento, Italy, pp 1190–1195

National Weather Service. January 24, 2019 heavy rain and flooding. From https://www.weather.gov/aly/24Jan19HeavyRainFlood. Retrieved May 1 2022

Neupane D, Seok J (2020) Bearing fault detection and diagnosis using case western reserve university dataset with deep learning approaches: A review. IEEE Access 8:93155–93178. https://doi.org/10.1109/ACCESS.2020.2990528

Nilsson F (2022) Joint human-machine exploration of industrial time series using the matrix profile. In: Halmstad university, school of information technology, Halmstad embedded and intelligent systems research (EIS), CAISR—center for applied intelligent systems research

Palpanas T (2022) Personal communication June 4th 2022

Paparrizos J, Kang Y, Boniol P et al (2022) TSB-UAD: An end-to-end benchmark suite for univariate time-series anomaly detection. In: Proceedings of the VLDB endowment (PVLDB) journal

Park D, Hoshi Y, Kemp CC (2018) A multimodal anomaly detector for robot-assisted feeding using an LSTM-based variational autoencoder. IEEE Robot Autom Lett 3:1544–1551. https://doi.org/10.1109/LRA.2018.2801475

Park JY, Wilson E, Parker A, Nagy Z (2020) The good, the bad, and the ugly: data-driven load profile discord identification in a large building portfolio. Energy Build 215:109892

Silive.com. Wild storm pelts Staten Island with giant hail—'threat of tornado has passed' from https://www.silive.com/news/2019/05/nws-issues-tornado-warning-for-staten-island.html. Retrieved May 1 2022

Su Y, Zhao Y, Niu C et al (2019) Robust anomaly detection for multivariate time series through stochastic recurrent neural network pp 2828–2837

Thill M, Konen W, Bäck T (2020) Time series encodings with temporal convolutional networks. Springer, Cham, pp 161–173

Truong HT, Ta BP, Le QA et al (2022) Light-weight federated learning-based anomaly detection for time-series data in industrial control systems. Comput Ind 140:103692. https://doi.org/10.1016/j.compind.2022.103692

Wastewater News. Valentine's day storm slams California, pushing water agencies to the edge. From www.news.cornell.edu/Chronicle/00/5.18.00/wireless_class.html. Retrieved Dec 1 2021

Wikipedia. Leap year problem. from https://en.wikipedia.org/wiki/Leap_year_problem. Retrieved December 1, 2021

Wu R, Keogh E (2021) Current time series anomaly detection benchmarks are flawed and are creating the illusion of progress. IEEE Trans Knowl Data Eng. https://doi.org/10.1109/TKDE.2021.3112126

Yeh C-CM, Zheng Y, Wang J et al (2021) Error-bounded approximate time series joins using compact dictionary representations of time series. CoRR abs arXiv:2112.12965

Yeh C-CM, Zhu Y, Dau HA et al (2019) Online amnestic dtw to allow real-time golden batch monitoring. pp 2604–2612

Zheng X, Xu N, Trinh L et al (2021) PSML: a multi-scale time-series dataset for machine learning in decarbonized energy grids. arXiv preprint arXiv: 2110.06324

Zhu Y, Yeh C-CM, Zimmerman Z et al (2018) Matrix profile XI: SCRIMP++: time series motif discovery at interactive speeds. In: IEEE pp 837–846