Fast and Scalable Mining of Time Series Motifs with Probabilistic Guarantees

Matteo Ceccarello
Free University of Bozen/Bolzano
Bolzano, Italy
mceccarello@unibz.it

Johann Gamper
Free University of Bozen/Bolzano
Bolzano, Italy
gamper@inf.unibz.it

ABSTRACT

Mining time series motifs is a fundamental, yet expensive task in exploratory data analytics. In this paper, we therefore propose a fast method to find the top-k motifs with probabilistic guarantees. Our probabilistic approach is based on Locality Sensitive Hashing and allows to prune most of the distance computations, leading to huge speedups. We improve on a straightforward application of LSH to time series data by developing a *self-tuning* algorithm that adapts to the data distribution. Furthermore, we include several optimizations to the algorithm, reducing redundant computations and leveraging the structure of time series data to speed up LSH computations. We prove the correctness of the algorithm and provide bounds to the cost of the basic operations it performs. An experimental evaluation shows that our algorithm is able to tackle time series of one billion points on a single CPU-based machine, performing orders of magnitude faster than the GPU-based state of the art.

PVLDB Reference Format:
Matteo Ceccarello and Johann Gamper. Fast and Scalable Mining of Time Series Motifs with Probabilistic Guarantees. PVLDB, 15(13): 3841 - 3853, 2022.
doi:10.14778/3565838.3565840

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Cecca/attimo.

1 INTRODUCTION

Finding structure in large time series is an important task in several domains. In particular, finding repeated subsequences in long time series, also known as *motifs*, is an important primitive in domains as diverse as entomology [32], meteorology [28], nematology [10], and seismology [12]. Furthermore, motif discovery is also used as a preprocessing step in anomaly detection [8].

Given the presence of noise, subsequences are usually compared by *similarity* rather than by exact equality. At its essence, motif discovery entails finding k pairs of subsequences of a given length which are pairwise maximally similar, for a parameter k specified by the user. This problem lends itself to a straightforward solution: iterate over all pairs of subsequences, keeping track of the closest k pairs. However, for a time series of length n and subsequence length w, this naïve approach has a $O(w \cdot n^2)$ complexity, which is clearly untenable for long time series.

We propose a *self-tuning* algorithm for motif discovery based on Locality Sensitive Hashing that adapts to the data distribution. Intuitively, we map each subsequence of the time series to hash values, with the property that similar subsequences are assigned the same hash value: in particular, the k motif pairs are likely to have the same hash value under this scheme. We can then restrict distance computations to pairs of subsequences with the same hash value, thus hopefully pruning away a lot of unneeded and expensive computations.

One of the challenges in using LSH is the number of parameters involved. Setting all the parameters appropriately affects the quality of the solution and the efficiency of the computation. Unfortunately, the best value for each parameter is data dependent. To mitigate this issue, our algorithm takes the *maximum allowed values* for two parameters, and automatically finds the configuration minimizing distance computations that both respects the given constraints and meets a user-provided failure probability.

Our approach is data-adaptive: the algorithm carries out a number of distance computations that depends on the separation between the motifs and the average subsequence pair. For a fixed time series length, our algorithm finishes more quickly on datasets with motifs that *stand out*. This is in contrast with state of the art approaches [48] that make the data-independence of their execution time one of their strong points.

Our contributions are the following:

- We present a novel self-tuning and data-adaptive LSH-based algorithm for the discovery of the top-k motifs with a user controlled failure probability.
- We introduce several optimizations that greatly improve the performance of our approach, speeding up and reducing the number of hash function evaluations, and removing redundant distance computations.
- We prove the correctness and complexity of our algorithm.
- We provide an open source implementation of our algorithm that scales to time series with billions of data points on a single CPU machine.
- We carry out an extensive experimental evaluation, showing that our CPU-based implementation can scale to large time series better than the state-of-the art GPU-based implementation.

Organization. After reviewing the related work (§2), we introduce background concepts on time series and LSH (§3), giving an example naïve application of LSH. Then, we describe our contribution (§4), presenting several optimizations separately (§ 5) for clarity. Finally, we evaluate our approach experimentally (§ 7).

Proceedings of the VLDB Endowment, Vol. 15, No. 13 ISSN 2150-8097.
doi:10.14778/3565838.3565840

2 RELATED WORK

A survey on motif discovery is given by Mueen [29], who identifies two variants of the problem: *similarity* based and *support* based. In the former a motif is defined as the pair of subsequences with the smallest distance in a time series; in the latter a distance threshold R is additionally considered, and motifs are defined as being balls of radius R containing the most subsequences. In this paper we adopt the *similarity* based definition, that foregoes the need of parameter R, which is domain specific and difficult to set.

The motif discovery problem in time series has been introduced in [38] along with the EMMA algorithm. Subsequences are organized in a hash table using their SAX [26] representation as keys: all subsequences assigned to the same bucket are candidates for being the motif. The algorithm also inspects neighboring buckets (using the lower bound on the Euclidean distance provided by SAX). While resorting to the SAX representation of time series allows to prune some unnecessary distance computations, there is no upper bound on the number of buckets that need to be evaluated.

Shortly after, a probabilistic algorithm has been proposed [15], borrowing techniques from a seminal work in approximate matching in DNA strings [11]. The algorithm is also based on the SAX representation of subsequences to transform a time series into a sequence of discrete symbols. The basic idea is similar to the EMMA algorithm, but random characters from the SAX alphabet are used as a key in the hash table. By repeating the procedure several times, similar subsequences are likely to collide, thus allowing to discover the motif. In contrast to our algorithm, this approach, which is reminiscent of LSH, provides no formal guarantees on the quality of the result and features a large number of parameters, which are difficult to tune.

The above solutions change the *representation* of the subsequences to prune unnecessary distance computations. A different strategy is investigated in [32], where the straightforward double-nested loop algorithm is optimized by considering a sample of *reference* subsequences, which are used to early reject candidate pairs. While this heuristic works well, in the worst case it still requires time $O(w \cdot n^2)$. A parallel version of this algorithm has been developed in [36].

For online settings, where new points are continuously appended at the end of the time series, an algorithm to maintain information about the *top* motif has been developed in [31] and later extended to the top-k case [24].

The *matrix profile* [45] is a data structure that stores, for each subsequence, the index and distance of its nearest neighbor. It provides a lot of information for several questions about time series, including motifs: one just needs to look at the minimum distance in the matrix profile to locate the top motif. Approaches based on the matrix profile require time $\Omega(n^2)$, making them challenging to apply to large scale data. An *anytime* version of the Matrix Profile, named SCRIMP++ [46], provides empirically good approximations of the matrix profile (and hence of the motifs). One of the key elements of this approach is the PreSCRIMP subroutine, which computes an approximation to the matrix profile. While the approach proves to be fast and accurate in practice, it lacks theoretical guarantees on the quality of the result. The most efficient computation of the matrix profile (and of motifs) is provided by Scamp [48], which can employ clusters of GPUs to speed up the computation. Scamp outperforms all previously proposed approaches, including PreSCRIMP. Nonetheless, the algorithm is inherently $\Omega(n^2)$, thus requiring a lot of computational resources to solve large problems.

Recently, a randomized exact approach has been proposed [25] with linear running time in expectation. The approach is based on building a set of grids with different widths to index the subsequences of the time series. The grids are populated incrementally, inserting subsequences one at a time, maintaining an ever-decreasing guess on the distance of the motif. During insertion, the partially-constructed grids are used to answer nearest-neighbor queries. At the end of this procedure, the motif pair with the highest similarity is returned. General ℓ_p norms are supported, including the z-normalized Euclidean distance, but the implementation and experiments use the Chebyshev distance.

In recent years, most of the research on time series motifs has shifted to the discovery of variable-length motifs [18, 19, 27, 30] and multi-dimensional motifs [18], problems which are outside of the scope of this paper, where we present substantial improvements to the running time of the fixed-length motif discovery problem.

To summarize, compared to previous works, our approach provides better scalability than the state of the art Scamp and provides theoretical guarantees. This is in contrast to approximation algorithms such as PreSCRIMP and random projection [15], that do not offer theoretical guarantees on the quality of the result.

Locality Sensitive Hashing (LSH for short) has been introduced by Indyk and Motwani [23] as a general technique to address nearest neighbor queries. For an in-depth survey we refer the interested reader to [16, 44]. Particularly relevant for the approach presented in this paper is the family of hash functions for the Euclidean distance presented in [17]. Note that in [17] LSH is applied to the approximate R-near neighbor search problem. In Section 4 we shall discuss how to employ LSH in an exact nearest neighbor search setting. A family with better theoretical properties has been presented in [1], but it is less amenable to an efficient implementation in the context of time series.

Using LSH correctly requires setting several parameters, many of which are data dependent. The PUFFINN [5] algorithm represents an approach that is able to automatically tune most parameters depending on the data at hand for k-nearest neighbor queries. The approach presented in this paper is inspired by PUFFINN, with several notable differences. PUFFINN is implemented for cosine and Jaccard similarity, whereas our approach works with the Euclidean distance between subsequences. This has an important impact on the implementation, since the LSH families for Euclidean distance feature additional parameters that need to be automatically tuned. Furthermore, our approach is fine-tuned to discover top-k pairs within a dataset, whereas PUFFINN aims at answering arbitrary nearest neighbor queries.

LSH has been applied to time series in the context of earthquake detection [41]. Waveforms associated with earthquakes are represented by means of two-dimensional *fingerprints* derived from their spectrogram. Similar fingerprints correspond to similar earthquakes. To improve the running time, fingerprints are processed using MinHash [9]. Compared to the approach proposed in this paper, [41] works in a different metric space and does not verify distances of colliding fingerprints, relying on a threshold on the

fraction of collisions with respect to the number of repetitions. In [48], the authors comment on this line of work saying that *"this approach [...] produced false positive and false negative results. In addition, LSH requires the careful selection of multiple dataset specific tuning parameters [...]"*. In the present work we address both concerns, by providing an algorithm with guaranteed recall that is able to self-tune to the data distribution.

There are LSH families for several popular distance measures, including the cosine similarity [14] and the Fréchet distance [13]. For the sake of clarity, in this work we focus our attention on the Euclidean distance [17].

3 PRELIMINARIES

3.1 Time series and motifs

A *time series* T is an ordered collection of n values in $\mathbb{R}$. Given a time series T of length n and a window length w, a *subsequence* is a collection of w contiguous values from T. The set of all subsequences of length w of T is denoted with T^w. With T_i^w we denote the subsequence of length w starting from the i-th point of T. When clear from context, we omit the superscript w.

For any given pair of subsequences $m \in T^w \times T^w$, we denote with $d(m)$ the distance between the two subsequences. Two overlapping subsequences are referred to as *trivial match*. The definition can be adjusted to suit different needs (e.g., a trivial match could be given by two subsequences with an overlap of at most $w/4$) with no changes in the algorithms. By extension, we say that two pairs of subsequences overlap if any subsequence in one pair overlaps with either subsequence in the other pair.

We define a total ordering over the pairs of subsequences: given two pairs m_1 and m_2, we have $m_1 < m_2$ if $d(m_1) < d(m_2)$, breaking ties in favor of the pair having the subsequence occurring earliest.

In this paper we focus on computing, for a given integer $k > 0$, the k pairs of subsequences with the smallest distance, ignoring trivial matches. Our definition of the problem is along the lines of [24, 29, 32, 38].

Definition 3.1 (Top-k motifs). Given a time series T and a length w, the top-k motifs are the k pairs of subsequences of length w with smallest distance, such that no two subsequences in any pair overlap with each other.

Observation 1. The matrix profile [45], which reports the nearest neighbor of each subsequence, is often used to address the problem above: the first motif is identified as the pair at minimum distance in the matrix profile, the second motif is the pair at the next minimum not overlapping with the first pair, and so on. We note that this procedure might miss some motifs defined as above. This happens when a motif is composed of two subsequences whose nearest neighbors are overlapping with a higher-ranked motif. In this situation, the matrix profile does not contain the information to identify such a pair. We will provide an example of such case in our experimental section.

As a distance function between subsequences $x, y \in T^w$ we will use the popular z-normalized Euclidean distance

$$d(x,y) = \sqrt{\sum_{i=0}^{w-1} \left(\left(\frac{x_i - \mu_x}{\sigma_x} \right) - \left(\frac{y_i - \mu_y}{\sigma_y} \right) \right)^2}$$

The z-normalized Euclidean distance is equivalent to the Pearson correlation coefficient ρ. In particular, we have $\rho(x,y) = 1 - \frac{d^2(x,y)}{2w}$, for positive correlation values [33, 39]. Therefore, minimizing the z-normalized Euclidean distance is equivalent to maximizing the Pearson correlation coefficient. All our arguments and algorithms also work with the non-normalized Euclidean distance simply by skipping the normalization step.

Example 3.2. Figure 1 reports the first 13 000 points of a classic example in the motif discovery literature [29, 32]: a time series of length 18 666 derived from an experiment modeling the behavior of the Beet leafhopper insect. We highlight the top-3 motifs of length $w = 400$ in blue, orange, and green.

3.2 Locality Sensitive Hashing

In this section, we give a brief overview of the LSH framework. We refer the interested reader to [16, 44] for a more in-depth discussion. Furthermore, we provide interactive online supplemental material[1] to provide more background.

Definition 3.3 (Locality Sensitive Hashing [23]). Let (X, dist) be a distance space and $\mathcal{H}$ be a family of functions $h : X \to R$. We say that $\mathcal{H}$ is locality-sensitive if for any three $x, y, z \in X$ and a random $h \in \mathcal{H}$ we have that

$$\text{dist}(x,y) \leq \text{dist}(x,z) \implies \Pr[h(x) = h(y)] \geq \Pr[h(x) = h(z)]$$

In other words, under a random hash function sampled from $\mathcal{H}$, vectors in X similar to each other are more likely to hash to the same value than vectors that are far away. We denote the event of two vectors hashing to the same value with *collision*, and by extension two vectors *collide* when they hash to the same value *under the same hash function*.

An LSH scheme for the Euclidean distance was introduced by Datar et al. [17]. For $\mathbb{R}^w$ and a parameter $r \in \mathbb{R}$ (the *quantization* parameter), a hash function is constructed by sampling a random vector $a \in \mathbb{R}^w$ and a random value $b \in [0, r]$. The hash function is then defined as

$$h_{a,b}(x) = \left\lfloor \frac{a \cdot x + b}{r} \right\rfloor \quad \text{for } x \in \mathbb{R}^w \tag{1}$$

where $a \cdot x$ is the dot product between a and x. The collision probability of two vectors at Euclidean distance d is

$$p(d) = 1 - 2 \cdot norm\left(-\frac{r}{d}\right) - \frac{2}{\sqrt{2\pi} r/d} \left(1 - e^{-(r^2/2d^2)}\right) \tag{2}$$

where *norm* is the cumulative distribution function of the Standard Normal distribution[2] [17]. By z-normalizing vectors before computing the hash function, the LSH scheme above can also be applied to the z-normalized Euclidean distance.

To modulate the collision probability between two vectors x and y in X, usually the so-called *powering* technique is used: for an integer $K \geq 1$, we sample pairs $(a_1, b_1), \ldots, (a_K, b_K)$ as described above and build a composite hash function

$$h'(x) = \left(h_{a_1,b_1}(x), \ \ldots, \ h_{a_K,b_K}(x) \right)$$

[1] https://cecca.github.io/attimo/VLDB-supplemental/

[2] We remark that this LSH family samples from a Gaussian distribution because of its 2-stable properties, and is unrelated to the particular distribution of data.

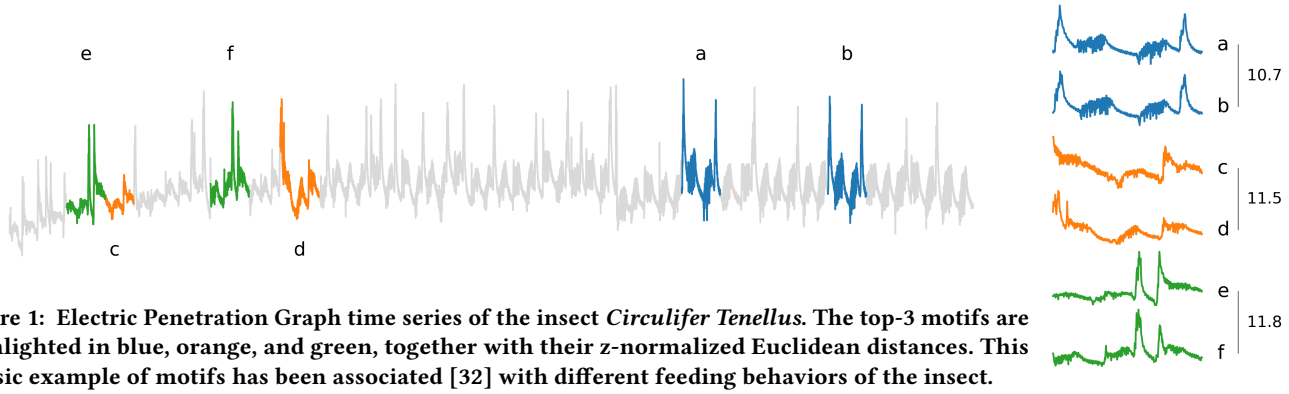

Figure 1: Electric Penetration Graph time series of the insect *Circulifer Tenellus*. The top-3 motifs are highlighted in blue, orange, and green, together with their z-normalized Euclidean distances. This classic example of motifs has been associated [32] with different feeding behaviors of the insect.

Table 1: Summary of notation.

T	time series
T^w	set of subsequences of T of length w
T_h^w	subsequence of T at index h
$m \in T^w \times T^w$	pair of subsequences
$\mathcal{H}_{\ell_2}$	family of LSH functions on the Euclidean space
K	number of LSH concatenations
L	number of LSH repetitions
i	index over K concatenations
j	index over L repetitions
$h_{K,j}(T_a^w)$	hash value of length K for T_a^w at repetition j
$\vec{h}_{i,j}(T_a^w)$	prefix of length i of $h_{K,j}(T_a^w)$
$p(d(m))$	collision probability of pair m, at distance $d(m)$

by concatenating the outputs of the K individual hash functions into a tuple of length K. The collision probability becomes then $(p(d))^K$. Using larger values of K lowers the probability that dissimilar points collide, thus reducing the number of distance computations to be performed, but also lowers the probability that similar points collide.

Repeating the above process L times with a new set of K hash functions being sampled in each repetition allows to balance the effect of using a large value of K. In fact, a pair of subsequences at distance d will collide in at least one of the L repetitions with probability

$$1 - \left(1 - p(d)^K\right)^L. \tag{3}$$

3.3 A naïve application of LSH to time series

Before presenting our adaptive algorithm, we will show a straightforward way of applying LSH to time series data to find the top-k motifs of length w.

First, we need to set three parameters: the quantization width r, the hash length K, and the number of repetitions L. Then, we perform L iterations. In each iteration j, we slide a window of length w over the entire time series, thus enumerating all its subsequences. For each subsequence T_a, we compute its hash value $h_{K,j}(T_a)$ of length K and use it as a key in a dictionary, with the subsequence itself being the associated value. Next, we compute the pair-wise distances between all the subsequences that collide on the same hash key, keeping track of the k subsequences with smallest distance. After L iterations, the algorithm returns the top-k motifs found.

In the above algorithm we assume that there is a sufficient number of collisions to find at least k motifs avoiding non-trivial matches. In fact, setting the three parameters changes the probability of any pair to collide, and thus the number of pair-wise distances that need to be computed. They also have a more fundamental consequence on the correctness of the output. An inappropriate parameter setting might make it too unlikely for some of the top-k pairs to collide, implying a non-negligible probability of not reporting the true top-k motifs. Note that under such conditions k pairs can still be returned. Since pairs with a larger distance than the true top-k have a non-zero collision probability, it might happen by chance that false positive pairs collide, while some of the true top-k do not.

Example 3.4. Consider the algorithm above running on the time series in Figure 1 with $w = 400$ and $r = 20$. The task is to find the top-3 motifs, with the third motif being at distance 11.8. Setting $K = 6$ concatenations and $L = 10$ repetitions leads to the evaluation, in expectation, of approximately 880 000 distances (i.e., $L \cdot \sum_{m \in T^w \times T^w} p(d(m)^K)$, by the linearity of expectation), with a probability of finding the true third motif of about ≈ 0.24 (by Eq. (3)). Increasing K to 12 reduces the expected distance computations to only 1 900, but now the probability that the third motif is among the top-3 pairs is only 0.007. Increasing the number of repetitions to $L = 200$ would raise this probability to 0.14, which is still rather small. Therefore, one might set $K = 6$ and $L = 200$, which makes the probability of finding the third motif ≈ 0.99, at the expense of computing 17 million distances (about 10% of all possible pairwise distances between subsequences).

We stress two issues about the above example. (1) We are able to evaluate the probability of finding the correct set of top-3 motifs only because we already know their distances (see Equation (3)). Therefore, controlling the failure probability requires the knowledge of the answer to the problem. We will address this issue with a self-tuning algorithm in the next section. (2) Most of the distances computed across repetitions are duplicates, since close pairs will

collide in most repetitions; we will address this, along with other efficiency issues, in Section 5.

4 ADAPTIVE ALGORITHM

In this section, we present a self-tuning algorithm, termed Attimo, to find the top-k motifs in a time series with error probability δ (cf. Algorithm 1). To circumvent the issues highlighted in the previous section, we adopt an approach reminiscent of PUFFINN [5] and LSH Forest [7]: we set *maximum allowed* K and L, and then the algorithm will automatically select appropriate values in the specified range using information about the data distribution gathered during the execution. As for the *quantization width parameter* r (Eq. (1)), the algorithm automatically derives a reasonable value in the preprocessing phase, as we shall see.

4.1 High level structure

Let δ be a user defined failure probability. Similarly to the naïve algorithm, we run L repetitions with hash functions of length K, while maintaining a collection of the k pairs at smallest distance found so far in a priority queue sorted by increasing distance (Lines 5–13). Differently from the naïve algorithm, at the end of each repetition j we check whether the probability of having missed a pair at distance smaller than the k-th pair in the queue is smaller than δ. If this is the case, the algorithm terminates and returns the top-k pairs. If the algorithm does not terminate after L repetitions, the probability of collision using hashes of length K was too small for the dataset at hand. Therefore, we repeat the above procedure, maintaining the top-k queue built so far and considering hashes of length $K-1$. The process continues, using hash values of length $K-2$, $K-3$, ..., until the probability of missing a pair in the true top-k becomes smaller than δ. Eventually in the worst case, if even with hashes of length 1 the algorithm is unable to meet the failure probability, all pairs are considered as candidates, and therefore we can use any all-to-all exact algorithm to find the solution. Note that in the above procedure the hash values are computed only once for length K; for the hash values of length $K-1$ to 1 we consider the corresponding prefixes.

Within the allowed constraints on the parameter K and L, our algorithm thus selects the *most selective* (i.e. minimizing the number of distance computations) configuration that meets the required failure probability, based on the knowledge gained about the distance distribution of pairs gained in the process. Therefore, our algorithm is *self tuning*.

Example 4.1. Consider again the setup of Example 3.4, this time using $L = 9$, $K = 4$, and failure probability $\delta = 0.1$. Figure 2 depicts the arrangement of 8 subsequences and three repetitions. The algorithm starts by considering hash values of length 4. In the first repetition there is only one collision between `a` and `b`. Hence, the queue of top-k candidates becomes `TOP` $= \langle$`(a, b)`$\rangle$. In the second repetition, again only `a` and `b` collide. In the third iteration there is no collision, and lets assume that there are no new collisions in the remaining six repetitions.

Since `TOP` contains only one motif, the algorithm starts again from the first repetition, but now considering only the hash prefixes of length 3. In the first repetition, we see two new collisions, yielding `TOP` $= \langle$`(a, b)`, `(c, d)`, `(g, f)`$\rangle$, in increasing order of distance.

Algorithm 1: Attimo

Input: Time series T, window length w, number of motifs k, failure probability δ, maximum number of repetitions L, maximum hash length K.
Output: The top-k motifs, with probability $1 - \delta$

```
// Initialization and index construction
1  Estimate the parameter r
2  for j ← 1, ..., L do
3      for T_a ∈ T^w do
4          Precompute h_{K,j}(T_a)
   // Iterate over prefixes until top-k pairs are found
      with failure probability < δ
5  TOP ← empty priority queue
6  for i ← K, K − 1, ..., 1 do
7      for j ← 1, ..., L do
8          foreach (T_a, T_b) ∈ T^w × T^w : h⃗_{i,j}(T_a) = h⃗_{i,j}(T_b) do
9              TOP.insert((T_a, T_b))
10             if |TOP| > k then
11                 TOP.pop()
12         if |TOP| = k ∧ Stop(TOP.max(), i, j) then
13             return TOP
14 return true top-k by checking all pairs

15 Function Stop(d, i, j) is
16     if i = K then return (1 − p(d)^K)^j ≤ δ
17     else return (1 − p(d)^i)^j · (1 − p(d)^{i+1})^{L−j} ≤ δ
```

The queue now contains $k = 3$ items. The distance between `g` and `f` is ≈ 29.1. As we will argue in Lemma 4.3, the probability of having missed a pair at a smaller distance at this point of the execution is $\approx 0.93 > \delta$. Therefore, the algorithm continues with the second repetition. A new collision between `f` and `e` is found, which are going to replace `(g, f)` in the queue, since they are at distance ≈ 11.8.

The queue now contains the true top-3 motifs, but the algorithm has no way of knowing it. The probability of not having found a pair at distance ≤ 11.8 yet in the second repetition, for prefixes of length 3, is in fact ≈ 0.35. Hence, the algorithm repeats the above process, with the queue remaining unchanged, since it contains the top-k motifs already. When the 4-th repetition using hashes of length 2 is performed, the failure probability is $\approx 0.08 < \delta$, and thus the algorithm stops.

Observe in the above example that some pairs, such as `(a, b)`, collide in almost all repetitions for all prefix lengths, by virtue of their small distance. Computing the distance between such subsequences is wasted work, and we will discuss in Section 5 how to avoid such duplicate computations.

4.2 Details of the algorithm

Setting the quantization width (Line 1). Recall that the *quantization width* r (see Eq. (1)) is the parameter of hash functions that controls how the projection of subsequences on the real line is

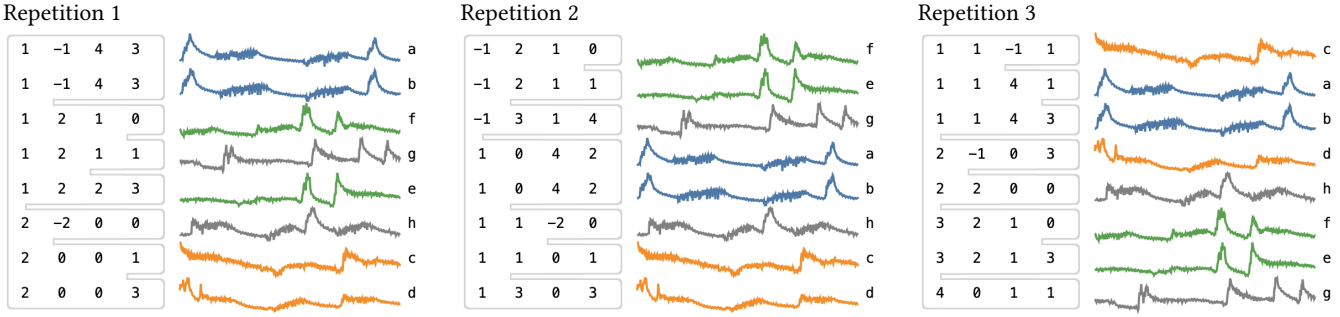

Figure 2: Three LSH repetitions with $K = 4$ for eight subsequences of the time series of Figure 1, including the top-3 motifs.

discretized into hash values. While the precise setting of this parameter does not affect the correctness of the algorithm, it may help with improving the performance. Specifically, the lower the value, the smaller the collision probability. Therefore, it plays a role similar to K. For a fixed K, a value too small will result in no collisions for long prefix lengths, thus wasting some of the hash computations. Conversely, a value too large will result in too many collisions already in the first iteration of the algorithm, when the full hashes are considered.

To set a reasonable value of r that strikes a good tradeoff between the two aforementioned extreme scenarios, we adopt the following simple heuristic. We set $r = 1$ in Eq.(1) and perform a single LSH repetition with hash values of length K. If there are no collisions, then we double r and repeat the process until we find at least one collision. We then use the resulting value of r in the rest of the algorithm.

Index data structure (Line 2). For each repetitions $j \leq L$, each subsequence T_a will be associated to the corresponding hash value $h_{K,j}(T_a)$. Later, the algorithm will need to access all the subsequences colliding on prefixes of these hash values. To efficiently support this access pattern, a possibility is to store all the hash values of a single repetition in a trie of height K, where the leaves contain the starting indices of subsequences with a given LSH hash value. However, one of the disadvantages of tries is that, being pointer-based data structures, they are not cache-efficient. Therefore, instead of building explicitly tries, we store the hash values in flat arrays, one per repetition, which are then sorted lexicographically. Given this ordering, for a given prefix length i, all the subsequences sharing the same prefix will be adjacent to one another. Therefore, the enumeration of colliding pairs at line 8 in Algorithm 1 can be implemented by considering slices of the array of hashes where all elements have the same prefix.

Example 4.2. In Figure 2 (left) hashes are sorted lexicographically from top to bottom. At prefix length 2, the slices of the array of hashes containing colliding pairs are $[a, b]$, $[f, g, e]$, $[h]$, $[c, d]$, from top to bottom.

Candidates data structure (Line 5). The data structure that stores the candidate motif pairs needs to support three operations: returning the maximum distance (`TOP.max()`), removing the pair at the maximum distance (`TOP.pop()`), and inserting a new pair (`TOP.insert()`). All of these operations are handled by standard priority queues, but in our setting we have the additional complication of trivial matches, which need to be ignored.

A simple implementation of this data structure to hold candidate pairs, supporting linear-time insertions and ignoring trivial matches is as follows. We maintain an array `TOP` of size $k+1$, where all entries are initially undefined. Retrieving the k-th element is a simple array lookup, and removing the last element is simply setting to undefined the last element which was not undefined. To insert a pair m, we first scan the array `TOP` until the position i such that $d(\texttt{TOP[i]}) < d(m) \leq d(\texttt{TOP[i+1]})$. If during this scan we encounter a pair overlapping with m, then m is not inserted. Otherwise, we insert m at i, and remove from the rest of the array (i.e. from $i+1$ onwards) any pair overlapping with m. An insertion might make the `TOP` queue grow too large: if this is the case, we remove the $k+1$ element, which might be the one we just inserted.

Observation 2. Algorithm 1 is *data adaptive* because its performance is determined by how pairwise distances are distributed. In fact, the number of iterations carried out by the algorithm depends on the collision probability of the top-k motif pairs, which is a function of their distance. Furthermore, if many subsequence pairs are at a distance similar to the top-k motifs, the algorithm will have to compute many distances to distinguish the true top-k motifs. Conversely, it will complete swiftly on an easier time series where the top motifs are markedly different from the other pairs.

As we shall see in Section 6, this intuition is formalized by the inclusion of the data-dependent collision probabilities in the expressions of the complexity.

4.3 Correctness of the algorithm

The lemma below states the failure probability of pairs in Algorithm 1, which will then be used as a stopping condition in the algorithm itself.

Lemma 4.3. *In the above construction, the probability of a pair at distance d* not colliding *in any of the repetitions considered so far, at prefix length i and repetition j, is*

$$\begin{cases} \left(1 - p(d)^i\right)^j \cdot \left(1 - p(d)^{i+1}\right)^{L-j} & \text{if } i < K \\ \left(1 - p(d)^K\right)^j & \text{otherwise} \end{cases}$$

Proof. In general, for a hash of length i, the probability of a pair at distance d to collide is $p(d)^i$. Therefore, the probability

of never colliding over j independent repetitions is $\left(1 - p(d)^i\right)^j$. The statement follows directly for the case $i = K$, in any of the j repetitions. For the case $i < K$, consider that we have performed j repetitions with hashes of length i, *but also* $L - j$ repetitions with hashes of length $i + 1$. The statement follows by multiplying the probabilities of never colliding in j repetitions with hashes of length i *and* never colliding in $L - j$ repetitions with hashes of length $i + 1$, which are independent. □

We now prove two lemmas corresponding to two different ways of employing our algorithm. In Lemma 4.4 we allow each returned motif to fail independently of the others. In this setting, the expected fraction of successful motifs corresponds to the expected recall of the algorithm. In Lemma 4.5, we set the algorithm so that *all* returned motifs are correct, an event that happens with a controlled probability.

Lemma 4.4. *Algorithm 1 finds the true top-k motifs, each with probability at least* $1 - \delta$.

Proof. If the algorithm reaches line 14, then it returns all true motifs with probability 1, since all the pairs of subsequences will be evaluated.

Consider now the case that the stopping condition is met at iteration i' and j' of the two nested loops, and let $m_1, \ldots, m_k$ be the motifs returned by the algorithm, sorted by increasing distance. By the monotonicity of the collision probabilities, we have that the failure probability of m_h for $h \in [1, k]$ is upper bounded by the failure probability of m_k. Such probability is given by Lemma 4.3, and the stopping condition ensures that it is $\leq \delta$. This holds for all returned pairs independently, and the statement follows. □

Given that each motif fails independently with probability $1 - \delta$, we have that the expected fraction of failing motifs is $1 - \delta$. In other words, Attimo has an expected recall of $1 - \delta$.

Lemma 4.5. *When invoked with failure probability* $\delta' = \delta/k$, *Algorithm 1 finds all true top-k motifs with probability at least* $1 - \delta$.

Proof. By Lemma 4.4 we have that each pair fails independently with probability $\leq \delta' = \delta/k$. The statement follows by applying a union bound on the k pairs being returned. □

5 OPTIMIZATIONS

In this section, we describe several optimizations that we incorporate into Algorithm 1 in order to reduce the running time, mainly by addressing the two most expensive operations of the algorithm: computing hash functions and distances between subsequences.

5.1 Speeding up LSH function evaluations

Evaluating LSH functions can be a costly operation, typically requiring time at least proportional to the size of the representation of the items at hand. In particular, for time series subsequences with the (z-normalized) Euclidean distance, evaluating the LSH functions described in Section 3.2 requires time linear in the subsequence length w, leading to an overall computation time of $O(wKL)$ to build the LSH index. Clearly this is undesirable, since it can hamper the extraction of long motifs.

	`[-1, 3]`	`[0, 2]`	`[1, 1]`
`[ 1, 4]`	`[ 1, -1, 4, 3]`	`[ 1, 0, 4, 2]`	`[ 1, 1, 4, 1]`
`[-2, 0]`	`[-2, -1, 0, 3]`	`[-2, 0, 0, 2]`	`[-2, 1, 0, 1]`
`[0, 1]`	`[ 0, -1, 1, 3]`	`[ 0, 0, 1, 2]`	`[ 0, 1, 1, 1]`

Figure 3: Application of tensoring to build, for $L = 9$ repetitions, hash values of length $K = 4$ out of $K\sqrt{L} = 12$ hash values.

To address this problem we combine two techniques. First, we reduce the number of hash functions to be evaluated using a technique known as *tensoring* [16]. Then, we decouple the complexity of hash function evaluations from the length of the subsequences by leveraging the fact that subsequences overlap.

5.1.1 Tensoring. We use a variant of the tensoring technique introduced in [16], which allows to reduce the number of hash function evaluations from $K \cdot L$ to $K \cdot \sqrt{L}$. This technique is also used in PUFFINN [5]: we provide a different statement of the failure probability. Furthermore, in our instantiation we extend it to support detection of redundant collisions, as we will see later on.

For simplicity of exposition, assume that K is even and that L is a perfect square. Consider a LSH family $\mathcal{H}$, and let $\mathcal{H}_\ell$ and $\mathcal{H}_r$ be two collections of $\sqrt{L}$ hash functions, each sampled from $\mathcal{H}^{K/2}$. We define the collection

$$\overline{h}_{K,j} = (h_{\frac{K}{2},\ell}, h_{\frac{K}{2},r}) \in \mathcal{H}_\ell \times \mathcal{H}_r \quad \text{where} \quad \begin{cases} \ell = j \div \sqrt{L} \\ r = j \bmod \sqrt{L} \end{cases} \tag{4}$$

for $1 \leq j \leq L$. For a subsequence T_a, the hash value $\overline{h}_{K,j}$ is obtained by interleaving values from $h_{\frac{K}{2},\ell}(T_a)$ and $h_{\frac{K}{2},r}(T_a)$. Therefore we have L repetitions of hash functions of length K using only $K \cdot \sqrt{L}$ hash function evaluations.

Example 5.1. Figure 3 illustrates the tensoring technique for computing the hash values of subsequence `a` (other subsequences are associated with similar tables) in our running example in Figure 2. For instance, the first row in the tensor data structure computes the hash values for subsequence `a` reported in Figure 2.

For $L = 9$ and $K = 4$ we can build all the required hash values using just 12 hash function evaluations instead of 36. Long hash values are built by interleaving shorter ones, using all possible combinations.

In a typical configuration used in our experimental evaluation with $K = 32$ and $L = 200$, we perform 453 hash function evaluations per subsequence instead of 6 400, i.e. only $\approx 7\%$.

Employing this tensoring technique changes the failure probability of our algorithm. Intuitively, since we no longer have independence between repetitions, a failure in one of the tensored repetitions has a broader impact, since it will make $\sqrt{L}$ repetitions fail (i.e. a full column or row in Figure 3). As a consequence, the number of iterations required to meet the stopping condition increases. The following lemma upper bounds the failure probability, and replaces the expression of the stopping condition in Algorithm 1. For space reasons, we defer the proof to the full version of the paper, hinting here at the high level idea.

Lemma 5.2. *With the tensoring approach, the probability that a pair at distance d has not collided in any of the repetitions considered so far, at prefix length i and repetition j, is upper bounded by*

$$Fc(i,j) \cdot Fp(i,j)$$

where

$$Fc(i,j) = 1 - Sp\left(\left\lfloor \frac{i}{2} \right\rfloor, j \div \sqrt{L}\right) \cdot Sp\left(\left\lceil \frac{i}{2} \right\rceil, j \bmod \sqrt{L}\right)$$

$$Fp(i,j) = 1 - Sp\left(\left\lfloor \frac{i+1}{2} \right\rfloor, \sqrt{L} - j \div \sqrt{L}\right) \cdot Sp\left(\left\lceil \frac{i+1}{2} \right\rceil, \sqrt{L} - j \bmod \sqrt{L}\right)$$

$$Sp(i',j') = 1 - \left(1 - p(d)^{i'}\right)^{j'}.$$

Proof. Recall that with tensoring we have two collections $\mathcal{H}_\ell$ and $\mathcal{H}_r$ of $\sqrt{L}$ hash functions each. We deem with *left* and *right* the repetitions from $\mathcal{H}_\ell$, and $\mathcal{H}_r$, respectively. Similarly to Lemma 4.3, the probability of failing at repetition $j \in [1, L]$ and prefix length i is the probability of failing both in all the repetitions up to j at prefix length i and all the repetitions from j to L at prefix length $i+1$. Using tensoring we no longer have independence between all repetitions. Therefore, we will focus on two subsets of independent repetitions: the ones given by the interleaving of the hash values from the first $j \div \sqrt{L}$ repetitions on the left (i.e. using hash functions from $\mathcal{H}_\ell$) and $j \bmod \sqrt{L}$ on the right (hash functions from $\mathcal{H}_r$), and the ones given by hash values built from the last $\sqrt{L} - j \bmod \sqrt{L}$ repetitions on the left and the $\sqrt{L} - j \bmod \sqrt{L}$ on the right.

With $Sp(i', j')$ we denote the probability of having at least one collision j' independent repetitions with hashes of length i'. In the lemma statement, Fc denotes the probability of failing in the first set of repetitions, at the *current* prefix length, whereas Fp is the probability of failing at the *previous* prefix length (i.e. $i+1$). These two failure events are independent, since the corresponding hash values are obtained from disjoint subsets of repetitions in the left and right sets of tensored repetitions. Therefore, the probability of their intersection is the product of their probabilities, and the statement follows. □

5.1.2 Leveraging subsequences structure. One of the main differences with general metric spaces is that consecutive subsequences from the same time series overlap with each other. Observe that evaluating a hash function as defined in Equation (1) involves a dot product between a subsequence and a random vector v. If we were to consider each subsequence of length w in isolation, this operation alone would have an overall complexity of $\Theta(w \cdot |T^w|)$, where $|T^w|$ is the number of subsequences. By using the Cyclic Convolution Theorem, instead, we can compute the dot product of v with all subsequences in T^w in time $O(n \log n)$. The idea, which has also been successfully leveraged for distance computations (see [34] and subsequent works), is to compute the Fourier Transform of the input time series T and of the reversed vector v, padded with zeros so to have the same length as T. Then, the inverse Discrete Fourier Transform of the element-wise product of the two transformed vectors holds in position i the dot product $v \cdot T_i^w$. Hence, for a LSH function defined by the vector a, the random offset b, and the width parameter r, we can compute all hash values in one go as follows:

$$z = DFT_n^{-1}\left(DFT_n(T) \odot DFT_n((a_w, a_{w-1}, \ldots, a_1)|\mathbf{0}_{n-w})\right) \quad (5)$$

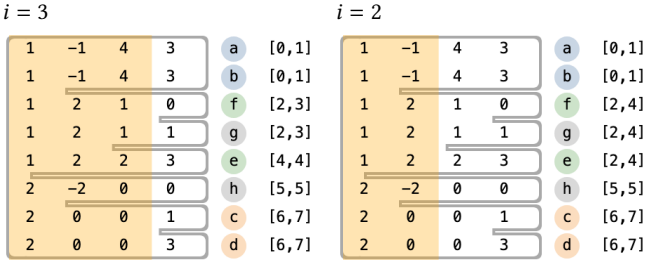

Figure 4: Ranges being used to detect collisions within repetition 0 of Figure 2.

$$h(T_i^w) = \left\lfloor \frac{z_i + b}{r} \right\rfloor \quad (6)$$

where $\odot$ is the element-wise product, z is the sliding dot product, $n = |T|$, DFT_n is the Discrete Fourier Transform of a vector of length n, and $|\mathbf{0}_{n-w}$ denotes padding a vector with $n - w$ zeros. Equation (5) takes time $O(n \log n)$ (for the DFT computation), whereas Equation (6) is a constant time operation for each i, and thus takes time $O(n)$ overall.

5.2 Reducing the number of distance computations

The fundamental property of LSH that close subsequences collide with high probability implies that we can expect to have a lot of duplicate collisions across repetitions. Furthermore, since we consider variable length prefixes, pairs colliding on longer prefixes will obviously collide on shorter prefixes in the same repetition. Both these facts lead to a lot of potential wasted work, where we compute the distance of the same pair several times.

A very simple solution to this issue is to maintain a flag for each pair, to be set when a distance is first computed. The quadratic space requirement makes this approach not applicable.

5.2.1 Duplicate collisions within the same repetition. To detect duplicates within the same repetition and across prefix lengths (i.e. to prevent pairs colliding on long prefixes to be considered again when evaluating shorter prefixes) we leverage the fact that we maintain the hash values for one repetition in a lexicographically sorted array. For repetition j and for every subsequence T_a we maintain a pair of indices tracking the range of the array of hash values with which T_a has collided for longer prefixes. When evaluating the collisions for T_a at prefix length $i < K$, the subsequences falling within said range are ignored altogether. When the prefix length is decreased to $i - 1$, the pair of indices for T_a is updated to the range that contained T_a at i.

Example 5.3. In Figure 2, consider the subsequences f and g in the first repetition. Once the algorithm reaches prefix length $i = 3$, the pair of indices is set to $[2, 3]$, as shown in Figure 4, left. In the next iteration with prefix length $i = 2$, the subsequences f and g will not be compared again, and they will only be compared with subsequences outside the range $[2, 3]$, i.e., only with subsequence e. The pair of indices for the subsequences f, g, and e is then updated to $[2, 4]$, as shown in Figure 4, right. As a consequence, for $i = 1$ these three subsequences will will only be compared to a and b.

5.2.2 Duplicate collisions across repetitions. To detect duplicate collisions across repetitions, we will exploit the tensoring data structure that we are already using to speed up LSH function evaluations, similarly to [4]. Tensoring allows to compactly store information about all hash values (and their prefixes) associated to a subsequence in all repetitions. The following lemma, which we adapt from [4] ensures that we can use this information to check for duplicates using only $O\left(|T^w|K\sqrt{L}\right)$ space overall, instead of quadratic.

Lemma 5.4. *Let T_p^w and T_q^w be two subsequences colliding at repetitions j on prefixes of length i. Then, we can decide in time $O\left(K\sqrt{L}\right)$ if there exist a repetition $j' < j$ where they collided as well, using space $O\left(|T^w|\ K\sqrt{L}\right)$ overall.*

Proof. For each subsequence we are storing $K\sqrt{L}$ hash values, and thus the space requirement follows. Let $j \in [0, L)$ be a repetition in which the subsequences T_p^w and T_q^w collide on the prefix of length i. Let $\ell = j \div \sqrt{L}$ and $r = j \bmod \sqrt{L}$, and consider the definition of tensored LSH function of Equation (4). If there exist values $0 \leq \ell' < \ell$ and $0 \leq r' < r$ such that T_p^w and T_q^w collide under $(h_{i,\ell'}, h_{i,r'})$, or it exists $0 \leq b' < b$ such that T_p^w and T_q^w collide under $(h_{i,\ell}, h_{i,r'})$, then the collision at the j-th repetition is a duplicate. Since we are storing the hash values associated to each subsequence and since $\ell, r < \sqrt{L}$, evaluating the above conditions requires time $O(K/2)$ for each hash value, yielding an overall time of $O\left(K\sqrt{L}\right)$. □

6 COMPLEXITY

For clarity of exposition, we study the different contributions to the running time of Algorithm 1 separately. In particular, we consider the time to build the index, the number of candidate pairs that are evaluated for removing duplicates, and the number of distances that are computed after duplicates are removed.

Theorem 6.1. *The index construction at line 4 of Algorithm 1 takes time $O\left(K \cdot \sqrt{L} \cdot n \log n\right)$.*

Proof. For each subsequence we need to compute $K\sqrt{L}$ hash values, thanks to tensoring. For a fixed hash function out of the $K\sqrt{L}$, we can compute all the dot products in $O(n \log n)$ time, using the cyclical convolution theorem. The theorem follows. □

Theorem 6.2. *Let m_k be the top-k motif, and let $i' \leq K$ and $j' \leq L$, respectively, be the first prefix length and repetition such that the stopping condition is met. Algorithm 1 considers*

$$j' \sum_{m \in T^w \times T^w} p(d(m))^{i'} + (L - j') \sum_{m \in T^w \times T^w} p(d(m))^{i'+1}$$

candidate pairs, in expectation.

Proof. Recall that for any of the L repetitions, the expected number of collisions at prefix length i is $\sum_{(x,y) \in T^w \times T^w} p(d(m_k))^i$. Furthermore, for any given repetition a pair colliding at prefix length i will not be considered again at any prefix length $< i$. Therefore, the collisions of all prefixes $\geq i$ can be accounted for once at prefix length i. Now, partition the repetitions in two groups: the first j' repetitions are evaluated until prefixes of length i' to confirm the top-k motif; the last $L - j'$ repetitions consider prefixes of $i' + 1$. The theorem follows by the linearity of expectation over the expected number of collisions in these two groups. □

Theorem 6.3. *With the optimizations presented in Section 5, Algorithm 1 evaluates*

$$\binom{|T^w|}{2} - \sum_{m \in T^w \times T^w} \left(1 - p(d(m))^{i'}\right)^{j'}$$

distances in expectation, where i' and j' are the first prefix length and number of repetitions meeting the stopping condition, respectively.

Proof. Since we detect duplicate collisions both across repetitions and prefix lengths, each pair's distance is computed at most once. Therefore, at prefix length i' we compute (once) the distances of all the pairs colliding in at least one repetition. Given that the probability of a pair at distance d to collide in at least one out of j' repetitions at prefix length i' is $1 - (1 - p(d)^{i'})^{j'}$, the theorem follows by the linearity of expectation. □

Observe that in the above statement the number of distance computations is upper bounded by the total number of pairs in the set of subsequences. The second term accounts for the pairs that *do not* collide in any repetition: the larger the value of j', the smaller this number, simply because with more repetitions there are more chances of collision; the larger the value of i', the larger the number of non-colliding pairs, since for longer prefixes collisions are less likely.

Example 6.4. To make the above two theorems more concrete, consider again our running example time series, and suppose that the algorithm stops at prefix length $i' = 6$ at repetition $j' = 165$. The number of pairs that are considered, including duplicates is $\approx$ 16 million (Theorem 6.2), the number of distance computations actually carried out is $\approx$ 13 million, in expectation (Theorem 6.3). The total number of pairs of subsequences is in the order of 167 millions, therefore we are computing only about 7% of the possible distances. As we shall see in the experimental section, on larger time series the computational savings are even more substantial.

7 EXPERIMENTS

This experimental evaluation aims at answering the following questions:

- How does our proposed approach compare with the state of the art in terms of running time?
- How does the number of repetitions influence the performance of our algorithm?
- What is the influence of the number of returned motifs on the running time?
- How do the algorithms scale with respect to the input size?
- What is the actual recall of the algorithm?

Baselines. We compare our solution to Scamp [48], which is the state of the art implementation of the matrix profile approach. Scamp has been independently [47, 48] found to be faster than previous algorithms for motif discovery, including [15, 26, 31, 32]. Along with it we consider PreSCRIMP [46], which provides an empirically good approximation to the matrix profile and hence, the motifs, albeit with no theoretical guarantees. Furthermore, we consider as

Table 2: Information about the benchmark datasets.

dataset	n	window	RC_1	RC_{10}
freezer	7 430 755	5 000	23.79	7.95
ASTRO	1 151 349	100	11.04	8.63
GAP	2 049 279	600	11.08	9.17
Whales	308 941 605	140	33.69	21.66
ECG	7 871 870	1 000	146.31	109.06
Seismic	1 000 000 000	100	2 658 166.60	274.44
HumanY	26 415 045	18 000	1 574.28	581.03

a baseline for the top-1 case the LL algorithm recently introduced in [25]. Finally, we include the random projection algorithm RPROJ[3] of [15] based on [11].

Experimental setup. We run our experiments on two machines. The first machine is equipped with a Intel(R) Xeon(R) CPU E5-2667 v3 clocked at 3.20GHz, with 94GB of RAM. To run the SCAMP GPU-based implementation in a favorable setup, we also use a Nvidia Tesla V100 PCIe with 32GB of memory.

Software implementation. Our implementation of the algorithm is freely available at https://github.com/Cecca/attimo, and features all the optimizations described above. Furthermore, in our implementation we report each motif as soon as it meets the stopping condition, rather than waiting for the k-th pair to report the entire solution. In the following, we refer to our implementation as ATTIMO, which stands for Adaptive Timeseries Motifs.

Datasets. We consider the following datasets in our experimental evaluation. `ECG` is an electrocardiogram from a stress recognition study in automobile drivers [21]. `ASTRO` is a dataset of celestial objects [42]. `GAP` is a time series of the global active electric power in a household in France over a period of 47 months, sampled every minute [22]. `HumanY` is the human Y chromosome transformed into a time series using the approach of [40] and made available by [19]. Similarity between subsequences of this time series suggest similarity in the original DNA subsequences `Freezer` records the power usage of a freezer home appliance over the course of two years [35]. `Whales` is derived from the signal of an underwater microphone [37] by considering the energy in the frequency band between 360 and 370 Hz. Motifs in this dataset correspond to whales vocalizations. We provide the preprocessing scripts in our code repository. `Seismic` is the largest dataset in our study, comprising 1 billion measurements at 20Hz of a seismometer at VCAB station of the Parkfield High Resolution Seismic Network [6, 43]. To the best of our knowledge, this is the largest dataset used to date for the motif discovery problem [48].

Table 2 reports statistics about the datasets, including the motif length we considered. To measure the *difficulty* of a dataset for LSH to solve, we adapt the definition of *relative contrast* [20] to our setting: for a given pair of subsequences at distance d, the relative contrast of the pair is $\bar{d}/d$, where $\bar{d}$ is the average pairwise distance between any two non-overlapping subsequences in the time series. In the context of k-nearest neighbor queries, this measure was found to be a good predictor of the difficulty of answering a query [2, 3]. Intuitively, it shows how much the distance between the motif subsequences differs from the other distances in the dataset. Smaller values correspond to a smaller difference: in such cases the LSH approach will have a harder time identifying the motif pair. In Table 2 the columns RC_1 and RC_{10} report the relative contrast of the first and 10-th motif pairs, respectively. Given this measure, we can expect that finding the 10-th top motif in `freezer` will be much harder than in `HumanY`, and that finding the top motif in `freezer` will be comparatively easier than finding the 10-th. Furthermore, `Seismic` features a top motif that truly stands out, given its very high relative contrast.

Default parameter values. Unless otherwise stated, we set the failure probability parameter $\delta = 0.01$ under the stricter stopping condition of Lemma 4.5. The maximum hash length is $K = 32$ in all experiments, and the default number of repetitions is $L = 200$, except for `Whales` and `Seismic` for which we set $L = 100$ and $L = 16$, due to their size. The parameter r, instead, is automatically estimated by the software.

7.1 Finding the top motif

In the first set of experiments, we report on the time to return the top-1 motif, as shown in Table 3. Numbers in parentheses report estimated performance metrics for algorithms that scale predictably with the input size.

We note that, on all datasets, ATTIMO is consistently faster than SCAMP on the CPU (up to two orders of magnitude). When run on the GPU, SCAMP sees improved running times, beating ATTIMO on the smaller datasets. In particular, on the billion-scale `Seismic` dataset ATTIMO finds the top motif in around 12 minutes, compared to one month (estimated) to compute the full matrix profile using SCAMP. On this same dataset, the original SCAMP paper [48] reports a running time of one day using a cluster of 5 GPUs, making our algorithm very competitive even in a scenario where it uses much less resources. The better performance of ATTIMO on large datasets is mainly due to the reduced number of distance computations it performs: it is just a very small fraction of the total number of distances between subsequences. For instance, on `Whales` ATTIMO computes around 746 thousands distances, compared to $4.77 \cdot 10^{16}$ computed by SCAMP, on `Seismic` just 26 thousands instead of $5.02 \cdot 10^{17}$. This also highlights the *data adaptive* nature of ATTIMO discussed in Observation 2: ATTIMO runs faster on `Seismic` than on `Whales`, despite the former being three times larger. The reason is that `Seismic` is comparatively easier, since it has a higher relative contrast. Thus, ATTIMO is able to exploit this feature of the data to compute fewer distances. The speedup of ATTIMO compared to SCAMP is not directly proportional to the reduction in the number of distances computed because SCAMP is *very* efficient at computing distances using the GPU, and ATTIMO requires to build the LSH index, so it performs more work than just computing distances. Still, filtering out unneeded computations by means of LSH proves beneficial, and these results suggest that, if one is interested only in the top motif(s), then our approach provides an efficient solution.

[3] For this algorithm, we test all combinations of the following parameters: size of the SAX alphabet $\in [3, 6]$, number of characters for the projections $\in [3, 6]$; the PAA window length is fixed to 1/10 of the motif length, and the number of repetitions to 10.

Table 3: Time and memory to find the top motif. Numbers in parentheses are estimates.

	Time (s)						Memory (Gb)					
dataset	Attimo	Scamp-gpu	Scamp	PreSCRIMP	LL	RProj	Attimo	Scamp-gpu	Scamp	PreSCRIMP	LL	RProj
ASTRO	19.1	9.4	87.1	88.6	6.2	650.9	0.7	5.8	3.3	3.0	0.4	69.1
GAP	50.3	20.4	264.4	47.6	85.8	649.4	1.1	5.9	3.8	3.7	0.7	89.7
freezer	57.9	180.9	3 444.1	87.6	1 313.9	2 430.8	4.0	6.4	6.8	7.8	2.8	157.4
ECG	64.3	196.6	3 733.0	411.1	2 124.6	-	3.9	6.5	7.0	8.1	23.0	-
HumanY	131.4	2 201.6	(11.6 h)	483.2	-	-	14.4	8.5	(17.6)	22.9	-	-
Whales	1 424.5	(2.9 days)	(70.4 days)	(51.4 days)	-	-	103.7	(38.1)	(177.7)	(237.1)	-	-
Seismic	729.5	(32.1 days)	(2.1 years)	(2.1 years)	-	-	129.1	(110.6)	(569.5)	(762.7)	-	-

PreSCRIMP is slower than Attimo in all cases. We remark that its running time depends on the window size, which explains why it has very different running times on datasets of comparable size, like `freezer` and `ECG`.

As for LL, we note that the running times vary a lot in comparison with both Attimo and Scamp: in some cases LL is comparable with Attimo, in some others it is slower than Scamp despite providing less information. For datasets larger than 8 million points, LL times out after two hours. We are unable to provide runtime estimates, since the implementation does not seem to scale regularly with the input size.

RProj is also unable to scale to larger time series, mainly due to the cost of storing the large sparse collision matrix. Table 3 reports the result for the configuration of RProj finding the best motif.

Considering the quality of the returned solutions, we have that Scamp is an exact algorithm for the top-1 motif, and PreSCRIMP is an approximate algorithm with no theoretical approximation guarantee. In practice in this set of experiments it always found the correct top motif. The random projection algorithm RProj, in the best configuration for each dataset could not find the correct motif, returning pairs of points with a distance between 2 and 9 times larger the one of the true top motif. This can be improved by increasing the number of repetitions at the expense of proportionally larger running times and increased memory usage.

Recall that we set Attimo with a failure probability of $\delta = 0.01$, implying that with probability 0.99 the returned pair is correct. In practice our algorithm always returned the exact top motif.

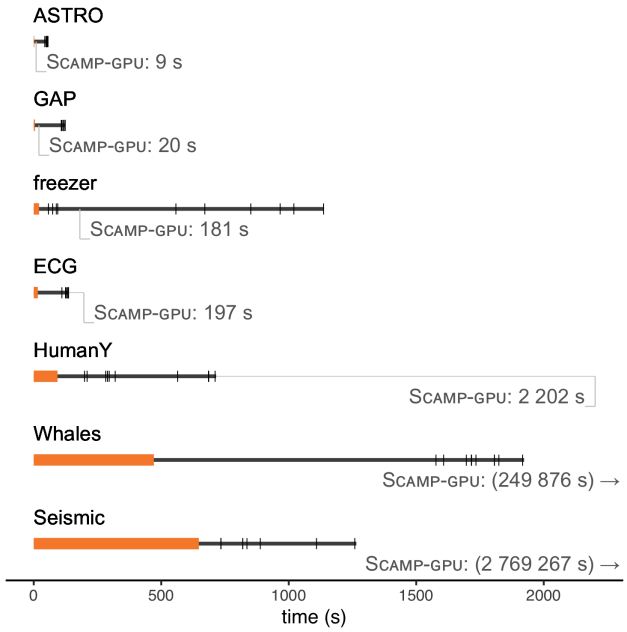

Figure 5: Time that Attimo employs to find the top-10 motifs in each time series.

7.2 Finding more than one motif

Finding the top motif might provide too little information to the data analyst, and the RC_{10} values reported in Table 2 suggest that in some cases finding the top 10 motifs can be considerably harder. Figure 5 reports the time required by Attimo to report the top 10 motifs using 400 repetitions (100 repetitions for `Whales` and 16 for `Seismic`). The datasets are arranged, from top to bottom, by increasing size. The orange bar marks the index construction time and the black bar the time to find the motifs for Attimo. Each tick on the black bar marks the time when one or more motifs are found; if less than 10 ticks are visible it means that multiple motifs are discovered at the same time. The thin gray line reports the runtime for Scamp running on the GPU. For some datasets such line is omitted for reasons of scale, and we only report the time.

First, note that motifs are not discovered at regular intervals. Rather, motifs at larger distances might require considering shorter hash prefixes to be confirmed, hence they require more time to be reported since more distance computations are needed. This is explained by the fact that lower rank motifs are much closer to the bulk of the distribution of distances of subsequences pairs, and therefore are more difficult for LSH to separate from other pairs.

As for the running times, we note that Scamp running on the GPU is faster than Attimo on the two smallest datasets we consider, i.e. `ASTRO` and `GAP`. On the most difficult dataset under consideration (in terms of RC_{10}), `freezer`, Attimo finds the first 4 motifs faster than Scamp. Motifs from the 5-th onwards are instead found first by Scamp. This is an instance of our earlier observation: such motifs are closer to the majority of the other subsequences, and hence more difficult to separate. At the same time, this might also mean that such motifs are not very significant: notice that the 10-th motif pair is 3 times farther away than the top motif. Since Attimo discovers motifs by increasing distance, the user has the chance of stopping the computation early if such situation occurs.

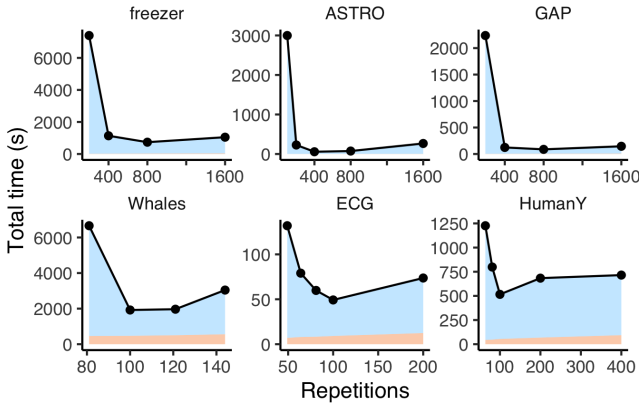

Figure 6: Running time for different numbers of repetitions.

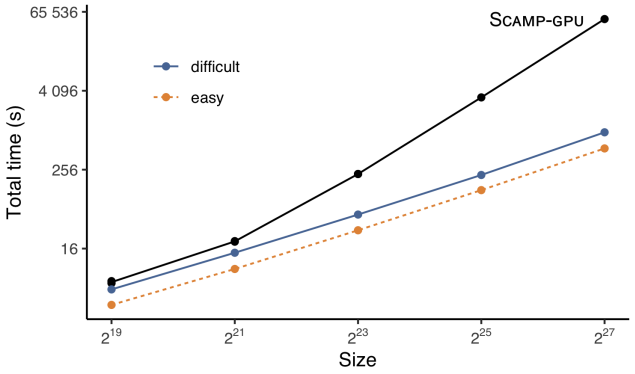

Figure 7: Scalability vs. input size (log scale both axes).

We remark that in accordance with Lemma 4.5 the algorithm stops when the failure probability of all pairs to be returned is $\leq \delta/k = 0.001$ (which explains the slightly different running times between Table 3 and Figure 5 for finding the top motif). This implies that we find *all* top-10 pairs with probability at least 0.99. However, the analysis is fairly conservative. As a result, in practice our algorithm always returned the exact top motifs.

On the other hand, as we discussed in Observation 1, in some cases SCAMP might not identify the true top-k motifs. In our experiments this happens on the `freezer` dataset, where the 8-th motif found by SCAMP is formed by subsequences 5 169 982 and 6 429 402, at distance 11.4624. By contrast, the 8-th motif found by ATTIMO is between subsequences 3 815 625 and 5 170 040, at distance 11.3378. The nearest neighbors of both these sequences overlap with sequences participating in higher-ranked motifs, thus preventing SCAMP to consider them. Nonetheless, these two subsequences are at distance smaller than the 8-th motif found by SCAMP, and should be part of the top-10 motifs.

7.3 Influence of the number of repetitions on the running time

In this set of experiments we investigate the influence of L on the running time for finding the top-10 motifs. The tradeoff in play here is the following. Allowing for a larger number of repetitions allows to stop the exploration at longer hash prefixes, where LSH is more selective and therefore able to prune away more distance computations. The price to pay is a longer preprocessing time, a higher memory usage, and more repetitions to be explored. However, remember that thanks to our use of the tensoring technique the preprocessing time is proportional to $\sqrt{L}$ rather than L. We exclude `Seismic` from this evaluation, since due to its size it fills the memory of our system with $L = 16$ repetitions.

Figure 6 reports our findings. The datasets can be divided in two groups. Those with high RC_{10} values (bottom three in the plot) perform best with few repetitions (≈ 100), whereas those with low RC_{10} values (top three in the plot) see their best performance with more repetitions, around 400 or 800. In other words, datasets in which the motifs are less distinguished from the average subsequence pair benefit from investing more repetitions.

7.4 Scalability

In Figure 7 we report on a scalability experiment on the `Seismic` dataset. Given that ATTIMO's running time depends on the relative contrast, and that different prefixes of the same time series might have different relative contrasts, we build synthetic datasets to test the scalability of the algorithms. In particular, we consider random walks of increasing size where we inject as a motif two instances of a sine wave of length 100 with added random Gaussian noise, using the variance to control the relative contrast of the motif. Using this construction, we set up an easy and a difficult dataset with relative contrasts 100 and 20, respectively. As expected, the running time of SCAMP depends only on the input size and scales quadratically. ATTIMO's running time, on the other hand is essentially linear for a fixed relative contrast. For a fixed input size, the running time increases as the relative contrast decreases. This is expected since more distance computations need to be carried out.

Therefore our algorithm exhibits a much better scalability with respect to the size of the input, allowing it to tackle large time series using considerably fewer computational resources.

8 CONCLUSIONS

In contexts where objects with extremal properties are sought, such as motif discovery, being adaptive to the data distribution allows to weed out many unnecessary computations. Our LSH based algorithm is able to adapt to the data distribution so to compute only a small fraction of all possible distances, thus enabling large speedups on large time series compared to the state of the art.

Our approach guarantees that the output is correct with a user-defined probability: given that our analysis is conservative, in practice our implementation always found the exact top-k motifs.

ACKNOWLEDGMENTS

Partially funded by the European Fund for Regional Development (EFRE 2014-2020) and the Autonomous Province of Bozen-Bolzano (EFRE1164).

REFERENCES

[1] Alexandr Andoni and Piotr Indyk. 2006. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *FOCS*. IEEE Computer Society, 459–468.

[2] Martin Aumüller and Matteo Ceccarello. 2019. The Role of Local Intrinsic Dimensionality in Benchmarking Nearest Neighbor Search. In *SISAP (Lecture Notes in Computer Science, Vol. 11807)*. Springer, 113–127.

[3] Martin Aumüller and Matteo Ceccarello. 2021. The role of local dimensionality measures in benchmarking nearest neighbor search. *Inf. Syst.* 101 (2021), 101807.

[4] Martin Aumüller and Matteo Ceccarello. 2022. Implementing Distributed Similarity Joins using Locality Sensitive Hashing. In *EDBT*. OpenProceedings.org, 1–13.

[5] Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Michael Vesterli. 2019. PUFFINN: Parameterless and Universally Fast FInding of Nearest Neighbors. In *ESA (LIPIcs, Vol. 144)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:16.

[6] William H Bakun and Allan G Lindh. 1985. The Parkfield, California, earthquake prediction experiment. *Science* 229, 4714 (1985), 619–624.

[7] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. 2005. LSH forest: self-tuning indexes for similarity search. In *WWW*. ACM, 651–660.

[8] Paul Boniol, Michele Linardi, Federico Roncallo, Themis Palpanas, Mohammed Meftah, and Emmanuel Remy. 2021. Unsupervised and scalable subsequence anomaly detection in large data series. *VLDB J.* 30, 6 (2021), 909–931.

[9] Andrei Z. Broder. 1997. On the resemblance and containment of documents. In *SEQUENCES*. IEEE, 21–29.

[10] André EX Brown, Eviatar I Yemini, Laura J Grundy, Tadas Jucikas, and William R Schafer. 2013. A dictionary of behavioral motifs reveals clusters of genes affecting Caenorhabditis elegans locomotion. *Proceedings of the National Academy of Sciences* 110, 2 (2013), 791–796.

[11] Jeremy Buhler and Martin Tompa. 2001. Finding motifs using random projections. In *Proceedings of the fifth annual international conference on Computational biology*. 69–76.

[12] Carmelo Cassisi, Marco Aliotta, Andrea Cannata, Placido Montalto, Domenico Patanè, Alfredo Pulvirenti, and Letizia Spampinato. 2013. Motif discovery on seismic amplitude time series: The case study of mt etna 2011 eruptive activity. *Pure and Applied Geophysics* 170, 4 (2013), 529–545.

[13] Matteo Ceccarello, Anne Driemel, and Francesco Silvestri. 2019. FRESH: Fréchet Similarity with Hashing. In *WADS (Lecture Notes in Computer Science, Vol. 11646)*. Springer, 254–268.

[14] Moses Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *STOC*. ACM, 380–388.

[15] Bill Yuan-chi Chiu, Eamonn J. Keogh, and Stefano Lonardi. 2003. Probabilistic discovery of time series motifs. In *KDD*. ACM, 493–498.

[16] Tobias Christiani. 2019. Fast Locality-Sensitive Hashing Frameworks for Approximate Near Neighbor Search. In *SISAP (Lecture Notes in Computer Science, Vol. 11807)*. Springer, 3–17.

[17] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG*. ACM, 253–262.

[18] Yifeng Gao and Jessica Lin. 2019. Discovering Subdimensional Motifs of Different Lengths in Large-Scale Multivariate Time Series. In *ICDM*. IEEE, 220–229.

[19] Yifeng Gao and Jessica Lin. 2019. HIME: discovering variable-length motifs in large-scale time series. *Knowl. Inf. Syst.* 61, 1 (2019), 513–542.

[20] Junfeng He, Sanjiv Kumar, and Shih-Fu Chang. 2012. On the Difficulty of Nearest Neighbor Search. In *ICML*. icml.cc / Omnipress.

[21] J.A. Healey and R.W. Picard. 2005. Detecting stress during real-world driving tasks using physiological sensors. *IEEE Transactions on Intelligent Transportation Systems* 6, 2 (2005), 156–166. https://doi.org/10.1109/TITS.2005.848368

[22] Hebrail, Georges and Berard, Alice. 2022. Individual household electric power consumption Data Set. Accessed: 2022-05-30. https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption. https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption

[23] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. ACM, 604–613.

[24] Hoang Thanh Lam, Toon Calders, and Ninh Pham. 2011. Online Discovery of Top-k Similar Motifs in Time Series Data. In *SDM*. SIAM / Omnipress, 1004–1015.

[25] Xiaosheng Li and Jessica Lin. 2019. Linear Time Motif Discovery in Time Series. In *SDM*. SIAM, 136–144.

[26] Jessica Lin, Eamonn J. Keogh, Li Wei, and Stefano Lonardi. 2007. Experiencing SAX: a novel symbolic representation of time series. *Data Min. Knowl. Discov.* 15, 2 (2007), 107–144.

[27] Michele Linardi, Yan Zhu, Themis Palpanas, and Eamonn J. Keogh. 2018. VALMOD: A Suite for Easy and Exact Detection of Variable Length Motifs in Data Series. In *SIGMOD Conference*. ACM, 1757–1760.

[28] Amy McGovern, Derek H. Rosendahl, Rodger A. Brown, and Kelvin Droegemeier. 2011. Identifying predictive multi-dimensional time series motifs: an application to severe weather prediction. *Data Min. Knowl. Discov.* 22, 1-2 (2011), 232–258.

[29] Abdullah Mueen. 2014. Time series motif discovery: dimensions and applications. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* 4, 2 (2014), 152–159. https://doi.org/10.1002/widm.1119

[30] Abdullah Mueen and Nikan Chavoshi. 2015. Enumeration of time series motifs of all lengths. *Knowl. Inf. Syst.* 45, 1 (2015), 105–132.

[31] Abdullah Mueen and Eamonn J. Keogh. 2010. Online discovery and maintenance of time series motifs. In *KDD*. ACM, 1089–1098.

[32] Abdullah Mueen, Eamonn J. Keogh, Qiang Zhu, Sydney Cash, and M. Brandon Westover. 2009. Exact Discovery of Time Series Motifs. In *SDM*. SIAM, 473–484.

[33] Abdullah Mueen, Suman Nath, and Jie Liu. 2010. Fast approximate correlation for massive time-series data. In *SIGMOD Conference*. ACM, 171–182.

[34] Abdullah Mueen, Yan Zhu, Michael Yeh, Kaveh Kamgar, Krishnamurthy Viswanathan, Chetan Gupta, and Eamonn Keogh. 2017. The Fastest Similarity Search Algorithm for Time Series Subsequences under Euclidean Distance. http://www.cs.unm.edu/ mueen/FastestSimilaritySearch.html.

[35] David Murray, Jing Liao, Lina Stankovic, Vladimir Stankovic, Richard Hauxwell-Baldwin, Charlie Wilson, Michael Coleman, Tom Kane, and Steven Firth. 2015. A data management platform for personalised real-time energy feedback. In *Proceedings of the 8-th international conference on energy efficiency in domestic appliances and lighting*. IET, 1–15.

[36] Ankur Narang and Souvik Bhattacherjee. 2010. Parallel Exact Time Series Motif Discovery. In *Euro-Par (2) (Lecture Notes in Computer Science, Vol. 6272)*. Springer, 304–315.

[37] NOAA National Centers for Environmental Information. 2022. NOAA Pacific Islands Fisheries Science Center. 2021. Pacific Islands Passive Acoustic Network (PIPAN) 10kHz Data. Accessed: 2022-05-30. https://doi.org/10.25921/Z787-9Y54. https://doi.org/10.25921/Z787-9Y54

[38] Pranav Patel, Eamonn J. Keogh, Jessica Lin, and Stefano Lonardi. 2002. Mining Motifs in Massive Time Series Databases. In *ICDM*. IEEE Computer Society, 370–377.

[39] Davood Rafiei. 1999. On Similarity-Based Queries for Time Series Data. In *ICDE*. IEEE Computer Society, 410–417.

[40] Thanawin Rakthanmanon, Bilson J. L. Campana, Abdullah Mueen, Gustavo E. A. P. A. Batista, M. Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn J. Keogh. 2012. Searching and mining trillions of time series subsequences under dynamic time warping. In *KDD*. ACM, 262–270.

[41] Kexin Rong, Clara E. Yoon, Karianne J. Bergen, Hashem Elezabi, Peter Bailis, Philip Alexander Levis, and Gregory C. Beroza. 2018. Locality-Sensitive Hashing for Earthquake Detection: A Case Study Scaling Data-Driven Science. *Proc. VLDB Endow.* 11, 11 (2018), 1674–1687. https://doi.org/10.14778/3236187.3236214

[42] S Soldi, Volker Beckmann, WH Baumgartner, Gabriele Ponti, Chris R Shrader, P Lubiński, HA Krimm, F Mattana, and Jack Tueller. 2014. Long-term variability of agn at hard x-rays. *Astronomy & Astrophysics* 563 (2014), A57.

[43] UC Berkeley Seismological Laboratory. Dataset. 2014. High Resolution Seismic Network. https://doi.org/doi:10.7932/HRSN

[44] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. 2014. Hashing for Similarity Search: A Survey. *CoRR* abs/1408.2927 (2014). arXiv:1408.2927 http://arxiv.org/abs/1408.2927

[45] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn J. Keogh. 2016. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets. In *ICDM*. IEEE Computer Society, 1317–1322.

[46] Yan Zhu, Chin-Chia Michael Yeh, Zachary Zimmerman, Kaveh Kamgar, and Eamonn J. Keogh. 2018. Matrix Profile XI: SCRIMP++: Time Series Motif Discovery at Interactive Speeds. In *IEEE International Conference on Data Mining, ICDM 2018, Singapore, November 17-20, 2018*. IEEE Computer Society, 837–846. https://doi.org/10.1109/ICDM.2018.00099

[47] Yan Zhu, Zachary Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth J. Funning, Abdullah Mueen, Philip Brisk, and Eamonn J. Keogh. 2016. Matrix Profile II: Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins. In *ICDM*. IEEE Computer Society, 739–748.

[48] Zachary Zimmerman, Kaveh Kamgar, Nader Shakibay Senobari, Brian Crites, Gareth J. Funning, Philip Brisk, and Eamonn J. Keogh. 2019. Matrix Profile XIV: Scaling Time Series Motif Discovery with GPUs to Break a Quintillion Pairwise Comparisons a Day and Beyond. In *SoCC*. ACM, 74–86.