Scalable, Variable-Length Similarity Search in Data Series: The ULISSE Approach

Michele Linardi
LIPADE, Paris Descartes University
michele.linardi@parisdescartes.fr

Themis Palpanas
LIPADE, Paris Descartes University
themis@mi.parisdescartes.fr

ABSTRACT

Data series similarity search is an important operation and at the core of several analysis tasks and applications related to data series collections. Despite the fact that data series indexes enable fast similarity search, all existing indexes can only answer queries of a single length (fixed at index construction time), which is a severe limitation. In this work, we propose *ULISSE*, the first data series index structure designed for answering similarity search queries of *variable length*. Our contribution is two-fold. First, we introduce a novel representation technique, which effectively and succinctly summarizes multiple sequences of different length (irrespective of Z-normalization). Based on the proposed index, we describe efficient algorithms for approximate and exact similarity search, combining disk based index visits and in-memory sequential scans. We experimentally evaluate our approach using several synthetic and real datasets. The results show that *ULISSE* is several times (and up to orders of magnitude) more efficient in terms of both space and time cost, when compared to competing approaches.

PVLDB Reference Format:
Michele Linardi, Themis Palpanas. Scalable, Variable-Length Similarity Search in Data Series: The ULISSE Approach. *PVLDB*, 11 (13): 2236-2248, 2018.
DOI: https://doi.org/10.14778/3275366.3275372

1. INTRODUCTION

Motivation. Data sequences are one of the most common data types, and they are present in almost every scientific and social domain (example application domains include meteorology, astronomy, chemistry, medicine, neuroscience, finance, agriculture, entomology, sociology, smart cities, marketing, operation health monitoring, human action recognition and others) [1, 2, 3, 4, 5]. This makes data series a data type of particular importance.

Informally, a data series (a.k.a data sequence, or time series) is defined as an ordered sequence of points, each one associated with a position and a corresponding value[1]. Recent advances in sensing, networking, data processing and storage technologies have significantly facilitated the processes of generating and collecting tremendous amounts of data sequences from a wide variety of domains at extremely high rates and volumes.

The *SENTINEL-2* mission [6] conducted by the European Space Agency (ESA) represents such an example of massive data series collection. The two satellites of this mission continuously capture multi-spectral images, designed to give a full picture of earth's surface every five days at a resolution of *10m*, resulting in over five trillion different data series. Such recordings will help monitor at fine granularity the evolution of the properties of the surface of the earth, and benefit applications such as land management, agriculture and forestry, disaster control, humanitarian relief operations, risk mapping and security concerns.

Data series analytics. Once the data series have been collected, the domain experts face the arduous tasks of processing and analyzing them [7] in order to identify patterns, gain insights, detect abnormalities, and extract useful knowledge. Critical part of this process is the data series similarity search operation, which lies at the core of several analysis and machine learning algorithms (e.g., clustering [8], classification [9], outliers [10], and others).

However, similarity search in very large data series collections is notoriously challenging [11, 12, 13, 14, 14], due to the high dimensionality (length) of the data series. In order to address this problem, a significant amount of effort has been dedicated by the data management research community to data series indexing techniques, which lead to fast and scalable similarity search [15, 16, 17, 18, 19, 20, 11, 21, 22, 23, 24, 25, 26].

Predefined constraints. Despite the effectiveness and benefits of the proposed indexing techniques, which have enabled and powered many applications over the years, they are restricted in different ways: either they only support similarity search with queries of a fixed size, or they do not offer a scalable solution. The solutions working for a fixed length, require that this length is chosen at index construction time (it should be the same as the length of the series in the index).

Evidently, this is a constraint that penalizes the flexibility needed by analysts, who often times need to analyze patterns of slightly different lengths (within a given data series collection) [20, 27, 28, 29, 30]. This is true for several applications. For example, in the *SENTINEL-2* mission data, oceanographers are interested in searching for similar coral bleaching patterns[2] of different lengths; at Airbus[3] engineers need to perform similarity search queries for patterns of variable length when studying aircraft takeoffs and land-

[1]If the dimension that imposes the ordering of the sequence is time then we talk about time series. Though, a series can also be defined over other measures (e.g., angle in radial profiles in astronomy, mass in mass spectroscopy in physics, etc.). We use the terms *data series*, *time series*, and *sequence* interchangeably.

[2]http://www.esa.int/Our_Activities/Observing_the_Earth/

[3]http://www.airbus.com/

Proceedings of the VLDB Endowment, Vol. 11, No. 13
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3275366.3275372

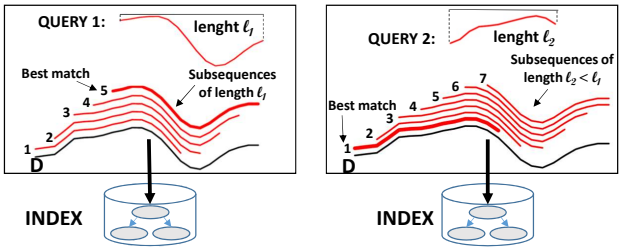


Figure 1: Indexing for supporting queries of 2 different lengths.

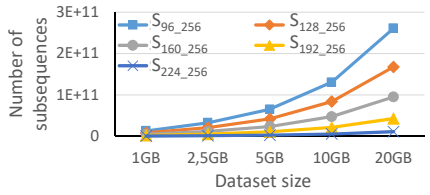


Figure 2: Search space evolution of variable length similarity search. Each dataset contains series of length 256

ings [31]; and in neuroscience, analysts need to search in Electroencephalogram (EEG) recordings for Cyclic Alternating Patterns (CAP) of different lengths (duration), in order to get insights about brain activity during sleep [32]. In these applications, we have datasets with a very large number of fixed length data series, on which analysts need to perform a large number of ad hoc similarity queries of (slightly) different lengths (as shown in Figure 1).

A straightforward solution for answering such queries would be to use one of the available indexing techniques. However, in order to support (exact) results for variable-length similarity search, we would need to (i) create several distinct indexes, one for each possible query length; and (ii) for each one of these indexes, index all overlapping subsequences (using a sliding window). We illustrate this in Figure 1, where we depict two similarity search queries of different lengths (ℓ and ℓ'). Given a data series from the collection, D_i (shown in black), we draw in red the subsequences that we need to compare to each query in order to compute the exact answer. Using an indexing technique implies inserting all the subsequences in the index: since we want to answer queries of two different lengths, we are obliged to use two distinct indexes.

Nevertheless, this solution is prohibitively expensive, in both space and time. Space complexity is increased, since we need to index a large number of subsequences for each one of the supported query lengths: given a data series collection $C = D^1, ..., D^{|C|}$ and a query length range $[\ell_{min}, \ell_{max}]$, the number of subsequences we would normally have to examine (and index) is:

$S_{\ell_{min},\ell_{max}} = \sum_{\ell=1}^{(\ell_{max}-\ell_{min})+1} \sum_{i=1}^{|C|} (|D^i| - (\ell - 1))$. Figure 2 shows how quickly this number explodes as the dataset size and the query length range increase: considering the largest query length range (S_{96-256}) in the 20GB dataset, we end up with a collection of subsequences (that need to be indexed) 5 orders of magnitude larger than the original dataset! Computational time is significantly increased as well, since we have to construct different indexes for each query length we wish to support.

In the current literature, a technique based on multi-resolution indexes [27, 20] has been proposed in order to mitigate this explosion in size, by creating a smaller number of distinct indexes and performing more post-processing. Nonetheless, this solution works exclusively for *non* Z-normalized series[4] (which means that it cannot return results with similar trends, but different absolute values), and thus, renders the solution useless for a wide spectrum of applications. Besides, it only mitigates the problem, since it still leads to a space explosion (albeit, at a lower rate), and therefore, it is not scalable, either.

We note that the technique discussed above (despite its limitations) is indeed the current state of the art, and no other technique has been proposed since, even though during the same period of time we have witnessed lots of activity and a steady stream of papers on the *single-length* similarity search problem (e.g., [17, 18, 19, 34, 11, 23, 24, 25, 26]). This attests to the challenging nature of the problem we are tackling in this paper.

Contributions. In this work, we propose *ULISSE* (ULtra compact Index for variable-length Similarity SEarch in data series), which is the first single-index solution that supports fast answering of variable-length similarity search queries for both non Z-normalized and Z-normalized data series collections. *ULISSE* produces exact (i.e., correct) results, and is based on the following key idea: a data structure that indexes data series of length ℓ, already contains all the information necessary for reasoning about any subsequence of length $\ell' < \ell$ of these series. Therefore, the problem of enabling a data series index to answer queries of variable-length, becomes a problem of how to reorganize this information that already exists in the index. To this effect, *ULISSE* proposes a new summarization technique that is able to represent contiguous and overlapping subsequences, leading to succinct, yet powerful summaries: it combines the representation of several subsequences within a single summary, and enables fast (approximate and exact) similarity search for variable-length queries.

Our contributions can be summarized as follows: (I) We introduce the problem of Variable-Length Subsequences Indexing, which calls for a single index that can inherently answer queries of different lengths. (II) We provide a new data series summarization technique, able to represent several contiguous series of different lengths. This technique produces succinct, discretized envelopes for the summarized series, and can be applied to both non Z-normalized and Z-normalized data series. (III) Based on this summarization technique, we develop an indexing algorithm, which organizes the series and their discretized summaries in a hierarchical tree structure, namely, the *ULISSE* index. (IV) We propose efficient exact and approximate K-NN algorithms, suitable for the *ULISSE* index. (V) Finally, we perform an experimental evaluation with several synthetic and real datasets. The results demonstrate the effectiveness and scalability of *ULISSE* to dataset sizes that competing approaches cannot handle.

Paper Organization. The rest of this paper[5] is organized as follows. Section 2 discusses related work, and Section 3 formulates the problem. In Section 4, we describe the *ULISSE* summarization techniques, and in Sections 5 and 6 we explain our indexing and query answering algorithms. Section 7 describes the experimental evaluation, and we conclude in Section 8.

2. RELATED WORK

Data series indexes. The literature includes several techniques for data series indexing [15, 16, 19, 36, 18, 11], which are all based on the same principle: they first reduce the dimensionality of the data

[4]Z-normalization transforms a series so that it has a mean value of zero, and a standard deviation of one. This allows similarity search to be effective, irrespective of shifting (i.e., offset translation) and scaling[33].

[5]A high level (4-page poster paper) discussion of the *ULISSE* general idea has appeared elsewhere [35].

series by applying some summarization technique (e.g., Piecewise Aggregate Approximation (PAA) [37], or Symbolic Aggregate approXimation (SAX) [19]. However, all the approaches mentioned above share a common limitation: they can only answer queries of a fixed, predetermined length, which has to be decided before the index creation.

Indexing for variable length query. Faloutsos et al. [15] proposed the first indexing technique suitable for variable length similarity search query. This technique extracts subsequences that are grouped in MBRs (Minimum Bounding Rectangles) and indexed using an R-tree. We note that this approach works only for non Z-normalized sequences. An improvement of this approach was proposed by Kahveci and Singh [27]. They described MRI (Multi Resolution Index), which is a technique based on the construction of multiple indexes for variable length similarity search query. Storing subsequences at different resolutions (building indexes for different series lengths) provided a significant improvement over the earlier approach, since a greater part of a single query is considered during the search. Subsequently, Kadiyala and Shiri [20] redesigned the MRI construction, in order to decrease the indexing size and construction time. This new indexing technique, called Compact Multi Resolution Index (CMRI), has a space requirement, which is 99% smaller than the one of MRI. The authors also redefined the search algorithm, guaranteeing an improvement of the range search proposed upon the MRI index. In contrast to CMRI, our approach uses a single index that is able to answer similarity search queries of variable length over larger datasets, and works for both non Z-normalized and Z-normalized series (a feature that is not supported by any of the previously introduced indexing techniques).

Sequential scan techniques. Even though recent works have shown that sequential scans can be performed efficiently [28, 38], such techniques are mostly applicable when the dataset consists of a single, very long data series, and queries are looking for potential matches in small subsequences of this long data series. Such approaches, in general, do not provide any benefit when the dataset is composed of a large number of small data series, like in our case. Therefore, indexing is required in order to efficiently support data exploration tasks, which involve ad-hoc queries, i.e., the query workload is not known in advance.

3. PROBLEM FORMULATION AND PRELIMINARIES

Let a data series $D = d_1,...,d_{|D|}$ be a sequence of numbers $d_i \in \mathbb{R}$, where $i \in \mathbb{N}$ represents the position in D. We denote the length, or size of the data series D with $|D|$. The subsequence $D_{o,\ell}$=d_o,...,$d_{o+\ell-1}$ of length ℓ, is a contiguous subset of ℓ points of D starting at offset o, where $1 \leq o \leq |D|$ and $1 \leq \ell \leq |D|-o+1$. A subsequence is itself a data series. A data series collection, C, is a set of data series.

We say that a data series D is Z-normalized, denoted D^n, when its mean μ is 0 and its standard deviation σ is 1. The normalized version of $D = d_1, ..., d_{|D|}$ is computed as follows: $D^n = \{\frac{d_1-\mu}{\sigma}, ..., \frac{d_{|D|}-\mu}{\sigma}\}$. Z-normalization is an essential operation in several applications, because it allows similarity search irrespective of shifting and scaling [33, 28].

Given two data series $D = d_1, ..., d_{|D|}$ and $D' = d'_1, ..., d'_{|D'|}$ of the same length (i.e., $|D| = |D'|$), we can calculate their Euclidean Distance as follows: $ED(D, D') = \sqrt{\sum_1^{|D|}(d_1 - d'_1)^2}$.

The problem we wish to solve in this paper is the following.

PROBLEM 1 (VARIABLE-LENGTH SUBSEQUENCES INDEXING). *Given a data series collection C, and a series length range $[\ell_{min}, \ell_{max}]$, we want to build an index that supports exact similarity search for queries of any length within the range $[\ell_{min}, \ell_{max}]$.*

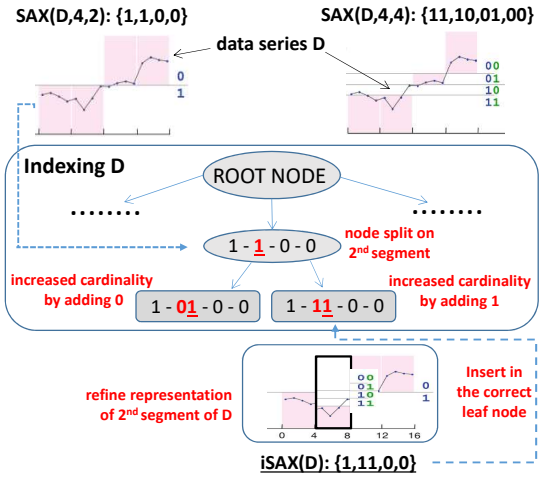


Figure 3: Indexing of series D (and an inner node split).

In our case similarity search is formally defined as follows:

DEFINITION 1 (SIMILARITY SEARCH). *Given a data series collection $C = \{D^1, ..., D^C\}$, a series length range $[\ell_{min}, \ell_{max}]$, a query data series Q, where $\ell_{min} \leq |Q| \leq \ell_{max}$, and $k \in \mathbb{N}$, we want to find the set $R = \{D^i_{o,\ell} | D^i \in C \wedge \ell = |Q| \wedge (\ell + o - 1) \leq |D^i|\}$, where $|R| = k$. We require that $\forall D^i_{o,\ell} \in R$ $\nexists D^{i'}_{o',\ell'}$ s.t. $ED(D^{i'}_{o',\ell'}, Q) < ED(D^i_{o,\ell}, Q)$, where $\ell' = |Q|$, $(\ell' + o' - 1) \leq |D^{i'}|$ and $D^{i'} \in C$. We informally call R, the k nearest neighbors set of Q.*

In this study, we use Euclidean Distance as the measure for conducting similarity search, which is a widely used and accepted measure [15, 16, 17, 18, 19, 20, 11, 21, 23, 24]. Though, our approach could be extended to work with other distance measures as well (e.g., Dynamic Time Warping, through the use of the corresponding envelopes [39]).

3.1 The iSAX Index

The Piecewise Aggregate Approximation (PAA) of a data series D, $PAA(D) = \{p_1, ..., p_w\}$, represents D in a w-dimensional space by means of w real-valued segments of length s, where the value of each segment is the mean of the corresponding values of D [37]. We denote the first k dimensions of $PAA(D)$, $(k \leq w)$, as $PAA(D)_{1,..,k}$. Then, the $iSAX$ representation of a data series D, denoted by $SAX(D, w, |alphabet|)$, is the representation of $PAA(D)$ by w discrete coefficients, drawn from an alphabet of cardinality $|alphabet|$ [19].

The main idea of the $iSAX$ representation (see Figure 3, top), is that the real-values space may be segmented by $|alphabet| - 1$ breakpoints in $|alphabet|$ regions that are labeled by distinct symbols: binary values (e.g., with $|alphabet| = 4$ the available labels are $\{00, 01, 10, 11\}$). $iSAX$ assigns symbols to the PAA coefficients, depending in which region they are located.

The iSAX data series index is a tree data structure [19, 21], consisting of three types of nodes (refer to Figure 3). (i) The root node points to n children nodes (in the worst case $n = 2^w$, when the series in the collection cover all possible iSAX representations). (ii)

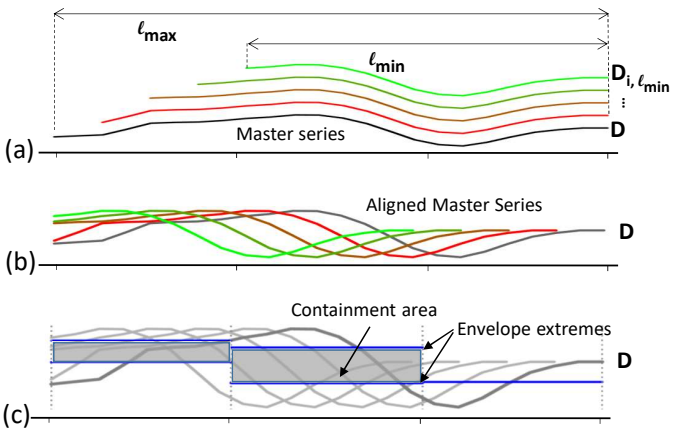


Figure 4: ***a)*** **master series of** D **in the length interval** ℓ_{min}, ℓ_{max}**.** ***b)*** **Zero-aligned master series.** ***c)*** **Envelope built over the master series.**

Each inner node contains the iSAX representation of all the series below it. (iii) Each leaf node contains both the iSAX representation *and* the raw data of all the series inside it (in order to be able to prune false positives and produce exact, correct answers). When the number of series in a leaf node becomes greater than the maximum leaf capacity, the leaf splits: it becomes an inner node and creates two new leaves, by increasing the cardinality of one of the segments of its iSAX representation. The two refined iSAX representations (new bit set to *0* and *1*) are assigned to the two new leaves.

4. THE ULISSE FRAMEWORK

The key idea of the *ULISSE* approach is the succinct summarization of *sets* of series, namely, overlapping subsequences. In this section, we present this summarization method.

4.1 Representing Multiple Subsequences

When we consider, contiguous and overlapping subsequences of different lengths within the range $[\ell_{min}, \ell_{max}]$ (Figure 4.a), we expect the outcome as a bunch of similar series, whose differences are affected by the misalignment and the different number of points. We conduct a simple experiment in Figure 4.b, where we zero-align all the series shown in Figure 4.a; we call those *master series*.

DEFINITION 2 (MASTER SERIES). *Given a data series* D*, and a subsequence length range* $[\ell_{min}, \ell_{max}]$*, the master series are subsequences of the form* $D_{i,min(|D|-i+1,\ell_{max})}$*, for each* i *such that* $1 \leq i \leq |D| - (\ell_{min} - 1)$*, where* $1 \leq \ell_{min} \leq \ell_{max} \leq |D|$*.*

We observe that the following property holds for the master series.

LEMMA 1. *For any master series of the form* $D_{i,\ell'}$*, we have that* $PAA(D_{i,\ell'})_{1,...,k} = PAA(D_{i,\ell''})_{1,...,k}$ *holds for each* ℓ'' *such that* $\ell'' \geq \ell_{min}$*,* $\ell'' \leq \ell' \leq \ell_{max}$ *and* $\ell', \ell''\%k = 0$*.*

PROOF. It trivially follows from the fact that, each non master series is always entirely overlapped by a master series. Since the subsequences are not subject to any scale normalization, their prefix coincides to the prefix of the equi-offset master series. □

Intuitively, the above lemma says that by computing only the PAA of the master series in D, we are able to represent the PAA prefix of any subsequence of D.

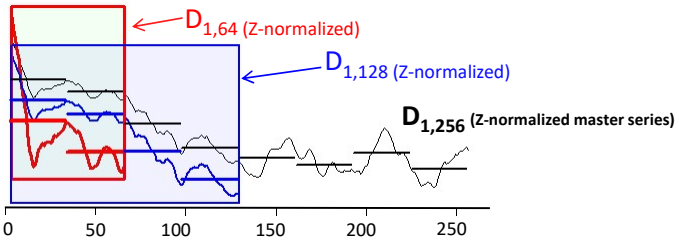


Figure 5: Master series $D_{1,256}$ **with marked PAA coefficients.**

When we zero-align the PAA summaries of the master series, we compute the minimum and maximum PAA values (over all the subsequences) for each segment: this forms what we call an *Envelope* (refer to Figure 4.c). (When the length of a master series is not a multiple of the PAA segment length, we compute the PAA coefficients of the longest prefix, which is multiple of a segment.) We call *containment area* the space in between the segments that define the Envelope.

4.2 PAA Envelope

In this subsection, we formalize the concept of the *Envelope*, introducing a new series representation.

We denote by L and U the PAA coefficients, which delimit the lower and upper parts, respectively, of a containment area (see Figure 4.c). Furthermore, we introduce a parameter γ, which corresponds to the number of master series we represent by the Envelope. This allows to tune the number of subsequences of length in the range $[\ell_{min}, \ell_{max}]$, that a single Envelope represents, influencing both the tightness of a containment area and the size of the Index (number of computed Envelopes). We will show the effect of the relative tradeoff i.e., Tightness/Index size in the Experimental evaluation. Given a, the point from where we start to consider the subsequences in D, and s, the chosen length of the PAA segment, we refer to an Envelope using the following signature:

$$paaENV_{[D,\ell_{min},\ell_{max},a,\gamma,s]} = [L, U] \quad (1)$$

4.3 PAA Envelope for Z-Normalized subsequence

So far we have considered that each subsequence in the input series D is not subject of any scale normalization, i.e., is not Z-normalized. We introduce here a negative result, concerning the *unsuitability* of a generic $paaENV_{[D,\ell_{min},\ell_{max},a,\gamma,s]}$ to describe subsequences that are Z-normalized.

Intuitively, we argue that the PAA coefficients of a single master series $D_{i,a}$, generate a containment area, which may not embed the coefficients of the Z-normalized subsequence in the form $D'_{i,a'}$, for $a' < a$. This happens, because Z-normalization causes the subsequences of different lengths to change their shape, and even shift on the y-axis. Figure 5 depicts such an example.

We can now formalize this negative result.

LEMMA 2. *A* $paaENV_{[D,\ell_{min},\ell_{max},a,\gamma,s]}$ *is not guaranteed to contain all the* PAA *coefficients of the Z-normalized subsequences of lengths* $[\ell_{min}, \ell_{max}]$*, of* D*.*

PROOF. To prove the correctness of the lemma, it suffices to pick such a case where a subsequence of D, namely $D_{a,\ell'}$, with $\ell_{min} \leq \ell' \leq \ell_{max}$, is not encoded by $paaENV_{[D,\ell_{min},\ell_{max},a,\gamma,s]}$. Formally, we should consider the case where $\exists k$ such that $PAA(D_{i,\ell'})_k > U_k$ or $PAA(D_{i,\ell'})_k < L_k$. We may pick a Z-normalized series D choosing $\ell_{max} = |D| = \ell_{min} + 1$ and $\gamma = 0$. The resulting $paaENV_{[D,\ell_{min}=\ell_{max}-1,\ell_{max}=|D|,i=1,\gamma=0,s]}$ obtains equal

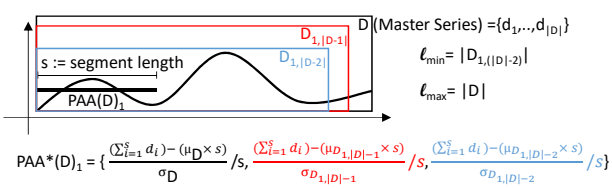


Figure 6: $PAA^*(D)_1$ computation. Since the first PAA segment (of length s) of the master series D, is also the first one of the two non master series $D_{1,|D-1|}$, $D_{1,|D-2|}$, three PAA coefficients are computed with the different normalizations.

bounds, namely $L = U$. Let consider the z-normalized subsequence $D_{1,\ell_{min}}$. Its PAA coefficients must be in the envelope. This implies that, $PAA(D_{1,\ell_{min}})_1 = L_1 = U_1$ (2) must hold. If s is the PAA segment length, in the case of Z-normalization, $PAA(D_{1,\ell_{min}})_1 = (((\sum_{i=1}^{s} d_i) - (\mu_{D_{1,\ell_{min}}} \times s))/\sigma_{D_{1,\ell_{min}}})/s$ and $U_1 = (((\sum_{i=1}^{s} d_i) - (\mu_D \times s))/\sigma_D)/s$. Therefore, the following equation: $(\mu_{D_{1,\ell_{min}}} \times s)/\sigma_{D_{1,\ell_{min}}} = (\mu_D \times s)/\sigma_D$ holds, which is equivalent to $\mu_{D_{1,\ell_{min}}}/\sigma_{D_{1,\ell_{min}}} = \mu_D/\sigma_D$. At this point we may have that $\mu_D = \mu_{D_{1,\ell_{min}}}$, when $d_{\ell_{max}} = \mu_{D_{1,\ell_{min}}}$. This clearly leads to have a smaller dispersion on D than $D_{1,\ell_{min}}$ and thus $\sigma_D < \sigma_{D_{1,\ell_{min}}} \Longrightarrow$ (2) does not hold. □

If we want to build an Envelope, containing all the Z-normalized sequences, we need to take into account the shifted coefficients of the Z-normalized subsequences, which are not master series. Hence, each PAA segment coefficient (in a master series) will be represented by the set of values resulting from the Z-normalizations of all the subsequences of length in $[\ell_{min}, \ell_{max}]$ that are not master series and contain that segment.

Given a generic master series $D_{i,\ell} = \{d_i, ..d_{i+\ell-1}\}$, and s the length of the segment, its k^{th} PAA coefficient set is computed by: $PAA^*(D_{i,\ell})_k = \{(\frac{(\sum_{p=s(k-1)+1}^{s(k-1)+s} d_p) - (\mu_{D_{i,\ell'}} \times s)}{\sigma_{D_{i,\ell'}}})/s | \ell_{min} \leq \ell' \leq \ell_{max}, \ell' \geq (s(k-1)+s-(i-1))\}$ (3).

In Figure 6, we depict an example of PAA^* computation for the first segment of the master series D.

We can then follow the same procedure as before (in the case of non Z-normalized sequences), computing the minimum and maximum PAA coefficients for each segment given by the above formula, in order to get the Envelope for the Z-normalized sequences (which we also denote with $paaENV$).

4.4 Indexing the Envelopes

Here, we define the procedure used to index the Envelopes. In that regard, we aim to adapt the $iSAX$ indexing mechanism (depicted in Figure 3).

Given a $paaENV$, we can translate its PAA extremes into the relative iSAX representation: $uENV_{paaENV_{[D,\ell_{min},\ell_{max},a,\gamma,s]}} = [iSAX(L), iSAX(U)]$, where $iSAX(L)$ ($iSAX(U)$) is the vector of the minimum (maximum) PAA coefficients of all the segments corresponding to the subsequences of D.

The *ULISSE* Envelope, $uENV$, represents the principal building block of the *ULISSE* index. Note that, we might remove for brevity the subscript containing the parameters from the $uENV$ notation, when they are explicit.

In Figure 7, we show a small example of envelope building, given an input series D. The picture shows the PAA coefficients computation of the master series. They are calculated by using a sliding window starting at point $a = 1$, which stops after γ steps. Note that the Envelope generates a containment area,

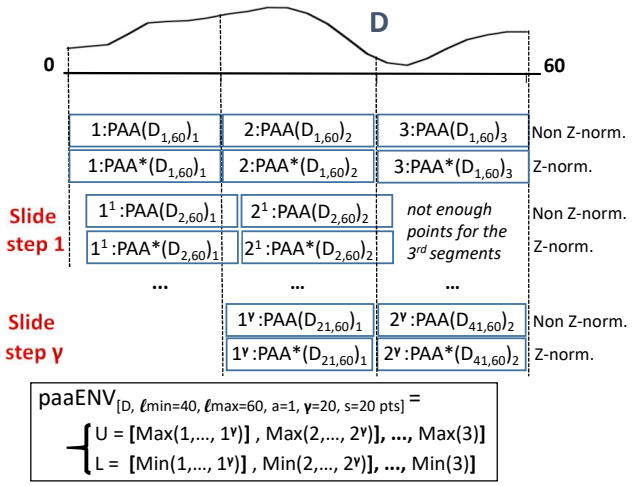


Figure 7: $uENV$ building, with input: data series D of length 60, PAA segment size $= 20$, $\gamma = 20$, $\ell_{min} = 40$ and $\ell_{max} = 60$.

Algorithm 1: *$uENV$ computation*

Input: float[] D, int s, int ℓ_{min}, int ℓ_{max}, int γ, int a
Output: $\mathbf{uENV}[\mathbf{iSAX_{min}}, \mathbf{iSAX_{max}}]$

```
1  int w ← ⌊ℓmax/s⌋ ;
2  int segUpdateList[S] ← {0,...,0};
3  float U[w] ← {−∞, ..., −∞}, L[w] ← {∞, ..., ∞};
4  if |D| − (i − 1) ≥ ℓmin then
5      float paaRSum ← 0;
       // iterate the master series.
6      for i ← a to min(|D|,a + ℓmax + γ) do
           // running sum of paa segment
7          paaRSum ← paaRSum + D[i];
8          if (j-a) > s then
9              paaRSum ← paaRSum - D[i-s];
10         for z ← 1 to min( ⌊i-(a-1)] / s⌋,w) do
11             if segUpdatedList[z] ≤ γ then
12                 segUpdateList[z] ++;
13                 float paa ← (paaRSum / s);
14                 L[z] ← min(paa, L[z]);
15                 U[z] ← max(paa, U[z]);
16     uENV ← [iSAX(L),iSAX(U)];
17 else
18     uENV ← ∅;
```

which embeds all the subsequences of D of all lengths in the range $[\ell_{min}, \ell_{max}]$.

5. INDEXING ALGORITHM

5.1 Non Z-Normalized Subsequences

We are now ready to introduce the algorithms for building an $uENV$. Algorithm 1 describes the procedure for non-Z-normalized subsequences. As we noticed, maintaining the running sum of the last s points, i.e., the length of a PAA segment (refer to Line 7), allows us to compute all the PAA values of the expected envelope in $O(w(\ell_{max} + \gamma))$ time in the worst case, where $\ell_{max} + \gamma$ is the points window we need to take into account for processing each master series, and w is the number of PAA segments in the maximum subsequence length ℓ_{max}. Since w, is usually a very small number (ranging between 8-16), it essentially plays the role of a constant factor. In order to consider not more than γ steps for each segment position, we store how many times we use it, to update the final envelope in the vector, in Line 2.

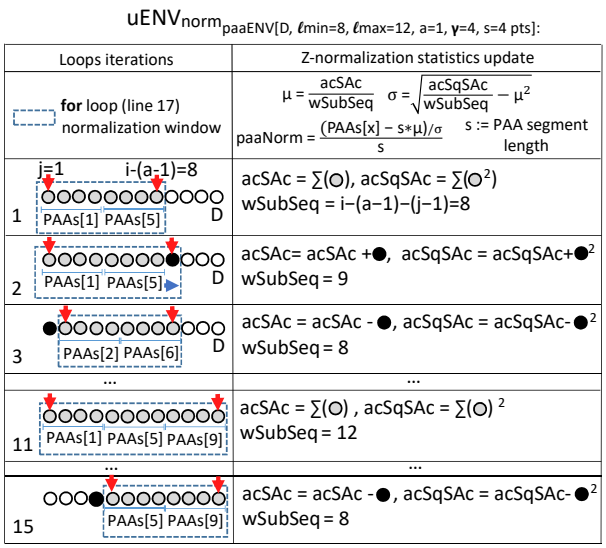


Figure 8: Running example of Algorithm 2. *Left column*) Points iteration, the dashed squared contours the subsequence used to normalize the PAA coefficients in the Second loop. *Right column*) Statistics update at each step, which serve the computation of μ and σ of each possible coefficients normalization.

5.2 Z-Normalized Subsequences

In Algorithm 2, we show the procedure that computes an indexable Envelope for Z-normalized sequences, which we denote as $uENV_{norm}$. This routine iterates over the points of the overlapping subsequences of variable length (*First loop* in Line 7), and performs the computation in two parts. The first operation consists of computing the sum of each PAA segment we keep in the vector $PAAs$ defined in Line 2. When we encounter a new point, we update the sum of all the segments that contain that point (Lines 8-11). The second part, starting in Line 16 (*Second loop*), performs the segment normalizations, which depend on the statistics (mean and std.deviation) of all the subsequences of different length (master and non-master series), in which they appear. During this step, we keep the sum and the squared sum of the window, which permits us to compute the mean and the standard deviation in constant time (Lines 19,20). We then compute the Z-normalizations of all the PAA coefficients in Line 25, by using Equation 3.

In Figure 8, we show an example that illustrates the operation of the algorithm. In *1*, the *First loop* has iterated over *8* points (marked with the dashed square). Since they form a subsequence of length ℓ_{min}, the *Second Loop* starts to compute the Z-normalized PAA coefficients of the two segments, computing the mean and the standard deviation using the sum ($acSAc$) and squared sum ($acSqAc$) of the points considered by the *First loop* (gray circles). The second step takes place after that the *First Loop* has considered the 9^{th} point (black circle) of the series. Here, the *Second Loop* updates the sum and the squared sum, with the new point, calculating then the corresponding new Z-normalized PAA coefficients. At step *3*, the algorithm considers the second subsequence of length ℓ_{min}, which is contained in the nine points window. The *Second Loop* considers in order all the overlapping subsequences, with different prefixes and length. This permits to update the statistics (and all possible normalizations) in constant time. The algorithm terminates, when all the points are considered by the *First loop*, and the *Second Loop* either encounters a subsequence of length ℓmin (as depicted in the step *15*), or performs at most γ iterations, since all the subsequences starting at position $a+\gamma+1$ or later (if any) will be represented by other Envelopes.

Algorithm 2: $uENV_{norm}$ computation

Input: **float**[] D, **int** s, **int** ℓ_{min}, **int** ℓ_{max}, **int** γ, **int** a
Output: $\mathbf{uENV_{norm}[iSAX_{min}, iSAX_{max}]}$

```
1  int w ← ⌊ℓmax/s⌋ ;
   // sum of PAA segments values
2  float PAAs [ℓmax + γ − (s − 1)] ← {0,...,0};
3  float U[w] ← {−∞, ..., −∞}, L[w] ← {∞, ..., ∞};
4  if |D| − (a − 1) ≥ ℓmin then
5      int nSeg← 1;
6      float accSum, accSqSum ← 0;
       // First loop:  Iterate the points.
7      for i ← a to min(|D|,(a+ℓmax+γ)) do
           // update sum of PAA segments values
8          if i − a > s then
9              nSeg++;
10             PAAs[nSeg] ← PAAs[nSeg-1] - D[i-s];
11         PAAs[nSeg] += D[i];
           // keep sum and squared sum.
12         accSum += D[i], accSqSum += (D[i])^2;
           // the window contains enough points.
13         if i-(a-1) ≥ ℓmin then
14             acSAc ← accSum, acSqSAc ← accSqSum;
15             int nMse ← min(γ+1,(i-(a-1)-ℓmin) + 1);
               // Normalizations of PAA coefficients.
16             for j ← 1 to nMse do
17                 int wSubSeq ← i-(a-1)-(j-1) ;
18                 if wSubSeq ≤ ℓmax then
19                     float μ ←acSAc/wSubSeq;
20                     float σ ← √((acSqSAc/wSubSeq) − μ²);
21                     int nSeg ← ⌊wSubSeq÷s⌋;
22                     for z ← 1 to nSeg do
23                         float a ← PAAs[j+[(z-1)×s]];
24                         float b ← s×μ;
25                         float paaNorm ← ((a−b)/σ)/s ;
26                         L[z] ← min(paaNorm, L[z]);
27                         U[z] ← max(paaNorm, U[z]);
28                 acSAc -= D[j], acSqSAc -= (D[j])^2;
29     uENV_norm ← [iSAX(L),iSAX(U)];
30 else
31     uENV_norm ← ∅;
```

5.2.1 Complexity Analysis

Given w, the number of PAA segments in the window of length ℓ_{max}, and $M = \ell_{max} - \ell_{min} + \gamma$, the number of master series we need to consider, building a normalized Envelope, $uENV_{norm}$, takes $O(M\gamma w)$ time.

5.3 Building the index

We now introduce the algorithm, which builds a *ULISSE* index upon a data series collection. We maintain the structure of the $iSAX$ index [21], introduced in the preliminaries.

Each *ULISSE* internal node stores the Envelope $uENV$ that represents all the sequences in the subtree rooted at that node. Leaf nodes contain several Envelopes, which by construction have the same $iSAX(L)$. On the contrary, their $iSAX(U)$ varies, since it get updated with every new insertion in the node. (Note that, inserting by keeping the same $iSAX(U)$ and updating $iSAX(L)$ represents a symmetric and equivalent choice.)

In Figure 9, we show the structure of the *ULISSE* index during the insertion of an Envelope (rectangular/yellow box). Note that insertions are performed based on $iSAX(L)$ (underlined in the figure). Once we find a node with the same $iSAX(L) = (1-0-0-0)$ (Figure 9, $1^{st} step$) if this is an inner node, we

Algorithm 3: *ULISSE* index computation

Input: Collection C, **int** s, **int** ℓ_{min}, **int** ℓ_{max}, **int** γ, **bool** $bNorm$
Output: ***ULISSE*** **index** I

```
1  foreach D in C do
2      int a' ← ∅;
3      uENV E ← ∅;
4      while true do
5          if bNorm then
6              E ← uENV_norm(D, s, ℓ_min, ℓ_max, γ, a');
7          else
8              E ← uENV(D, s, ℓ_min, ℓ_max, γ, a');
9          a' ← a' + γ + 1 ;
10         if E == ∅ then
11             break;
12         bulkLoadingIndexing(I, E);
13         I.inMemoryList.add(maxCardinality(E));
```

descend its subtree (always following the $iSAX(L)$ representations) until we encounter a leaf. During this path traversal, we also update the $iSAX$ representation of the Envelope we are inserting, by increasing the number of bits of the segments, as necessary. In our example, when the Envelope arrives at the leaf, it has increased the cardinality of the second segment to two bits: $iSAX(L) = (1{-}\mathbf{10}{-}0 - 0)$, and similarly for $iSAX(U)$ (Figure 9, $2^{nd}step$). Along with the Envelope, we store in the leaf a pointer to the location on disk for the corresponding raw data series. We note that, during this operation, we do not move any raw data into the index.

To conclude the insertion operation, we also update the $iSAX(U)$ of the nodes visited along the path to the leaf, where the insertion took place. In our example, we update the upper part of the leaf Envelope to $iSAX(U) = (1{-}\mathbf{11}{-}0 - 0)$, as well as the upper part of the Envelope of the leaf's parent to $iSAX(U) = (1{-}\mathbf{1}{-}0 - 0)$ (Figure 9, $3^{rd}step$). This brings the *ULISSE* index to a consistent state after the insertion of the Envelope.

Algorithm 3 describes the procedure, which iterates over the series of the input collection C, and inserts them in the index. Note that function $bulkLoadingIndexing$ in Line 12 may use different bulk loading techniques. In our experiments, we used the iSAX 2.0 bulk loading algorithm [34]. Alongside the index, we also keep in memory (using the raw data order) all the Envelopes, represented by the symbols of the highest $iSAX$ cardinality available (Line 13). This information is used during query answering.

5.3.1 Space complexity analysis

The index space complexity is equivalent for the case of Z-normalized and non Z-normalized sequences. The choice of γ determines the number of $Envelopes$ generated and thus the index size. Hence, given a data series collection $C = \{D^1, ..., D^{|C|}\}$ the number of extracted Envelopes is given by $N = (\sum_i^{|C|} \lfloor \frac{|D^i|}{\ell_{min}+\gamma} \rfloor)$. If w PAA segments are used to discretize the series, each $iSAX$ symbol is represented by a single byte (binary label) and the disk pointer in each Envelope occupies b bytes (in general *8* bytes are used). The final space complexity is $O((2w)bN)$.

6. SIMILARITY SEARCH WITH ULISSE

In this section, we present the building blocks of the similarity search algorithms we developed for the *ULISSE* index.

6.1 Lower Bounding Euclidean Distance

The iSAX representation allows the definition of a distance function, which lower bounds the true Euclidean [19]. This function compares the PAA coefficients of the first data series, against the

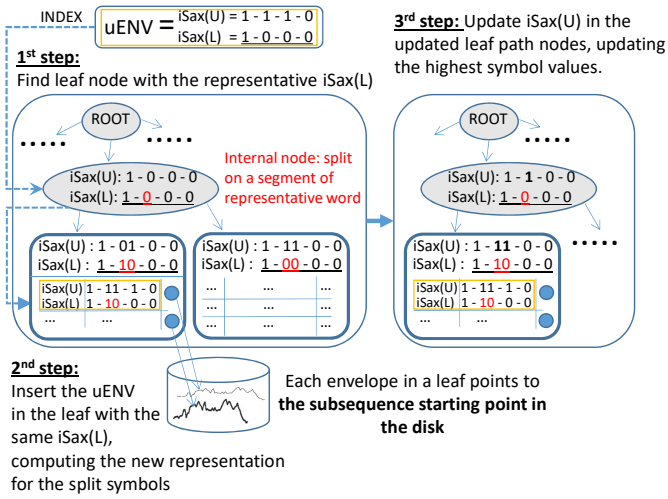


Figure 9: Envelope insertion in an *ULISSE* index. $iSAX(L)$ is chosen to accommodate the Envelopes inside the nodes.

iSAX breakpoints (values) that delimit the symbol regions of the second data series.

Let $\beta_u(S)$ and $\beta_l(S)$ be the breakpoints of the iSAX symbol S. We can compute the distance between a PAA coefficient and an iSAX region using:

$$distLB(PAA(D)_i, iSAX(D')_i) =$$

$$\begin{cases} (\beta_u(iSAX(D')_i) - PAA(D)_i)^2 & if \beta_u(iSAX(D')_i) < PAA(D)_i \\ (\beta_l(iSAX(D')_i) - PAA(D)_i)^2 & if \beta_l(iSAX(D')_i) > PAA(D)_i \\ 0 & \text{otherwise.} \end{cases}$$

In turn, the lower bounding distance between two equi-length series D,D', represented by w PAA segments and w $iSAX$ symbols, respectively, is defined as:

$$mindist_{PAA\_iSAX}(PAA(D), iSAX(D')) = \sqrt{\frac{|D|}{w}} \sqrt{\sum_{i=1}^{w} distLB(PAA(D)_i, iSAX(D')_i)}. \quad (4)$$

We rely on the following proposition [40]:

PROPOSITION 1. *Given two data series D, D', where $|D| = |D'|$, $mindist_{PAA\_iSAX}(PAA(D), iSAX(D')) \leq ED(D, D')$.*

Since our index contains Envelope representations, we need to adapt Equation 4, in order to lower bound the distances between a data series Q, which we call query, and a set of subsequences, whose iSAX symbols are described by the Envelope $uENV_{paaENV_{[D,\ell_{min},\ell_{max},a,\gamma,s]}} = [iSAX(L), iSAX(U)]$.

Therefore, given w, the number of PAA coefficients of Q, that are computed using the Envelope PAA segment length s on the longest multiple prefix, we define the following function:

$$mindist_{ULiSSE}(PAA(Q), uENV_{paaENV_{...}}) =$$

$$\sqrt{s} \sqrt{\sum_{i=1}^{w} \begin{cases} (PAA(Q)_i - \beta_u(iSAX(U)_i))^2, & if \beta_u(iSAX(U)_i) < PAA(Q)_i \\ (PAA(Q)_i - \beta_u(iSAX(L)_i))^2, & if \beta_l(iSAX(L)_i) > PAA(Q)_i \\ 0 & \text{otherwise.} \end{cases}} \quad (5)$$

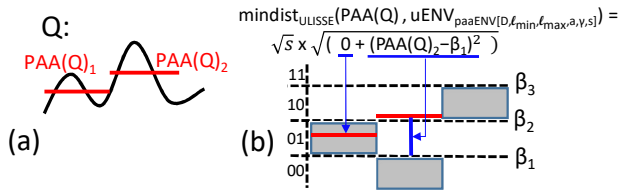


Figure 10: Given the PAA representation of a query Q (*a*) and $uENV_{paaENV_{[D,\ell_{min},\ell_{max},a,\gamma,s]}}$ (*b*) we compute their $mindist_{ULiSSE}$. The $iSAX$ space is delimited with dashed lines and the relative breakpoints β_i.

In Figure 10, we report an example of $mindist_{ULiSSE}$ computation between a query Q, represented by its PAA coefficients, and an Envelope in the iSAX space.

PROPOSITION 2. *Given two data series Q,D,* $mindist_{ULiSSE}(PAA(Q), uENV_{paaENV_{[D,\ell_{min},\ell_{max},a,\gamma,s]}}) \leq ED(Q, D_{i,|Q|})$*, for each i such that* $a \leq i \leq a+\gamma+1$ *and* $|D|-(i-1) \geq \ell min$.

PROOF. (sketch) We may have two cases, when $mindist_{ULiSSE}$ is equal to zero, the proposition clearly holds, since Euclidean distance is non negative. On the other hand, the function yields values greater than zero, if one of the first two branches is true. Let consider the first (the second is symmetric). If we denote with D'' the subsequence in D, such that $\beta_l(iSAX(U)_i) \leq PAA(D'')_i \leq \beta_u(iSAX(U)_i)$, we know that the upper breakpoint of the i^{th} iSAX symbol, of each subsequence in D, which is represented by the Envelope, must be less or equal than $\beta_u(iSAX(U)_i)$. It follows that, for this case, Equation 5 is equivalent to $distLB(PAA(Q)_i, iSAX(D'')_i)$, which yields the shortest lower bounding distance between the i^{th} segment of points in D and Q. □

6.2 Approximate search

Similarity search performed on *ULISSE* index relies on Equation 5 to prune the search space. This allows to navigate the tree in order, visiting first the most promising nodes.

We thus provide a fast approximate search procedure we report in Algorithm 4. In Line 4, we start to push the internal nodes of the index in a priority queue, where the nodes are sorted according to their lower bounding distance to the query. Note that in the comparison, we use the largest prefix of the query, which is a multiple of the PAA segment length, used at the index building stage (Line 1). Then, the algorithm pops the ordered nodes from the queue, visiting their children in the loop of Line 6. In this part, we still maintain the internal nodes ordered (Lines 21,22).

As soon as a leaf node is discovered (Line 8), we check if its $mindist$ to the query is shorter than the *bsf*. If this is verified, the dataset does not contain any data series that are closer than those already compared with the query. In this case, the approximate search result coincides with that of the exact search. Otherwise, we can load the raw data series pointed by the Envelopes in the leaf, which are in turn sorted according to their position, to avoid random disk reads. We visit a leaf only if it contains Envelopes that represent sequences of the same length as the query. Each time we compute the true Euclidean distance, the best-so-far distance (*bsf*) is updated, along with the R^{approx} vector. Since priority is given to the most promising nodes, we can terminate our visit, when at the end of a leaf visit the k *bsf*'s have not improved (Line 15). Hence, the vector R^{approx} contains the k approximate query answers.

Algorithm 4: *ULISSE K-nn-Approx*

```
   Input: int k, float [] Q, ULISSE index I
   Output: float [k][|Q|] R^approx, float [] bsf
1  float [] Q* ← PAA(Q_{1,..,⌊|Q|/I.s⌋});
2  float[k] bsf ← {∞, ..., ∞} ;
3  PriorityQueue nodes;
4  foreach node in I.root.children() do
5  |  nodes.push(node, mindist_ULiSSE(Q*, node));
6  while n = nodes.pop() do
7  |  if n.isLeaf() and n.containsSize(|Q|) then
8  |  |  if n.mindist < bsf[k] then
   |  |  |  // sort according disk pos.
9  |  |  |  uENV [] Envelopes = sort(n.Envelopes);
   |  |  |  // iterate the Env.  and compute true ED
10 |  |  |  oldBSF ← bsf[k];
11 |  |  |  foreach E in Envelopes do
12 |  |  |  |  float [] D ← readSeriesFromDisk(E);
13 |  |  |  |  for i ← E.a to min(E.a+E.γ+1,|D| − (|Q| − 1)) do
14 |  |  |  |  |  ED_updateBSF(Q, E.D_{i,|Q|}, k,bsf, R^approx);
   |  |  |  // if bsf has not improved end visit.
15 |  |  |  if oldBSF == bsf[k] then
16 |  |  |  |  break;
17 |  |  else
18 |  |  |  break; // Approximate search is exact.
19 |  |
20 |  else
21 |  |  nodes.push(n.right, mindist_ULISSE(Q*, n.right));
22 |  |  nodes.push(n.left, mindist_ULISSE(Q*, n.left));
```

Algorithm 5: *ULISSE K-nn-Exact*

```
   Input: int k, float [] Q, ULISSE index I
   Output: float [k][|Q|] R
1  float [] Q* ← PAA(Q_{1,..,⌊|Q|/I.s⌋});
2  float [] bsf, float [k][|Q|] R ← K-nn-Approx(k, Q, I) ;
3  if bsf is not exact then
4  |  foreach E in I.inMemoryList do
5  |  |  if mindist_ULiSSE(Q*, E) < bsf[k] then
6  |  |  |  float [] D ← readSeriesFromDisk(E);
7  |  |  |  for i ← E.a to min(E.a+E.γ+1,|D| − (|Q| − 1)) do
8  |  |  |  |  ED_updateBSF(Q, E.D_{i,|Q|}, k,bsf, R);
```

6.3 Exact search

Note that the approximate search described above may not visit leaves that contain answers better than the approximate answers already identified, and therefore, it will fail to produce exact, correct results. We now describe an exact nearest neighbor search algorithm, which finds the k sequences with the absolute smallest distances to the query.

In the context of exact search, accessing disk-resident data following the lower bounding distances order may result in several leaf visits: this process can only stop after finding a node, whose lower bounding distance is greater than the *bsf*, guaranteeing the correctness of the results. This would penalize computational time, since performing many random disk I/O might unpredictably degenerate.

We may avoid such a bottleneck by sorting the Envelopes, and in turn the disk accesses. Moreover, we can exploit exploit the *bsf* provided by approximate search, in order to perform a sequential search with pruning over the sorted Envelopes list (this list is stored across the *ULISSE* index). Intuitively, we rely on two aspects. First, the *bsf*, which can translate into a tight-enough bound for pruning the candidate answers. Second, since the list has no hierarchy structure, any Envelope is stored with the highest cardinality available, which guarantees a fine representation of the series, and can contribute to the pruning process.

Algorithm 5 describes the exact search procedure. In Line 5, we compute the lower bounding distance between the Envelope and the query. If it is not better than the k^{th} *bsf*, we do not access the

disk, pruning Euclidean Distance computations as well. We note that, while we are computing the true Euclidean distance, we can speed-up computations using the *Early Abandoning* technique [28], which is effective especially for Z-normalized data series.

6.4 Complexity of query answering

We provide now the time complexity analysis of query answering with *ULISSE*. Both the approximate and exact query answering time strictly depend on data distribution as shown in [41]. We focus on exact query answering, since approximate is part of it.

Best Case. In the best case, an exact query will visit one leaf at the stage of the approximate search (Algorithm 4), and during the second leaf visit will fulfill the stopping criterion (i.e., the *bsf* distance is smaller than the $mindist$ between the second leaf and the query). Given the number of the first layer nodes (root nodes) N, the length of the first leaf path L, and its size S, the best case complexity is given by the cost to iterate the first layer node and descend to the leaf keeping the nodes sorted in the heap: $O(w(N + LlogL))$, where w is the number of symbols checked at each *mindist* computation. Moreover we need to take into account the additional cost of computing the true distances in the leaf, which is $O(S(logS+\ell_{max}))$ (including both the cost of sorting the disk accesses, and the cost of computing the Euclidean distances).

Worst Case. The worst case for exact search takes place when at the approximate search stage, the complete set of leaves that we denote with T, need to be visited. This has a cost of $O(w(N + TLlogL))$ plus the cost of computing the true Euclidean distances, which in this case takes $O(T(S(logS+\ell_{max})))$. Note though that this worst case is pathological: for example, when all the series in the dataset are the same straight lines (only sligthly perturbed). Evidently, the very notion of indexing does not make sense in this case, where all the data series look the same. As we show in our experiments on several datasets, in practice, the approximate algorithm always visits a very small number of leaves.

***ULISSE* K-nn Exact complexity.** So far we have considered the exact K-nn search with regards to Algorithm 4 (approximate search). When this algorithm produces approximate answers, providing just an upper bound *bsf*, in order to compute exact answers we must run Algorithm 5 (exact search). The complexity of this procedure is given by the cost of iterating over the Envelopes and computing the *mindist*, which takes $O(Mw)$ time, where M is the total number of Envelopes. Let's denote with V the number of Envelopes, for which the raw data are retrieved from disk and checked. Then, the algorithm takes an additional $O(V\ell_{max})$ time to compute the true Euclidean distances.

7. EXPERIMENTAL EVALUATION

Setup. All the experiments presented in this section are completely reproducible: the code and datasets we used are available online [42]. We implemented all algorithms (indexing and query answering) in C (compiled with gcc 4.8.2). We ran experiments on an Intel Xeon E5-2403 (4 cores @ 1.9GHz), using the x86_64 GNU/Linux OS environment.

Algorithms. We compare *ULISSE* to the *Compact Multi-Resolution Index (CMRI) [20]*, which is the current state-of-the-art index for similarity search with varying-length queries (recall that *CMRI* constructs a limited number of distinct indexes for series of different lengths). We note though, that in contrast to our approach, *CMRI* can only support non Z-normalized sequences. In addition, we compare to the current state-of-the-art algorithms for subsequence similarity search, the *UCR suite* [28], and *MASS* [38]. These algorithms do not use an index, but are based on optimized serial scans, and are natural competitors, since they can process overlapping subsequences very fast.

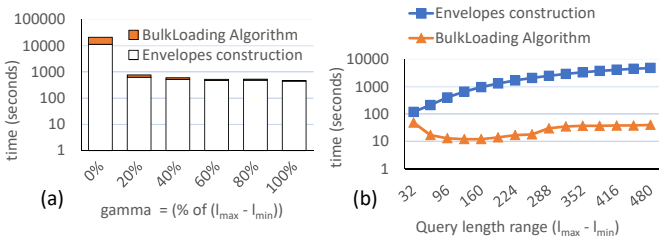


Figure 11: a) Construction and bulk Loading time (log scale) of Envelopes in 5GB datasets varying γ (5M of series of length 256), $\ell_{min} = 160$, $\ell_{max} = 256$. b) Construction and Bulk Loading time (log scale) of Envelopes in 5GB dataset (2.5M of series of length 512) varying $\ell_{max} - \ell_{min}$ (lengths range), $\gamma = 256$, fixed $\ell_{max} = 512$.

Datasets. For the experiments, we used both synthetic and real data. We produced the synthetic datasets with a generator, where a random number is drawn from a Gaussian distribution $N(0, 1)$, then at each time point a new number is drawn from this distribution and added to the value of the last number. This kind of data generation has been extensively used in the past [41], and has been shown to effectively model real-world financial data [15].

The real datasets we used are astrophysics and seismic data series. The first contains *100* Million astronomical data series of length 256 (*100*GB), representing celestial objects (ASTRO)[43]. The second real dataset contains *100* Million seismic data series (*100*GB) of length 256, collected from the IRIS Seismic Data Access repository (SEISMIC) [44]. In our experiments, we test queries of lengths *160-4096* points, since these cover at least 90% of the ranges explored in works about data series indexing in the last two decades [33, 45, 46]. Moreover, of the 85 datasets in the UCR archive [47], only four are (slightly) longer than *1,024*.

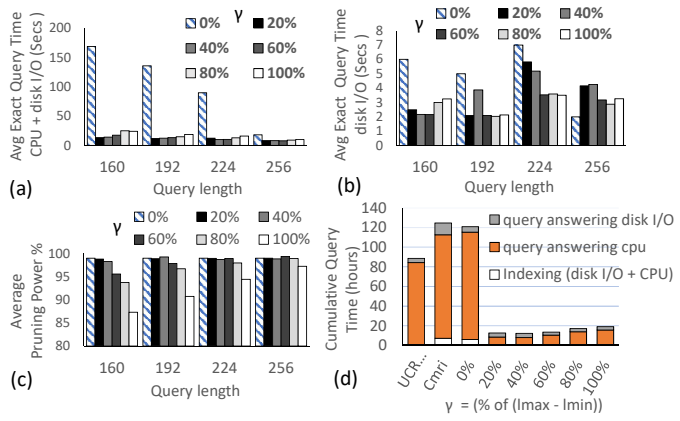
7.1 Envelope Building

In the first set of experiments, we analyze the performance of the *ULISSE* indexing algorithm. In Figure 11.a) we report the indexing time (Envelope Building and Bulk loading operations) when varying γ. We use a dataset containing *5M* series of length *256*, fixing $\ell_{min} = 160$ and $\ell_{max} = 256$. We note that, when $\gamma = 0$, the algorithm needs to extract as many Envelopes as the number of master series of length ℓ_{min}. This generates a significant overhead for the index building process (due to the maximal Envelopes generation), but also does not take into account the contiguous series of same length, in order to compute the statistics needed for Z-normalization. A larger γ speeds-up the Envelope building operation by several orders of magnitude, and this is true for a very wide range of γ values (Figure 11.a)). These results mean that the $uENV_{norm}$ building algorithm can achieve good performance in practice, despite its complexity that is quadratic on γ.

In Figure 11.b) we report an experiment, where γ is fixed, and the query length range ($\ell_{max} - \ell_{min}$) varies. We use a dataset, with the same size of the previous one, which contains *2.5M* series of length *512*. The results show that increasing the range has a linear impact on the final running time.

7.2 Exact Search Similarity Queries

We now test *ULISSE* on exact 1-Nearest Neighbor queries. We have repeated this experiment varying the *ULISSE* parameters along predefined ranges, which are (default in bold) γ : $[0\%, 20\%, 40\%, 60\%, 80\%, \mathbf{100}\%]$, where the



(e)

Method	γ	Indexing time (h)	Indexing size (GB)	No. of Records/ Envelopes	No. of indexes
Cmri	-	7	90	2B	4
Ulisse	0%	6	58	485M	1
Ulisse	20%	0.2	3	25M	1
Ulisse	40%	0.11	1.7	15M	1
Ulisse	60%	0.08	1	10M	1
Ulisse	80%	0.08	1	10M	1
Ulisse	100%	0.04	0.5	5M	1

Figure 12: Query answering time performance, varying γ on non Z-normalized data series. *a) ULISSE* average query time (CPU + disk I/O). *b) ULISSE* average query disk I/O time. *c) ULISSE* average query pruning power. *d)* Comparison of *ULISSE* to other techniques (cumulative indexing + query answering time). *e)* Table resuming the indexes' properties.

percentage is referring to its maximum value, ℓ_{min} : $[96, 128, \mathbf{160}, 192, 224, 256]$, ℓ_{max} : $[256]$, dataset series length (ℓ_S): $[\mathbf{256}, 512, 1024, 1536, 2048, 2560]$ and dataset size of $5GB$. Here, we use synthetic datasets containing random walk data in binary format, where a single point occupies 4 bytes. Hence, in each dataset C, where $|C|^{Bytes}$ denotes the corresponding size in bytes, we have a number of subsequences of length ℓ given by $N^{seq} = (\ell_S - \ell + 1) \times ((|C|^{Bytes}/4)/\ell_S)$. For instance, in a *5GB* dataset, containing series of length 256, we have ∼*500 Million* subsequences of length *160*.

We record the average *CPU time*, *query disk I/O time* (time to fetch data from disk: Total time - CPU time), and *pruning power* (percentage of the total number of Envelopes in the index that do not need to be read), of *100* queries, extracted from the datasets with the addition of Gaussian noise. For each index used, the *building time* and the relative *size* are reported. Note that we clear the main memory cache before answering each set of queries. We have conducted our experiments using datasets that are both smaller and larger than the main memory.

In all experiments, we report the cumulative running time of *1000* random queries for each query length.

Varying γ. We first present results for similarity search queries on *ULISSE* when we vary γ, ranging from *0* to its maximum value, i.e., $\ell_{max} - \ell_{min}$. In Figure 12, we report the results concerning non Z-normalized series (for which we can compare to *CMRI*). We observe that grouping contiguous and overlapping subsequences under the same summarization (Envelope) by increasing γ, affects positively the performance of index construction, as well as query answering (Figure 12.a,d)). The latter may seem counterintuitive, since γ influences in a negative way pruning power, as depicted in Figure 12.c). Indeed, inserting more master series into a single $Envelope$ is likely to generate large containment areas, which are not tight representations of the data series. On the other hand, it leads to an overall number of $Envelopes$ that is several orders of magnitude smaller than the one for $\gamma = 0\%$. In this last case, when $\gamma = 0$, the algorithm inserts in the index as many records as the number of master series present in the dataset (*485*M), as reported in (Figure 12.e)).

We note that the disk I/O time on compact indexes is not negatively affected at the same ratio of pruning power. On the contrary, in certain cases it becomes faster. For example, the results in Figure 12.b) show that for query length 160, the $\gamma = 100\%$ index is more than 2x faster in disk I/O than the $\gamma = 0\%$ index, despite the fact that the latter index has an average pruning power that is 14% higher (Figure 12.c)). This behavior is favored by disk caching, which translates to a higher hit ratio for queries with slightly larger disk load. We note that we repeated this experiment several times, with different sets of queries that hit different disk locations, in order to verify this specific behavior. The results showed that this disk I/O trend always holds.

While disk I/O represents on average the $3 - 4\%$ of the total query cost, computational time significantly affects the query performance. Hence, a compact index, containing a smaller number of $Envelopes$, permits a fast in memory sequential scan, performed by Algorithm 5.

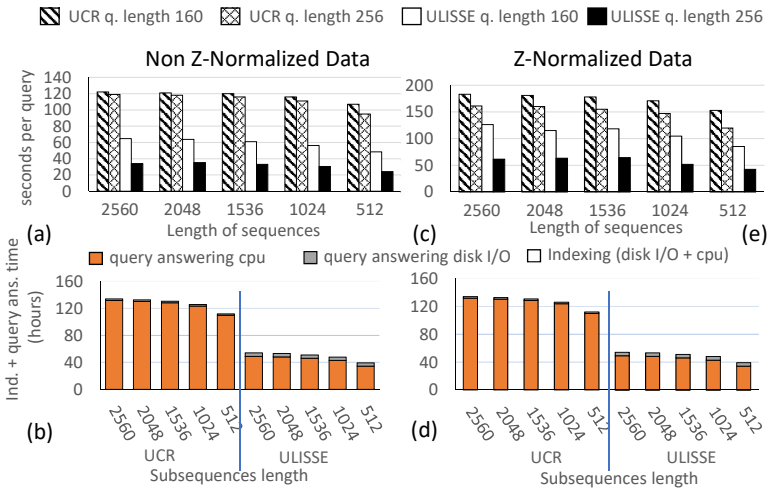
In Figure 12.d) we show the cumulative time performance (i.e., $4,000$ queries in total), comparing *ULISSE*, *CMRI*, and *UCR Suite*. Note that in this experiment, *ULISSE* indexing time is negligible w.r.t. the query answering time. *ULISSE*, outperforms both *UCR Suite* and *CMRI*, achieving a speed-up of up to 12x.

Further analyzing the performance of *CMRI*, we observe that it constructs four indexes (for four different lengths), generating more than $2B$ index records! Consequently, it is clear that the size of these indexes will negatively affect the performance of *CMRI*, even if it achieves reasonable pruning ratios. These results suggest that the idea of generating multiple copies of an index for different lengths, is not a scalable solution.

Varying Length of Data Series. In this part, we present the results concerning the query answering performance of *ULISSE* and *UCR Suite*, as we vary the length of the sequences in the indexed datasets, as well as the query length (refer to Figure 13). In this case, varying the data series length in the collection, leads to a search space growth, in terms of overlapping subsequences, as reported in Figure 13.e). This certainly penalizes index creation, due to the inflated number of Envelopes that need to be generated. On the other hand, *UCR Suite* takes advantage of the high overlapping of the subsequences during the in-memory scan. Note that we do not report the results for $CMRI$ in this experiment, since its index building time would take up to *1 day*. In the same amount of time, *ULISSE* answers more than $1,000$ queries.

Observe that in Figures 13.a) and .c), *ULISSE* shows better query performance than the UCR suite, growing linearly as the search space gets exponentially larger. This demonstrates that *ULISSE* offers a competitive advantage in terms of pruning the search space that eclipses the pruning techniques *UCR Suite*. The aggregated time for answering $4,000$ queries ($1,000$ for each query length) is 2x for *ULISSE* when compared to *UCR Suite* (Figures 13.b) and .d)).

Varying Range of Query Lengths. In the last experiment of this subsection, we investigate how varying the length range $[\ell min; \ell max]$ affects query answering performance. In Figure 14, we depict the results for Z-normalized sequences. We observe that enlarging the range of query length, influences the number of Envelopes we need to accommodate in our index.Moreover, a larger query length range corresponds to a higher number of Series (dif-



# subsequences in 5GB			Indexing Ulisse		
Length	160	256	Non Z-norm. time (secs)	Z-norm. time (secs)	#Envelopes
2560	1,200M	1,152M	377	2273	13,5M
2048	1,180M	1,120M	382	2187	13,75M
1536	1,147M	1,067M	392	2106	13,3M
1024	1,081M	961M	355	1977	13,75M
512	882M	642M	307	1445	12.5M

Figure 13: Query answering time performance of *ULISSE* and *UCR Suite*, varying the data seres size. Average query (CPU time + disk I/O) (*a*) for non Z-normalized, (*c*) for Z-normalized series). Cumulative indexing + query answering time (*b*) for non Z-normalized, (*d*) for Z-normalized series). *e)* Table resuming the indexes' properties. *f)* Comparison between MASS algorithm, *UCR Suite* and *ULISSE*.

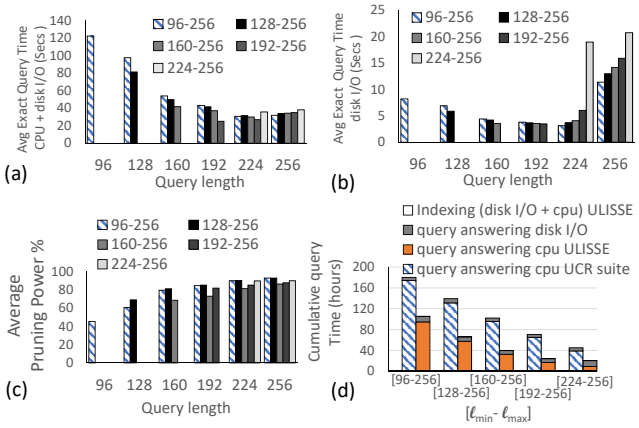


Figure 14: Query answering time, varying the range of query length on Z-normalized data series. (*a) ULISSE* average query time (CPU + disk I/O). (*b) ULISSE* average query disk I/O time. (*c) ULISSE* average query pruning power. (*d) ULISSE* comparison to other techniques (cumulative indexing + query answering time).

ferent normalizations), which the algorithms needs to consider for building a single Envelope (loop of line 16 of Algorithm 2). This leads to large containment areas and in turn, coarse data summarizations. In contrast, Figure 14.c) indicates that pruning power slightly improves as query length range increases. This is justified by the higher number of Envelopes generated, when the query length range gets larger. Hence, there is an increased probability to save disk accesses. In Figure 14.a) we show the average query time (CPU + disk I/O) on each index, observing that this latter is not significantly affected by the variations in the length range. The same is true when considering only the average query disk I/O time (Figure 14.b), which accounts for $3-4\%$ of the total query cost. We note that the cost remains stable as the query range increases, when the query length varies between *96-192*. For queries of length *224* and *256*, when the range is the smallest possible the disk I/O time increases. This is due to the high pruning power, which translates into a higher rate of cache misses.

In Figure 14.d), the aggregated time comparison shows *ULISSE* achieving an up to 2x speed-up over *UCR Suite*.

7.3 Comparison to Serial Scan Algorithms

We now perform further comparisons to serial scan algorithms, namely, *MASS* and *UCR Suite*, with varying query lengths.

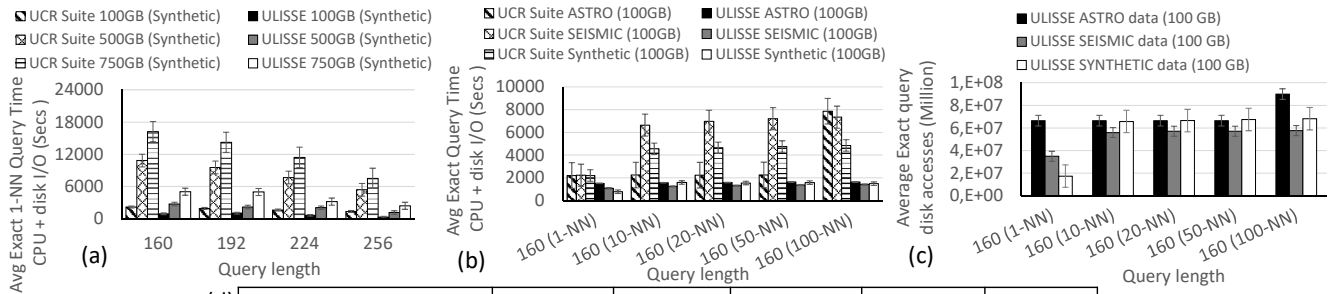
MASS [38] is a recent data series similarity search algorithm that computes the distances between a Z-normalized query of length l and all the Z-normalized overlapping subsequences of a single sequence of length $n \geq l$. *MASS* works by calculating the dot products between the query and n overlapping subsequences in frequency domain, in $logn$ time, which then permits to compute each Euclidean distance in constant time. Hence, the time complexity of *MASS* is $O(nlogn)$, and is independent of the data characteristics and the length of the query (l). In contrast, the *UCR Suite* effectiveness of pruning computations may be significantly affected by the data characteristics.

We compared *ULISSE* (using the default parameters), MASS and *UCR Suite* on a dataset containing *5M* data series of length *4096*. In Figure 14.f), we report the average query time (CPU + disk/io) of the three algorithms.

We note that MASS, which in some cases is outperformed by *UCR Suite* and *ULISSE*, is strongly penalized, when ran over a high number of non overlapping series. The reason is that, although MASS has a low time complexity of $O(nlogn)$, the Fourier transformations (computed on each subsequence) have a non negligible constant time factor that render the algorithm suitable for computations on very long series.

7.4 Approximate Search Similarity Queries

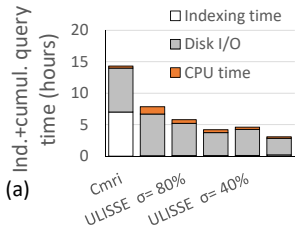
In this subsection, we evaluate *ULISSE* approximate search. Since we compare our approach to CMRI, Z-normalization is not applied. Figure 15.a) depicts the cumulative query answering time for $4,000$ queries. As previously, we note that the indexing time for *ULISSE* is relatively very small. On the other hand, the time that CMRI needs for indexing is 2x more than the time during which *ULISSE*s has finished indexing and answering $4,000$ queries.



(d)

	Synthetic data (100 GB)	Synthetic data (500 GB)	Synthetic data (750 GB)	Seismic data (100 GB)	Astro data (100 GB)
Indexing time (hours)	2.93	15.00	41.79	2.93	2.83
Leaf nodes in the ULISSE index	39,260	36,141	58,108	43,920	44,090
Envelopes in the ULISSE index	100M	453M	750M	100M	100M

Figure 16: Exact and Approximate similarity search on Z-normalized synthetic and real datasets. *a)* Average exact query time (CPU + disk I/O) on synthetic datasets. *b)* Average exact $K-NN$ query time (CPU + disk I/O) on real datasets (100 GB) varying K. *c)* Average disk accesses of $K-NN$ query. *d)* Indexing measures for all datasets.



Query length		160	192	224	256
Ranking position		Ulisse			
	1-25	92%	90%	92%	100%
	26-50	8%	10%	8%	0%
		Cmri			
	1-25	100%	100%	100%	100%

(b)

Figure 15: Approximate query answering on non Z-normalized data series. (*a)* Cumulative Indexing + approximate search query time (CPU + disk I/O) of $4,000$ queries ($1,000$ per each query length in [160,192,224,256]). (*b)* Approximate quality: percentage of answers in the relative exact search range.

In Figure 15.b), we measure the quality of the Approximate search. In order to do this, we consider the exact query results ranking, showing how the approximate answers are distributed along this rank, which represents the ground truth. We note that CMRI answers have slightly better positions than the *ULISSE* ones. This happens thanks to the tighter representation generated by the complete sliding window extraction of each subsequence, employed by CMRI. Nevertheless, this small penalty in precision is balanced out by the considerable time performance gains: *ULISSE* is up to 15x faster than CMRI. When we use a smaller γ, (e.g., 20), *ULISSE* shows its best time performance. This is due to tighter $Envelopes$ containment area, which permits to find a better best-so-far with a shorter tree index visit.

7.5 Experiments with Real

In this last part, we test *ULISSE* on three large synthetic datasets of sizes *100*GB, *500*GB, and *750*GB, as well as on two real series collections, i.e., ASTRO and SEISMIC (described earlier). The other parameters are the default ones. For each generated index and for the *UCR Suite*, we ran a set of 100 queries, for which we report the average exact search time. In Figure 16.a) we report the average query answering time ($1-NN$) on synthetic datasets, varying the query length. These results demonstrate that *ULISSE* scales better than *UCR Suite* across all query lengths, being up to 5x faster. In Figure 16.b), we report the $K-NN$ exact search time performance, varying K and picking the smallest query length, namely *160*. Note that, this is the largest search space we consider in these datasets, since each query has *9.7* billion of possible candidates (subsequences of length *160*). The experimental results on real datasets confirm the superiority of *ULISSE*, which scales with stable performance, also when increasing the number K of nearest neighbors. Once again it is up to 5x faster than *UCR Suite*, whose performance deteriorates as K gets larger. In Figure 16.c) we report the number of disk accesses of the queries considered in Figure 16.b). Here, we are counting the number of times that we follow a pointer from an envelope to the raw data on disk, during the sequential scan in Algorithm 5. Note that the number of disk accesses is bounded by the total number of Envelopes, which are reported in Figure 16.d) (along with the number of leaves and the building time for each index). We observe that in the worst case, which takes place for the ASTRO dataset for $K=100$, we retrieve from disk $\sim$*82%* of the total number of subsequences. This still guarantees a remarkable speed-up over *UCR Suite*, which needs to consider all the raw series. Moreover, since *ULISSE* can use Early Abandoning during exact query answering, we observe during our empirical evaluation that disposing of the approximate answer distance prior the start of the exact search, permits to abandon on average *20%* of points more than *UCR Suite* for the same query.

8. CONCLUSIONS

Similarity search is one of the fundamental operations for several data series analysis tasks. Even though much effort has been dedicated to the development of indexing techniques that can speed up similarity search, all existing solutions are limited by the fact that they can only support queries of a fixed length.

In this work, we proposed *ULISSE*, the first index able to answer similarity search queries of variable-length, over both Z-normalized and non Z-normalized sequences. We experimentally evaluated, our indexing and similarity search algorithms, on synthetic and real datasets, demonstrating the effectiveness and efficiency (in space and time cost) of the proposed solution. In our future work, we will adapt our technique in order to allow the use of more elastic measures, such as *Dynamic Time Warping*. We also plan to study extensions that work for datasets containing a few very long sequences, as well as solutions adapted to multi-core and multi-socket architectures.

References

[1] K. Kashino, G. Smith, and H. Murase, "Time-series active search for quick retrieval of audio and video," in *ICASSP*, 1999.

[2] U. Raza, A. Camerra, A. L. Murphy, T. Palpanas, and G. P. Picco, "Practical data prediction for real-world wireless sensor networks," *IEEE Trans. Knowl. Data Eng.*, 2015.

[3] D. Shasha, "Tuning time series queries in finance: Case studies and recommendations," *IEEE Data Eng. Bull.*, 1999.

[4] P. Huijse, P. A. Estévez, P. Protopapas, J. C. Principe, and P. Zegers, "Computational intelligence challenges and applications on large-scale astronomical time series databases," 2014.

[5] T. Palpanas, "Data series management: The road to big sequence analytics," *SIGMOD Rec.*, 2015.

[6] ESA. SENTINEL-2 mission. [Online]. Available: https://sentinel.esa.int/web/sentinel/missions/sentinel-2

[7] K. Zoumpatianos and T. Palpanas., "Data series management: Fulfilling the need for big sequence analytics." in *ICDE*, 2018.

[8] V. Niennattrakul and C. A. Ratanamahatana, "On clustering multimedia time series data using k-means and dynamic time warping," ser. MUE '07, 2007.

[9] J. Lines and A. Bagnall, "Time series classification with ensembles of elastic distance measures," *Data Mining and Knowledge Discovery*, 2015.

[10] P. Senin, J. Lin, X. Wang, T. Oates, S. Gandhi, A. P. Boedihardjo, C. Chen, and S. Frankenstein, "Time series anomaly discovery with grammar-based compression," in *EDBT*, 2015.

[11] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang, "A data-adaptive and dynamic segmentation index for whole matching on time series," *PVLDB 6(10):793-804*, 2013.

[12] K. Zoumpatianos, S. Idreos, and T. Palpanas, "Indexing for interactive exploration of big data series," in *SIGMOD*, 2014.

[13] T. Palpanas, "Big sequence management: A glimpse of the past, the present, and the future," in *SOFSEM*, 2016.

[14] ——, "The parallel and distributed future of data series mining," in *High Performance Computing & Simulation (HPCS)*, 2017.

[15] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases," in *SIGMOD*, 1994.

[16] D. Rafiei and A. Mendelzon, "Efficient retrieval of similar time sequences using dft," in *ICDE*, 1998.

[17] E. J. Keogh, T. Palpanas, V. B. Zordan, D. Gunopulos, and M. Cardle, "Indexing large human-motion databases," in *VLDB*, 2004.

[18] I. Assent, R. Krieger, F. Afschari, and T. Seidl, "The ts-tree: Efficient time series search and retrieval," in *EDBT*, 2008.

[19] J. Shieh and E. J. Keogh, "*i*sax: indexing and mining terabyte sized time series," in *KDD*, 2008, pp. 623–631. [Online]. Available: http://doi.acm.org/10.1145/1401890.1401966

[20] S. Kadiyala and N. Shiri, "A compact multi-resolution index for variable length queries in time series databases," *KAIS*, 2008.

[21] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. J. Keogh, "Beyond one billion time series: indexing and mining very large time series collections with isax2+," *KAIS*, 2014.

[22] M. Dallachiesa, T. Palpanas, and I. F. Ilyas, "Top-k nearest neighbor search in uncertain data series," *PVLDB(8)1:13-24*, 2014.

[23] K. Zoumpatianos, S. Idreos, and T. Palpanas, "RINSE: interactive data series exploration with ADS+," *PVLDB (8)12:1912-1915*, 2015.

[24] ——, "ADS: the adaptive data series index," *VLDB J. 25(6): 843-866*, 2016.

[25] D. E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas, "Dpisax: Massively distributed partitioned isax," in *ICDM*, 2017, pp. 1135–1140.

[26] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas, "Coconut: A scalable bottom-up approach for building data series indexes," *PVLDB (11)6:677-690*, 2018.

[27] T. Kahveci and A. Singh, "Variable length queries for time series data," in *Proceedings 17th International Conference on Data Engineering*, 2001.

[28] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. A. P. A. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh, "Searching and mining trillions of time series subsequences under dynamic time warping," in *SIGKDD*, 2012.

[29] M. Linardi, Y. Zhu, T. Palpanas, and E. J. Keogh, "Matrix profile X: VALMOD - scalable discovery of variable-length motifs in data series," in *SIGMOD Conference 2018*.

[30] ——, "VALMOD: A suite for easy and exact detection of variable length motifs in data series," in *SIGMOD Conference 2018*.

[31] A. G. H. of Operational Intelligence Department Airbus., "Personal communication." 2017.

[32] "Automatic detection of cyclic alternating pattern (cap) sequences in sleep: preliminary results," *Clinical Neurophysiology*, 1999.

[33] E. J. Keogh and S. Kasetty, "On the need for time series data mining benchmarks: A survey and empirical demonstration," *Data Min. Knowl. Discov.*, 2003.

[34] A. Camerra, T. Palpanas, J. Shieh, and E. J. Keogh, "isax 2.0: Indexing and mining one billion time series," in *ICDM 2010*.

[35] M. Linardi and T. Palpanas, "ULISSE: ULtra compact Index for Variable-Length Similarity SEarch in Data Series," in *ICDE 2018*. [Online]. Available: http://www.mi.parisdescartes.fr/~themisp/publications/icde18-ulisse.pdf

[36] Y. Bu, T. wing Leung, A. W. chee Fu, E. Keogh, J. Pei, and S. Meshkin, "Wat: Finding top-k discords in time series database," in *SDM*, 2007, pp. 449–454.

[37] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra, "Dimensionality reduction for fast similarity search in large time series databases," *KAIS*, vol. 3, 2000.

[38] A. Mueen, H. Hamooni, and T. Estrada, "Time series join on subsequence correlation," in *ICDM 2014*, 2014.

[39] E. J. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping," *Knowl. Inf. Syst.*, 2005.

[40] J. Lin, E. Keogh, L. Wei, and S. Lonardi, "Experiencing sax: a novel symbolic representation of time series," *Data Mining and Knowledge Discovery*, 2007.

[41] K. Zoumpatianos, Y. Lou, T. Palpanas, and J. Gehrke, "Query workloads for data series indexes," in *SIGKDD*, 2015, pp. 1603–1612.

[42] www.mi.parisdescartes.fr/~mlinardi/ULISSE.html.

[43] W. e. a. G. C. P. L. H. F. M. J. T. S. Soldi, V. Beckmann, "Long-term variability of agn at hard x-rays," *Astronomy & Astrophysics*, 2014.

[44] IRIS. Seismic Data Access 2016. [Online]. Available: http://ds.iris.edu/data/access

[45] A. Bagnall, J. Lines, A. Bostrom, J. Large, and E. J. Keogh, "The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances," *Data Min. Knowl. Discov.*, 2017.

[46] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. J. Keogh, "Experimental comparison of representation methods and distance measures for time series data," *Data Min. Knowl. Discov.*, 2013.

[47] Y. Chen, E. Keogh, B. Hu, N. Begum, A. Bagnall, A. Mueen, and G. Batista, "The ucr time series classification archive," July 2015, www.cs.ucr.edu/~eamonn/time_series_data/.