UNIVERSITY OF CALIFORNIA
RIVERSIDE

The Matrix Profile: Scalable Algorithms and New Primitives for Time Series Data Mining

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Yan Zhu

September 2018

Dissertation Committee:
- Dr. Eamonn Keogh, Chairperson
- Dr. Christian Shelton
- Dr. Stefano Lonardi
- Dr. Evangelos Papalexakis

The Dissertation of Yan Zhu is approved:

Committee Chairperson

University of California, Riverside

ACKNOWLEDGEMENTS

My deepest gratitude goes to my advisor Dr. Eamonn Keogh, who has guided me through an amazing Ph.D. journey with immense support. Eamonn is the best Ph.D. advisor I could ever ask for: he is the most knowledgeable person that I know, and he is always there whenever I need him. When I get stuck and need inspirations, he is there; when I am stressed and need encouragement, he is there. He patiently taught me how to identify interesting and impactful research problems, guided me through numerous obstacles and most importantly, showed me what a great researcher should be like. Eamonn, I can never thank you enough. It is my greatest fortune to have you as my advisor.

I humbly thank my committee members Dr. Christian Shelton, Dr. Stefano Lonardi and Dr. Evangelos Papalexakis for their valuable suggestions throughout my Ph.D. study. I would also like to thank Dr. Abdullah Mueen for all the inspirations, encouragement and support; Dr. Daniel Nikovski, Dr. Ethan Jackson and Dr. Kang Li for their great mentorship during my summer internships at MERL, MSR and Google.

In the last five years, I have met a lot of fantastic people at UCR and received great friendship. I owe many thanks to my lovely colleagues from the data mining lab: Chin-Chia Michael Yeh, Hoang Anh Dau, Zachary Zimmerman, Nader Shakibay Senobari, Diego Silva, Shaghayegh Gharghabi, Shima Imani, Frank Madrid, Alireza Abdoli, Sara AlaeeJordehi, Kaveh Kamgar, Renjie Wu, Dr. Shailendra Singh, Dr. Liudmila Ulanova, Dr. Nurjahan Begum, Dr. Yanping Chen and Dr. Mohammad Shokoohi-Yekta. I would

also like to thank Jing Sun, Meng Zhao, Fei Ye, Xing Zheng and many other friends for accompanying me through this wonderful journey.

Last but not the least, I would like to thank my dad Gehai Zhu, my mum Sufen Tan, and my husband Zhuobo Feng for their unconditional love and continuous support. They are my biggest motivation to chase my dreams and strive for excellence.

To my dad Gehai, my mum Sufen and my husband Zhuobo

ABSTRACT OF THE DISSERTATION

The Matrix Profile: Scalable Algorithms and New Primitives for Time Series Data Mining

by

Yan Zhu

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2018
Dr. Eamonn Keogh, Chairperson

Primitives such as motifs, discords, shapelets, etc., are widely used in time series data mining. A versatile approach to find these primitives is through computing *similarity joins* for time series subsequences. The last decade has seen a significant amount of research effort on similarity joins in domains such as text and DNA, but not much progress has been made on similarity joins for time series subsequences. The lack of progress is probably due to the daunting nature of the problem: for even modest sized datasets the brute-force algorithm can take months to complete. Typical speed-up techniques such as indexing and lower-bounding at best produce only one or two orders of magnitude speedup, and their performance can degrade significantly in the face of an unfavorable dataset.

In this dissertation we introduce a suite of algorithms that significantly expand the limit of scalability for time series subsequences similarity joins. These algorithms not

only provide the fastest exact solution to the discovery of *time series motifs*, one of the most extensively-studied time series data mining primitives in the last decade, but also allow the invention of new primitives that the state-of-the-art could not support.

Specifically, we present a novel batch algorithm which, when combined with the GPU framework, can find the full set of exact motifs on a dataset two orders of magnitude larger than the literature limit in feasible time. A novel fast-converging anytime algorithm further expands this scalability, allowing motif discovery for million-scale datasets to be performed in *interactive* sessions with an off-the-shelf desktop. We also show how these techniques can be combined with a novel lower bound to allow fast motif discovery in the presence of missing data. Furthermore, we introduce *time series chains*, a new time series data mining primitive that can capture the evolution of systems and help predict the future.

We demonstrate the utility of our ideas in domains as diverse as seismology, entomology, human activity monitoring, electrical power-demand monitoring and medicine.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1 Introduction

The explosion of new sensing technology is continuously generating a massive amount of time series data in every aspect of our lives, from seismometers monitoring earthquake activities to smartwatches measuring our heartbeats. In order to turn such massive and diverse time series data into actionable insights, the research community has developed primitives such as motifs [15][49][37], discords [13], shapelets [95], etc., to extract important features/patterns from the data. These primitives can be used both as standalone exploratory tools, and as sub-routines in higher-level data mining tasks.

A basic requirement for time series data mining primitives is that they must be computed very fast. Recent algorithmic advances have largely improved the computational efficiency for each of the fore-mentioned primitives [49][37][32][50]. However, existing approaches suffer from a lack of generality, as they are highly optimized for individual primitives based on their unique characteristics. If an analyst would like to obtain a more comprehensive view of the data and compute several different primitives, she must run each of the specialized algorithms from scratch, and carefully tune their parameters based on the specific dataset she is trying to analyze.

Is there a way to unify *all* these useful primitives? The answer is affirmative: we can find them through computing the *similarity joins* for time series subsequences. The basic statement for the similarity join problem is this: *Given a collection of objects, retrieve the nearest neighbor for every object*. Time series subsequences similarity joins

encode all the information needed to answer both time series motif and time series discord queries, and can be used in other tasks such as shapelet discovery and semantic segmentation. However, the brute force solution to this problem is intractable for large datasets. For example, to obtain the similarity join for a dataset with 500,000 subsequence objects, the obvious nested loop algorithm requires 250,000,000,000 pairwise Euclidean distance computations. If each one took 0.0001 second, then the join would take 289 days, which is infeasible.

In this dissertation we introduce the Matrix Profile, a scalable, general and versatile time series data mining tool which solves the similarity join problem for time series subsequences. We present a suite of highly efficient algorithms to compute the Matrix Profile, and demonstrate that our algorithms not only outperform the state-of-the-art specialized methods in discovering *time series motifs*, arguably the most important time series data mining primitive, but also allows the invention of *time series chains*, a useful new primitive for time series data mining.

Specifically, our contributions can be divided into the following four aspects:

- We propose an efficient *batch* algorithm to compute the Matrix Profile. As we shall demonstrate, our algorithm incidentally provides the fastest *exact* solution for time series motif discovery. When combined with the GPU framework, our algorithm can find the full set of exact motifs in hundred-million-scale time series in feasible time. This expands the largest size considered in the literature by two orders of magnitude.

- We propose a fast-converging *anytime* algorithm to compute the Matrix Profile. For the first time, our algorithm allows the possibility of real-time *interactive* discovery of motifs in million-scale time series, using an off-the-shelf consumer desktop.
- Given the ubiquity of missing data in real world applications, we propose the first algorithm that can find motifs in the presence of missing data without producing false negatives.
- We introduce time series chains, a new time series data mining primitive built on top of the Matrix Profile. Time series chains can capture the evolution/drift of systems and help predict the future.

In the next two sections, we briefly review time series motifs and introduce time series chains.

1.1 Time Series Motifs

Time series motifs are approximately repeating subsequences found within a longer time series. Since their formulation in 2002 [56] they have emerged as one of the most important primitives in time series data mining. Each year there are at least a dozen new research efforts that exploit this primitive. Motif discovery has been used as a sub-routine in algorithms as diverse as classification, clustering, complex-event-processing [8], visualization [26], and rule-discovery [68]. Moreover, in recent years motif discovery has received significant attention *beyond* the data mining community, and has been applied to a wide variety of problems such as understanding the network of genes affecting the

locomotion of the *C. elegans* nematode [10], cataloging speech pathologies in humans [6], severe weather prediction [44], robotics, music processing, medicine [78] and seismology [101]. **Figure 1.1** shows an example of a repeating earthquake sequence pair (essentially a time series motif) we discovered from seismic data.

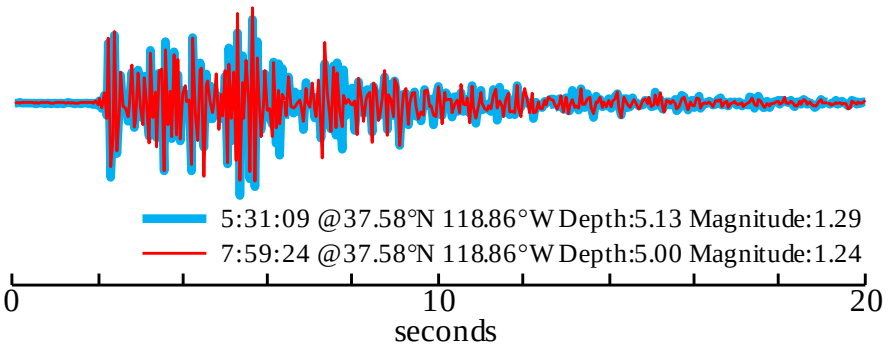

Figure 1.1. A pair of repeating earthquake sequences (motifs) we discovered from seismic data recorded at a station near Mammoth Lakes on February 17th, 2016. One occurrence (fine/red) is overlaid on top of another occurrence (bold/blue) that happened hours earlier.

With a little introspection, it is easy to see why time series motifs are so useful and widely used. If a pattern is repeated (or *conserved*), there must be a latent system that occasionally produces the conserved behavior. For example, this system may be an overcaffeinated heart, sporadically introducing a motif pattern containing an extra beat (Atrial Premature Contraction [40]), or the system may be an earthquake fault, infrequently producing highly repeated seismograph traces because the local geology produces unique wave reflection/refractions (see **Figure 1.1**/Chapter 3). Time series motifs are a commonly used technique to gain insight into such latent systems. In essence, they can be best seen as "*generalizing the notion of a regulatory motif to operate robustly on non-genomic data*" [78].

Although significant progress has been made in how we score, rank, and visualize motifs, *discovering* them in large datasets remains a computational bottleneck. To date, we are unaware of any attempts to mine any dataset larger than one million data points [37]. In Chapter 3, we show how we can significantly improve the scalability of exact motif discovery both by exploiting a novel batch algorithm and by leveraging GPU hardware.

We further argue that while all data mining algorithms benefit from improvements in speed, for the particular case of motif discovery, improvements in speed are game-changing. Motif discovery benefits from *interactivity* more than most data mining processes. In Chapter 4, we propose an anytime algorithm which, for the first time, allows motif discovery for million-scale time series to be performed in an *interactive* session (i.e. *real-time*) with an off-the-shelf desktop.

Moreover, despite the well-documented ubiquity of missing data in scientific, industrial, and medical datasets, there is still no technique to allow the discovery of motifs in the presence of missing data. We address this problem in Chapter 5. Our method is admissible, producing no false negatives.

1.2 Time Series Chains

We expand the notion of time series motifs to the new primitive of *time series chains* (or just *chains*). Time Series Chains are related to, but distinct from, time series motifs. Informally, time series chains are a temporally ordered set of subsequence patterns, such that each pattern is similar to the pattern that preceded it, but the first and

last patterns can be arbitrarily dissimilar. In the discrete space, this is similar to extracting the text chain "data, date, cate, cade, code" from text stream. The first and last words have nothing in common, yet they are connected by a chain of words with a small mutual difference. **Figure 1.2** symbolically illustrates the difference between time series motifs and time series chains (we defer formal definitions until Chapter 6).

Both motifs and chains have the property that each subsequence is relatively close to its nearest neighbor. However, the motif set also has a relatively small diameter (the maximum distance between any pair in the set). In contrast, the set of points in a chain has a diameter that is much larger than the mean of each member's distance to its nearest neighbor. Moreover, the chain has the property of *directionality*. For example, in **Figure 1.2**.*left*, if a tenth member was added to the motif set, its location will also be somewhere near the platonic ideal, but independent of the previous subsequences. In contrast, in **Figure 1.2**.*right*, the location of the tenth member of the chain would be somewhere just North-West of item nine. This potential regularity immediately suggests *exploitability*; we can use chains to (weakly) predict the future.

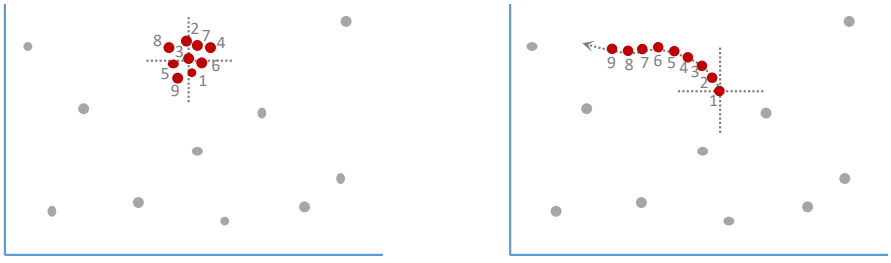

Figure 1.2. Visualizing time series subsequences as points in high-dimensional space. *left*) A time series motif can be seen as a collection of points that approximate a platonic ideal, represented here as the crosshairs. *right*) In contrast, a time series chain may be seen as an evolving trail of points in the space. Here the crosshairs represent the first link in the chain, the *anchor*.

While we can clearly define *chains*, it may not be obvious that such constructs actually exist in the real-world. In fact, as we preview in **Figure 1.3**, time series chains appear to be near ubiquitous in many domains, so long as the data trace is sufficiently long.

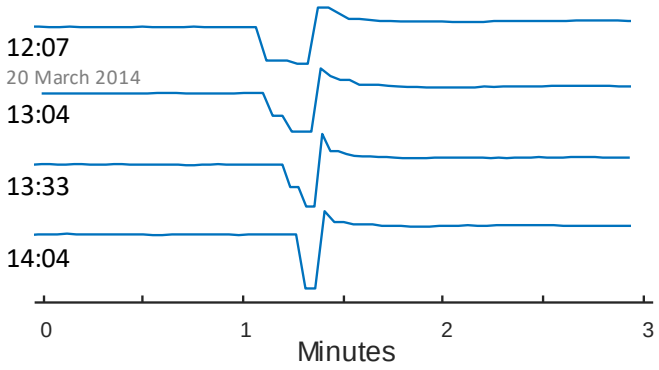

Figure 1.3. A time series chain discovered in an electrical power demand dataset monitoring domestic freezer usage [52]. Note that through the early afternoon, the valley becomes narrower and the peak that follows it becomes sharper.

In Chapter 6, we introduce two robust definitions of time series chains, and scalable algorithms that allow us to discover them in massive complex datasets.

1.3 Organization of the Dissertation

The rest of the dissertation is organized as follows. In Chapter 2 we introduce the Matrix Profile, a general and versatile time series data mining tool which has implications for a variety of time series data mining tasks. In Chapter 3 we present an ultra-fast algorithm to compute the Matrix Profile, and demonstrate its effectiveness and efficiency in exact motif discovery. In Chapter 4 we describe a fast-converging anytime algorithm which further expands the scalability limits of the Matrix Profile and allows motifs for million-scale time series to be discovered at interactive speeds with a standard desktop. In

Chapter 5 we consider motif discovery in the presence of missing data. Chapter 6 introduces time series chains, a new time series data mining primitive built on top of the Matrix Profile. Chapter 7 concludes the dissertation and discusses future directions.

Chapter 2 The Matrix Profile: Basic Concepts and Applications

In this chapter, we introduce the Matrix Profile, a general and versatile time series data mining tool that solves the similarity join problem for time series subsequences. We formally define the Matrix Profile, and briefly show its implication for various time series data mining tasks.

2.1 Notation and Definitions

We begin by defining the data type of interest, *time series*:

Definition 2.1: A *time series* T is a sequence of real-valued numbers t_i: $T = t_1, t_2, ..., t_n$, where n is the length of T.

We are interested in *local*, not *global* properties of a time series. A local region of a time series is called a *subsequence*.

Definition 2.2: A *subsequence* $T_{i,m}$ of a time series T is a continuous subset of the values from T of length m, which begin at position i. Formally, $T_{i,m} = t_i, t_{i+1}, \ldots, t_{i+m-1}$, where $1 \leq i \leq n-m+1$.

We can take a subsequence and compute its distance to *all* subsequences in the same time series. This is called a *distance profile*.

Definition 2.3: A *distance profile* D_i of time series T is a vector of the Euclidean distances between a given query subsequence $T_{i,m}$ and each subsequence in time series T. Formally, $D_i = [d_{i,1}, d_{i,2}, \ldots, d_{i,n-m+1}]$, where $d_{i,j}$ ($1 \leq i, j \leq n-m+1$) is the distance between $T_{i,m}$ and $T_{j,m}$.

We assume that the distance is measured by Euclidean distance between z-normalized subsequences [89].

We are interested in finding the nearest neighbors of all subsequences in T, as the closest pairs of this are the classic definition of time series motifs [15][49]. Note that by definition, the i^{th} location of distance profile D_i is zero, and it is close to zero just before and after this location. Such matches are defined as *trivial matches* [49]. We avoid such matches by ignoring an "exclusion zone" of length $m/4$ before and after the location of

the query. In practice, we simply set $d_{i,j}$ to infinity (i-$m/4 \leq j \leq i$+$m/4$) while evaluating D_i.

We use a vector called *matrix profile* to represent the distances between all subsequences and their nearest neighbors.

Definition 2.4: A *matrix profile* P of time series T is a vector of the Euclidean distances between each subsequence $T_{i,m}$ and its nearest neighbor (i.e. the closest match) in time series T. Formally, P = [min(D_1), min(D_2), …, min(D_{n-m+1})], where D_i ($1 \leq i \leq n$-m+1) is the distance profile D_i of time series T.

We call this vector a matrix profile, since it could be computed by using the full distance matrix of all pairs of subsequences in time series T, and evaluating the minimum value of each column (although this method is naïve and space-inefficient). **Figure 2.1** illustrates both a *distance profile* and a *matrix profile* created on the same raw time series T.

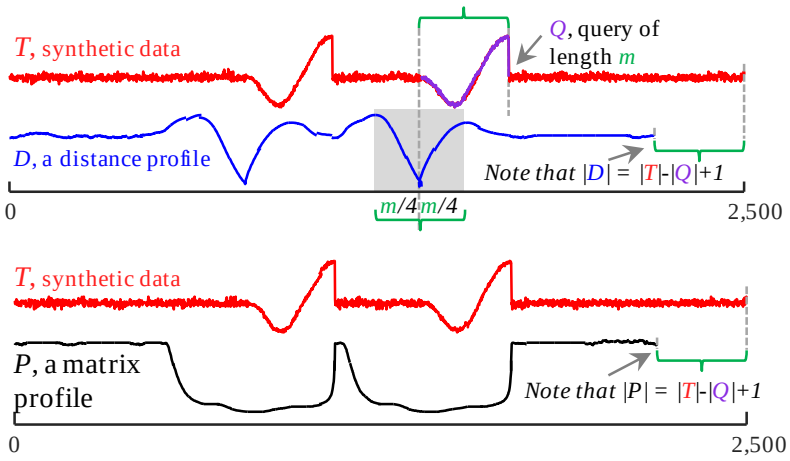

Figure 2.1. *top*) One distance profile (Definition 2.3) created from a random query subsequence *Q* of *T*. If we created distance profiles for all possible query subsequences of *T*, the element-wise minimum of this set would be the matrix profile (Definition 2.4) shown at (*bottom*). Note that the two lowest values in *P* are at the location of the 1st motif [15][49].

It is important to note that the full distance matrix is symmetric: D_i is both the i^{th} row and the i^{th} column of the full distance matrix. **Figure 2.2** shows this more concretely.

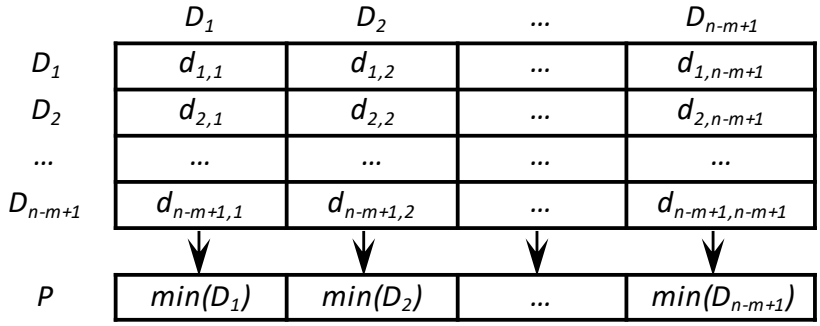

Figure 2.2. An illustration of the relationship between the distance profile, the matrix profile and the full distance matrix. For clarity, note that we do *not* actually create the full distance matrix, as this would have untenable memory requirements.

The i^{th} element in the matrix profile P indicates the Euclidean distance from subsequence $T_{i,m}$ to its nearest neighbor in time series T. However, it does not indicate the location of that nearest neighbor. This information is recorded in a companion data structure called the *matrix profile index*.

Definition 2.5: A *matrix profile index I* of time series T is a vector of integers: $I=[I_1, I_2, \ldots I_{n-m+1}]$, where $I_i=j$ if $d_{i,j} = \min(D_i)$.

By storing the neighboring information in this manner, we can efficiently retrieve the nearest neighbor of query $T_{i,m}$ by accessing the i^{th} element in the matrix profile index.

Note that as we presented it, the matrix profile is a *self-join* [96]: for every subsequence in a time series T, it records information about its (non-trivial-match) nearest neighbor in the same time series T. However, we can trivially generalize it to be an AB-join [96]: given two different time series A and B, for every subsequence in time

series *A*, record information about its nearest neighbor in time series *B*. Note that *A* and *B* can be of different lengths, and that in general, AB-join ≠ BA-join.

For clarity of presentation, we have confined this work to the single dimensional case; however, nothing about our work intrinsically precludes generalizations to multidimensional data.

To briefly summarize this section, we can create two Meta time series, the *matrix profile* and the *matrix profile index,* to annotate a time series *T* with the distance and location of all its subsequences' nearest neighbors in itself (in the self-join case), or in another time series (in the AB-join case). The reader may already have realized, the smallest pair of values in the self-join *matrix profile* correspond to the best motif pair by the classical definition [15][37][49], and the corresponding values in the *matrix profile index* indicate the location of the motif. As Mueen et al. [49] argues, the *top-k* motifs, *range* motifs, and any other reasonable variant of motifs can also trivially be computed from the information encoded in the matrix profile. Moreover, the matrix profile also yields the exact solution for *time series discords*, a popular definition for *anomalies* in time series [13]. In the next section, we use a concrete example to show how we can discover motifs and discords with the matrix profile.

2.2 Discovering Time Series Motifs and Discords with the Matrix Profile

Unlike other motif/discord discovery systems, the matrix profile computes a score for *every* subsequence in the dataset. Here, we use an example to demonstrate the utility of this more comprehensive annotation of data. Consider the New York Taxi dataset of

Rong and Bailis [63]. As shown in **Figure 2.3**.*top*, the data is the normalized number of NYC taxi passengers over 10 weeks, October 1st to December 15th 2014. The authors show this dataset to demonstrate the versatility of their "Attention Prioritization" technique for finding unusual patterns [4][63]. In essence, they transform the data (not shown here) in a way to make the discovery of anomalies easier. They note that Thanksgiving, on Thursday, November 27th, can be considered an "anomaly" in this dataset, since the patterns of travel apparently change during this important US holiday.

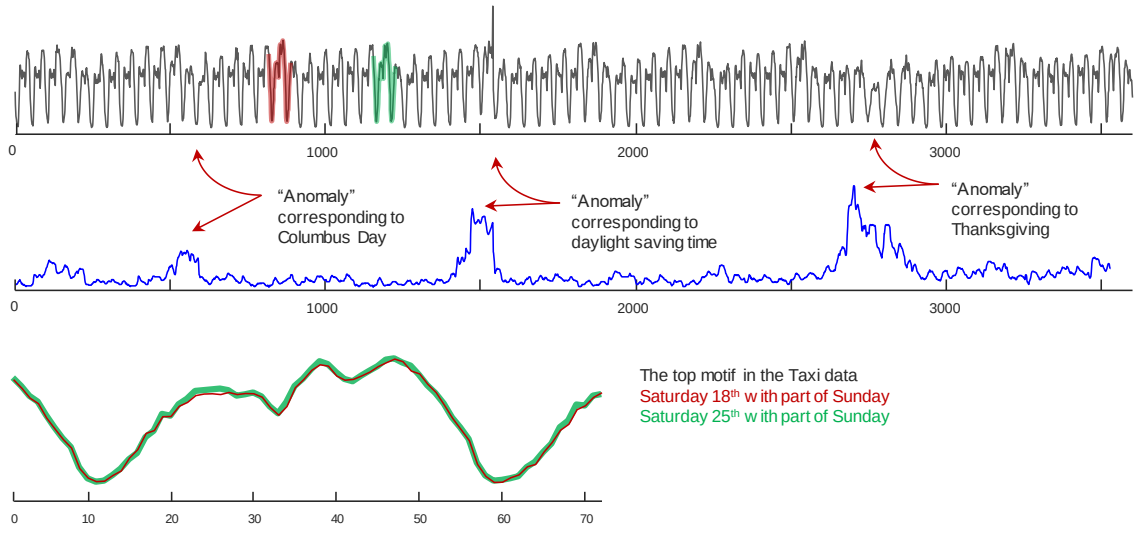

Figure 2.3. *top*) Normalized number of NYC taxi passengers over 10 weeks [4][63]. *middle*) The matrix profile produces high values where the corresponding subsequences are unusual. *bottom*) The top motif corresponds to two consecutive Saturdays.

We computed the matrix profile for this dataset, with a subsequence length of one and a half days. As **Figure 2.3**.*middle* shows, the matrix profile peaks at the location that indicates Thanksgiving. However, there are additional observations that we can make with the matrix profile. There is a secondary anomaly occurring on Sunday, November 2nd; there appears to be a spike in taxi demand at about 2:00 am. With a little thought, we

realize this is exactly the hour in which daylight saving time is observed in the US. Setting the clock back one hour gives the appearance of doubling the normal demand for taxis at that hour. There is arguably a third anomaly in the dataset, with a more subtle, but still significant peak at October 13th. This day corresponds to Columbus Day. This holiday is all but ignored in most of the US, but it is still observed in New York, which has a strong and patriotic Italian community.

In **Figure 2.3**.*bottom*, we show the top-1 *motif* from the dataset (corresponding to the minimum value of the matrix profile), which is extremely well conserved. In many natural datasets, for example the circadian rhythm of an animal, the best motifs are typically exactly twenty-four hours apart (a phenomenon known as *persistence*). However, because this motif's two occurrences are exactly *seven* days apart, the importance of artificial divisions of the calendar on human behaviors becomes apparent. It is possible that the regions of lower conservation with the motif are also telling. For example, from 24 to 26 (about 10 to 11am), the motif corresponding to the 25th (green/bold) is a little higher than the previous week. It was lightly raining (about 0.12 inches) at the time, which may explain the slightly higher taxi demand in the late morning.

Besides motif discovery and discord discovery, the matrix profile also has implications for a host of other time series data mining tasks. In the next section, we show how we can use the matrix profile to discover time series shapelets, another useful time series data mining primitive.

2.3 Discovering Time Series Shapelets with the Matrix Profile

Time series shapelets are time series subsequences that best represent a class [95]. Here we use the GunPoint dataset [14] to show how we can use the matrix profile to quickly identify good shapelet candidates. This dataset has two classes, Gun and NoGun (also known as Point, hence the name GunPoint). As shown in **Figure 2.4**, we construct time series *A* by concatenating all the instances of the Gun class, and time series *B* by concatenating all the instances of the NoGun class. Between every two concatenated exemplars, we insert a *NaN* value to avoid introducing artificial subsequences that are not present in the original data. We compute two matrix profiles P_{BB} and P_{BA}: P_{BB} is the self-join matrix profile for time series *B*; P_{BA} is the BA-join matrix profile for time series *B* and time series *A*. For simplicity, we use a subsequence length of 38, which is the length of the best shapelet reported for this dataset [95].

We evaluate the difference between P_{BA} and P_{BB} (we denote it as $P_{diff} = P_{BA} - P_{BB}$), as shown in **Figure 2.4**.*bottom.left*. Intuitively, the peak values in P_{diff} are indicators of good shapelet candidates, because they suggest patterns that are well conserved in the NoGun class but are very different from their closest match in the Gun class. We pick the top-10 candidates from time series *B* (corresponding to the top-10 peaks in **Figure 2.4**.*bottom*), and among them select the one that renders the highest classification accuracy on the training data. As shown in **Figure 2.4**.*bottom.left,* the selected shapelet reflects a distinct characteristic of the NoGun class, as discussed by Ye and Keogh [95]: "*the NoGun class* "*has a "dip" where the actor put her hand down by her side, and*

inertia carries her hand a little too far and she is forced a correct it…a phenomenon known as 'overshoot'". In contrast, in the opposite Gun class, the actor carries a gun; she needs to put the gun back in the holster and then bring her hand to a complete rest position, generating a different pattern.

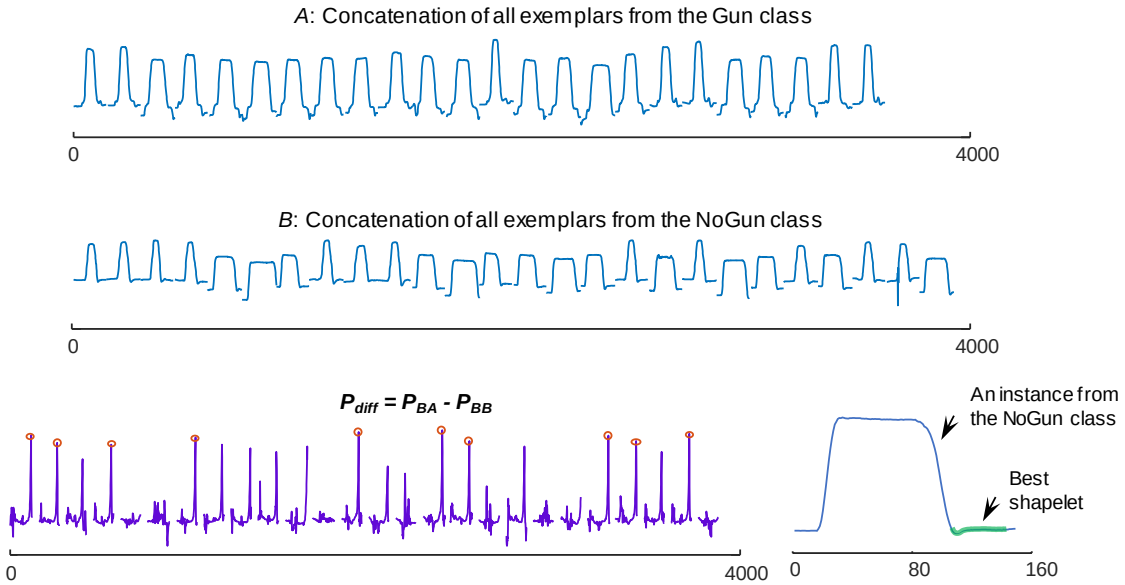

Figure 2.4. *top and middle*): Two time series *A* and *B* formed by concatenating instances of each class of GunPoint dataset. *bottom.left*) The difference between P_{BA} and P_{BB}. The top-10 peak values (highlighted with red circles) are suggestive of good shapelet candidates. *bottom.right*) The best shapelet found.

The selected shapelet achieves the same classification accuracy (93.33%) on the test data as the original shapelet algorithm [95]. However, note that while the original shapelet algorithm needs to go through a time-consuming process to exhaustively evaluate the classification power of every possible shapelet candidate in the dataset, the matrix profile readily finds us the most promising shapelet candidates for free.

2.4 Conclusions

We introduced two meta time series, the *matrix profile* and the *matrix profile index*, to annotate a time series *T* with the distance and location of all its subsequences' nearest neighbors within *T*. We briefly showed the implication of the matrix profile on various time series data mining tasks without explaining how we *computed* it. In the next four chapters, we introduce a suite of highly scalable algorithms to compute the matrix profile, and discuss in detail the effectiveness and efficiency of our ideas in discovering two useful time series data mining primitives: time series motifs and time series chains.

Chapter 3 Computing the Exact Matrix Profile with STOMP

In this chapter, we introduce STOMP, an ultra-fast exact algorithm to compute the Matrix Profile. We demonstrate the scalability and effectiveness of STOMP in discovering time series motifs, one of the most studied primitives in time series data mining. When combined with a high-performance GPU, STOMP can find the full set of exact motifs on a dataset with one hundred and forty-three million subsequences, which requires ten quadrillion pairwise comparisons and is by far the largest dataset ever mined for time series motifs/joins, in feasible time.

Though this chapter only considers motif discovery, as introduced in Chapter 2, the Matrix Profile is a versatile tool that can also be applied to other time series data mining tasks such as discord discovery and shapelet discovery. As a fundamental algorithm to compute the Matrix Profile, STOMP can also benefit these tasks.

3.1 Background and related work

3.1.1 Motif Discovery Background

Motif discovery for time series was introduced in 2003 [15] (although the classic paper of Agrawal, Faloutsos and Swami foreshadows motifs by computing all-pair similarity for time series [2]). Since then, it increased in research activity. One critical direction has been applying motifs to solve problems in a wide variety of domains such as bioinformatics [10], speech processing [6], robotics, human activity understanding [80][85], severe weather prediction [44], neurology, and entomology [49]. The other key research focus has been in the extensions and generalizations of the original work, especially in the attempts to improve scalability [37][49]. These attempts to improve the scalability of motif discovery fall into two broad classes; *approximate* and *exact* motif discovery [37][47][49].

Clearly approximate motifs *can* be much faster to compute, and this may be useful in some domains. However, there are domains in which the risk of false negatives is unacceptable. Consider seismology, which is the domain motivating our work [101]. This is a domain in which false negatives could affect public policy, change insurance rates for customers, and conceivably cost lives by allowing a dangerous site to be developed for dwellings. Given that the task at hand is to find *exact* motifs, all known methods based on *hashing* [101] and/or *data discretization* [15][47] can be dismissed from consideration.

Beyond being exact, the proposed approach has many advantages that are not shared by rival methods.

- The proposed method is **simple** and **parameter-free**: In contrast, other methods typically require building and tuning spatial access methods and/or hash functions [15][37][41][47][80][85][101].
- It is **space efficient**: Our algorithm requires an inconsequential space overhead, just linear in the time series length, with a small constant factor. In particular, we avoid the need to actually explicitly extract the individual subsequences [15][47][49], something that would increase the space complexity by two or three orders of magnitude.
- It is **incrementally maintainable**: Having computed motifs for a dataset, we can incrementally update the best motifs very efficiently if new data arrives.
- It can leverage hardware: As we show below, our algorithm is embarrassingly **parallelizable** on multicore processors.
- Our algorithm has **time complexity that is *constant* in subsequence length**: This is a very unusual and desirable property; virtually all time series algorithms scale poorly as the subsequence length grows (the classic *curse of dimensionality*) [15][37][41][47][80][101].
- Our algorithm takes **deterministic time**, dependent on the data's length, but completely independent of the data's structure / noise level etc. This is also an unusual and desirable property for an algorithm in this domain. For example, even for a fixed time series length, and a fixed subsequence length, all other algorithms we are aware of can take radically different times to finish on two (even slightly) different datasets [15][37][41][47][80][101]. In contrast, given

only the length of the time series, we can predict precisely how long it will take our algorithm to finish in advance.

Virtually every time series data mining technique has been applied to the motif discovery problem, including indexing [37][88], data discretization [15], triangular-inequality pruning [49], hashing [80][85][101], early abandoning, etc. However, all these techniques rely on the assumption that the *intrinsic* dimensionality of the time series is much lower than the *recorded* dimensionality [15][80][85][89][101]. This is generally true for data such as short snippets of heartbeats and gestures, etc.; however, it is not true for seismographic data, which is intrinsically high dimensional. To ascertain this, we performed a simple experiment.

We measured the *Tightness of Lower Bounds* (TLB) for three types of data, using the two most commonly used dimensionality reduction representations for time series, DFT and PAA. Additionally, PAA is essentially equivalent to the Haar wavelets for this purpose [89]. The TLB is defined as:

$$TLB = LowerBoundDist(A,B) / TrueEuclideanDist(A,B)$$

It is well understood that the TLB is near perfectly (inversely) correlated with wall-clock time, CPU operations, number of disk access or any other performance metric for similarity search, all-pair-joins, motifs discovery, etc. [89]. As the mean TLB decreases, we quickly degrade to simple brute-force search. The absolute minimum value of TLB is dependent on the data, the search algorithm, and the problem setting (main-memory based vs disk based). However as [89] demonstrates, lower bound values less than 0.5 generally do not "break even."

Figure 3.1 shows unambiguous results. There is *some* hope that we could avail current speed-up techniques when considering (relatively smooth and simple) short snippets of ECGs, but there is *little* hope that the noisy and more complex human activity would yield to such optimizations, and there is *no* hope that anything currently in the literature will help with seismological data. This claim is further proven in our detailed experiments in Section 3.3.

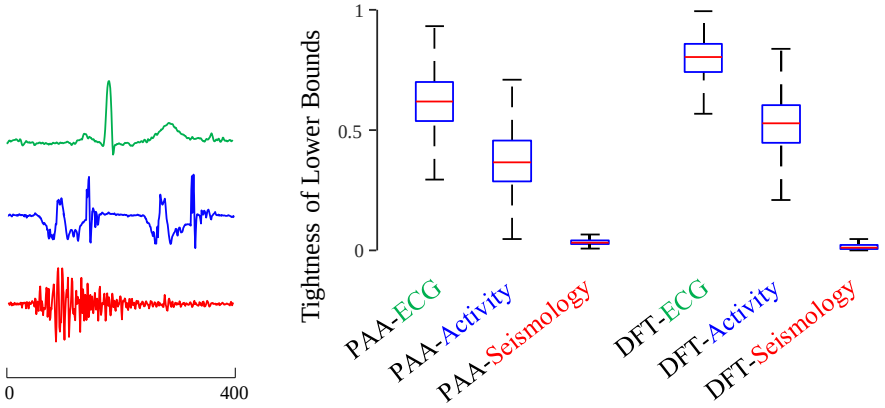

Figure 3.1. *left*) Samples from three datasets, ECG, Human Activity, and Seismology (available in [59]). *right*) The tightness of lower bounds, averaged over 10,000 random pairs, using PAA and DFT.

Even if we ignore this apparent death-knell for indexing/spatial access techniques, we could still dismiss them for other reasons, including memory considerations. As demonstrated in Section 3.2, a critical property of our algorithm is that it does not need to explicitly *extract* the subsequences, which is unlike the indexing/spatial access methods. For example, consider a time series of length 100 million, with eight bytes per value, requiring 0.8 GB. Our algorithm requires an overhead of seven other vectors of the same size (including the output), for an easily manageable total of 6.4 GB (if memory *was* a bottleneck, we could reduce this by using reduced precision vectors or compression). However, any indexing algorithm that needs to extract the subsequences will increase

memory requirements by *at least* O(*d*), where *d* is the reduced dimensionality used in the index [41]. Given that *d* may be 20 or greater, this indicates the memory requirements grow to at least 16 GB. With such a large memory footprint, we are almost certainly condemned to a random-access disk-based algorithm, dashing any hope of any speedup.

A related advantage of our framework is that we can choose the subsequence length *just* prior to performing the motif discovery. In contrast, any index-based technique must commit to a subsequence length before *building* the index, and it could take hours/days to build the data structure before any actual searching could begin [61][89]. If such an index is built to support subsequences of say length 200, it cannot be used to join subsequences of length 190 or 205, etc. (See Section 1.2.3 of Rakthanmanon et al. [61]). Thus, if we change our mind about the length of patterns we are interested in, we are condemned to a costly rebuilding of the entire index. It is difficult to overstate the utility of this feature. In Section 3.3.8, we will demonstrate how we can use STOMP to explore the behavior of a penguin. At the beginning of this case study, we had no idea of what time frame the penguin's behavior might be manifest. However, with no costly index to build, we simply tried a few possible lengths until it was obvious that we found a reasonable value.

In summary, while we obviously are unable to absolutely guarantee that there is no other scalable solution to our task-at-hand, we are confident that there is no existing off-the-shelf technology that can be used or adapted to allow us to get within two orders of magnitude of the results we obtain on the largest datasets.

3.1.2 Seismology Background

While our algorithms are completely general and can be applied to any domain, seismological data is of particular interest to us, due to its sheer scale and importance in human affairs.

In the early 1980s, it was discovered that in the telemetry of seismic data recorded by the same instrument from sources in given region, there will be many similar seismograms [22]. Geller and Mueller [22] have suggested that, "*The physical basis of this clustering is that the earthquakes represent repeated stress release at the same asperity, or stress concentration, along the fault surface.*" These patterns are called "repeating earthquake sequences" in seismology, and correspond to the more general term "*time series motifs*."

A more recent paper notes that many fundamental problems in seismology can be solved by joining seismometer telemetry in locating these repeating earthquake sequences [101], which includes the discovery of foreshocks, aftershocks, triggered earthquakes, swarms, volcanic activity, and induced seismicity. However, the paper further notes that an *exact* join with a query length of 200 on a data stream of length 604,781 requires 9.5 days. Their solution, a transformation of the data to allow LSH based techniques, does achieve significant speedup, but at the cost of false negatives and necessary careful parameter tuning. For example, Yeh et al. [96] notes that they need to set the threshold to precisely 0.818 to achieve decent results. While we defer a full discussion of experimental results to Section 3.3, the ideas introduced in this paper can reduce the quoted 9.5 days for exact motif discovery from a dataset of size 604,781 to less than one

minute, without tuning any parameters and also guaranteeing that false negatives will not occur.

It is vital to note that this kind of speed up really is game changing in this domain. It allows seismologists to quickly identify or detect earthquakes that are identical or similar in location without needing trilateration, and it can also provide useful information on relative timing and location of such events [3][33][34].

More controversially, some researchers have suggested that the slow slip on the fault accompanying non-volcanic tremors (a sequence of Low Frequency Earthquakes, many of which are repeated) may temporarily increase the probability of triggering a large earthquake. Therefore, detecting and locating these repeating LFEs allows more accurate short-term earthquake forecasting [33].

Finally, we note that seismologists have been early adopters of GPU technology [46] and other high-performance computing paradigms. However, their use of this technology has been limited to similarity search, not motif search.

3.1.3 A Brief Review of the STAMP Algorithm

The recently introduced STAMP algorithm can compute the *full* and *exact* matrix profile and matrix profile index of a time series [96]. The STAMP algorithm essentially evaluates the distance profile D_i of a query subsequence $T_{i,m}$ by utilizing the FFT(Fast Fourier Transform) to calculate the dot product between $T_{i,m}$ and all of the subsequences of the time series T. The overall time complexity of the algorithm is $O(n^2 \log n)$, and the space complexity is $O(n)$, where n is the length of time series T. The STAMP algorithm can process a time series with up to a million data points in feasible time. However, to

solve the problems in our motivating domain seismology, it is necessary to process even larger datasets. It would take STAMP more than 20 years to analyze a seismology time series sampled at 20Hz for about 2 months, which is of length 100 million (see **Table 3.2**). In the next section, we will show a new and fast algorithm, which can finish processing the same time series in only 4 days when it is built on top of a GPU.

3.2 Algorithms

In this section, we begin by demonstrating that we can improve upon the STAMP algorithm [96] to create the much faster STOMP algorithm. Then we demonstrate that the structure of STOMP lends itself to porting to GPUs.

3.2.1 The STOMP Algorithm

As explained below, STOMP is similar to STAMP [96] in that it can be viewed as highly optimized nested loop searches with repeating calculations of distance profiles in the inner loop. However, while STAMP must evaluate the distance profiles in a random order (to allow its anytime behavior), STOMP performs an *ordered* search. By exploiting the locality of these searches, we can reduce the time complexity by a factor of O(log*n*).

Before we explain the details of the algorithm, we first introduce a formula to calculate the z-normalized Euclidean distance $d_{i,j}$ of two time series subsequences $T_{i,\mathrm{m}}$ and $T_{j,m}$ by using their dot product, $Q_{i,j}$:

$$d_{i,j} = \sqrt{2m\left(1 - \frac{Q_{i,j} - m\mu_i\mu_j}{m\sigma_i\sigma_j}\right)} \tag{3.1}$$

Here m is the subsequence length, μ_i is the mean of $T_{i,m}$, μ_j is the mean of $T_{j,\mathrm{m}}$, σ_i is the standard deviation of $T_{i,m}$, and σ_j is the standard deviation of $T_{j,\mathrm{m}}$.

The technique introduced in Rakthanmanon et al. [61] allows us to obtain the means and standard deviations with O(*1*) time complexity; thus, the time required to compute $d_{i,j}$ depends only on the time required to compute $Q_{i,j}$. Here, we claim that $Q_{i,j}$ can also be computed in O(*1*) time, once $Q_{i-1,j-1}$ is known.

Note that $Q_{i-1,j-1}$ can be decomposed as the following:

$$Q_{i-1,j-1} = \sum_{k=0}^{m-1} t_{i-1+k} t_{j-1+k} \tag{3.2}$$

And $Q_{i,j}$ can be decomposed as the following:

$$Q_{i,j} = \sum_{k=0}^{m-1} t_{i+k} t_{j+k} \tag{3.3}$$

Thus we have:

$$Q_{i,j} = Q_{i-1,j-1} - t_{i-1} t_{j-1} + t_{i+m-1} t_{j+m-1} \tag{3.4}$$

Therefore, our claim is proved.

Figure 3.2 visualizes the algorithm. Based on (3.1), we can map the distance matrix in **Figure 2.2** (also shown in **Figure 3.2**.*left*) to its corresponding dot product matrix (shown in **Figure 3.2**.*right*).

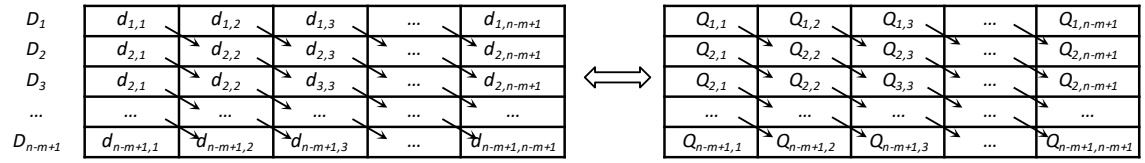

Figure 3.2. Mapping the computation of the distance matrix (*left*) to the computation of its correponding dot product matrix (*right*).

The arrows in **Figure 3.2**.*right* show the data dependency indicated by (3.4): once we have $Q_{i-1, j-1}$, we can compute $Q_{i,j}$ in O(*1*) time. However, note that (3.4) does not apply to the special case when $i=1$ or $j=1$ (the first row and the first column of **Figure 3.2**.*right*, shown in red). This problem is easy to solve: we can pre-compute the dot product values in these two special cases with FFT, as shown in **Algorithm 1**. Concretely, we use SlidingDotProduct($T_{1,\mathrm{m}}$, *T)* to calculate the first dot product vector: $Q_1 = [Q_{1,1}, Q_{1,2}, \ldots, Q_{1,n-m+1}] = [Q_{1,1}, Q_{2,1}, \ldots, Q_{n-m+1,1}]$. The dot product vector is stored in memory and used as needed.

Algorithm 1: SlidingDotProduct(*S*, *T*)

Input: A query subsequence ***S***, and a user provided time series ***T***
Output: The dot product between ***S*** and all subsequences in ***T***

```
1  n ← Length(T), m ← Length(S)
2  T_a ← Append T with n zeros
3  S_r ← Reverse(S)
4  S_ra ← Append S_r with 2n-m zeros
5  S_raf ← FFT(S_ra), T_af ← FFT(T_a)
6  Q ← InverseFFT(ElementwiseMultiplication(S_raf, T_af))
7  return Q_{m:n}
```

After the first row and the first column in **Figure 3.2**.*right* are computed, the rest of the dot product matrix is evaluated row after row.

We are now in the position to introduce our STOMP algorithm (**Algorithm 2**).

The algorithm begins in line 1 by computing the matrix profile length *l*. In line 2, the mean and standard deviation of every subsequence in *T* are pre-calculated. Line 3 calculates the first dot product vector *Q* with the algorithm in **Algorithm 1**. In line 5, we initialize the matrix profile *P* and matrix profile index *I*. The loop in lines 6-13 calculates the distance profile of every subsequence of *T* in sequential order. Lines 7-9 update *Q* according to (3.4). We update Q_1 in line 10 with the pre-computed Q'_i in line 3. Line 11

calculates distance profile D according to (3.1). Finally, line 12 compares every element of P with D: if $D_j < P_j$, then $P_j = D_j$, $I_j = i$.

Algorithm 2: STOMP(T, m)

Input: A time series $\boldsymbol{T}$ and a subsequence length $\boldsymbol{m}$
Output: Matrix profile $\boldsymbol{P}$ and the associated matrix profile index $\boldsymbol{I}$ of T

```
1  n ← Length(T), l ← n – m + 1
2  μ, σ ← ComputeMeanStd(T, m)                          // see [61]
3  Q ← SlidingDotProduct(T_{1,m}, T), Q' ← Q
4  D ← CalculateDistanceProfile(Q, μ, σ, 1)             // see (3.1)
5  P← D, I← ones                                        // initialization
6  for i = 2 to l                                       // in-order evaluation
7      for j= l downto 2                                // update dot product, see (3.4)
8          Q_j ← Q_{j-1} -T_{j-1}×T_{i-1}+T_{j+m-1}×T_{i+m-1}
9      end for
10     Q_1 ← Q'_i
11     D ← CalculateDistanceProfile(Q, μ, σ, i)         // see (3.1)
12     P, I ← ElementWiseMin(P, I, D, i)
13 end for
14 return P, I
```

The time complexity of STOMP is $O(n^2)$; thus, we have achieved a $O(\log n)$ factor speedup over STAMP [96]. Moreover, it is clear that $O(n^2)$ is optimal for any exact motif algorithm in the general case. The $O(\log n)$ speedup makes little difference for small datasets and for those with just a few tens of thousands of data points [15]. However, as we consider the datasets with millions of data points, this $O(\log n)$ factor begins to produce a very useful order-of-magnitude speedup.

To better understand the efficiency of STOMP, it is important to clarify that the time complexity of the classic brute force algorithm is $O(n^2 m)$. The value of m depends on the domain, but in Section 3.3.8, we consider real world applications where it is 2,000. Most techniques in the literature gain speedup by slightly reducing the n^2 factor; however, we gain speedup by reducing the m factor to $O(1)$. Moreover, it is important to

remember that the techniques in the literature can only reduce this n^2 factor if the data has a low intrinsic dimensionality (recall **Figure 3.1**), *and* the domain requires a short subsequence length. In contrast, the speedup for STOMP is completely independent of *both* the structure of the data and the subsequence length.

3.2.2 Incrementally Maintaining the Matrix Profile with STOMP*I*

Up to this point we have discussed the *batch version* of STOMP. By *batch*, we mean that the STOMP algorithm needs to see the *entire* time series T before creating the matrix profile. However, in many situations it would be advantageous to build the matrix profile incrementally. Given that we have performed a batch construction of matrix profile, when a new data point arrives, it would clearly be preferable to incrementally *adjust* the current profile, rather than starting from scratch.

Because the matrix profile solves both the times series motif and the time series discord problems, an incremental version of STOMP would automatically provide the first incremental versions of both these algorithms. In this section, we demonstrate that we *can* create such an incremental algorithm.

We name the incremental algorithm STOMP*I* (STOMP *I*ncremental, shown in **Algorithm 3**).

As a new data point t arrives, the size of the original time series T increases by one. We denote the new time series as T^{new}, and we need to update the matrix profile P^{new} and its associated matrix profile index I^{new} corresponding to T^{new}. For clarity, note that the input variables Q, μ and σ are all vectors, where Q_i is the dot product of the i^{th} and the last

subsequences of T; μ_i and σ_i are, respectively, the mean and standard deviation of the i^{th} subsequence of T.

Algorithm 3: STOMPI(T, t, m, P, I, μ, σ)

Input: The original time series $\boldsymbol{T}$, a new data point $\boldsymbol{t}$ following T, subsequence length $\boldsymbol{m}$, the matrix profile $\boldsymbol{P}$ and its associated matrix profile index $\boldsymbol{I}$ of T, dot product vector $\boldsymbol{Q}$, mean vector $\boldsymbol{\mu}$ and standard deviation vector $\boldsymbol{\sigma}$
Output: The updated matrix profile $\boldsymbol{P^{new}}$ and its matrix profile index $\boldsymbol{I^{new}}$ corresponding to the new time series $\boldsymbol{T^{new}}$= [T, t], the updated dot product vector $\boldsymbol{Q^{new}}$, updated mean vector $\boldsymbol{\mu^{new}}$ and standard deviation vector $\boldsymbol{\sigma^{new}}$

1 $n \leftarrow$ Length(T), $l \leftarrow n$-m+1, T^{new} = [T, t], $S \leftarrow T^{new}_{l+1:n+1}$
2 $t_{drop} \leftarrow T_l$ *// t_{drop} is the first item of the last subsequence of T*
3 **for** j= l+1 **downto** 2 *// update dot products with* (3.4)
4 $Q_j^{new} \leftarrow Q_{j-1} - T_{j-1}^{new} \times t_{drop} + T_{j+m-1}^{new} \times t$
5 **end for**
6 $Q_1^{new} \leftarrow 0$
7 **for** j= 1 **to** m *// calculate the first dot product with simple brute-force*
8 $Q_1^{new} \leftarrow Q_1^{new} + T_j^{new} \times S_j$
9 **end for**
10 $\mu_S \leftarrow \mu_l + (t - t_{drop}) / m$ *// update mean of S*
11 $\sigma_S \leftarrow \sigma_l^2 + \mu_l^2 + (t^2 - t_{drop}^2) / m - \mu_S^2$ *// update standard deviation of S*
12 $\mu^{new} \leftarrow [\mu, \mu_S]$, $\sigma^{new} \leftarrow [\sigma, \sigma_S]$
13 $D \leftarrow$ CalculateDistanceProfile(Q^{new}, μ^{mew}, σ^{new}, l+1) *// see* (3.1)
14 $P, I \leftarrow$ ElementWiseMin(P, I, $D_{1:l}$, l+1) *// note that we ignore trivial match here*
15 $p_{new}, i_{new} \leftarrow$ Min(D) *// note that we ignore trivial match here*
16 $P^{new} \leftarrow [P, p_{new}]$, $I^{new} \leftarrow [I, i_{new}]$
17 **return** P^{new}, I^{new}

In line 1, S is a new subsequence generated at the end of T^{new}. Lines 2-5 evaluate the new dot product vector Q^{new} according to (3.4), where Q_i^{new} is the dot product of S and the i^{th} subsequence of T^{new}. Note that the length of Q^{new} is one item longer than that of Q. The first dot product Q_1^{new} is a special case where (3.4) is not applicable, so lines 6-9 calculate it with simple brute-force. In lines 10-12 we evaluate the mean and standard deviation of the new subsequence S, and update the vectors μ^{new} and σ^{new}. After that we calculate the distance profile D with regard to S and T^{new} in line 13. Then, similar to STOMP, line 14 performs a pairwise comparison between every element in D and the

corresponding element in *P* to see if the corresponding element in *P* needs to be updated. Note that we only compare the first *l* elements of *D* here, since the length of *D* is one item longer than that of *P*. Line 15 finds the nearest neighbor of *S* by evaluating the minimum value of *D*. Finally, in line 16, we obtain the new matrix profile and associated matrix profile index by concatenating the results in line 14 and line 15.

The time complexity of the STOMP*I* algorithm is O(*n*) where *n* is the length of size of the current time series *T*. Note that as we maintain the profile, each incremental call of STOMP*I* deals with a one-item longer time series, thus it gets *very* slightly slower at each time step. Therefore, the best way to measure the performance is to compute the Maximum Time Horizon (MTH), in essence the answer to this question: "*Given this arrival rate, how long can we maintain the profile before we can no longer update fast enough?*"

Note that the subsequence length *m* is not considered in the MTH evaluation, as the overall time complexity of the algorithm is O(*n*), which is independent of *m*. We have computed the MTH for two common scenarios of interest to the community.

- **House Electrical Demand** [52]: This dataset is updated every eight seconds. By iteratively calling the STOMP*I* algorithm, we can maintain the profile for at least twenty-five years.
- **Oil Refinery**: Most telemetry in oil refineries and chemical plants is sampled at once a minute [83]. The relatively low sampling rate reflects the "inertia" of massive boilers/condensers. Even if we maintain the profile for 40 years, the update time is only around 1.36 seconds. Moreover, the raw data, matrix profile

and index would only require 0.5 gigabytes of main memory. Thus the MTH here is forty-plus years.

For both these situations, given projected improvements in hardware, these numbers effectively mean we can maintain the matrix profile forever.

As impressive as these numbers are, they are actually quite pessimistic. For simplicity we assume that *every* value in the matrix profile index will be updated at each time step. However, empirically, much less than 0.1% of them need to be updated. If it is possible to *prove* an upper bound on the number of changes to the matrix profile index per update, then we could greatly extend the MTH, or, more usefully, handle much faster sampling rates. We leave such considerations for future work.

3.2.3 Porting STOMP to a GPU Framework

As we will show in Section 3.3, STOMP is extremely efficient, much faster than real time for many motif discovery tasks. Nevertheless, it still takes STOMP approximately 5-6 hours to process a time series of length one million. Can we further reduce the time?

It is important to note that the STOMP algorithm is extremely amenable to parallel computing frameworks. This is not a coincidence; the algorithm was conceived with regards to eventual hardware acceleration. Recall that the space requirement for the algorithm is only O(n); there is no data dependency in the main inner loop of the algorithm (lines 7-9 of **Algorithm 2**), so we can update all entries of Q in parallel. The evaluation of each entry in vectors D, P, and I in lines 11 and 12 are also independent of each other. In the next section, we will introduce a GPU-based version of STOMP,

utilizing these observations to further speed up the evaluation of the matrix profile and thus motif discovery.

The Graphic Processor Unit, or GPU, is "*especially well-suited to address problems that can be expressed as data-parallel computations*" [54]. It has its own memory, and it can launch multiple threads in parallel. Here, we use the ubiquitous Single Instruction Multiple Data (SIMD) NVIDIA CUDA architecture, where we can assign multiple threads to process the same set of instructions on multiple data.

The threads on the GPU are managed in thread blocks. Threads in a thread block run simultaneously, and they can cooperate with each other through shared local resources. A CUDA function is called a *kernel*. When we launch a kernel, we can specify the number of blocks and the number of threads in each block to run on GPU. For example, the NVIDIA Tesla K80 allows launching at most 1024 threads within a block and as many as 2^{63} blocks (a total of 2^{73} threads), which is plentiful for processing a time series of length 100 million.

The GPU implementation of the STOMP algorithm in **Algorithm 2** can be decomposed into four steps:

- CPU copies the time series to GPU global memory.
- CPU launches GPU kernels to evaluate μ, σ, the initial Q, D, P and I.
- CPU iteratively launches GPU kernels to update QT, D, P, and I.
- CPU copies the final output (P and I) from GPU.

In the first step, the CPU copies time series T (input vector of **Algorithm 2**) to the global memory of GPU. The time used to copy a time series of length 100 million takes

less than a second. Note that in order to run the STOMP algorithm, we need to allocate space to store eight vectors in the GPU global memory: T, μ, σ, Q, Q', D, P and I. A double-precision time series of length 100 million is approximately 0.8GB, so the algorithm utilizes approximately 6.4GB global memory space. This is feasible for NVIDIA Tesla K40 and K80 cards; however, if the device used has less memory space available, we can split the time series into small sections and evaluate one section at a time with the GPU.

In the second step, the CPU launches GPU kernels to evaluate the vectors in parallel. The mean and standard deviation vectors in line 2 of **Algorithm 2** can be efficiently evaluated by CUDA Thrust [54]. The first QT vector in line 3 can be evaluated in parallel by applying cuFFT, the NVIDIA CUDA Fast Fourier Transform [55] to the *SlidingDotProduct* function in **Algorithm 1**. We assign a total of n-m+1 threads to evaluate Q', D, P, and I in lines 3-5 in parallel. The j^{th} thread processes the j^{th} entry of these vectors one by one.

Now that we have initialized Q, D, P, and I, we update them iteratively. In the third step, the CPU runs the outer loop in lines 6-13 of **Algorithm 2** iteratively. In every iteration, the CPU launches a GPU kernel with n-m+1 threads, parallelizing the evaluation of Q, D, P, and I. As shown in **Figure 3.3**, the first thread reads Q_1 from the pre-computed Q' vector, while the second to the last threads evaluate their corresponding entry of Q using (3.4).

Note that in contrast to the CPU STOMP algorithm, which uses only one vector Q to store both Q_{i-1} and Q_i, here we use two vectors to separate them. This is necessary

because as the threads evaluate entries in Q in parallel, we need to avoid writing entries *before* they are read. A simple and efficient way to accomplish this is to create two vectors, *Qodd* and *Qeven*. When the outer loop variable i in line 6 is even, the threads read from *Qodd* and write to *Qeven*; when i is odd, the threads read data from *Qeven* and write to *Qodd*. Following this, the threads evaluate D with (3.1), and the j^{th} thread updates P and I if $D_j < P_j$.

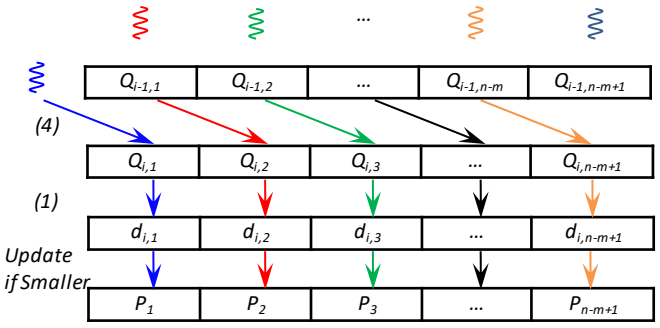

Figure 3.3. Division of work among threads in the third step of GPU STOMP.

When all of the iterations are complete, we have reached the last step of GPU STOMP, where the CPU copies P and I back to the system memory.

3.2.4 Further Parallelizing STOMP with multiple GPUs

The parallelization scheme above is suitable if we only have one GPU device. Can we further reduce the processing time if there are two or more GPUs available?

Thus far, we have been using CPU to iteratively control the outer loop of the STOMP algorithm in **Algorithm 2**. We start by computing the first distance profile (the first row) in **Figure 2.2** and its corresponding Q vector. Then in each iteration, we compute a new row of the distance matrix in **Figure 2.2**, and maintain the minimum-so-

far values of each column in vector *P*. When the iteration is complete, *P* becomes the exact matrix profile.

This outer loop computation can be further parallelized. Assume we have *k* independent GPU devices, and we also have $(n-m+1)/k = q$. We can then divide the distance matrix in **Figure 2.2** into *k* sections: device *1* evaluates the 1^{th} to the q^{th} rows, device *2* evaluates the $(q+1)^{th}$ to the $(2q)^{th}$ rows, etc. Essentially, device *k* uses the parallelized version of *SlidingDotProduct* function in **Algorithm 1** to calculate $Q_{q(k-1)+1}$ and $D_{q(k-1)+1}$, then it evaluates the following *q-1* rows iteratively. The *k* devices can run in parallel, and after the evaluation completes, we can simply find the minimum among all the *k* matrix profile outputs. In summary, we can achieve a *k*-times speed up by using *k* identical GPU devices.

By porting all the introduced techniques to NVIDIA Tesla K80, which contains two GPU devices on the same unit, we are able to obtain the matrix profile and matrix profile index of a seismology time series of length 100 million within 19 days. Are there any further optimizations left?

3.2.5 A Technique to Further Accelerate GPU-STOMP

Figure 3.3 showed the process to compute the i^{th} row of the distance matrix in **Figure 2.2** by *n-m+1* parallel threads. Recall that the distance matrix corresponding to a self-join matrix profile is symmetric; half of the distance computations can be saved if we only evaluate the i^{th} to the last columns. We show this strategy in **Figure 3.4**.*top*.

However, note that it is desirable to maintain the $O(n)$ space complexity of our algorithm; if we move on to the $(i+1)^{th}$ row of **Figure 2.2** without further processing,

then $P_i = min(d_{1,i}, d_{2,i}, \ldots, d_{i,i})$, and it would no longer be updated. To correct this, it is necessary to launch another kernel after **Figure 3.4**.*top* is completed. The new kernel is shown in **Figure 3.4**.*bottom*.

Essentially, we have used an analogous reduction technique as in [27] to obtain $d_{min} = min(d_{i,i+1}, d_{i,i+2}, \ldots, d_{i,m+n-1})$, which also is equivalent to $min(d_{i+1,i}, d_{i+2,i}, \ldots, d_{n-m+1,i})$ as a result of symmetry. If $d_{min} < P_i$, we set $P_i = d_{min}$, so $P_i = min(D_i)$. Although it is necessary to launch an additional kernel to process each row, which will require extra time, the extra time is still less than what is saved when handling large time series.

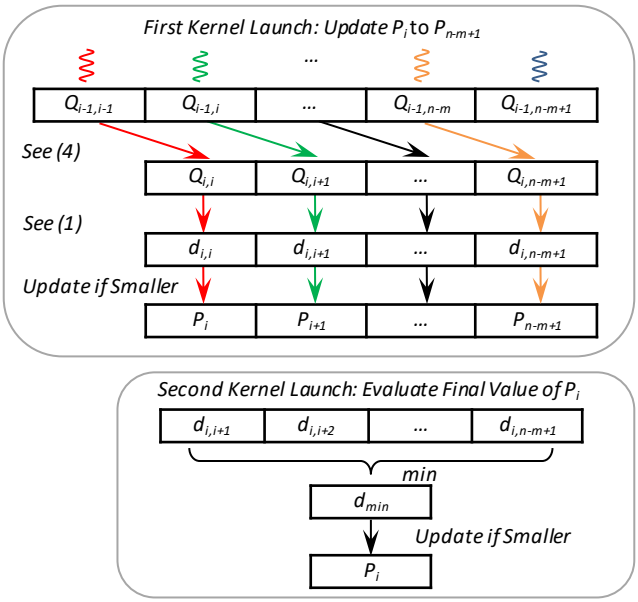

Figure 3.4. Modifying the third step of GPU-STOMP. *top)* Launch only *n-m-i+2* threads (instead of the *n-m+1* threads in Figure 3.3) this time at the i^{th} iteration. *bottom)* Launch another kernel to evaluate the final value of P_i.

For example, this new technique reduced the time to process a time series of length 100 million from 19 days to approximately 12 days on NVIDIA Tesla K80. This indicates that it is possible to finish five quadrillion pairwise comparison of subsequences within 12 days.

Note that fewer and fewer threads are being launched in each iteration. To apply this new technique to multiple GPUs, it is necessary to ensure that each GPU is loaded with similar amount of work, so they will finish in similar time. Here, for NVIDIA Tesla K80, we computed the first $(n-m+1)(1-1/\sqrt{2})$ distance profiles with the first GPU and the last $(n-m+1)/\sqrt{2}$ distance profiles with the second GPU.

3.2.6 A Final Optimization: Breaking the Ten Quadrillion Pairwise Comparison Barrier

In the last section, we demonstrated a technique to use parallel threads to evaluate the rows of the distance matrix in **Figure 2.2** iteratively. Note that to compute one row, the technique needs to launch two kernels, all threads need to be synchronized following the evaluation, and the corresponding *Q* vector needs to be updated in GPU global memory. As there are $n-m+1$ rows in **Figure 2.2**, when *n* becomes large, the time cost for kernel launch, and the thread synchronization and memory writing becomes nontrivial.

As impressive as the results are in the last section, which breaks the 5 quadrillion pairwise comparison barrier, there is one more optimization we can perform to further speed up the GPU code. We denote this optimized version GPU-STOMP$_{OPT}$. To help the reader better understand how the GPU-STOMP$_{OPT}$ works, we will first show our initial optimization scheme in **Figure 3.5**, then further refine it in **Figure 3.6** and **Figure 3.7**.

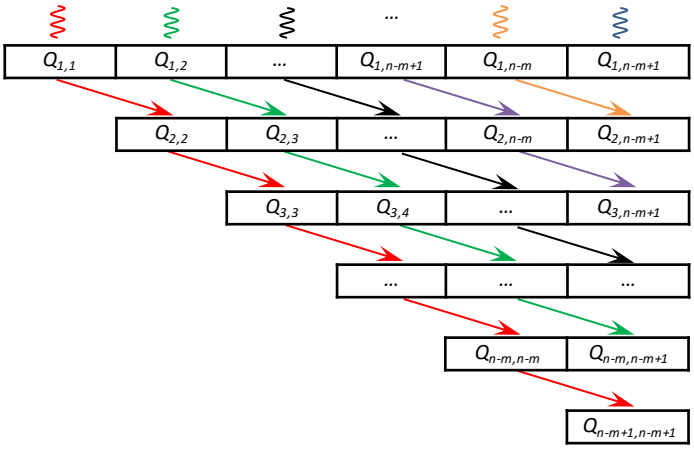

Figure 3.5. An optimization scheme for the the third step of GPU-STOMP. We only need to launch one kernel to evaluate all the rows of the distance matrix in Figure 2.2.

Figure 3.5 shows our key scheme to save the kernel launch and thread synchronization time: instead of launching a kernel for every single row in **Figure 2.2**, we issue only one single kernel to generate the entire matrix profile. Note that based on the one-one correspondence between $d_{i,j}$ and $Q_{i,j}$ (as shown in (3.1)), we can convert the symmetric distance matrix computation into **Figure 3.5**, where we evaluate the upper-right half of the dot product matrix. Since the value of $Q_{i,j}$ is only dependent on $Q_{i-1,j-1}$ (according to (3.4)), the computation of each diagonal in **Figure 3.5** is independent of any other diagonal. Thus, we assign $n-m+1$ threads to compute these diagonals in parallel.

Once we obtain $Q_{i,j}$, we can easily evaluate $d_{i,j}$ based on (3.1). Then we examine two elements of the matrix profile: if $d_{i,j} < P_i$, we set $P_i = d_{i,j}$; and if $d_{i,j} < P_j$, we set $P_j = d_{i,j}$. Note that as each thread in **Figure 3.5** operates independently, multiple threads may attempt to update the same entry of the matrix profile at the same time. We need to use CUDA atomic operations to organize this. Essentially, we set a lock for each entry of the

matrix profile. When multiple threads try to update the same matrix profile entry, they line up to get the lock, and perform an atomic Min operation in order. The reader may doubt that this can result in a significant cost of time, as it is possible that all threads can be lining up to update the same single matrix profile entry. However, in practice, we find that a large portion of these atomic operations can be pruned from the calculation.

Assume we have twenty atomic operations lined up to update a matrix profile entry, which has an initial value of 6.81, with the following distance values in order:

0.6, 4.46, 1.99, 6.98, 2.29, 2.95, 7.05, 1.47, 6.04, 2.72, 2.31, 3.2, 6.25, 9.33, 0.27, 2.62, 2.00, 2.74, 6.67, 2.34.

Since the matrix profile entry keeps track of the minimum distance value, only two updates would be executed: 0.6 and 0.27. That is only 10% of this short sequence of data. Now let us randomly shuffle the data:

7.05, 2.29, 1.47, 0.27, 2.74, 2.95, 9.33, 2.34, 4.46, 2.00, 6.04, 2.72, 2.31, 3.2, 6.25, 6.98, 0.6, 2.62, 1.99, 6.67.

This time three updates would be executed: 2.29, 1.47, 0.27. That is only 15% of the data; so again, it is only a small portion.

Note that our toy example here is a very short data sequence. In practice, for most time series only less than 0.1% distance values end up smaller than their corresponding matrix profile elements. For example, for a random-walk time series of length one million, we executed on average only 39 atomic calls for each matrix profile entry; more than 99.996% of the atomic operations are pruned.

By implementing the optimization scheme shown in **Figure 3.5**, we have obtained about 3X speedup over GPU-STOMP for medium-size time series (i.e. with less than 4 million data points). However, as the time series gets even longer, less speedup is observed, as the time spent on atomic operations and global memory writes become nontrivial.

To solve this, we use two strategies to refine our optimization scheme in **Figure 3.5**.

The first strategy aims to accelerate each atomic write. As stated previously, multiple independent threads can be attempting to update the matrix profile at the same time, so we are using CUDA atomic Min operation to organize them. Note that when a matrix profile entry (which is a 64-bit double precision value) is updated, the corresponding matrix profile index value (a 32-bit integer value) also needs to be updated. However, currently CUDA only supports atomic operations on either one single 32-bit value or one single 64-bit value. To tackle this, we initially set a lock on every entry of the matrix profile, and used a critical section to update both the matrix profile entry and the matrix profile index value when a thread gets the lock; however, this solution is not scalable with longer time series inputs. As a result, we turned to a better solution as shown in **Figure 3.6**. Instead of using a time-consuming critical section, we lower the precision of the matrix profile to 32 bits. We then combine the matrix profile and the matrix profile index into one double-precision vector in the global memory that can be atomically updated. For the *i*th entry of the double-precision vector, 32 bits are

used to store the *i*th matrix profile value, and another 32 bits are used to store the *i*th matrix profile index.

This refinement strategy largely accelerated the speed for atomic operations. Note that the strategy will not result in large precision loss, as only the precision of the output is reduced; we are still using 64 bits to store all the intermediate results during the evaluation process.

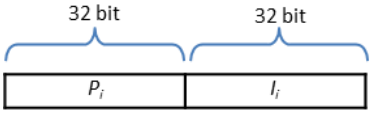

Figure 3.6. We reduced the matrix profile to 32 bits, then combined each matrix profile entry and its corresponding matrix profile index entry into a double-precision value to allow fast atomic updates.

The second strategy is to utilize the CUDA shared memory to ease the contention for global memory writes. The strategy, as shown in **Figure 3.7**, can be viewed as 2-level hierarchy of **Figure 3.5**. Here we define *TPB* as the number of threads per block on CUDA.

Different from **Figure 3.5**, in which each thread evaluates one single diagonal of the distance matrix, here we divide the distance matrix into *k* meta diagonals (as shown in **Figure 3.7**.*a*, a meta diagonal consists of *TPB* diagonals of the distance matrix; ($k-1)\times TPB<n-m+1\leq k\times TPB$). Each meta diagonal is evaluated by one CUDA thread block. As shown in **Figure 3.7**.*b*, the thread block evaluates one parallelogram at a time, managing a local copy of the matrix profile in the shared memory. The threads in a block (shown in **Figure 3.7**.*c*) work very similarly as those in **Figure 3.5**, except that they atomically update the shared memory instead of the global memory. After a parallelogram (**Figure 3.7**.*b*) is evaluated, all the threads in the block are synchronized. If

any value in the shared memory is smaller than its corresponding entry in the global memory, the global memory is updated.

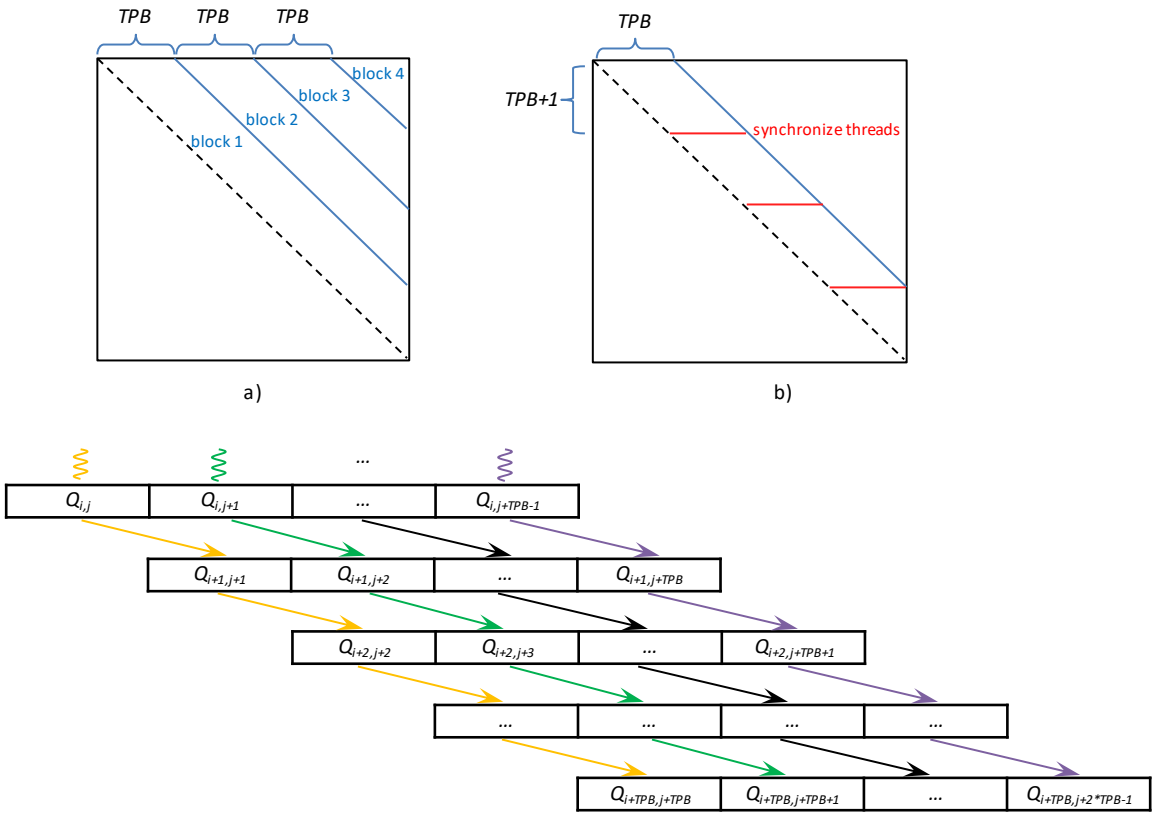

Figure 3.7. ***a)*** **Each thread block evaluates one meta diagonal of the distance matrix.** ***b)*** **The parallelograms in a meta diagonal are evaluated iteratively by a thread block.** ***c)*** **The threads in a block evaluate diagonals of a parallogram in parallel.**

With this refinement strategy, the contention of atomic updates in **Figure 3.5** is largely relieved. The original scheme in **Figure 3.5** allowed a global memory location to be visited by all active threads in all the thread blocks (which can be as many as $(n\text{-}m+1)$ threads) simultaneously. In contrast, with the refined scheme in **Figure 3.7**, the number of threads racing for a shared memory location cannot be larger than *TPB*, and a global memory location cannot receive more than k atomic update requests at the same time. This brings about a large performance gain.

Similar to GPU-STOMP, GPU-STOMP$_{OPT}$ can easily be adapted to multiple GPUs as well. For example, to evenly divide the work for an NVIDIA Tesla K80, we compute the odd (1st. 3rd, 5th, etc. from the left) meta diagonals in **Figure 3.7**.*a* with the first GPU, and the even (2nd, 4th, 6th, etc. from the left) meta diagonals in **Figure 3.7**.*a* with the second GPU.

With all the optimization strategies, GPU-STOMP$_{OPT}$ achieved more than 2X speedup over GPU-STOMP for large datasets. Concretely, it further reduces the time to process a time series of length 100 million from 12 days to about 4 days on NVIDIA Tesla K80. Furthermore, for the first time in the literature, we are able to process a time series of length 143 million, which is slightly more than ten quadrillion pairwise comparison of subsequences, within just 9 days.

3.3 Empirical Evaluation

Although some parts of our experiments require access to a GPU, we have designed them so they can be reproduced easily. To allow for the reproduction of our experiments, we have constructed a webpage [59], which contains all datasets and code used in this work. We begin with a careful comparison to STAMP [96], which is obviously the closest competitor, and we consider more general rival methods later.

Unless otherwise noted, we used an Intel i7@4GHz PC with 4 cores to evaluate all the CPU-based algorithms; we used a server with two Intel Xeon E5-2620@2.4GHz cores and an NVIDIA Tesla K80 GPU to evaluate GPU-STOMP.

3.3.1 STAMP vs STOMP

We begin by demonstrating that STOMP is faster than STAMP, and also that this difference grows as we consider increasingly large datasets. Furthermore, we measure the gains made by using GPU-STOMP. In **Table 3.1**, we measure the performance of the three algorithms on increasingly long random walk time series with a fixed subsequence length 256.

Table 3.1. Time required for motif discovery with $m = 256$, varying n, for the three algorithms under consideration

Algorithm \ Value of n	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
STAMP	15.1 min	1.17 hours	5.4 hours	24.4 hours	4.2 days
STOMP	4.21 min	0.3 hours	1.26 hours	5.22 hours	0.87 days
GPU-STOMP	10 sec	18 sec	46 sec	2.5 min	9.25 min

Note that we choose *m*'s length as a power-of-two only to offer the best case for (the FFT-based) STAMP; our algorithm is agnostic to such issues.

A recent paper on finding motifs in seismograph datasets also considers a dataset of about 2^{19} in length and reports taking 1.6 hours, which is approximately the same time it takes STOMP [101]. However, their method is probabilistic and allows false negatives (twelve of which were actually observed, after checking against the results of a 9.5 day brute-force search [101]). Moreover, it requires careful tuning of several parameters, and it does not lend itself to GPU implementation.

We wish to consider the scalability of even larger datasets with GPU-STOMP. However, in order to do so, we must estimate the time it takes the other two other algorithms. Fortunately, both of the other algorithms allow for an approximate prediction of the time needed, given the data length *n*. To obtain the estimated time, we evaluated

only the first 100 distance profiles of both STAMP and STOMP and multiplied the time used by (*n-m+1*)/100. In **Table 3.2**, we consider even larger datasets, one of which reflects the data used in a case study in Section 3.3.4.

Table 3.2. Time required for motif discovery with various m and various *n*, for the three algorithms under consideration

Algorithm \ *m* | *n*	2000 | 17,279,800	400 | 100,000,000
STAMP (*estimated*)	*36.5 weeks*	*25.5 years*
STOMP (*estimated*)	*8.4 weeks*	*5.4 years*
GPU-STOMP (actual)	9.27 hours	12.13 days

Note that the 100-million-length dataset is one hundred times larger than the largest motif search in the literature [37].

In all three algorithms under consideration, the time required is independent of the subsequence length *m*, which is desirable. This is demonstrated in **Table 3.3**, where we measure the time required with *n* fixed to 2^{17}, for increasing *m*.

Table 3.3. Time required for motif discovery with $n = 2^{17}$, varying *m*, for the three algorithms under consideration

Algorithm \ Value of *m*	64	128	256	512	1,024
STAMP	15.1 min	15.1 min	15.1 min	15.0 min	14.5 min
STOMP	4.23 min	4.33 min	4.21 min	4.23 min	2.92 min
GPU-STOMP	10 sec	10 sec	10 sec	10 sec	10 sec

Note that the time required for the longer subsequences is slightly shorter. This is true since the number of pairs that must be considered for a time series join [96] is $(n-m+1)^2$, so as *m* becomes larger, the number of comparisons becomes slightly smaller.

3.3.2 GPU-STOMP$_{OPT}$ Breaks the Ten Quadrillion Pairwise Comparison Barrier

In **Table 3.4**, we measure the performance of STAMP, STOMP, GPU-STOMP, and GPU-STOMP$_{OPT}$ on increasingly long random walk time series with a fixed subsequence length 256. The shaded cells are duplicated from **Table 3.1**, but they are included for comparison. Note that while some numbers are estimated, as explained in the next section, we can predict the time and memory requirement of STAMP and STOMP very precisely (with less than 5% error) for large datasets.

Table 3.4. Time required for motif discovery with *m* = 256, varying *n*, for the three algorithms under consideration

Algorithm \ Value of *n*	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	17,279,800	100,000,000	143,000,000
STAMP	15.1 min	1.17 hours	5.4 hours	24.4 hours	4.2 days	36.5 weeks (estimated)	25.5 years (estimated)	51.2 years (estimated)
STOMP	4.21 min	0.3 hours	1.26 hours	5.22 hours	0.87 days	8.4 weeks (estimated)	5.4 years (estimated)	10.9 years (estimated)
GPU-STOMP	10 sec	18 sec	46 sec	2.5 min	9.25 min	9.27 hours	12.13 days	24.5 days (estimated)
GPU-STOMP$_{opt}$	8 sec	9 sec	17 sec	49 sec	2.93 min	3.29 hours	4.51 days	9.33 days

We note in passing that this experiment on a time series of length 143 million is the largest time series ever searched for exact motifs. Moreover, we are confident that this is the first time ten quadrillion pairwise comparisons have been made on a single dataset, in any context.

The time required for GPU-based algorithms can be divided into two parts. The first part includes the data reading time and computation time; the second part includes the time needed for kernel launch, data synchronization and memory writes. GPU-STOMP and GPU-STOMP$_{OPT}$ spent the same time in the first part. To further compare

the effectiveness of the two methods, in **Table 3.5** we measure their run time in the second part.

Table 3.5. Time required for kernel launch, data synchronization and memory writes with *m* = 256, varying *n*, for the two GPU-based algorithms

Algorithm \ Value of *n*	2^{18}	2^{19}	2^{20}	2^{21}	17,279,800	100,000,000
GPU-STOMP	17 sec	41 sec	2.2 min	8.17 min	8.04 hours	10.39 days
GPU-STOMP$_{opt}$	8 sec	12 sec	32 sec	1.85 min	2.06 hours	2.77 days

We can wee that GPU-STOMP$_{OPT}$ achieved more than 4X speedup over GPU-STOMP in the second part for large datasets.

3.3.3 STOMP vs State-of-the-Art Motif Discovery Algorithms

Beyond the independence of the subsequence length demonstrated in **Table 3.3**, all three matrix profile-based algorithms are also independent of the intrinsic dimensionality of the *data*, which is also desirable. To demonstrate this, we will compare to the recently introduced Quick-Motif framework [37] and the more widely known MK algorithm [49]. The Quick-Motif method was the first technique to perform an exact motif search on one million subsequences.

To level the playing field, we do not avail of GPU acceleration, but instead, we use the identical hardware (a PC with Intel i7-2600@3.40GHz) and programming languages for all algorithms. Note that for a fair comparison with STAMP [96], which is written in Matlab, in Section 3.3.1, we measured the performance of STOMP based on its Matlab implementation. However, because the two rival methods in this section (Quick-Motif and MK) are written in C/C++, here we measure the runtime of (the CPU version of) STOMP based on its C++ implementation.

We use the original author's executables [60] to evaluate the runtime of both MK and Quick-Motif. The reader may wonder why the experiments here are less ambitious than in the previous sections. The reason is that beyond *time* considerations, the rival methods have severe *memory* requirements. For example, for a seismology data with m = 200, $n = 2^{18}$, we measured the Quick-Motif memory footprint as large as 1.42 GB. In contrast, STOMP requires only 14MB memory for the same data, which is less than 1/100 of this. If this ratio linearly interpolates, Quick-Motif would need more than ½ terabyte of main memory to tackle the one hundred million benchmark, which is infeasible. Moreover, for Quick-Motif, it is possible that a different dataset of the same size could require a larger or smaller footprint. In contrast, the space required for STOMP is independent of both the structure of data and the subsequence length.

This severe memory requirement makes it impossible to compare the STOMP algorithm with Quick-Motif on the seismology data, since Quick-Motif often crashed with an *out-of-memory* error as we varied the value of m. However, we noticed that the memory footprint for Quick-Motif tends to be much smaller with smooth data. Therefore, instead of comparing performance of the algorithms on seismology data, in **Table 3.6**, we utilized the much smoother ECG dataset (used in Rakthanmanon et al. [61]), which is an ideal dataset for both MK and Quick-Motif to achieve their *best* performance.

Table 3.6. Time required for motif discovery with $n = 2^{18}$, varying m, for various algorithms

Algorithm \ m	512	1,024	2,048	4,096
STOMP	501s (14MB)	506s (14MB)	490s (14MB)	490s (14MB)
Quick-Motif	27s (65MB)	151s (90MB)	630s (295MB)	695s (101MB)
MK	2040s (1.1GB)	N/A (>2GB)	N/A (>2GB)	N/A (>2GB)

Clearly, both the runtime and memory requirement for STOMP are independent of the subsequence length. In contrast, Quick-Motif and MK both poorly scale in subsequence length in both runtime and memory usage. Note that the memory requirement of Quick-Motif is not monotonic in *m*, as reducing *m* from 4,096 to 2,048 requires three times as much memory. This is not a flaw in implementation (we used the author's own code), but a property of the algorithm itself.

As indicated in **Figure 3.1**, the Quick-Motif algorithm [37], the MK algorithm [49], and the original motif discovery by *projection* algorithm [15] can all be fast in the *best* case. For example, if there happens to be a perfect (zero Euclidean distance) motif in the dataset, they will all discover it with O(n) work (with high constants), and all algorithms can use this zero-valued best-so-far to prune all other possibilities for motif pairs. While we generally do not expect to have a zero-distance motif in real-valued data, a very close motif pair in a dataset with low intrinsic dimensionality (recall **Figure 3.1**) can offer similar speed ups. However, that describes the *best* case for all three algorithms. Consider instead the *worst* case (for example, the input signal is white noise, and all subsequences are effectively equidistant from each other), all three rival algorithms degenerate to O(mn^2) (again, with high constants). In contrast, STOMP is unique in that its best case and worse case are identical, just O(n^2). Because *m* can be as large as 2,000 (see **Figure 3.8**), this can produce a significant speedup. Moreover, as we will show in the next two sections, STOMP computes much more useful information than the two rival methods.

Before demonstrating this, we show that the experiments in the previous table were spurious for STOMP. We do not need to *measure* its time or memory footprint, because we can predict it precisely. To the best of our knowledge, this property is unique among all motif discovery algorithms proposed in the literature [15][37][49].

For STOMP (assuming only that $m \ll n$), given only n, we can predict how long the algorithm will take to terminate and how much memory it will consume, which is completely independent of the value of m and the data.

To do this, we need to do a single calibration run on the machine in question. With a time series of length n, we measure T, the time taken to compute the matrix profile, and M, the (maximum) amount of memory consumed. Then, for any new length n_{new}, we can compute $T_{required}$, the time needed as the following:

$$T_{required} = \frac{T}{n^2} \times n_{new}^2 \tag{3.5}$$

and we can compute $M_{required}$, the memory needed as the following:

$$M_{required} = \frac{M}{n} \times n_{new} \tag{3.6}$$

As long as we avoid trivial cases, such as $m \sim n$, n_{new} is very small or n is very small, this formula will predict the resources needed with an error of less than 5%. To demonstrate this, we performed the following experiment. On our machine (a PC with Intel i7-2600@3.40GHz) we ran STOMP (Matlab version) on a random walk dataset of size 2^{18}, measuring the resources consumed. Then, as shown in **Table 3.7**, we use the formulas above to predict the resources needed to compute the Matrix Profile for datasets of size $\{2^{18}, 2^{19}, 2^{20}, 2^{21}\}$. Then we measured these values with actual experiments on

random walk data. From **Table 3.7**, the agreement between our predictions and the observed values is clear.

Table 3.7. Time and memory required for STOMP, with *m* = 256, varying *n*

Resources / *n*	2^{18}	2^{19}	2^{20}	2^{21}
STOMP time (memory) measured	19.0 min (30MB)	75.6 min (60MB)	313 min (121MB)	1253 min (242MB)
STOMP time (memory) predicted	19.0 min (30MB)	76.0 min (60MB)	304 min (120MB)	1216 min (240MB)
Relative Error	0% (0%)	0.5 % (0%)	3% (0.8%)	3% (0.8%)

This property has several desirable implications: we can carefully plan resources when performing analytics on large data archives; we can easily divide the work to parallel computing resources to finish our task in time; and we can show a perfectly accurate "progress bar" to a user who is using STOMP interactively.

3.3.4 Case Studies in Seismology: Infrequent Earthquake Case

To allow confirmation of the correctness and utility of STOMP, we begin by considering a dataset for which we know the result from external sources. On April 30th 1996, there was an earthquake of magnitude 2.12 in Sonoma County, California[1]. Then on December 29th 2009, about 13.6 years later, there was another earthquake with a similar magnitude. We concatenated the two full days in question to create a single time series of length 17,279,800 (see **Table 3.2** for timing results) and examined the top motifs with m = 2,000 (twenty seconds). Note that we are using the raw data as provided

[1] A small earthquake of that magnitude would only be felt by attentive humans in the immediate vicinity of the epicenter.

to us by the seismologists, we are not preprocessing it in anyway. As **Figure 3.8**.*top* illustrates, the top motif here is *not* an earthquake but an unusual sensor artifact [28].

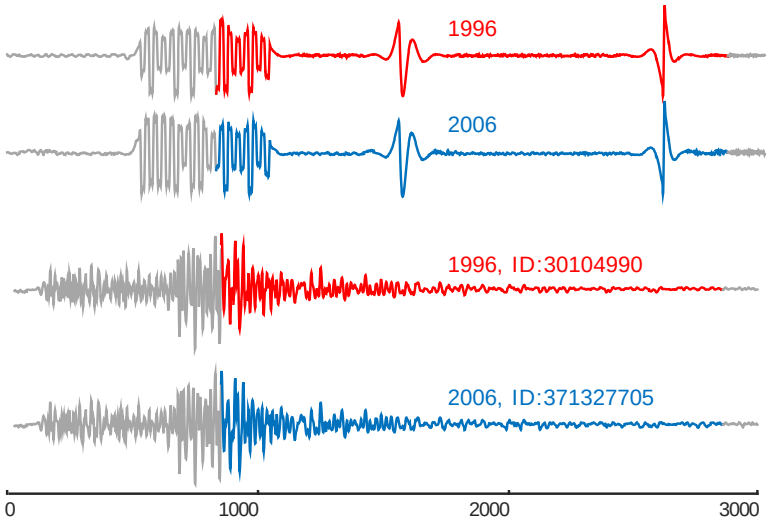

Figure 3.8. Motifs (colored) shown in context (gray). *top*) The top motif discovered in the Sonoma County dataset is a sensor artifact, as are the next three motifs (not shown). *bottom*) The fifth motif is two true occurrences of an earthquake that happen 4,992 days apart.

There are a handful of other such artifacts; however, as shown in **Figure 3.8**.*bottom*, the fifth best motif *is* the two occurrences of the earthquake. These misleading sensor artifacts are common, but they could be eliminated easily [28]. For example, the sensors could have a zero crossing rate that is an order of magnitude lower than true earthquakes.

This example allows us to demonstrate yet another advantage of STOMP over rival methods. All the existing rival techniques can be expanded from top-1 motif discovery to top-*k* motif discovery; however, increasing *k* by a modest amount will significantly degrade their speed.

Furthermore, consider again the example in **Figure 3.8**. It is not possible to have known the "magic" value of $k = 5$ beforehand. If *k* was set to a large value to "be on the

safe side," say k = 10, then all existing techniques would slow down because the best-so-far lower bound that prunes unnecessary computations would be much looser. If we set k as a more conservative value, say k = 3, then we would miss the most valuable information in this seismology dataset. You might imagine that the rival methods could slowly increase from k to $k+1$ based on the user's lack of satisfaction with the k motifs she has examined thus far; however, each adjustment of k will require all existing techniques to perform significant extra computation, *even* if they have cached the results of every calculation they have performed.

In contrast, the time needed for STOMP is completely independent of k. We only need to run STOMP once; as the matrix profile obtained already contains all necessary information, and it takes minimal additional effort to find the top k motif, no matter how large k is.

3.3.5 Parameter Settings

As we have previously noted, STOMP (together with STAMP) is unique among motif discovery algorithms because it is parameter-free. In contrast, Random Projection [15] has four parameters, Quick-Motif [37] has three parameters, Tree-Motif has four parameters [88], MK [49] has one parameter, and FAST has three parameters [101].

That being said, the reader may wonder about the only input value besides the time series of interest: the subsequence length m. Note that this is also a required input for all the other existing techniques. However, we do not consider m to be a true parameter, as it is a *user choice,* reflecting her prior knowledge of the domain. Nevertheless, it is

interesting to ask how sensitive motif discovery is to this choice; at least in the seismology domain that motivates us.

To test this, we edited the data above such that the two earthquakes in **Figure 3.8**.*bottom* happen exactly 13 minutes 20 seconds apart. We reran motif discovery with m=2,000 (twenty seconds), with double that length (m=4,000), and with half that length (m=1,000). **Figure 3.9** shows the result.

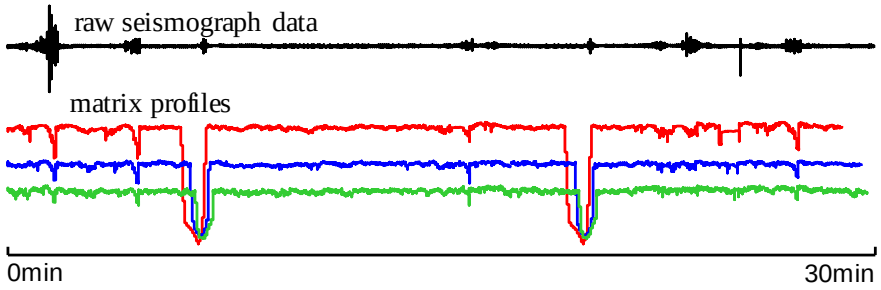

Figure 3.9. *top*) Thirty minutes of seismograph data that has the two earthquakes from Figure 3.8.*bottom* occur at 6min-40s and 20min. *bottom*) The matrix profile computed if we use the suggested subsequence length 2,000 (blue), or if we use twice the length (red), or half that length (green).

The results are reassuring. At least for earthquakes, motif discovery is not sensitive to the user input. Even a poor guess as to the best value for m, it will likely give accurate results.

3.3.6 Case Studies in Seismology: Earthquake Swarm Case

In the previous section, we discovered a repeating earthquake source that has a frequency of about once per 13.6 years. Here, we consider earthquakes that are tens of millions of times more frequent.

Forecasting volcanic eruptions is of critical importance in many parts of the world [74]. For example, on May 18th, 1980, Mount St. Helens had a paroxysmal eruption that

killed 57 people [34]. It is conjectured that explosive eruptions are commonly preceded by elevated or accelerated gas emissions and seismicity; thus, seismology is a major tool for both monitoring and predicting such events.

In **Figure 3.10**, we illustrate a short section of the matrix profile of a seismograph recording at Mount St Helens. It is important to restate that this is *not* the raw seismograph data, but it is the matrix profile that STOMP computed from it.

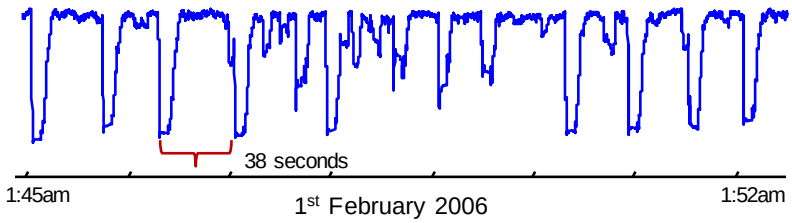

Figure 3.10. The matrix profile of a seven-minute snippet from a seismograph recording at Mount St Helens.

The image demonstrates a stunning regularity. Repeated earthquakes are occurring approximately once every thirty-eight seconds. This is consistent with the findings of a team from the US Geological Survey who reported that the earthquakes, which accompanied a dome-building eruption, appeared "... *so regularly that we dubbed them 'drumbeats'. The period between successive drumbeats shifted slowly with time, but was 30–300 seconds*" [34].

This example shows a significant advantage of our approach that we share with STAMP but no other motif discovery algorithm. Instead of computing only O(*1*) distance values for the top *k* motifs, STOMP is computing *all* O(*n*) distances from *every* subsequence to their nearest neighbors. By plotting the entire matrix profile, we gain unexpected insights by viewing the motifs *in context*. For example, in the example above,

we can see both the surprising periodicity of the earthquakes, and by comparing the smallest values in the matrix profile with the mean or maximum values, we can get a sense of how well the motifs are conserved relative to "chance" occurrences. It could also potentially indicate whether there were changes to the earthquake source, reflecting changes in eruptive behavior over time.

A recent paper performed a similar analysis on the Mount Rainier volcano, making the interesting and unexpected discovery that the frequency of earthquakes is correlated with snowfall [3]. However, the paper bemoans at the number of ad-hoc "hacks" that needed to make such an exploration tenable. For example, "*In order to save on computing time, we cut out detections that are unlikely to contain a repeating earthquake event by excluding events with a signal width,"* and "*To save on computing time, we define that in order to be detected*…" etc. [3]. However, the results in **Table 3.4** indicate that we could bypass these issues by spending a few hours computing the *full* exact answers. This would eliminate the risk that some speedup "trick" erases an interesting and unexpected pattern.

3.3.7 Case Studies in Seismology: Detection of Repeated Low Frequency Earthquakes

In the previous sections, we showed how STOMP could help us detect repeating earthquake sources by evaluating the matrix profile of a *single* seismograph recording time series. Here we show that by providing the matrix profiles of *multiple* seismograph recording time series, STOMP allows us to detect low frequency earthquakes (LFEs). LFEs are of great importance to the seismology community, as they could "*potentially*

contribute to seismic hazard forecasting by providing a new means to slow slip at depth" [66]. LFEs recur episodically, often during bursts of tectonic 'tremor', which are considered superpositions of many LFEs in a short period of elevated seismic activity [65]. One traditional approach, known as 'matched filtering' identifies repeated LFEs by evaluating the cross-correlation between continuous waveform data (time series) and a template waveform (subsequence) (e.g. [67]). However, this requires a suitable, carefully recorded template waveform of an LFE (an LFE subsequence) to have been identified in advance, which is very difficult or even impossible in many cases. In the face of this, similarity-join search through autocorrelation (e.g. [11]) has been used to detect LFEs in several studies. However, the traditional similarity-join search approach is computationally intensive (typically only one hour or less of continuously waveform data can be searched in feasible time), severely limiting the number and range of LFEs that can be detected.

Consider an example of LFE detection along the central San Andreas fault near Parkfield, CA. We search for LFEs in waveform data from a tremor burst that occurred on October, 6, 2007, in which many LFEs were detected by matched filtering [67]. As before, note that we are using the raw data as provided to us by the seismologists, we are not preprocessing it in anyway. The LFE template (subsequence) in Shelly et al. [67] was found by careful visual examination of seismic recording from multiple temporary seismic stations located close to the source (the green triangles in **Figure 3.11**; temporary stations were set up near a well-known earthquake source in this area), and subsequently also identified on more distant, permanent High Resolution Seismic Network (HRSN, the

red triangles). Note that our task here is to detect all the LFEs automatically, and the only data available are those from the HRSN stations (the red triangles in **Figure 3.11**), since in most applications we do not know the earthquake source location (thus the data from the temporary stations) until well after the event.

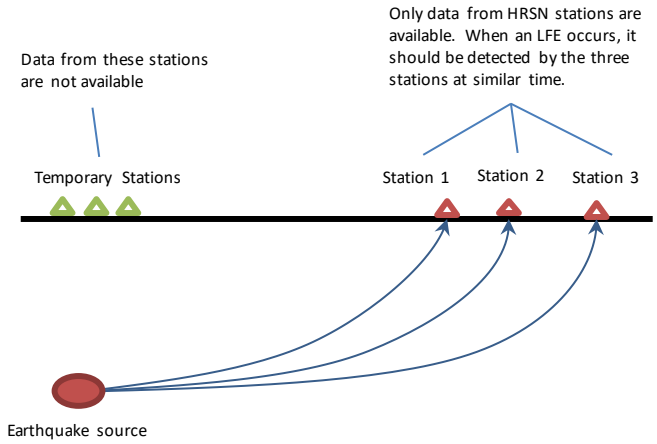

Figure 3.11. LFEs can be detected from the seismograph recording of HRSN stations.

Apart from the lack of the temporary station data, what makes our task even more difficult is that the data from HRSN stations are noisy and many contain a lot of false positives. For example, the top 15 motifs (repeating templates) found from the data of an HRSN station near central San Andreas fault are either sensor artifacts (similar to **Figure 3.8**) or instrument noise in the station itself. However, in spite of all these difficulties, we will demonstrate that STOMP allows us to detect LFEs from long seismic recordings.

We ran GPU-STOMP$_{OPT}$ on the seismic recording time series from three HRSN stations for a 24-hour period spanning the tremor burst. The three HRSN stations are located close to each other. The data was sampled at 20Hz, for a total of ~1.7 million samples per station time series. **Figure 3.12** shows the sum of the three matrix profiles obtained.

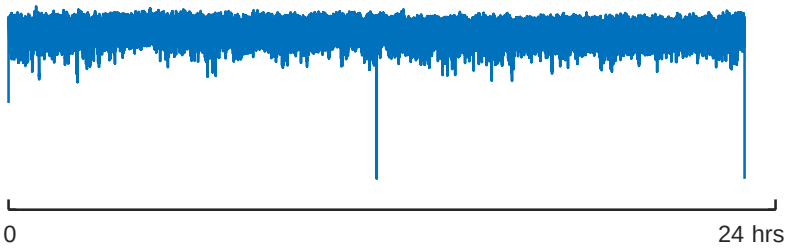

Figure 3.12. The sum of three matrix profiles of the 24-hour seismograph recording at three HRSN stations near the central San Andreas fault.

The reader may wonder why we are summing the three matrix profiles here. This simple step greatly reduces the false positives in the data. As the three HRSN stations are located close to each other, when an LFE occurs, the stations should detect it at a similar time. As a result, the matrix profile values of the three stations should all be low at the occurrence of the LFE. The sum of the matrix profiles shows low values at such time instants, which strengthens the LFE signal and thus weakens the false positives, which are local to each sensor. We discovered that the top seven motifs identified in this way were either glitches in the waveform data (sensor artifacts, again, recall **Figure 3.8**), or signals that could not be separated into individual LFEs; however, as shown in **Figure 3.13**, the 8th best motif showed strong characteristics, in terms of frequency content, waveform shape and duration, of an LFE, and the origin time of this LFE is consistent with the results in Shelly et al. [67], which may be regarded as the ground truth.

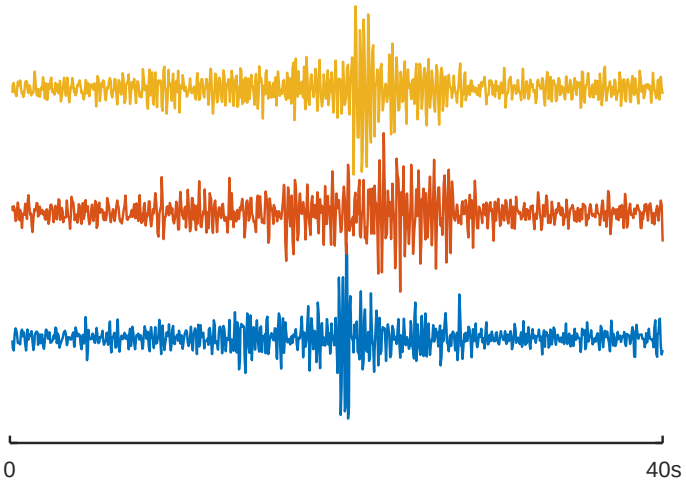

Figure 3.13. The 40-second LFE snippet detected from the three HRSN station time series.

In contrast to Shelly et al. [67], which detects the LFE pattern with *weeks* of enormous human effort, we are able to complete the same task automatically in approximately 3 minutes with GPU-STOMP$_{OPT}$ on NVIDIA Tesla K80.

3.3.8 A Case Study in Animal Behavior

While seismology is the primary motivator for this work, nothing about our algorithm assumes anything about the data's structure, or precludes us from considering other datasets. To demonstrate this, in this section, we briefly consider telemetry collected from Magellanic penguins (*Spheniscus magellanicus*). Adult Magellanic penguins can regularly dive to depths of between 20m to 50m deep in order to forage for prey, and may spend as long as fifteen minutes under water. The data was collected by attaching a small multi-channel data-logging device to the bird. The device recorded tri-axial acceleration, tri-axial magnetometry, pressure, etc. As shown in **Figure 3.14**, for simplicity we consider only Y-axis magnetometry. Note that, as with the seismology, we are not preprocessing this data source in anyway, no smoothing, not down sampling, etc.

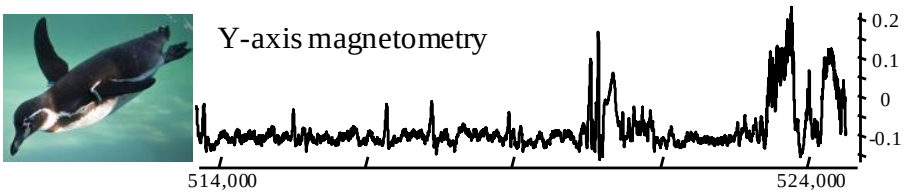

Figure 3.14. ***left*****) The Magellanic penguin is a strong swimmer.** ***right*****) A four-minute snippet of the full dataset reveals high levels of noise and no obvious structure.**

An observer with binoculars labels the data; thus, we have a coarse ground truth for the animal's behavior. The full data consists of 1,048,575 data points recorded at 40 Hz (about 7.5 hours). We ran GPU-STOMP$_{OPT}$ on this dataset, using a subsequence length of 2,000. This took our algorithm just 49 seconds to compute. As shown in **Figure 3.15**, the top motif is a surprisingly well conserved "shark fin" like pattern.

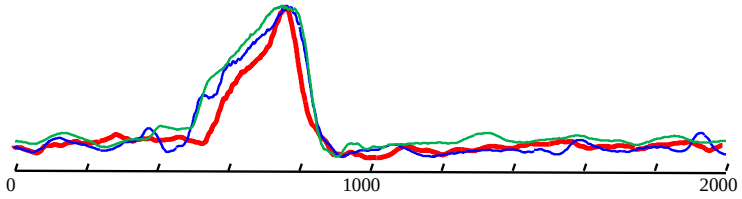

Figure 3.15. The top motif of length 2,000 discovered in the penguin dataset. Only three examples are shown for visual clarity, there are eight such patterns. This behavior may be part of a 'porpoise' maneuver.

What (if anything) does this pattern indicate? Suggestively, we observed this pattern does not occur in any of the regions labeled as *nesting*, *walking*, *washing*, etc., but only during regions labeled *foraging*. Could this motif be related to a diving (for food) behavior?

Fortunately, *diving* is the one behavior we can unambiguously determine from the data, as the *pressure* sensor reading increases by orders of magnitude when the penguin is under water. We discovered that the motif occurs moments before each dive and nowhere else. This pattern appears to be part of a ritual behavior made by the bird before diving. It

has been reported that "*The only time penguins are airborne is when they leap out of the water. Penguins will often do this to get a gulp of air before diving back down for fish*." Thus, we suspect this pattern is part of a 'porpoise' behavior [71].

Generally speaking, we see this example as typical of the interactions that motif discovery supports. In most cases, motif discovery is not the *end* of analyses, but only the *beginning*. By correlating the observed motifs with other (internal or external) data, we can form hypotheses and open avenues for further research. Recall the previous section; this is rather similar to the team studying Mount Rainier's seismology discovered that its earthquakes are correlated with snowfall [3]. We believe that the STOMP algorithm may enable many such unexpected discoveries in a vast array of domains.

3.3.9 Incrementally Maintaining Motifs

In the previous sections, we have demonstrated the ability and efficiency to detect time series motifs using the matrix profile. However, we assumed that the entire time series was available beforehand. Here we remove this assumption and show how STOMP*I* allows us to incrementally maintain time series motifs in an online fashion. There are many attempts of this task in the literature [7][82], but they are all approximate and allow false dismissals.

In Section 3.2.2, we introduced the STOMP*I* algorithm. The ability to incrementally maintain the matrix profile implies the ability to *exactly* maintain the time series motif [49] in streaming data. We simply need to keep track of the *minimum* value of the incrementally-growing matrix profile, and report a new pair of motifs when a new minimum value is detected.

We demonstrate the utility of this idea on the AMPds dataset [42]. While this is a real dataset, it lacks ground truth annotation so we slightly contrived it such that we can check the plausibility of the outcomes. For simplicity, we assume that the kitchen fridge and the heat pump are both plugged into a single metered power supply. For the first week, only the refrigerator is running. At the end of the week, the weather gets cold and the heat pump is turned on. The sampling rate is one sample/minute, and the subsequence length is 100 (i.e. one hour and forty minutes). We apply the STOMP algorithm to the first three days of data, then invoke the STOMP*I* algorithm to handle newly arriving data. Whenever we detect a new minimum value, we report a new motif.

As shown in **Figure 3.16**, a new minimum value is detected at the 9,864th minute (6 day 20 hour 24 minute), which indicates a new time series motif. The just-arrived 100-minute-long pattern looks very similar to another pattern that occurred five hours earlier. While there is a lot of regularity in the fridge data in general, the exceptional similarity observed here suggested some underlying physical mechanism that caused such a perfectly-conserved pattern, perhaps a mechanical ice-making "subroutine."

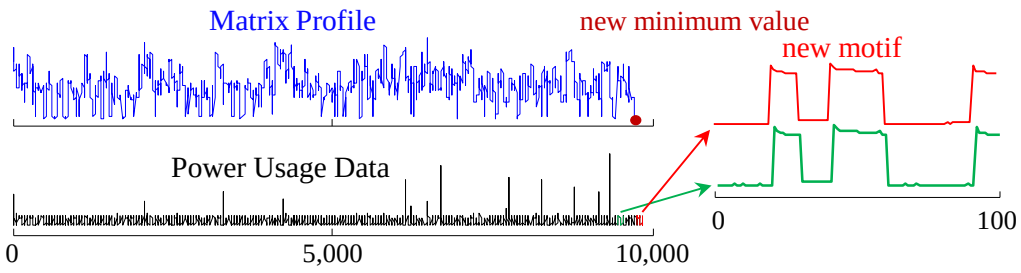

Figure 3.16. *top*) The matrix profile of the first 9,864 minutes of data. *bottom*) The minimum value of the matrix profile corresponds to a pair of time series motifs in the power usage data. *right*) The time series motif detected.

The maximum time needed to process a single data point with STOMP*I* in this dataset is 0.0003 seconds, which is less than 0.004% of the data sampling rate. Thus, on

this dataset we could continue monitoring with the STOMP*I* algorithm for several decades before running out of time or memory.

STOMP*I* may have implications for real-time earthquake warning systems, which will reduce the probability of false alarms by quickly searching dictionaries of previously confirmed events [101]. We leave such consideration for future work.

3.4 Conslusions

In this chapter we introduced STOMP, a new algorithm for time series motif discovery, and showed that it is theoretically and empirically faster than its strongest rivals in the literature, STAMP [96], Quick-Motif [37] and MK [49]. In the limited domain of seismology, we showed that STOMP is at least as fast as the recently introduced FAST algorithm [101], but STOMP does not allow false negatives and does not need careful parameter tuning. Moreover, for datasets and subsequences lengths encountered in the real world, STOMP requires one to three orders of magnitude less memory than rival methods. Thus, even if we are willing to wait a longer period of time for the rival methods to search a large (ten million-plus) dataset, we will almost certainly run out of main memory. Given that these algorithms require random access to the data, disk-based implementations are infeasible. This is not a gap that is likely to be closed by a new implementation of these algorithms, because STOMP is unique among motif discovery algorithms in *not* extracting subsequences, but performing all the computations in-situ.

We also introduced STOMP*I*, the incremental version of STOMP that allows us to maintain time series motifs in an online fashion, and demonstrated its utility in electricity power monitoring.

We further demonstrated optimizations that allow STOMP to take advantage of GPU architecture, opening an even greater performance gap and allowing the first exact motif search in a time series of length one hundred and forty-three-million.

In future work, we plan to investigate the multidimensional version of our algorithms.

Chapter 4 SCRIMP++: An Anytime Algorithm to Compute the Matrix Profile

In previous chapters, we have shown that the Matrix Profile is a flexible and generic data tool to solve a host of time series data mining problems, including motif discovery. There are two algorithms to compute the Matrix Profile, STOMP (Chapter 3), which requires $O(n^2)$ time, and STAMP [96], which is an $O(\log n)$ factor slower. In spite of being slower, STAMP is actually the preferred solution for some applications, as it is a fast converging *anytime* algorithm. In favorable scenarios STAMP needs only to be run to a small fraction of completion to provide a very accurate approximation of the top-*k* motifs. In this chapter we introduce SCRIMP++, an $O(n^2)$ time algorithm that is *also* an anytime algorithm, combining the best features of STOMP and STAMP. As we shall show, SCRIMP++ maintains all the desirable properties of the original algorithms, but converges much faster, in almost all scenarios producing the correct output after spending a tiny fraction of the full computation time. SCRIMP++ further expands the purview of the Matrix Profile and allows us to consider even larger datasets. More critically, SCRIMP++ allows us to perform motif discovery *interactively*, rather than the typical offline *batch* processing that is the norm.

4.1 Motif Analytics: An Insatiable Need for Speed

While all data mining algorithms benefit from improvements in speed, here we argue that for the particular case of motif discovery, improvements in speed are game-changing. Motif discovery benefits from *interactivity* more than most data mining processes. To see this, consider the following analytics session scenario, which while slightly fictionalized, is based on an ongoing project supporting data-intensive entomology [92].

An entomologist wants to examine a five-hour, 1,080,000-point time series (as shown in **Figure 4.1**) she recorded overnight. From her previous experience, she suspects that a motif length of 100, corresponding to one-second, is about the right scale for this insect's behavior to be manifest. However, because she notices the motifs discovered are so well conserved at this scale, she decides to consider two-second long motifs. When she sees these new motifs, she realizes that they correspond to snippets from the *setup* time, when her assistant was adjusting the conductive glue on the insect's back. She therefore crops off the first few minutes and runs motif discovery again. She then…

Figure 4.1. A five-hour sample of Electrical Penetration Graph (EPG) data hints at the difficulty of motif search. See also Figure 4.14/Figure 4.15.

If the entomologist was to use STOMP (Chapter 3), the state-of-the-art exact motif discovery algorithm, then on a modern desktop each run would take about 0.7 hours. This is an important data resource, and a diligent entomologist may find it worth the effort to visit her machine every hour or so, but clearly such long cycle time dashes any

hope of interactively. As [84] notes *"In interactive data analysis processes, the dialogue between the human and the computer is the enabling mechanism that can lead to actionable observations. It is of paramount importance that this dialogue is not interrupted by slow computation*".

As we will show in this work, SCRIMP++ allows us to perform the above analytic workflow *interactively*; in the above scenario, we can reduce the cycle time to just a few seconds.

Beyond the above anecdote that reflects *our* research interests, the literature is replete with examples that suggest the need for faster motif discovery. A recent paper considering several fundamental questions in neuroscience notes that some such questions reduce to determining if neural activity "repeats" happen more than expected by chance [35]. As **Figure 4.2** suggests, these *repeats* are simply time series motifs.

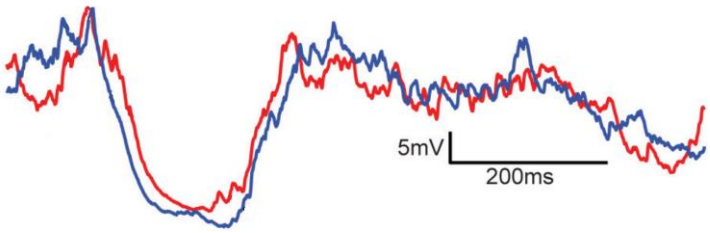

Figure 4.2. Adapted from [35]. "Repeats" in the neuroscience literature are simply time series motifs.

To find such motifs in even a minute's worth of data, the authors resorted to various approximations to "*increase processing speed.*" For example, they downsampled their data by 1 in 10, and rather than use a *sliding* window, they use a "*jumping*" window to reduce the number of comparisons. Even then, the authors noted that to obtain timely

answers their "*repeat-finding algorithm was parallelized and performed on a high-performance computing* (HPC) *cluster*." [35].

However, consider their 2-kHz data, and further assume that we search for their longest motif length of 2.7 seconds (5,400 datapoints), and test *all* possible subsequences (not just "jumping" overlaps) in their largest dataset, which is 8,258,064 data points corresponding to 68.8 minutes of wall clock time.

With an off-the-shelf desktop we can run SCRIMP++ to 1%, in 27.4 minutes, and reproduce their quality of results (cf. [75]). Note that even here, with the original authors' most challenging task, we can still process the data *faster than they can collect it* [35]. The authors go on to bemoan the fact that even with their approximations and use of HPC, that their findings "*represent a lower limit on the duration and prevalence of motifs which might be observed if longer segments of intracellular dynamics could be analyzed*". The algorithm presented in this paper will trivially allow this possibility to be explored, not with batch processing on an HPC, but in real-time interactive sessions on a laptop.

Before moving on, we note that the Matrix Profile has implication for other time series tasks, including discord discovery (Chapter 2), chain discovery (Chapter 6), semantic segmentation [23], etc. While SCRIMP++ can benefit these tasks, for simplicity and concreteness we only consider motif discovery in this chapter.

4.2 Related Work and Background

4.2.1 Definitions

In this chapter, we inherit all the definitions and notations for time series (**Definition 2.1**), time series subsequence (**Definition 2.2**), distance profile (**Definition 2.3**), matrix profile (**Definition 2.4**) and matrix profile index (**Definition 2.5**) from Chapter 2.

4.2.2 Matrix Profile Background

It has been shown in [96] that one can trivially compute all top-*k* motifs (for any *k*), range motifs (for arbitrary ranges), and a host of other useful time series primitives, if one has access to the *matrix profile*. Thus, fast motif discovery simply reduces to fast computation of the matrix profile.

To date there are two algorithms to compute the matrix profile, STAMP [96] and STOMP (Chapter 3).

The STAMP algorithm [96] evaluates the distance profiles (**Definition 2.3**; the columns/rows in **Figure 2.2**) in random order. Each distance profile D_i is evaluated by the MASS algorithm [51], which exploits Fast Fourier Transform (FFT) to calculate the dot product between $T_{i,m}$ and every subsequence in T. The evaluation of a distance profile thus takes $O(n\log n)$ time where n is the length of time series T, and the overall process takes $O(n^2\log n)$ time.

In contrast to STAMP, the STOMP algorithm introduced in Chapter 3 evaluates the distance profiles in **Figure 2.2** in-order by exploiting the computation dependency

between consecutive distance profiles. The algorithm only costs O(n^2) time, an O(log$n$) factor faster than STAMP. STOMP algorithm was forcefully demonstrated as more efficient than the previous state-of-the-art motif discovery algorithms, the Quick-Motif algorithm [37] and the MK algorithm [49] in both time and space (see Section 3.3.3 and Section 3.3.4).

Both STAMP and STOMP maintain the element-wise minimum-so-far values of the evaluated distance profiles in a running matrix profile. Note that although STAMP is an O(log$n$) factor slower than STOMP, it shows better interactivity. As shown in **Figure 4.3**, STAMP is able to locate the highlighted motifs in the time series T when it is only 10% completed, as the running matrix profile already contains two deep valleys in the vicinity of the motifs. In contrast, STOMP cannot locate the motifs even when it is 50% completed (no deep valleys show up), because the running matrix profile converges to the oracle from left to right in order.

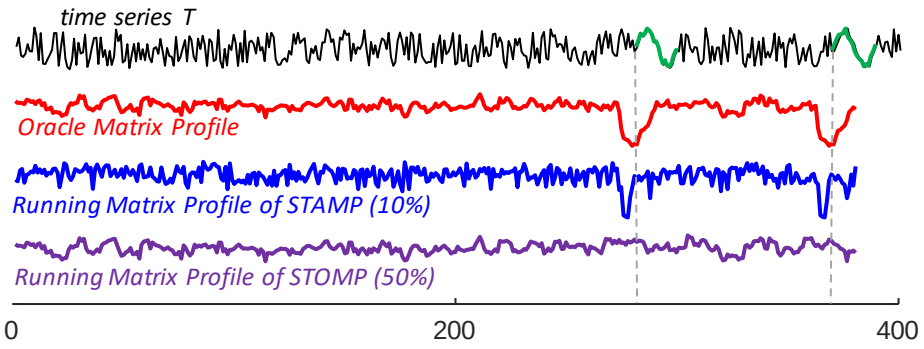

Figure 4.3. STAMP is able to detect the motifs located towards the right side of a time series when it is only 10% completed due to its random computation order. In contrast, STOMP's left-to-right sequential computation means it cannot detect them even when 50% completed.

However, when the time series is very long and motifs are rare, the probability of STAMP finding the top-k motifs within 10% of its computation greatly decreases.

Furthermore, as STOMP is a factor of O(log*n*) faster, by the time STAMP has completed 10% of its computation, STOMP may already converge to the exact solution. These conflicting strengths of the two algorithms require careful reasoning by the analyst, based on her goals and her tentative knowledge of the data. SCRIMP++ eliminates any dilemma, by combining the speed of STOMP with the anytime convergence property of STAMP.

4.2.3 General Motif Search

It is important to make the distinction between *approximate* algorithms (of which there are many, see [81] for a survey) and *anytime* algorithms for motif discovery [96]. Suppose a user runs a fast, but approximate algorithm on a large dataset. It is possible that when the motifs are returned, she is satisfied. However, suppose that the motifs are *not* as well conserved as she expected, given her domain knowledge and her intuitions for the data. She is now in a quandary, are the expected motifs simply not there, or did the algorithm fail to find them? The problem is compounded by the fact that no approximate motif discovery algorithm we are aware of come with any kind of probabilistic guarantees, and all require at least three unintuitive parameters to be set [81]. What can our user do? If the approximate algorithm was stochastic, she can run it again, and/or change the parameters, but she may repeatedly face the same problem. Otherwise, she is condemned to run the fastest *exact* algorithm she has access to (which is STOMP).

If the approximate algorithms took a tiny fraction of the time of the best exact algorithm, this issue would require some careful reasoning about trade-offs. However, as

we will show in Section 4.4.3, all approximate algorithms take a large fraction of the time needed by SCRIMP++, especially for longer motifs.

For this reason, we argue that an *anytime* algorithm is necessary. In most cases, in a few seconds the user has acceptable results. If she has any doubts, she can simply let the algorithm run a little longer. There is no need to start the fastest *exact* algorithm, because it is already running!

Finally, we need to qualify the claim that STOMP is the fastest exact algorithm for motif discovery. On "cooperative data" (relatively smooth data, motifs highly conserved relative to the rest of the data, short motif lengths etc.), other exact algorithms such as Quick-Motif [37], IMD [24], or MK [49] *can* be fast. But in less-than-cooperative data (e.g., the seismology data in Chapter 3) these algorithms degenerate to $O(n^2m)$, with very high constant factors. The authors of [24] are to be commended for stating this explicitly "…*in the worst case, the algorithm still has a time complexity of* $O(n^2m)$".

As we show in our case studies (see **Figure 4.14**), m can be as large as 15,000 or greater for real-world problems. In contrast STOMP (and SCRIMP++) takes $O(n^2)$ time, completely independent of the data and the value of m. Thus, for realistic problems with high dimensionality, STOMP can be thousands of times faster than Quick-Motif [37], IMD [24], or MK [49].

4.3 Algorithms

The SCRIMP++ Algorithm consists of two parts: PreSCRIMP and SCRIMP (as shown in **Figure 4.4**). In this section, we will first introduce the SCRIMP algorithm,

which is an O(n^2) anytime algorithm with better convergence characteristics than STOMP (Chapter 3). We will then further extend SCRIMP to SCRIMP++, a robust anytime algorithm which, thanks to the addition of an ultra-fast preprocessing algorithm PreSCRIMP, is capable of detecting essentially all the motifs within a time series at an early stage, even when the motifs are subtle and/or extremely rare. For simplicity we only consider self-join here; however, all the algorithms introduced can be easily extended to AB-join [96].

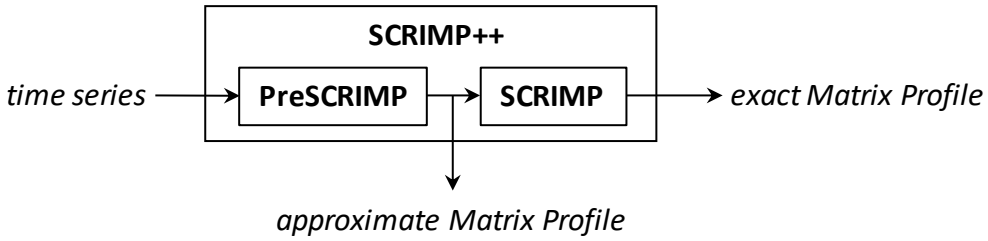

Figure 4.4. The SCRIMP++ algorithm consists of an ultra-fast preprocessing algorithm, PreSCRIMP, and an O(n^2) anytime algorithm, SCRIMP. PreSCRIMP provides a very accurate approximation of the matrix profile at an early stage; SCRIMP further refines the approximate matrix profile until it becomes the exact/final solution. The user can interupt the algorithm at any time (during either PreSCRIMP or SCRIMP) to inspect the current approximate solution. Thus overall, SCRIMP++ is also an anytime algorithm.

4.3.1 Our Initial Solution: The SCRIMP Algorithm

Before we introduce the SCRIMP algorithm, let us first briefly review the STOMP algorithm (Chapter 3). Based on (3.1) and (3.4), the STOMP algorithm evaluates the distance matrix in **Figure 2.2** row-by-row in-order and updates the matrix profile accordingly, rendering an O(n^2) time complexity. However, as indicated in **Figure 4.3**, this in-order computation prevents motifs at the end of a time series from being discovered at an early stage. Can we fix this undesirable property?

Note that (3.4) also implies that we can evaluate the *diagonals* of the distance matrix in **Figure 2.2** in random order. The SCRIMP algorithm (**Algorithm 4**) exploits this, evaluating the matrix profile in an anytime fashion while keeping the same $O(n^2)$ time complexity.

Algorithm 4: SCRIMP(T, m)

Input: A time series $\boldsymbol{T}$ and a subsequence length $\boldsymbol{m}$
Output: Matrix profile $\boldsymbol{P}$ and the associated matrix profile index $\boldsymbol{I}$ of T

```
1  n ← Length(T)
2  μ, σ ← ComputeMeanStd(T, m)                    // see [61]
3  P ← infs, I ← ones                             // initialization
4  Orders ← RandPerm(m/4+1 : n-m+1)               // randomize evaluation order
5  for k in Orders        //evaluating diagonals in random order
6     for i ← 1 to n-m+2-k
7        if i=1 do q ← DotProduct(T_{1,m}, T_{k,m})
8        else q ← q - t_{i-1} t_{i+k-2} + t_{i+m-1} t_{i+k+m-2}        // see (3.4)
9        end if
10       d ← CalculateDistance(q, μ_i, σ_i, μ_{i+k-1}, σ_{i+k-1})     // see (3.1)
11       if d < P_i do P_i ← d, I_i ← i+k-1 end if
12       if d < P_{i+k-1} do P_{i+k-1} ← d, I_{i+k-1} ← i end if
13    end for
14 end for
15 return P, I
```

Line 2 precomputes the means and standard deviations of all subsequences in T. The matrix profile P and matrix profile index I are initialized in line 3. In lines 5-14, we iteratively evaluate the diagonals of the distance matrix in **Figure 2.2** in random order. **Figure 4.5** visualizes this. The distance values $d_{1,k}$, $d_{2,k}$, …, $d_{n-m+2-k,n-m+1}$ are calculated one by one; if $d_{i,i+k-1}$ (denoted as d in line 10, $1 \leq i \leq n-m+2-k$) is smaller than P_i (line 11) or P_{i+k-1} (line 12), we update the corresponding matrix profile (and index) values. At any time, the user can interrupt the algorithm to inspect the current P and I.

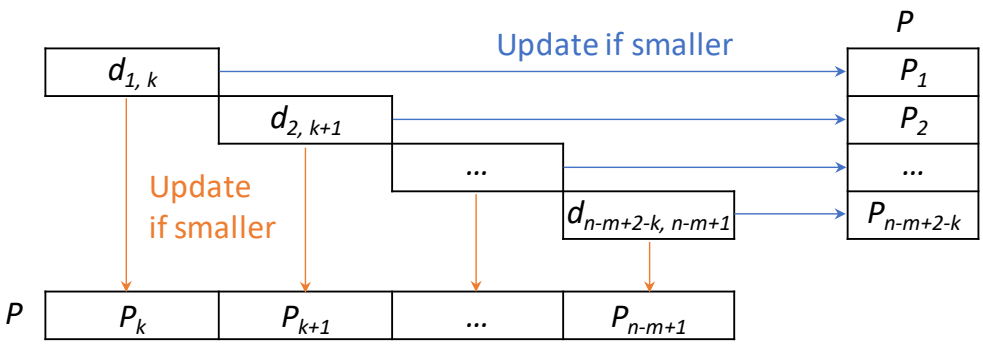

Figure 4.5. A single iteration of SCRIMP evaluates a randomly selected diagonal in Figure 2.2, thus updating the matrix profile in an *anytime* fashion.

4.3.2 Limitations of the SCRIMP Algorithm

As motifs in the time series correspond to the minimum points of the *oracle* (or *exact*) matrix profile (indicated in **Figure 4.6**.*top*), we hope that SCRIMP could "focus" on these minimum points rather than at other locations. This has an element of a *chicken-and-egg* paradox to it, we want the algorithm to focus on where the motifs are, but we are using the algorithm to discover where the motifs are.

Recall that in each iteration of SCRIMP (as shown in **Figure 4.5**), we evaluate a random diagonal of the distance matrix. To locate the motifs of time series T in **Figure 4.6**.*top*, we need to evaluate the diagonal starting from $d_{1,126}$ (126-1=137-12) as early as possible. As shown in **Figure 4.6**.*middle*, if SCRIMP evaluates that diagonal in its first iteration, the running matrix profile already overlaps perfectly with the oracle at the minimum points. However, if SCRIMP does not evaluate that diagonal until its very last iteration (**Figure 4.6**.*bottom* shows the running matrix profile before the last iteration), we need to wait until the algorithm is 100% completed to locate the motifs. In fact, the probability to evaluate the diagonal of $d_{1,126}$ before the kth iteration is $k/(n-m+1)$. While

SCRIMP has a chance to find the motif early no matter where they are located (which is its advantage over STOMP), that probability is not high.

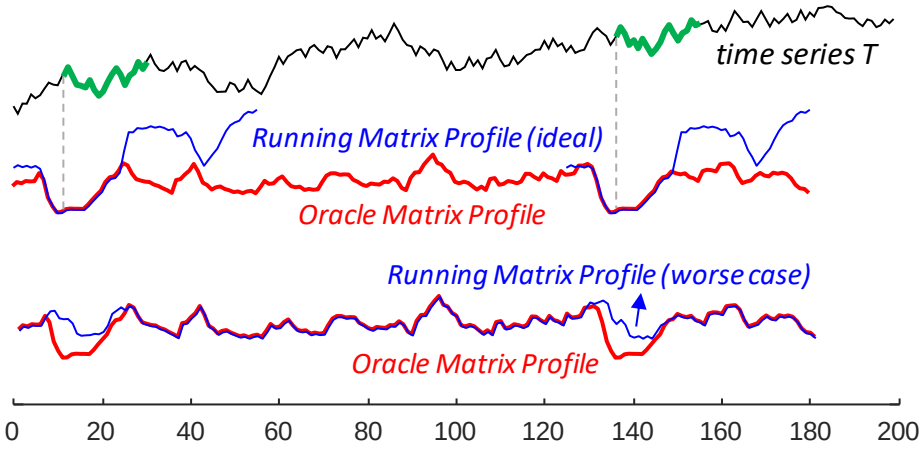

Figure 4.6. ***top)*** **Motifs (highlighted, located at 12 and 137) correspond to the minimum values of the matrix profile.** ***middle)*** **Ideally, SCRIMP can locate the motifs after its first iteration.** ***bottom)*** **In the pathological worst case, SCRIMP cannot locate the motifs until fully completed.**

However, note that **Figure 4.6**.*top* shows the hardest possible scenario for motif discovery; there is only a single pair of motifs in the time series. When the data contain more motifs, SCRIMP will perform much better. This is much like how the famous birthday paradox has an unexpectedly fast converge to probability 1 as we consider more individuals. The chance of SCRIMP making an early discovery of some pair from a motif set, increases dramatically if there are more members in that motif set. In the next section, we will introduce SCRIMP++, an extended version of SCRIMP which has a much higher probability of discovering not *some*, but *all* the true motifs at an early stage, even when the motifs are very rare.

4.3.3 Our Ultimate Solution: The SCRIMP++ Algorithm

The SCRIMP++ Algorithm is simply the SCRIMP algorithm (**Algorithm 4**) augmented by an additional preprocessing stage called PreSCRIMP (recall **Figure 4.4**). We begin by introducing the Consecutive Neighborhood Preserving Property of time series subsequences, upon which PreSCRIMP is based.

Let us examine the matrix profile index of the example time series T in **Figure 4.6**.*top*. **Figure 4.7** shows its first 25 entries.

Index	1	2	3	4	…	7	8	9	…	24	25	…
I	56	57	112	113	…	116	133	134	…	149	150	…

Figure 4.7. The matrix profile index of time series *T* in Figure 4.6.*top*.

Here *Index* = [1, 2, 3, …, n-m+1] is the locations of all the subsequences in T, I is the matrix profile index (**Definition 2.5**) of T. We can see that the matrix profile index can be divided into multiple sections of consecutive values: within each section, a set of consecutive subsequences find another set of consecutive subsequences as their nearest neighbors. We call this the *Consecutive Neighborhood Preserving (CNP) Property* of time series subsequences.

With a little introspection, one can see that the CNP property *should* exist: since consecutive subsequences overlap by a large portion, if the i^{th} subsequence is very similar to the j^{th} subsequence, then there is a very high probability that the $(i+1)^{th}$ subsequence is also very similar to the $(j+1)^{th}$ subsequence. In **Figure 4.8**, we can see that the 11^{th}, 12^{th}, 13^{th}, and 14^{th} subsequences find the 136^{th}, 137^{th}, 138^{th} and 139^{th} subsequences as their

nearest neighbors, respectively; the subsequence-neighbor pairs remain a constant location difference of 125.

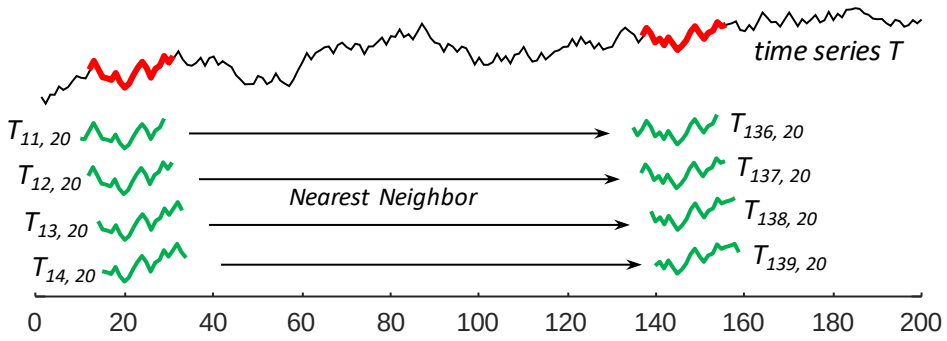

Figure 4.8. Visualizing the CNP property of time series subsequences in the vicinity of the 1st motif pattern.

Exploiting the CNP property, we propose a preprocessing algorithm *PreSCRIMP*, that produces a very close approximation of the oracle matrix profile while costing only a tiny fraction of its O(n^2) computation time. Essentially, we sample subsequences from the time series with a fixed interval s (**Figure 4.9**.*top* shows the starting location of these sampled subsequences). For each sampled subsequence, we find its exact nearest neighbor. Assume that $T_{i,\mathrm{m}}$ is a sampled subsequence, and its nearest neighbor is $T_{j,m}$, then according to the CNP property, there is a high probability that the nearest neighbor of $T_{i+k,\mathrm{m}}$ is $T_{j+k,m}$ (k=*-s+1, -s+2, …, -2, -1, 1, 2, …, s-2, s-1*). We compute the distances between these pairs of subsequences and update the matrix profile if a smaller distance value shows up.

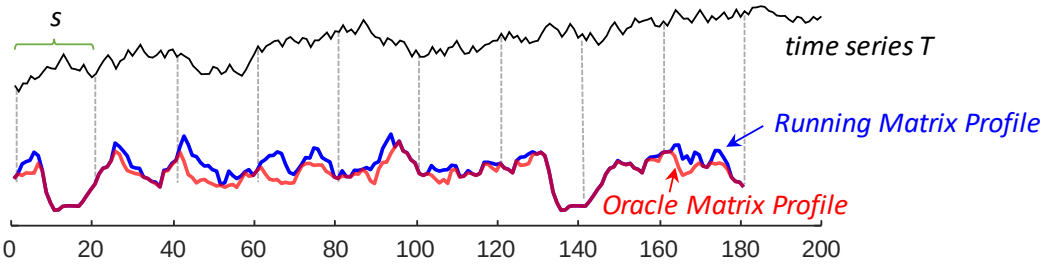

Figure 4.9. *top*) Subsequences are sampled from time series *T* with a fixed interval *s*. *bottom*) After running PreSCRIMP, the running matrix profile becomes *very* similar to the oracle matrix profile, especially at the low values we care about.

The overall algorithm is outlined in **Algorithm 5**. Line 2 precomputes the means and standard deviations of all subsequences in *T*. In line 3, we sample subsequences from time series *T* with a fixed interval *s* (**Figure 4.9**.*top* shows their starting position), then process these subsequences in random order. Each sample subsequence is processed with two stages (lines 4-22).

In the first stage (lines 4-7), we evaluate the distance profile corresponding to the current sample subsequence $T_{i,m}$ with the MASS algorithm [51], then update the running matrix profile (and index) if we find a smaller distance value. Note that after this stage, we already know the nearest neighbor of $T_{i,m}$ (assume it is $T_{j,m}$), and the matrix profile and matrix profile index are exact at the *i*th entry. As a result, we can see from **Figure 4.9**.*bottom* that the running matrix profile aligns perfectly with the oracle matrix profile at the sampled locations.

In the second stage (lines 8-22), we refine the running matrix profile (and index) near the i^{th} entry by exploiting the CNP property. Starting from the current sample subsequence $T_{i,m}$ and its nearest neighbor $T_{j,m}$, we move forward to evaluate the pairwise distances between ($T_{i+1,m}$, $T_{j+1,m}$), ($T_{i+2,m}$, $T_{j+2,m}$), …, until we reach the next sampled

location or the end of the time series (lines 10-15). After that, we traverse backward from $T_{i,m}$ and $T_{j,m}$ to evaluate the pairwise distance between ($T_{i-1,m}$, $T_{j-1,m}$), ($T_{i-2,m}$, $T_{j-2,m}$), …, until we reach an earlier sampled location or the beginning of the time series (lines 17-22). The corresponding running matrix profile (and index) entries are updated once we find a smaller distance value.

Algorithm 5: PreSCRIMP(*T*, *m*, *s*)

Input: A time series ***T***, a subsequence length ***m*** and a sampling interval ***s***
Output: The running matrix profile ***P*** and matrix profile index ***I*** of T

```
1  n ← Length(T), P← infs, I← ones // initialization
2  μ, σ ← ComputeMeanStd(T, m)   // precomputation, see [61]
3  for i ← RandPerm(1 : s : (n-m+1)) do   //sampling with interval s
4     seq ← T_{i,m}    //obtain a sample subsequence
5     D ← MASS(T, seq)   // evaluate a distance profile, see [51]
6     P, I ← ElementWiseMin(D, P, i)
7     P_i , I_i ← min(D)
8     j ← I_i    // the nearest neighbor of the sample subsequence
9     q ← CalculateDotProduct(P_i, μ_i, σ_i, μ_j, σ_j), q' ← q  // see (3.1)
10    for k ← 1 to min(s-1, n-m+1- max(i,j)) do
11       q ← q - t_{i+k-1} t_{j+k-1} + t_{i+k+m-1} t_{j+k+m-1}    // see (3.4)
12       d ← CalculateDistance(q, μ_{i+k}, σ_{i+k} , μ_{j+k}, σ_{j+k}) // see (3.1)
13       if d < P_{i+k} do P_{i+k} ← d, I_{i+k} ← j+k end if
14       if d < P_{j+k} do P_{j+k} ← d, I_{j+k} ← i+k end if
15    end for
16    q ← q'
17    for k ← 1 to min(s-1, i-1, j-1) do
18       q ← q - t_{i-k+m} t_{j-k+m} + t_{i-k} t_{j-k}    // see (3.4)
19       d ← CalculateDistance(q, μ_{i-k}, σ_{i-k} , μ_{j-k}, σ_{j-k}) // see (3.1)
20       if d < P_{i-k} do P_{i-k} ← d, I_{i-k} ← j-k end if
21       if d < P_{j-k} do P_{j-k} ← d, I_{j-k} ← i-k end if
22    end for
23 end for
24 return P, I
```

The overall time complexity of the algorithm is $O(n^2 \log n/s)$, where n is the length of the time series and s is the sampling interval. The space complexity is $O(n)$. From **Figure 4.9**.*bottom*, we can see that after running PreSCRIMP, the running matrix profile

aligns very well with the oracle matrix profile, especially at the minimum points, which for motif discovery, are all we care about.

The reader may wonder how we determine the sampling interval *s*. Note that any unsampled subsequence must overlap with one of the sampled subsequences by at least $1-s/(2m)$. Therefore, the smaller *s* is, the more accurate is our running matrix profile (and the longer PreSCRIMP takes to compute it). As a practical matter (as we will demonstrate later in Section 4.4), we set $s=m/4$, which guarantees that all the subsequences overlap with at least one sampled subsequence by at least 87.5%. This setting renders PreSCRIMP an $O(n^2 \log n/m)$ time complexity. As the subsequence length *m* is normally much larger than log*n*, the time needed for PreSCRIMP is a tiny fraction required for SCRIMP/STOMP.

After running PreSCRIMP, we continue to refine the matrix profile with SCRIMP, until it converges to the exact solution. We call the augmentation of SCRIMP with PreSCRIMP, *SCRIMP++* (recall **Figure 4.4**). Note that SCRIMP++ can be interrupted at any stage (including during the PreSCRIMP stage), to produce an approximate solution.

4.4 Empirical Evaluation

To ensure that our experiments are reproducible, we have built a website which contains all data/code/raw spreadsheets for the results, in addition to many experiments that are omitted here for brevity [75]. All experiments were run on a Dell XPS 8920, with Intel Core i7-7700 CPU @ 3.6GHz and 64GB RAM.

4.4.1 Comparing Convergence Behaviors

We begin by comparing the convergence behavior of STAMP [96], STOMP (Chapter 3) and SCRIMP++. Note that STOMP is not regarded as a true anytime algorithm but is included for completeness.

To stress-test these algorithms with different circumstances (different numbers and locations of motifs, different data type, etc.), we created four different synthetic datasets. **Figure 4.10** shows one example from each of the four datasets.

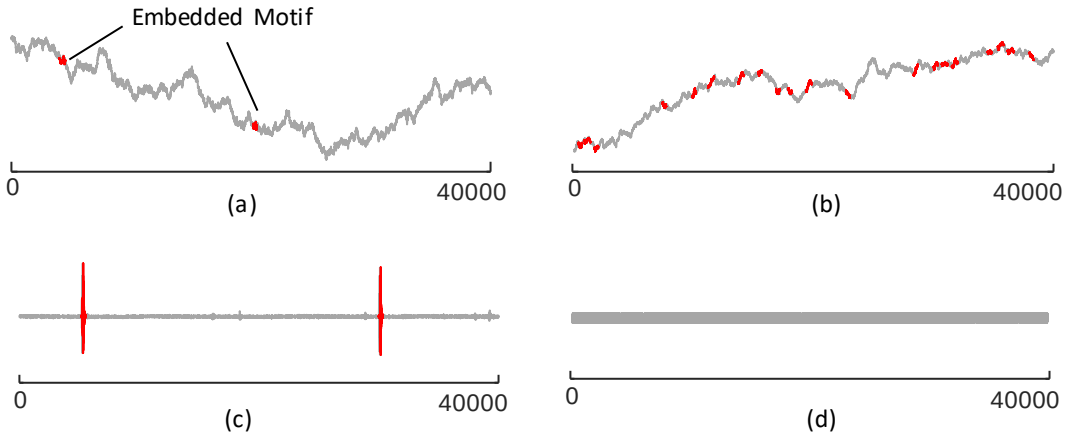

Figure 4.10. *a*) Random-walk data with one pair of embedded random-walk motif patterns. *b*) Random-walk data with 10 embedded random-walk motif pairs. *c*) Seismology data with two repeated earthquake signals. *d*) Random noise without any embedded motif patterns.

Each dataset includes 100 time series of length 40,000. Within each time series we embed various numbers of motif patterns of length m=400 *at random locations*. The first dataset (**Figure 4.10**.*a*) is a set of random-walk time series; within each of these time series we embed a single pair of random-walk motif patterns (they are similar, but not identical). The second dataset (**Figure 4.10**.*b*) is also random-walk data, but contains 10 pairs of different random-walk motif patterns. The third dataset (**Figure 4.10**.*c*) is adapted from Section 3.3.4, where we have a continuous recording of seismograph

background noise, and embed in it one pair of repeated earthquake signals (similar but not identical) at random locations. The fourth dataset (**Figure 4.10**.*d*) is random noise time series; we embed no motifs in it, but regard its natural top-1 motif pattern of length 400 as target.

As the algorithms evaluate the matrix profile of a time series, we constantly interrupt it, mark the current runtime *t*, then extract the top-*k* motif patterns (we set *k = 10* for **Figure 4.10**.*b; k = 1* for **Figure 4.10**.*a*, **Figure 4.10**.*c* and **Figure 4.10**.*d*) from the running matrix profile and check to see if the embedded motif patterns have been discovered. We regard an embedded motif pattern as discovered if it overlaps with one of the *k* extracted motif patterns by at least 95%. We use a value *p* to represent the percentage of embedded motif pairs discovered at each time instant *t*.

We first consider **Figure 4.10**.*b*, where the random-walk time series includes 10 pairs of embedded motifs. **Figure 4.11** shows the average value of *p* as the three algorithms search for motifs.

We can see that SCRIMP++ shows much faster convergence characteristics than STAMP or STOMP in locating the top 10 motif pairs. After the PreSCRIMP phase (requiring only 0.26 seconds) finishes, all the 10 embedded motifs randomly located in all 100 random-walk time series are successfully discovered. In contrast, to be just 99% sure that we have discovered all the true motifs, STAMP takes about 8 times longer and STOMP needs to almost run to completion (about 9 times longer).

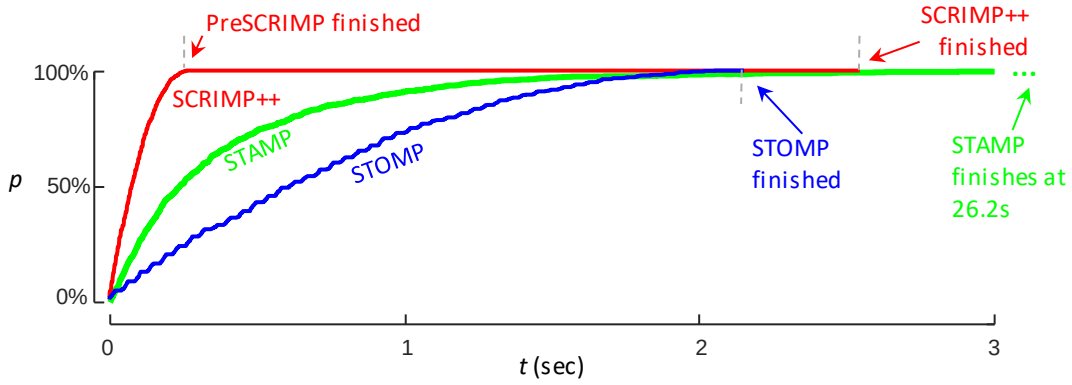

Figure 4.11. The average percentage of embedded motif pairs discovered at each time instant for the dataset shown in Figure 4.10.*b*. Note that the time for STAMP's convergence is truncated.

Now let us consider the harder scenarios in **Figure 4.10**.*a*, **Figure 4.10**.*c* and **Figure 4.10**.*d*, where there are only one pair of motif patterns in the data. We experimented in these scenarios because: 1) The top-1 motif in these datasets are hard to locate as they are rare. 2) The seismology data in **Figure 4.10**.*c* is a typical example of "less-than-cooperative" data discussed in Section 4.2.3, which would degenerate rival motif discovery methods such as Quick-Motif [37] or MK [49] to their worst case time complexity (recall **Figure 3.1**). 3) The random-noise data in **Figure 4.10**.*d* shows an extremely hard case for motif discovery, as essentially all pairs of time series subsequences are approximately equidistant. Nevertheless, as shown in **Figure 4.12**, SCRIMP++ shows a very fast convergence characteristic in all these datasets. After the PreSCRIMP phase is completed (0.26 seconds), all the top-1 motifs in all the time series within all three datasets are already successfully discovered, costing only a tiny fraction of time needed by STOMP or STAMP. Note that here STAMP does not perform as well as in **Figure 4.11**, as the motifs are very rare.

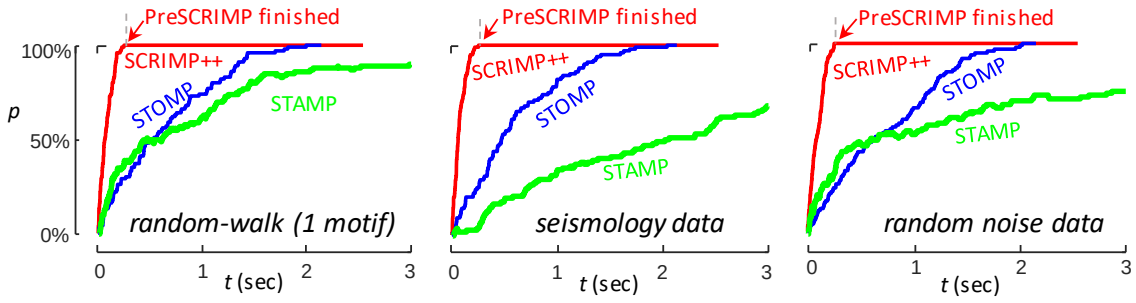

Figure 4.12. ***left-to-right)*** **The observed probability for the top-1 motif discovered at each time instant for the dataset shown in Figure 4.10.*a*, Figure 4.10.*c* and Figure 4.10.*d*. Note that the full time for STAMP's convergence is truncated.**

As we show in the next section, SCRIMP++ maintains this advantage over different lengths of time series and motif lengths. We chose to consider 40,000 data points here, because based on our informal survey of practitioners that *use* motif discovery, this is about the median size of datasets[2] considered [10][79]. Here we can find such motifs in just ¼ of a second, truly interactive time [84].

4.4.2 Runtime Comparison of SCRIMP++ and STOMP

In this section, we compare the run time of SCRIMP++ with the state-of-the-art exact motif discovery algorithm, STOMP (Chapter 3). The time measurements are based on the C++ implementation of both algorithms. Note that the runtime for both algorithms is invariant to the type of time series data we are using. **Table 4.1** shows the time required by both algorithms with a fixed subsequence length *m*, on random noise time series with increasing length *n*.

[2] To be clear, many biologists produce terabytes of data, but often each "run" or "treatment" is only of the order of tens to hundreds of thousands in length.

Table 4.1. Time Needed for Motif Discovery with m = 4096, varying n

Algorithm \ n		2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
STOMP		22.5 sec	1.78 min	7.37 min	37.1 min	2.22 hours
SCRIMP++	PreSCRIMP	0.51 sec	2.33 sec	17.2 sec	1.52 min	6.83 min
	SCRIMP	23.9 sec	1.94 min	7.96 min	40.9 min	2.46 hours

We can see that the runtime of the SCRIMP Algorithm is similar to the STOMP algorithm, as they vary only in evaluation order. The PreSCRIMP algorithm consumes only a very small fraction (less than 6%) of their time[3].

In **Table 4.2**, we fixed the time series length n and vary subsequence length m. We can see that the runtime of STOMP and SCRIMP are essentially invariant to the subsequence length m. PreSCRIMP, with a time complexity $O(n^2 \log n/m)$, costs less and less time while we increase m. As m can be in the thousands for real-world problems (cf. Sections 4.4.4-4.4.6), this is a desirable feature.

Table 4.2. Time Needed for Motif Discovery with $n = 2^{18}$, varying m

Algorithm \ m		1024	2048	4096	8192	16384
STOMP		1.83 min	1.78 min	1.78 min	1.8 min	1.67 min
SCRIMP++	PreSCRIMP	9.22 sec	4.81 sec	2.33 sec	1.23 sec	0.58 sec
	SCRIMP	2.17 min	2.12 min	1,94 min	2.05 min	1.96 min

Furthermore, as PreSCRIMP is based on iterative vector operations, the computation process is highly parallelizable. Implementing PreSCRIMP with high-performance computing platforms such as GPU is trivial, and we make the GPU version freely available at [75].

[3] Note that though we could further speed up PreSCRIMP with multi-threading or piece-wise FFT, we reported its run time here without any of these optimizations.

4.4.3 Comparison to Rival Methods

We have argued that there is a critical difference between *approximate* algorithms and *anytime* algorithms for motif discovery. By definition, anytime algorithms are also approximate algorithms (if stopped early), but the converse is not true. If the motifs returned by an approximate algorithm are not satisfactory, the user has no recourse but to adjust parameters and try again, or resort to the fastest exact algorithm STOMP (Chapter 3).

Nevertheless, it may be instructive to compare our proposed algorithm to the state-of-the-art *approximate* algorithm. But which algorithm *is* state-of-the-art for this task? A recent survey reviews more than a dozen algorithms without explicitly answering that question [81]. Fortunately, we can bypass this issue, and effectively compare to *all* of them. All such algorithms, whether they use hashing, grammars, Markov models, suffix trees etc. [81], must first convert the data into a symbolic representation. The time taken to do this is clearly a lower bound on the time to produce any motifs. Note that we cannot bypass this time requirement with any precomputation/indexing, as this is only possible if one knows the length of motifs, but as we have shown, this can be changed in an ad-hoc manner during the user's interactive session.

We used the code written by L. Wei [91] (which is the code used by the majority of papers reviewed in [81]), to discretize increasingly long time series, while keeping *m* fixed to 4,096 and a dimensionality of 8 and cardinality of 5 (typical values for most research efforts [81]). As **Figure 4.13** shows, we compare the runtime of this preprocessing discretization step of the rival algorithms to that of PreSCRIMP.

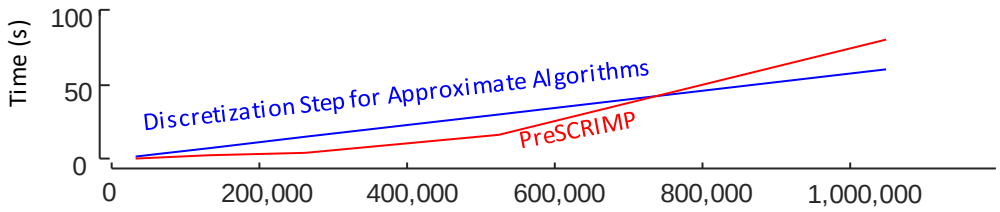

Figure 4.13. The time needed to discretize data and the time needed to perform PreSCRIMP for increasingly long data.

We can see that when the time series length is smaller than 2^{19}, SCRIMP++ has already reported a very high-quality solution with PreSCRIMP, before any approximate algorithm is even in a position to finally start the hashing or suffix tree construction that they hope will yield an approximate answer.

Note that this experiment offers an extremely weak lower bound for the cost of the rival approximate algorithms. In practice, the searching such algorithms take is 3 to 20 times longer than this preprocessing [81]. Finally, all these methods are reporting motifs found in a lossy data representation with the inherent error that produces, whereas SCRIMP++ is searching the *original* data.

4.4.4 Case Study: Multiscale-Motifs

We believe that the extraordinary speed of PreSCRIMP will allow the community to invent novel time series primitives. To give an example, we consider a question suggested by an entomologist collaborator: are there any *multiscale-motifs* in the EPG datasets [92] previously discussed in Section 4.1? We informally define a multiscale-motif as a pair of patterns that are very similar to each other but differ by at least a factor of two in length.

Clearly finding multiscale-motifs is computationally challenging, because beyond comparing all pairs of subsequences, we must now compare all pairs of subsequences, *and* at all possible combinations of scales. It may be possible to create a scalable novel algorithm to find multiscale motifs, but the speed of PreSCRIMP suggests a very easy "fast-enough" method that we can implement in a handful of lines of code, given PreSCRIMP as a primitive.

Recall we can use PreSCRIMP to do self-joins or AB-joins. Suppose we set B = rescale(A,300%) and compute an AB-join. The resulting motifs discovered will reflect a short pattern in A that matches a much longer pattern in B, after the patterns are scaled to a common length. In this case, we do not know what the "right" rescaling length is, but PreSCRIMP is fast enough to allow us to run it fifty times and simply test all possible scalings from 200% to 300%, in 2% increments. We have done this for a 1.8 hour (650,000 datapoints) long trace of Asian citrus psyllid (*Diaphorina citri*) feeding on Citrangor, a subspecies of orange. **Figure 4.14** shows that the best multiscale-motif occurs for a rescaling of 218%.

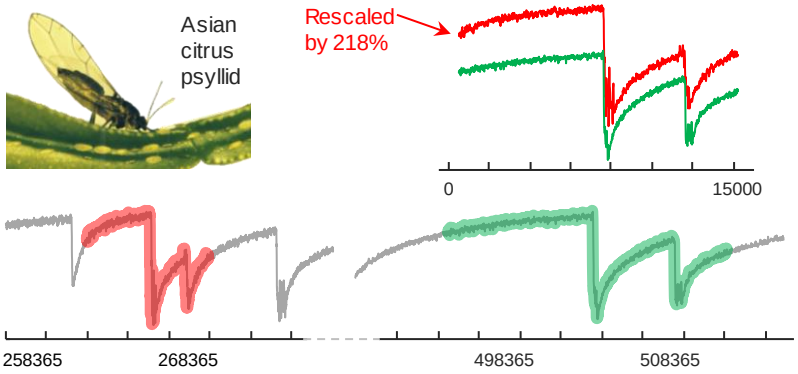

Figure 4.14. ***top.left*****) An Asian citrus psyllid feeding on a citrango leaf.** ***top.right*****) The top-1 multiscale-motif discovered.** ***bottom*****) the two motif occurrences in context.**

Note that one may wish to normalize the Euclidian distance for length when comparing multiscale-motifs (it happens to make no difference in this case). Further note that we are not claiming any particular entomological significance here, although it is interesting that this insect has behaviors that manifest themselves at such different scales. Our point is simply to show that PreSCRIMP is fast enough to be considered as a primitive we can call multiple times for higher-level analytics. The time taken for this entire experiment was just 84 seconds (m=15,000).

This ability to handle motifs that occur at different lengths may also be of interest to the neuroscience/neuroinformatics community, which has recently adopted time series motifs as one of their most used analytic tools [10][79]. However, some of these authors have criticized current motif discovery algorithms because they "*considers only exactly equal duration sequences as potential matches*" [79]. The authors of [79] note that motifs of "turning maneuvers" of *Drosophila* larval have a variable length scale, with μ = 0.83s and σ = 0.27s. Using the simple algorithm described above, we can find multiscale motifs in the range of $\mu \pm 2\sigma$ in a dataset of 40,000 points, searching *all* rescalings in 5% increments ([35%, 40%, … ,160%, 165%]) in just 17 seconds.

4.4.5 Case Study: Motif Joins

The EPG domain considered in the previous section is a rich source of fundamental problems that can be addressed with motif discovery, below we consider another such problem.

As shown in **Figure 4.15**.*top*, we consider three datasets, each of length 7,560,000, representing 21-hours of insect behavior. One of them, in which the insect was feeding on

Valencia (a type of orange), we designated as reference sample, $\text{Valencia}_{\text{Ref}}$. We are interested to know if any elements of this reference behavior are to be found in the two other datasets, in one of which an insect was feeding on a Yamaguchi (a different type of citrus), and the other in which a different insect is feeding on a Valencia. We hope to understand what elements of the Asian citrus psyllid may be attributed to the type of plant it is feeding on, and what may be attributed to simple differences between individuals. Such studies have implications for breeding resistant strains and hybrids.

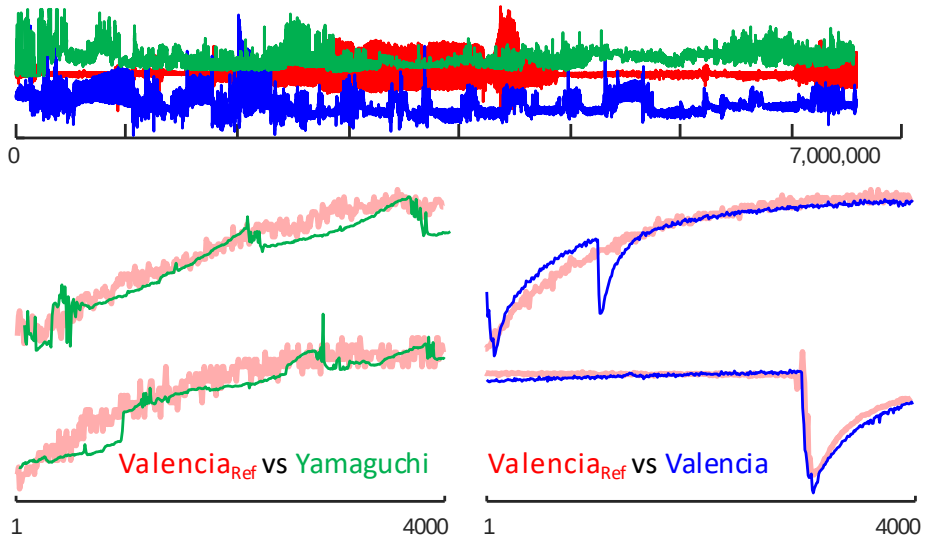

Figure 4.15. ***top*****) The three EPG time series under investigation.** ***bottom-left to right*****) There is little evidence of conserved patterns when the insects are feeding on different citrus plants, but there are strongly conserved patterns when feeding on a single plant type.**

It is instructive to think of the cost of a brute-force-search here. The motifs are of length 4,000, requiring (at least) 4,000 real-valued operations. Each AB-join requires about $5.71 * 10^{13}$ pairwise comparisons of subsequences, requiring $2.28 * 10^{17}$ real-valued operations. Even at one hundred gigaFLOPS, this would require 26.4 days. In contrast, PreSCRIMP took just 2 hours.

4.4.6 Case Study: Electrical Power Demand

As a final example of the scalability of SCRIMP++, and the potential actionability of motif discovery, we examined the electrical power demand dataset of [62]. Each trace corresponds to two calendar years or 8,198,756 datapoints, sampled once every 8 seconds. As shown in **Figure 4.16**, a pair of motifs from trace 3 of House-5 caught our attention.

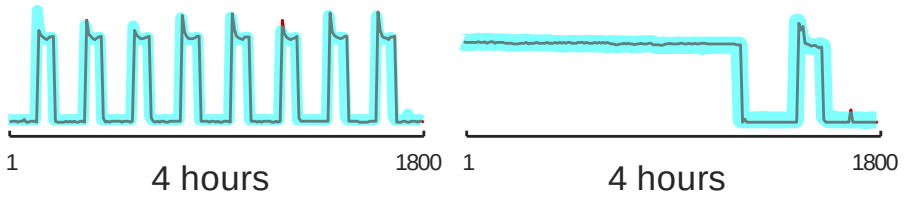

Figure 4.16. The top two motifs in an electrical power data set.

The first motif is the (near) binary switching on-and-off of a freezer compressor at very regular intervals. This unusually "perfect" motif has dozens of occurrences, almost all at night when there is no kitchen actively that would cause the compressor to "kick-in" after the freezer was opened and disrupt the perfect spacing. The second motif is more interesting. It suggests that the compressor was running continuously for at least three hours. Two common causes of a freezer motor running for a long time are a faulty thermostat, or the more prosaic explanation, the homeowner not fully closing the door. In either case this is clearly a low-hanging fruit for energy conservation.

SCRIMP++ allows us to find such patterns in real-time interactive sessions, something that no other tool allows [81].

4.4.7 When can PreSCRIMP fail?

The previous sections have shown the extraordinary alacrity and effectiveness of PreSCRIMP. To explore the limits of PreSCRIMP, in Section 4.4.1, we considered all the possible worst-case scenarios: when the motifs are very rare, when the dataset is of very high intrinsic dimensionality, when all the subsequence pairs are equidistant, etc. Nevertheless, PreSCRIMP succeeded in quickly locating all the true motifs in all these scenarios. It is natural to ask, can PreSCRIMP ever fail? Do we *ever* need to resort to running the SCRIMP phase of the SCRIMP++ algorithm, to refine the PreSCRIMP answer?

In spite of a diligent search of over 100 diverse datasets, we could not find any *real* dataset that prevents PreSCRIMP from quickly discovering motifs. However, with careful introspection, we can *create* a pathological example that is difficult for PreSCRIMP. As shown in **Figure 4.17**.*top*, we created a synthetic random walk time series of length 40,000, with a pair of motifs embedded at fixed locations ($T_{21842,400}$ and $T_{24871,400}$, shown in red and yellow respectively). We edited the first/red motif pattern such that *just* before and after the pattern, the level of the time series dramatically changed. In this scenario, the CNP property no longer hold at locations around the motif patterns. Though $T_{21842,400}$ is very similar to $T_{24871,400}$, $T_{21842+k,400}$ is very different from $T_{24871+k,400}$ (k=*-3, -2, -1, 1, 2, 3*) because of the dramatic level change. As a result, PreSCRIMP cannot discover the motif pair unless either $T_{21842,400}$ or $T_{24871,400}$ is sampled.

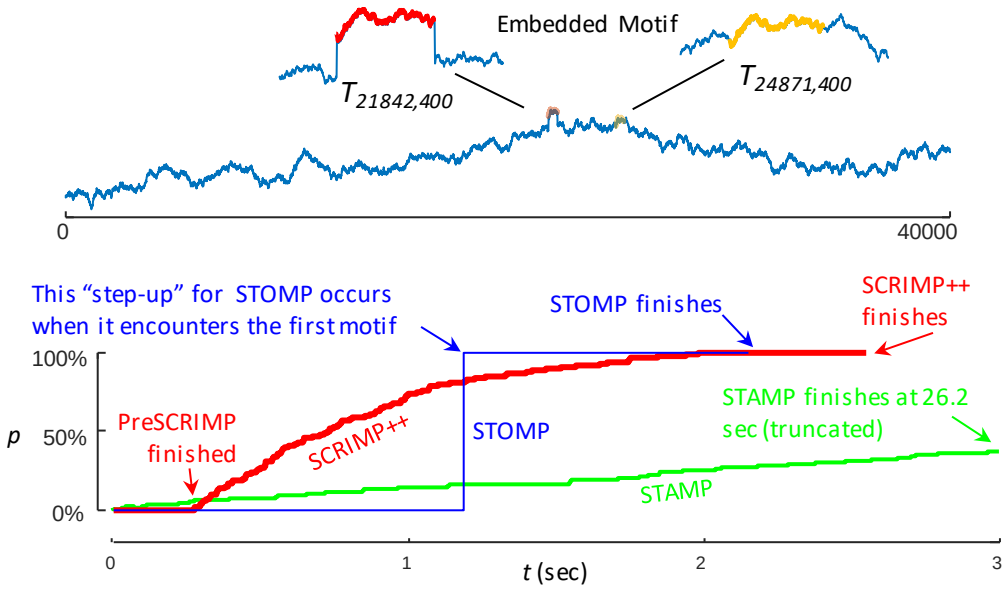

Figure 4.17. ***top*****) A pathological random walk time series with a pair of embedded motifs. The level of the data dramatically changes just before and after the first motif pattern, which invalidates the CNP property.** ***bottom*****) the observed probability for the top-1 motif discovered at each time instant. Note that the probability for STOMP is binary, and flips to 100% as soon as it encounters the first motif. That could happen arbitrarily late (i.e. to the far right) in the worse case.**

However, as **Figure 4.17**.*bottom* shows, the overall SCRIMP++ algorithm *still* converges much faster than STAMP [96] and STOMP (Chapter 3) at the early stage. Here the result is averaged over 100 runs, and the value p represents the probability that the embedded motif pair is discovered at each time instant t. Although SCRIMP++ fails to discover the motif at the PreSCRIMP phase, p quickly increases as the algorithm turns into the SCRIMP phase thanks to its random computation ordering. In contrast, STOMP shows a 0% probability in locating the motifs until after 1.2 seconds (recall that STOMP is deterministic, and reports the same result over the 100 runs); STAMP shows a very low probability in finding the motifs even when SCRIMP++ finishes. This example demonstrates the robustness of SCRIMP++, even in the most pathological and contrived cases that defeat PreSCRIMP.

4.5 Conclusions

In many domains, including neuroscience [10][35], entomology [79], medicine and consumer-level energy conservation [62], etc., analysts routinely deal with datasets that are in the range of a few million data points long. For the first time, SCRIMP++ allows the possibility of real-time *interactive* discovery of motifs in such datasets, using off-the-shelf consumer desktops. We believe that this ability will allow novel discoveries to be made in the relevant domains, and even new types of analytics to be invented. We have made all code and data freely available in perpetuity to allow the community to confirm and extend our findings [75].

Chapter 5 Lower-bounding the Matrix Profile: Admissible Time Series Motif Discovery with Missing Data

The discovery of time series motifs has emerged as one of the most useful primitives in time series data mining. Researchers have shown its utility for exploratory data mining, summarization, visualization, segmentation, classification, clustering, and rule discovery. Although there has been more than a decade of extensive research, there is still no technique to allow the discovery of time series motifs in the presence of missing data, despite the well-documented ubiquity of missing data in scientific, industrial, and medical datasets. In this chapter, we introduce a technique for motif discovery in the presence of missing data. We formally prove that our method is *admissible*, producing no false negatives. We also show that our method can "piggy-back" off the fastest known motif discovery algorithm STOMP with a small constant factor time/space overhead. We will demonstrate our approach on diverse datasets with varying amounts of missing data.

5.1 Related Work and Background

Time Series motifs are short approximately repeated patterns within a longer time series dataset. The fact that such patterns are *conserved* often suggests underlying structure and regularities that can be exploited in many ways. Some examples include: rule discovery [68][57], forecasting [44], or building better classifiers [12]. However, despite over a decade of active research [15][49][101], there is no known method to allow the discovery of motifs in the presence of missing data.

Paradoxically, in spite of improvements in sensor technology, missing data is becoming *more* prevalent. This is because sensors are now so cheap that we are willing to deploy them in hostile environments with intermittent and unreliable bandwidth [73]. In many cases, sensors have become "throwable" and disposable [21].

Figure 5.1 shows an example of a motif in a music processing domain [53]. Note that both occurrences of the motif contain sections of missing data.

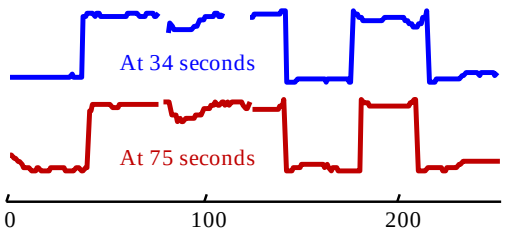

Figure 5.1. A four-second long motif that appears in the pitch contour time series of a Cypriot folk song, Kotsini Trantafillia (Red Rose-tree). Note that both occurrences have multiple instances of missing data [53].

Here the data could be missing for one of two reasons. It *might* be meaningful, i.e., the vocalist may not have produced a sound at these times. It is also possible that the missing data reflects poor audio quality, loose wires, etc. We are agnostic to such issues

in this work. We simply note that in either case, there is no motif discovery algorithm defined in the literature for such data.

More generally, in time series data mining, missing data is often handled by filling in the missing values with some interpolation or imputation method [31] and running the analysis unaltered. There are hundreds of imputation algorithms in the literature (see [90] and the references therein) to choose from, but no matter which one we use, we may obtain false negatives with respect to the *oracle* data (the true underlying data, without missing values).

To see this let us consider a simple example. Suppose we have a dataset that is composed of just three (sub)sequences:

$$A=\{0,2,0,2\},\ B=\{0,2,0,2\},\ C=\{0,-1,0,2\}$$

Clearly the pair A|B is a perfect motif. Now suppose that the second value in A is missing. The most obvious imputation technique is interpolation from the two neighbors of the missing data point, giving us $A_{miss} = \{0,0,0,2\}$. As we can see in **Figure 5.2**, this one change means that we no longer discover the pair A|B as a perfect motif, but instead we are led to believe that the pair A|C is the best motif in the dataset.

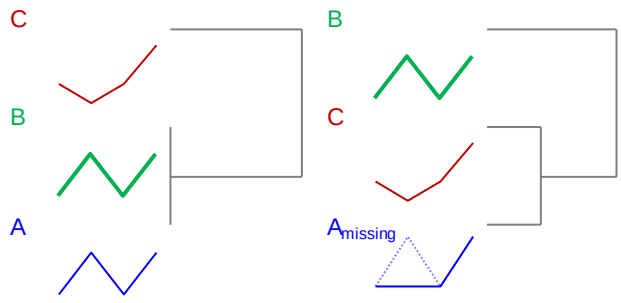

Figure 5.2. ***left)*** **A contrived dataset in which the pair A|B is a perfect motif.** ***right)*** **If A had its second value replaced by the most common imputation algorithm, we would fail to discover A|B as the motif.**

While this is a trivial example, it is easy to see that no matter what imputation algorithm is used, using a simple adversarial augment we can always construct datasets for which the classic motif algorithms would produce false negatives. This problem is even more severe with complex datasets that contain a lot of high-frequency patterns and noise, for example seismology datasets.

In **Figure 3.8** we showed two repeating earthquake patterns that appeared approximately 14 years apart. We have found that if there were just a handful of missing data points in one of the earthquake samples, we would be unable to detect a match between them with any common imputation method, as such high-frequency and noisy data defies the assumptions that most imputation techniques assume.

Figure 5.2 showed that imputation methods can produce possible false negatives even if we have a random single value missing in the data. Moreover, another disadvantage of current imputation methods is that they cannot predict *block-missing data* well [100]. In some circumstances, due to malfunction of the sensor or other anomaly factors, we may lose reading from a sensor at consecutive timestamps (instead of sparse, single missing timestamps at random locations of the data). This is often called *block-missing* data. In [100] the authors proposed a state-of-the-art spatial-temporal imputation method to predict the block-missing data by learning from not only the real-valued reading of the same sensor, but also from the reading of several geographically nearby sensors. The information from other sensors greatly improved their imputation accuracy. However, this method does not apply when we only have access to one single

sensor, or when all the sensors contain block-missing data at the same timestamps due to regional power outage or communication errors.

Finally, note that our problem is not artificial or contrived in anyway. The literature is replete with examples of data analysts frustrated by the inability to perform motif discovery in the presence of missing data. For example, a recent paper studies recurrent water consumption behavior by Australian consumers [87]. The authors observe "*A small proportion of all hourly readings are missing…, probably due to server failure or maintenance*". The authors realize that any imputation method used here has a risk of producing false negatives by noting that "*…hourly water consumption is highly unpredictable, we ignored the points of missing hours for the routine discovery, rather than approximating missing readings*." However, their solution of ignoring some data runs the risk of missing interesting patterns.

5.1.1 Dismissing Apparent Solutions

In this section, we continue the discussion of related work, while explicitly dismissing some proposed solutions to the task at hand.

The last decade has seen several distance measures for handling uncertain time series, including PROUD [99], DUST [64], PBRQ [1], PRRQ [1], etc. One might believe that these measures could be used to replace the Euclidean Distance subroutine in an existing motif discovery algorithm. However, we believe this is not possible for the following reasons:

- These methods assume not *missing* data, but *uncertain* data. For example, they can address the situation where an observation is not known precisely but comes from some known distribution. In [64] for example, they explicitly model the normal, exponential, and uniform error distributions. However, for generality, we do not wish to make such strong assumptions.
- Even if we assume that we could somehow avail of an existing uncertain distance measure, none of them are metrics (only *measures*). However, all speedup techniques for exact motif discovery that we are aware of require and exploit metric properties [15][49]. This suggests we must resort to a $O(n^2m)$ brute force search (n is the length of the time series, m the motif length). However, our proposed method is $O(n^2)$. As m may be in the thousands (see Section 5.3), this suggests a three orders of magnitude time difference.
- Finally, we want to be able to guarantee that our search produces no false negatives in the face of missing data. To the best of our knowledge, no existing uncertain time series similarity measures can support this requirement.

The reader may not appreciate why our task-at-hand is hard, because the analogue problem with *strings* is trivial. Suppose we are asked to compare the following text strings "*Norwegian blue*" and "*Norwegian wood*" under the Hamming Distance, and we encounter missing values, represented here by "*****".

Norwe*ian blu*

N*rwe*ian wood

We can easily compute both the lower and upper bound of the Hamming Distance. In the former case, we would assume that all the missing values in one word are the *same* as their counterparts in the other word. Given that the only letters we can be sure are differing are “woo” vs “blu,” and we have a lower bound of three. In the latter case, we would assume that all the missing values in one word are *different* than their counterparts in the other word. These three pessimistic differences combined with the three observed differences give us an upper bound of six.

However, consider the time series version of this problem. Suppose we have the following two time series:

```
[ 0.5, 0.1, ***, ***, -0.6, -0.7, 0.0 ]

[ 0.3, 0.1, 1.1, ***, -0.6,  ***, 0.1 ]
```

One might consider applying similar logic here. For example, accumulating 0.2 error (i.e. |**0.5** - **0.3**|) from the first pair of numbers, then 0.0 error (i.e. |**0.1** - **0.1**|) from the second pair, etc. However, the critical difference is that the time series must be *normalized* before comparison. This is because, aside from the rare and well understood exceptions [18], it is meaningless to compare time series without normalizing them first. This presents a problem as z-normalization (the most common normalization technique [17]) requires us to know the exact mean and standard deviation of the data, which are undefined when we have a single missing data point.

Thus, for any pair of corresponding *known* points (for example the **0.5** and the **0.3** in the above), it is possible that the true (had we known the mean and standard deviation of the data to allow the correct normalization) difference between them could stay the

same, increase, or decrease by arbitrary amounts. This suggests that producing tight upper or lower bounds will be nontrivial.

5.1.2 Pseudo Missing Data

Before introducing our solution to the missing data problem, we take the time to point out that the problem is more general than one may assume when we consider the generalization of the Pseudo Missing Data (PMD). Informally, we define PMD as data that technically is not missing, but *effectively* is. **Figure 5.3** illustrates three kinds of PMD frequently encountered.

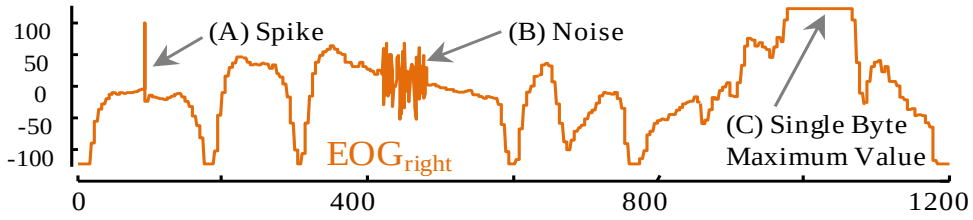

Figure 5.3. A snippet of an Electrooculogram (EOG) exhibits three kinds of pseudo missing data.

In **Figure 5.3**.A, we see a "spike." Given what we know about this domain, it is inconceivable that the human eye could move fast enough to produce this data, so it is clearly an artifact. Likewise, in **Figure 5.3**.B, the dramatic increase in variance suggests that this section of data is not likely to faithfully represent the underlying physical process. Finally, in **Figure 5.3**.C the perfectly flat plateau is not reflective of reality, but is simply a region where the physical process exceeds the 8-bit precision available to record it.

In all three cases, the best thing to do would be to treat the data as missing. Note that this decision is domain-dependent; there are clearly domains where a short spike

represents some physical event, or where a perfectly flat plateau represents a physical limitation, which is not a quirk of the hardware/software use.

5.1.3 Definitions and Notations

In this section, we introduce all the notations and definitions needed to explain our solution.

We first generalize the definition of time series and subsequence in Section 2.1 to allow for the possibly that at least one value is missing. For clarity, we differentiate such time series (including subsequences) with a "bar." To keep both in a common notation and for implementation purposes (i.e. Matlab, etc.), we use *NaN*s as a placeholder for missing values.

Definition 4.1: A *missing value time series* $\overline{T}$ is a sequence of values that are either real-valued numbers or *NaN*s, $\bar{t}_i$: $\overline{T} = \bar{t}_1, \bar{t}_2, ..., \bar{t}_n$, where *n* is the length of $\overline{T}$.

In the rest of the paper, we assume that *T* is the *actual* time series of $\overline{T}$ before the missing values were created by some process. That is to say, we would have obtained *T* instead of $\overline{T}$ if the sensors were functioning properly. We do not know *T* precisely, but we have $\bar{t}_i = t_i$ ($1 \leq i \leq n$) if $\bar{t}_i \neq NaN$.

For motif discovery, we are not interested in the *global* properties of the missing value time series, but in the *local* regions, known as *subsequences*:

Definition 4.2: A *subsequence* $\overline{T}_{i,m}$ of a missing value time series $\overline{T}$ is a continuous subset of the values from $\overline{T}$ of length *m* starting from position *i*. Formally, $\overline{T}_{i,m} = \bar{t}_i, \bar{t}_{i+1}, ..., \bar{t}_{i+m-1}$, where $1 \leq i \leq n\text{-}m+1$.

Note that subsequence $\overline{T}_{i,m}$ may or may not contain missing values (*NaN*s).

Recall that in Section 2.1, we have introduced two meta time series, the *matrix profile* and the *matrix profile index*, to annotate a time series *T* with the distance and location of all its subsequences' nearest neighbors within itself. For a time series *T* of length *n*, the STOMP algorithm introduced in Chapter 3 is able to compute the two meta files with a mere O(n^2) time complexity and O(*n*) space complexity, which enables a fast motif discovery in a massive time series. However, the STOMP algorithm is not applicable to any time series $\overline{T}$ with missing values (*NaN*s). To solve this problem, we introduce a novel algorithm that does not allow false negatives. The method *may* allow occasional false positives, but since the discovered motifs are typically examined by the human eye [49][57][101], or some subsequent analysis, the false positives (if any) can be filtered out at a later stage.

We call our algorithm Motif Discovery with Missing Data, MDMS. Our MDMS algorithm is built on top of the Matrix Profile data structure, and here we claim our MDMS algorithm can solve the missing data problem with the same time and space complexity as STOMP. We leave the detailed discussion of the algorithm to Section 5.2.

5.2 Algorithms

5.2.1 An Intuitive Preview

We begin by previewing our solution. As shown in Chapter 2, if *T* is used to compute a matrix profile then finding the motifs is trivial. The location of the *smallest* pair of values in the matrix profile is also the location in *T* of the optimal motif pair.

Moreover, other definitions of motifs, such as the top-K motifs or range motifs [49][96], can also be easily extracted from the matrix profile. Given this, our solution to the missing data problem is to build a special matrix profile using $\overline{T}$. This special matrix profile will be very similar to the true matrix profile, and be a (in general, *very* tight) lower bound for it. If we use the existing motif extraction algorithms [96] to pull out motifs from this matrix profile, we may have false positives, but we will have no false negatives [18]. Thus most of our contribution outlined below is to show how we can build this special matrix profile.

5.2.2 The Lower Bound Matrix Profile

To create the special matrix profile data structure, our MDMS algorithm evaluates the z-normalized Euclidean distance between every pair of subsequences within a missing value time series $\overline{T}$. Depending on whether or not the subsequences have missing values, we may encounter three different situations. Assume that the pair of subsequences under consideration is $\overline{T}_{i,m}$ and $\overline{T}_{j,m}$.

- **Case 1**: Neither $\overline{T}_{i,m}$ nor $\overline{T}_{j,m}$ contains any missing value (*NaN*). Normally, this applies to most subsequence pairs within time series $\overline{T}$ if $\overline{T}$ contains more real-valued numbers than *NaN*s. The traditional exact z-normalized Euclidean distance between $\overline{T}_{i,m}$ or $\overline{T}_{j,m}$ can be evaluated in this case.
- **Case 2**: $\overline{T}_{i,m}$ contains missing values (*NaN*s) while $\overline{T}_{j,m}$ does not, or vice versa.
- **Case 3**: Both $\overline{T}_{i,m}$ and $\overline{T}_{j,m}$ have missing values (*NaN*s).

In cases 2 and 3, the exact z-normalized Euclidean distance $d_{i,j}$ between $\overline{T}_{i,m}$ or $\overline{T}_{j,m}$ cannot be evaluated. However, we can evaluate a lower bound of the distance between $\overline{T}_{i,m}$and $\overline{T}_{j,m}$, $d_{i,j}^{LB}$, such that $d_{i,j}^{LB} \leq \min(d_{i,j})$. That is to say, no matter what the missing values in $\overline{T}_{i,m}$ or $\overline{T}_{j,m}$ are, we guarantee that $d_{i,j}$, the actual distance between $\overline{T}_{i,m}$ and $\overline{T}_{j,m}$, is no less than $d_{i,j}^{LB}$.

Now we are ready to redefine the distance profile (**Definition 2.3** in Section 2.1) in the context of missing values. We keep those $d_{i,j}$ values corresponding to Case 1 unchanged, and use $d_{i,j}^{LB}$ to replace the $d_{i,j}$ values corresponding to Cases 2 and 3. We will then obtain a *lower bound distance profile*:

Definition 4.3: A *lower bound distance profile* D_i^{LB} of a missing value time series $\overline{T}$ is a vector D_i^{LB}=[$\bar{d}_{i,1}$, $\bar{d}_{i,2}$,..., $\bar{d}_{i,n-m+1}$,], where $\bar{d}_{i,j}$=$d_{i,j}$ ($1 \leq i, j \leq n\text{-}m+1$) if neither $\overline{T}_{i,m}$ nor $\overline{T}_{j,m}$ contains *NaN*s (Case 1), and $\bar{d}_{i,j}$=$d_{i,j}^{LB}$ ($1 \leq i, j \leq n\text{-}m+1$) otherwise.

Similarly, we can redefine the matrix profile and the matrix profile index (**Definition 2.4** and **Definition 2.5** in Section 2.1) in the context of missing values.

Definition 4.4: A *lower bound matrix profile* P^{LB} of a missing value time series $\overline{T}$ is a vector of the *lower bound* Euclidean distances between each subsequence $\overline{T}_{i,m}$ and its nearest possible neighbor (closest possible match) in $\overline{T}$. Formally, P^{LB} = [min(D_1^{LB}), min(D_2^{LB}),..., min(D_{n-m+1}^{LB})], where D_i^{LB} ($1 \leq i \leq n\text{-}m+1$) is the lower bound distance profile D_i^{LB} of the time series *T*.

Definition 4.5: A *lower bound matrix profile index* I^{LB} of $\overline{T}$ *is a vector of integers:* I^{LB}=[I_1^{LB}, I_2^{LB}, ... I_{n-m+1}^{LB}], where I_i^{LB}=*j if* $\bar{d}_{i,j}$= *min*(D_i^{LB}).

The lower bound matrix profile gives us an optimistic approximation of the distance between every subsequence and its nearest possible neighbor in $\bar{T}$. This approximation may produce false positives, but it will not allow false negatives.

We believe this is the best approach for the task at hand: as Section 3.3.4 shows, the cost to filter out false positive motifs is very low once we have the matrix profile, but we cannot afford the occurrence of any false negatives, since they may include the most important patterns in the time series. In the following sections, we will introduce the lower bound Euclidean distances corresponding to Cases 2 and 3 respectively; then we will introduce our MDMS algorithm, which evaluates a lower bound of the matrix profile.

5.2.3 The Lower Bound Euclidean Distance

3.2.3.1 Case 2

Let us first consider Case 2, where $\overline{T}_{i,m}$ contains missing values (*NaN*s) while $\overline{T}_{j,m}$ does not, or vice versa. Without loss of generality, for now we assume $\overline{T}_{i,m}$ is the subsequence that contains missing values, and $\overline{T}_{j,m}$ is the subsequence without missing values. **Figure 5.4** shows a visual example of this case.

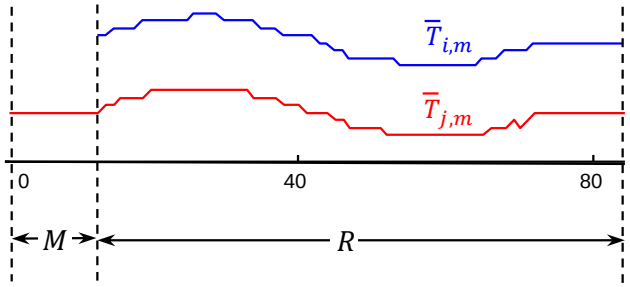

Figure 5.4. *top*) A subsequence with missing values. *bottom*) A subsequence without missing values.

Here $M = \{k \mid 1 \le k \le m \text{ and } t_{i+k-1} = NaN\}$ is a set of the locations of missing values within $\overline{T}_{i,m}$, and $R = \{k \mid 1 \le k \le m \text{ and } t_{i+k-1} \neq NaN \text{ and } t_{j+k-1} \neq NaN\}$ is the intersection of the real-valued locations within $\overline{T}_{i,m}$ and those within $\overline{T}_{j,m}$. To evaluate the lower bound distance of the two subsequences, first we need to z-normalize them [17][49][96].

We assume that μ_i and σ_i are the mean and standard deviation of $\overline{T}_{i,m}$, μ_j and σ_j are the mean and standard deviation of $\overline{T}_{j,m}$. Note that because $\overline{T}_{i,m}$ has missing values, we cannot evaluate μ_i and σ_i; however, we can treat them as variables. Assume $d_{i,j}$ is the distance between $\overline{T}_{j,m}$ and the oracle subsequence of $\overline{T}_{i,m}$, we can easily obtain a lower bound distance of $d_{i,j}$ by ignoring the missing part M in **Figure 5.4**:

$$d_{i,j}^{LB} = \sqrt{\min_{\mu_i,\sigma_i} \sum_{k \in R} \left(\frac{t_{i+k-1} - \mu_i}{\sigma_i} - \frac{t_{j+k-1} - \mu_j}{\sigma_j} \right)^2} \tag{5.1}$$

Assume $f_1 = {d_{i,j}^{LB}}^2$. We can linearly transform f_1 as:

$$f_1 = {d_{i,j}^{LB}}^2 = \left(\frac{\sigma_j^R}{\sigma_j} \right)^2 \min_{\mu,\sigma} \sum_{k \in R} \left(\frac{t_{i+k-1} - \mu}{\sigma} - \frac{t_{j+k-1} - \mu_j^R}{\sigma_j^R} \right)^2 \tag{5.2}$$

We assume $T_{i,m}^R$ is the real-valued part of $\overline{T}_{i,m}$, $T_{j,m}^R$ is the subset of $\overline{T}_{j,m}$ corresponding to R (as indicated in **Figure 5.4**), μ_i^R and σ_i^R are the mean and standard deviation of $T_{i,m}^R$, μ_j^R and σ_j^R are the mean and standard deviation of $T_{j,m}^R$.

Figure 5.5 visualizes (5.2). Note that $T_{j,m}^R$ is z-normalized in (5.2), so its offset and scale are fixed in **Figure 5.5**; to obtain $d_{i,j}^{LB}$, we would like to adjust μ (corresponding

to the offset of $T_{i,m}^R$) and σ (corresponding to the scale of $T_{i,m}^R$), such that the Euclidean distance between $T_{i,m}^R$ and $T_{j,m}^R$ is minimized.

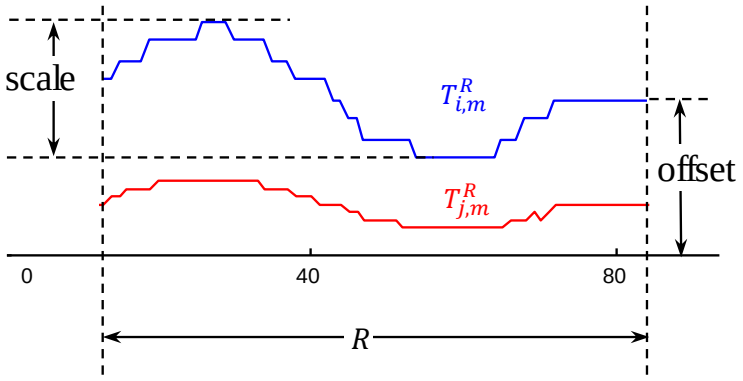

Figure 5.5. Different setting of μ and σ changes the offset and the scale of $T_{i,m}^R$. Note that the offset and scale of $T_{j,m}^R$ are fixed.

By solving $\frac{\partial f_1}{\partial \mu} = 0$ and $\frac{\partial f_1}{\partial \sigma} = 0$, and substituting σ and μ back into (5.2), we have:

$$d_{i,j}^{LB} = \begin{cases} \frac{\sigma_j^R}{\sigma_j}\sqrt{|R|} & if\ q_{i,j} \leq 0 \\ \frac{\sigma_j^R}{\sigma_j}\sqrt{|R|(1-q_{i,j}^2)} & if\ q_{i,j} > 0 \end{cases} \quad (5.3)$$

Here R is the intersection of the real-valued locations within $\overline{T}_{i,m}$ and those within $\overline{T}_{j,m}$, $q_{i,j}$ is the Pearson Correlation Coefficient between $T_{i,m}^R$ and $T_{j,m}^R$:

$$q_{i,j} = \frac{\sum_{k \in R} t_{j+k-1} t_{i+k-1} - |R|\mu_i^R \mu_j^R}{|R|\sigma_i^R \sigma_j^R} \quad (5.4)$$

The analysis of Case 2 is now complete. Let us turn to Case 3, where both $\overline{T}_{i,m}$ and $\overline{T}_{j,m}$ contain missing values.

3.2.3.2 Case 3

Figure 5.6 shows an example of Case 3.

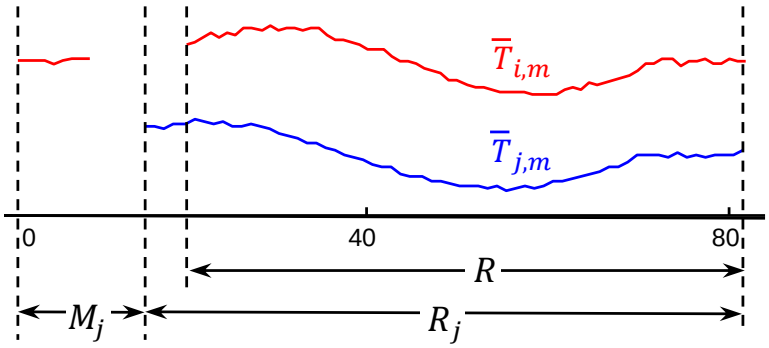

Figure 5.6. Two subsequences with missing values.

As both $\overline{T}_{i,m}$ and $\overline{T}_{j,m}$ have missing values, their means (μ_i, μ_j) and standard deviations (σ_i, σ_j) can be arbitrary values. We have:

$$d_{i,j} \geq \min_{\mu_j,\sigma_j} \sqrt{\min_{\mu_i,\sigma_i} \sum_{k \in R} \left(\frac{t_{i+k-1} - \mu_i}{\sigma_i} - \frac{t_{j+k-1} - \mu_j}{\sigma_j} \right)^2} = d_{i,j}^{LB} \tag{5.5}$$

Here $R = \{k \mid 1 \leq k \leq m, t_{i+k-1} \neq NaN, t_{j+k-1} \neq NaN\}$ is the intersection of the real-valued locations within $\overline{T}_{i,m}$ and $\overline{T}_{j,m}$ (see **Figure 5.6**). A diligent reader may have noticed that the lower bound expression $d_{i,j}^{LB}$ in (5.5) subsumes (5.1), the lower bound expression in Case 2. We can visualize this in **Figure 5.6**: if we remove M_j, the problem is transformed to Case 2. Therefore, we can directly substitute (5.3), the result of Case 2, into (5.5):

$$d_{i,j}^{LB} = \begin{cases} \sigma_j^R \sqrt{|R|} \min_{\sigma_j} \frac{1}{\sigma_j} & if\ q_{i,j} \leq 0 \\ \sigma_j^R \sqrt{|R|(1 - q_{i,j}^2)} \min_{\sigma_j} \frac{1}{\sigma_j} & if\ q_{i,j} > 0 \end{cases} \tag{5.6}$$

We can see from (5.6) that $d_{i,j}^{LB}$ is dependent on σ_j (which is controlled by the missing part of $\overline{T}_{j,m}$ in **Figure 5.6**): the larger σ_j, the smaller $d_{i,j}^{LB}$. Note that σ_j can be as

large as $+\infty$; in that case $d_{i,j}^{LB}$ becomes zero. This is a very undesirable lower bound, as *any* pair of missing value subsequences can be reported as a motif, even if they look very different from each other. **Figure 5.7** shows an example of this.

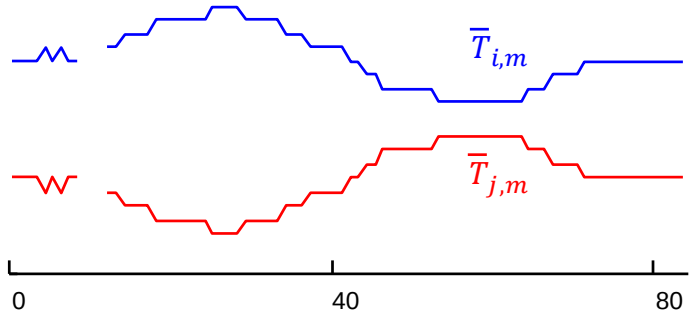

Figure 5.7. Two subsequences with missing values. The real-valued parts of the subsequences look very different from each other, but if we fill the missing parts with infinitely large numbers, the z-normalized Euclidean distance of the two subsequences will become zero.

Fortunately, sensor readings normally have physical limits. The accelerometer values on an iPhone 7 are limited to ± 8g (± 78.48 m/s^2); virtually all medical sensors come with carefully specified limits to meet regulations (i.e., EU directive 93/42/EEC mandates that a pediatric lung ventilator monitor produces values in the range of 0 to 125$_{\text{cmH20}}$), etc. Therefore, we can assume that the missing values in $\overline{T}_{j,m}$ are bounded by $[V_{min,j}, V_{max,j}]$. With this bound, we can derive the following inequality for ${\sigma_j}^2$ (we refer the interested readers to [77] for details):

$$ {\sigma_j}^2 \leq \frac{{C_j}^2}{4} + \frac{\sum_{k \in R_j} {t_{j+k-1}}^2 + |R_j| \left(B_j - \mu_j^{R_j} A_j \right)}{m} \tag{5.7} $$

Here R_j is a set of the locations of all the real values within $\overline{T}_{j,m}$ (see **Figure 5.6**), $\mu_j^{R_j}$ is the mean of the real-valued part of $\overline{T}_{j,m}$, $C_j = V_{max,j} - V_{min,j}$, $B_j = V_{max,j} V_{min,j}$,

$A_j = V_{max,j} + V_{min,j}$. In practice, we set $V_{min,j}$ and $V_{max,j}$ as the minimum and maximum value of the real-valued part of $\overline{T}_{j,m}$.

We can now evaluate $d_{i,j}^{LB}$ by substituting (5.7) back into (5.6):

$$d_{i,j}^{LB} = \begin{cases} \sqrt{|R| f_{LB}(j)} & if\ q_{i,j} \leq 0 \\ \sqrt{|R|(1 - q_{i,j}^2) f_{LB}(j)} & if\ q_{i,j} > 0 \end{cases} \tag{5.8}$$

Here $f_{LB}(j) = \sigma_j^{R^2} \Big/ \left[\frac{{C_j}^2}{4} + \frac{\sum_{k \in R_j} {t_{j+k-1}}^2 + |R_j| \left(B_j - \mu_j^{R_j} A_j \right)}{m} \right]$.

Note that $f_{LB}(j)$ is based on R_j, the real-valued part of $\overline{T}_{j,m}$ (recall **Figure 5.6**). However, as in Case 3 both $\overline{T}_{i,m}$ and $\overline{T}_{j,m}$ have missing values, we can analogously derive a lower bound expression similar as (5.8) based on $\overline{T}_{i,m}$, and set the larger expression as $d_{i,j}^{LB}$:

$$d_{i,j}^{LB} = \begin{cases} \sqrt{|R| max(f_{LB}(i), f_{LB}(j))} & if\ q_{i,j} \leq 0 \\ \sqrt{|R|(1 - q_{i,j}^2) max(f_{LB}(i), f_{LB}(j))} & if\ q_{i,j} > 0 \end{cases} \tag{5.9}$$

Here $f_{LB}(p) = \sigma_p^{R^2} \Big/ \left[\frac{{C_p}^2}{4} + \frac{\sum_{k \in R_p} {t_{p+k-1}}^2 + |R_p| \left(B_p - \mu_p^{R_p} A_p \right)}{m} \right]$.

The analysis of Case 3 is now complete. Finally, for completeness, let us briefly discuss Case 1.

3.2.3.3 Case 1

As neither $\overline{T}_{i,m}$ nor $\overline{T}_{j,m}$ contains any missing value in Case 1, we set $d_{i,j}^{LB}$ as the exact z-normalized Euclidean distance between $\overline{T}_{i,m}$ and $\overline{T}_{j,m}$, using the following equation (recall (3.1) in Chapter 3):

$$d_{i,j}^{LB} = \sqrt{2m\left(1 - q_{i,j}\right)} \tag{5.10}$$

Note that all three cases use the same expression of $q_{i,j}$ in (5.4).

Now that we have the lower bound distance for any subsequence pair in $\overline{T}$, we can also evaluate the lower bound matrix profile.

5.2.4 The MDMS Algorithm

3.2.3.1 Case 2

The STOMP algorithm introduced in Chapter 3 can obtain the matrix profile of a time series that is free of missing values, in O(n^2) time with only O(n) space; as we will now show, in the face of missing data, our MDMS algorithm (**Algorithm 6**) can obtain the lower-bound matrix profile with the same time and space complexity.

Before discussing the algorithm in detail, we first need to introduce three important auxiliary time series (shown in line 3 of **Algorithm 6**), Z, X and B.

- We define $Z = z_1, z_2, \dots z_n$, such that $z_i = t_i$ if $t_i \neq NaN$, and $z_i = 0$ if $t_i = NaN$. We can simply obtain Z by filling zeros in the locations of $\overline{T}$ where the data is missing.
- We define $X = x_1, x_2, \dots x_n = {z_1}^2, {z_2}^2, \dots {z_n}^2$.
- We define $B = b_1, b_2, \dots b_n$, such that $b_i = 1$ if $t_i \neq NaN$, and $b_i = 0$ if $t_i = NaN$. We can see that B indicates the locations of the real-valued numbers and missing values in $\overline{T}$.

With these three auxiliary time series and the techniques introduced in [61] and **Algorithm 2**, we can evaluate any lower bound distance introduced in the last section in O(*1*) time with O(*n*) space.

Algorithm 6: MDMS(*T*,*m*)

Input: A missing value time series ***T***, subsequence length ***m***
Output: Lower bound matrix profile ***P*** and the associated lower bound matrix profile index ***I*** of *T*

```
1  n ← Length(T), len ← n-m+1
2  vmax ← SlidingMax(T), vmin ← SlidingMin(T)
3  Z ← PadZero(T), B ← OneZero(T), X← ElementWiseSquare(Z)
4  μz, σz ← ComputeMeanStd(Z, m)                    // see [61]
5  μb, σb ← ComputeMeanStd(B, m)                    // see [61]
6  QZ ← SlidingDotProduct(Z_{1:m}, Z), QZ' ← QZ     //see Algorithm 1
7  QB ← SlidingDotProduct(B_{1:m}, B), QB' ← QB     //see Algorithm 1
8  BZ ← SlidingDotProduct(B_{1:m}, Z), BZ' ← BZ     //see Algorithm 1
9  ZB ← SlidingDotProduct(Z_{1:m}, B), ZB' ← ZB     //see Algorithm 1
10 BX ← SlidingDotProduct(B_{1:m}, X), BX' ← BX     //see Algorithm 1
11 XB ← SlidingDotProduct(X_{1:m}, B), XB' ← XB     //see Algorithm 1
12 P ← CalculateLBDistance(n, m, vmax, vmin, QZ, QB, BZ, ZB, BX, XB, μz, σz, μb, σb, i)
13 I← ones                                          // initialization
14 for i = 2 to len                                 // in-order evaluation
15     for j= len downto 2                          // update dot product, see Algorithm 2
16         QZ_j ← QZ_{j-1}-Z_{i-1}×Z_{j-1}+Z_{i+m-1}×Z_{j+m-1}
17         QB_j ← QB_{j-1}-B_{i-1}×B_{j-1}+B_{i+m-1}×B_{j+m-1}
18         BZ_j ← BZ_{j-1}-B_{i-1}×Z_{j-1}+B_{i+m-1}×Z_{j+m-1}
19         ZB_j ← ZB_{j-1}-Z_{i-1}×B_{j-1}+Z_{i+m-1}×B_{j+m-1}
20         BX_j ← BX_{j-1}-B_{i-1}×X_{j-1}+B_{i+m-1}×X_{j+m-1}
21         XB_j ← XB_{j-1}-X_{i-1}×B_{j-1}+X_{i+m-1}×B_{j+m-1}
22     end for
23     QZ_1←QZ'_i, QB_1←QB'_i, BZ_1←ZB'_i
24     ZB_1←BZ'_i, BX_1←XB'_i, XB_1←BX'_i
25     D ← CalculateLBDistance(n, m, vmax, vmin, QZ, QB, BZ, ZB, BX, XB, μz, σz, μb, σb, i)
26     P, I ← ElementWiseMin(P, I, D, i)
27 end for
28 return P, I
```

The MDMS algorithm is very similar to the STOMP framework introduced in Chapter 3. In line 2 we evaluate the maximum and minimum values of the real-value part of every subsequence in *T*. Lines 4-5 evaluate the mean and standard deviation of every subsequence in *Z* and *B*. In lines 6-11, SlidingDotProduct(*x*, *y*) computes a vector of dot

products between a query subsequence x and every subsequence in time series y (**Algorithm 1**). We call this the sliding dot product as we can extract all the subsequences in time series y by sliding a window of length m across y. Note that μz, σz, μb and σb and the sliding dot product vectors QZ, QB, BZ, ZB, BX, XB are sufficient statistics to compute the lower bound matrix profile. We initialize the lower bound matrix profile P and matrix profile index I in lines 12-13. Lines 14-27 iteratively evaluate the lower bound distance profile D, and update P and I if necessary. The CalculateLBDistance algorithm in lines 12 and 25 is shown in **Algorithm 7**.

Algorithm 7: CalculateLBDistance(*n, m, vmax, vmin, QZ, QB, BZ, ZB, BX, XB, μz, σz, μb, σb, i*)

Input: the length ***n*** of time series *T*, the subsequence length ***m***, the maximum/minimum possible value vector ***vmax/vmin***, sliding dot product vectors ***QZ***, ***QB***, ***BZ***, ***ZB***, ***BX***, ***XB***, means and standard deviations ***μz***, ***σz***, ***μb***, ***σb*** of time series *Z* and *B*, current subsequence index ***i***.
Output: Lower bound distance profile ***D***

```
1  for j= 1 to n-m+1
2      ui ← ZB_j / QB_j, uj ← BZ_j / QB_j                    //μ_i^R and μ_j^R
3      vi ← XB_j / QB_j - ui ^2, vj ← BX_j / QB_j - uj ^2    //σ_i^R and σ_j^R
4      q ← (QZ_j / QB_j - ui × uj) / sqrt(vi × vj)           // (5.4)
5      if QB_j == m then                                     // Case 1, |R| = m
6          D_j ←2×m×(1-q)                                    // (5.10)
7      else
8          if max(μb_i, μb_j)==1 then                        // Case 2
9              if μb_i >μb_j then vo ← vi, v ← σz_i ^2
10             else vo ← vj, v ← σz_j ^2
11             end if
12             if q <=0 then D_j ← QB_j × vo / v             // (5.3)
13             else D_j ← QB_j × vo / v × (1 - q^2)
14             end if
15         else                                              // Case 3
16             v1 ← vmax_i, v2 ← vmin_i, C ← v1 – v2, B ← v1 × v2, A ← v1 + v2
17             ur ← μz_i /μb_i, vr ← (σz_i ^2 + μz_i ^2) / μb_i – ur^2
18             f1 = vi / (ub_i × (vr + B + ur × (ur – A)) + C^2 / 4)      // f_LB(i) in (5.9)
19             v1 ← vmax_j, v2 ← vmin_j, C ← v1 – v2, B ← v1 × v2, A ← v1 + v2
20             ur ← μz_j / μb_j, vr ← (σz_j ^2+μz_j ^2) / μb_j -ur ^2
21             f2 = vj / (ub_j × (vr + B + ur × (ur – A)) + C^2 / 4)      // f_LB(j) in (5.9)
22             if q <=0 then D_j ← QB_j × max(f1, f2)          // (5.9)
23             else D_j ← QB_j × max(f1, f2) × (1 - q^2)       // (5.9)
```

```
24              end if
25          end if
26      end if
27  end for
28  return D
```

The CalculateLBDistance algorithm evaluates all *n-m+1* lower bound distance values in *D* with equations (5.3), (5.9) and (5.10). Line 4 evaluates (5.4). Case 1 is handled in lines 5-6. Lines 8-14 handle Case 2, and Case 3 is evaluated in lines 15-24.

We can see that each loop of the CalculateLBDistance algorithm in lines 2-26 can be evaluated in O(*1*) time, so the time complexity of CalculateLBDistance is O(*n*). The space needed to store all the vectors in the *MDMS* algorithm is O(*n*), and each loop in lines 14-25 of *MDMS* takes O(*n*) time. Therefore, the time complexity of *MDMS* algorithm is O(n^2) and the space complexity is O(*n*), the same as STOMP (Chapter 3).

Furthermore, we can see that unlike most imputation algorithms [90], our MDMS algorithm is extremely model-free and parameter-free. The only inputs to the algorithm are the time series and the subsequence length. In the next section, we will use two case studies to demonstrate the robustness and efficacy of our ultra-fast, parameter-free motif discovery algorithm in the face of missing data.

5.3 Experimental Evaluation

We begin by noting that all the code and data used in this work are archived in perpetuity at [77]. In the following two case studies, we consider both random-missing data (data with sparsely located, random missing timestamps) and block-missing data (data with consecutive missing timestamps).

For each case study, we compare our method with the commonly used strawman in the literature to handle missing data: linear imputation, as shown in **Figure 5.8**.

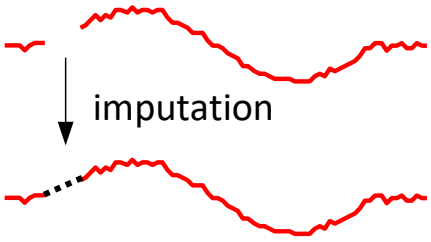

Figure 5.8. To evaluate our method, we compare our result with that of linear imputation.

5.3.1 Case Study: Seismological Data

Repeated pattern (i.e. *motif*) discovery is a fundamental tool in seismology, which allows for the discovery of foreshocks, triggered earthquakes, swarms, volcanic activity, and induced seismicity [101]. However, this domain is replete with missing data.

For example, a classic paper notes "*A frequent dilemma in spectral analysis* (in seismology) *is the incompleteness of the data record, either in the form of occasional missing data or as larger gaps*" (our emphasis) [5]. In this experiment, we demonstrate that we can handle both cases.

We consider a dataset for which we know the answer from external sources. On April 30th, 1996, there was an earthquake of magnitude 2.12 in Sonoma County, California. Then, on December 29th, 2009, about 13.6 years later, there was another earthquake with a similar magnitude. To allow the results to be visualized in a single plot, we edited this data such that the two earthquakes happen just 15 seconds apart. We set the subsequence length as 2,000, which corresponds to one second of data. As shown in

Figure 5.9 (red curve), when there is no missing data the matrix profile correctly discovers the locations of the two earthquakes.

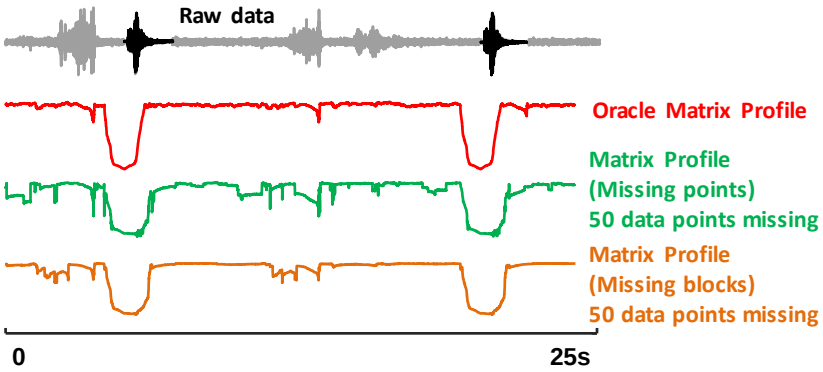

Figure 5.9. A raw seismograph contrived such that two earthquakes from the same region happen 15 seconds apart. The matrix profile computed with no missing data (red curve) finds the true event, as does MDMS even in the presence of missing points (green curve) or missing blocks (orange curve).

To test our algorithm for the "occasional missing data" case, we randomly deleted 50 data points. As **Figure 5.9** (green curve) shows, the matrix profile is still minimized at the correct location, and there are no false positives (no other small values in the matrix profile besides the two deep valleys). This shows the robustness of our algorithm in the face of random missing data.

Next, we consider the "larger gaps" (or the block-missing data) case. Here, instead of removing individual data points, we removed two blocks of length 25. As shown in **Figure 5.9** (orange curve), the shape of the lower bound matrix profile still looks very similar to that of the oracle matrix profile (red curve). We see only two deep valleys in the vicinity of the motifs, so no false positive patterns are discovered.

The result demonstrates that our lower bound matrix profile is robust against producing false positives.

To test the robustness of our algorithm against false negatives, we removed a block of missing data at the center of the second earthquake pattern. The length of the missing block is 400, which is 20% of the subsequence length. In **Figure 5.10**, we compared our lower-bound matrix profile result with the matrix profile generated by linear imputation.

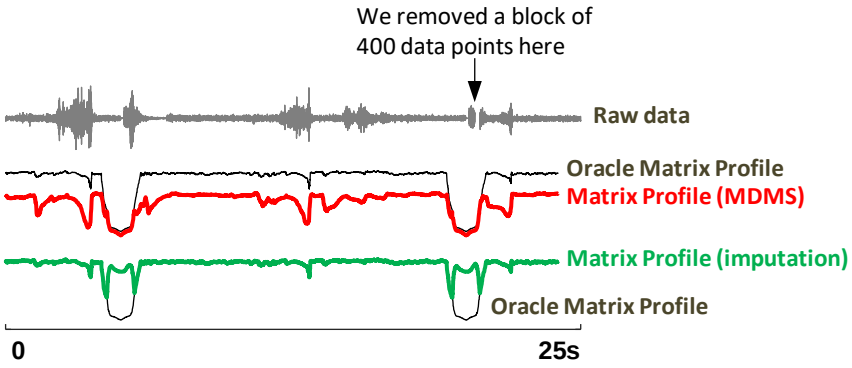

Figure 5.10. We removed 400 consecutive data points at the center of the second earthquake pattern. The oracle matrix profile computed with no missing data (black curve) finds the true event, as does MDMS (red curve) even in the presence of a large missing block. The Matrix Profile generated after linear imputation (green curve) fails to capture the minimum points within the oracle matrix profile.

We can see that the lower-bound matrix profile generated by our MDMS algorithm (red curve) agrees closely with the oracle matrix profile (black curve) in the vicinity of the two earthquake patterns, while the matrix profile generated after linear imputation (green curve) shows a high value at these locations.

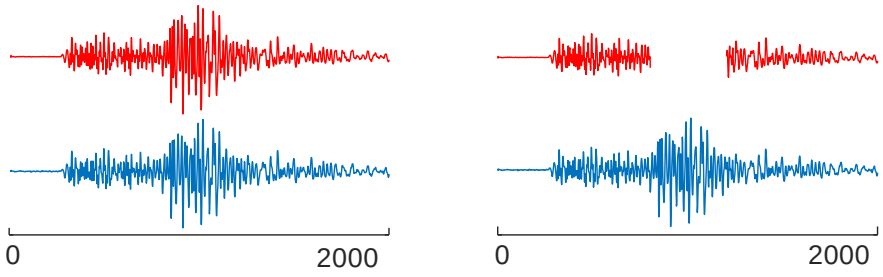

Figure 5.11. The first motif found by the MDMS algorithm (right) in the presences of a large missing block is identical with the first motif found in the oracle data (left).

As a result, the MDMS algorithm successfully captures the 1st motif (as shown in **Figure 5.11**) even in the presence of a large missing block within one earthquake pattern, while the imputation method misses the 1st motif within the oracle data. This illustrates two major strength of our algorithm over imputation methods. Firstly, our algorithm does not allow false negatives. Secondly, our algorithm is more robust to large missing blocks as it does not change the data, while imputation method can change the data a lot. In the next case study, we will further demonstrate the robustness of MDMS in the presence of missing blocks.

5.3.2 Case Study: Activity Data from Video

Time series extracted from video often has missing data reflecting "frame drops" due to bandwidth congestion [20]. To test our algorithm in this context, we examine the activity dataset of [86]. This dataset consists of a 13.3 minute 10-fps video sequence of an actor performing various activities. From this data, the original authors extracted 721 channels of the optical flow time series, and the length of each time series is 8,000. We consider the time series corresponding to the 533th channel, which is suggestive of the structure in places but is noisy. The data is shown in **Figure 5.12***.top,* the subsequence length is 120. From the oracle matrix profile in **Figure 5.12** (black curve), we can extract the 1st motif in the oracle data. To test the performance of the MDMS algorithm, here we remove 12 consecutive data points in the center of one of the 1st motif patterns. In **Figure 5.12**, we compare our lower-bound matrix profile result (red curve) with the matrix profile generated by linear imputation (green curve).

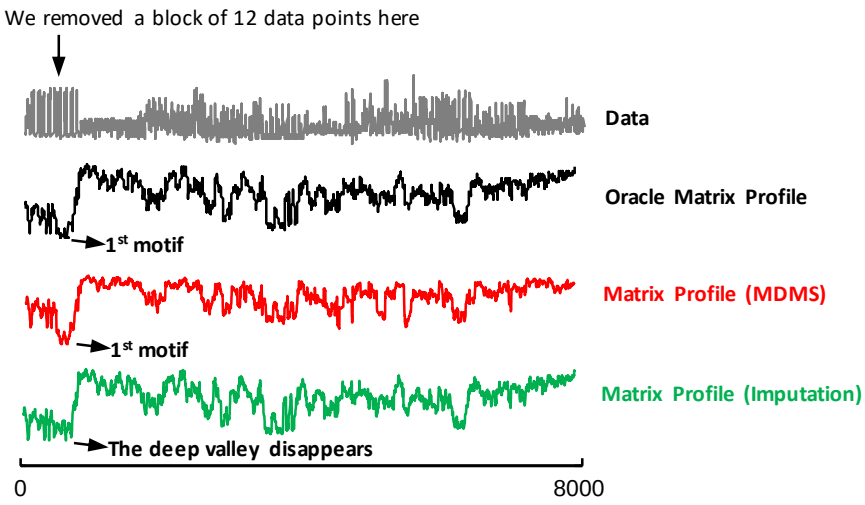

Figure 5.12. A raw activity time series. We removed 12 consecutive data points in one of the 1st motif patterns in the time series. The oracle matrix profile computed with no missing data (black curve) finds the true motif starting at the 540th and the 622nd data points. With the presence of 12 missing data points, the MDMS algorithm finds the same motif as the oracle result (red curve), starting at 520th and 602nd data points. The Matrix Profile generated after linear imputation (green curve) fails to capture the two deep valleys within the oracle matrix profile and thus misses the 1st motif.

We can see that the oracle matrix profile (black curve) shows two apparent valleys at the locations of the 1st motif, as does the lower bound matrix profile generated by MDMS (red curve). The 1st motif discovered by the MDMS algorithm (shown in **Figure 5.13**.*right*) is identical to the oracle motif (shown in **Figure 5.13**.*left*). In contrast, the matrix profile generated by imputation (green curve) does not have these valleys, and thus misses the 1st motif of the oracle data.

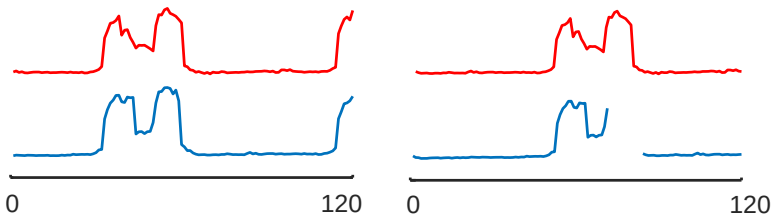

Figure 5.13. The first motif found by the MDMS algorithm (*right*) is identical to the first motif within the oracle data (*left*), despite a small phase shift.

We can see from **Figure 5.13** that though a large portion of the blue pattern is missing, our MDMS algorithm still finds it very similar with the red pattern. This example further demonstrates that our algorithm is robust against missing true motif patterns.

5.3.3 Quantifying the Robustness of MDMS

As MDMS evaluates the lower bound matrix profile, it naturally does not allow false negatives, but it can produce false positives. Here we perform two "stress tests" to evaluate the robustness and limitations of our MDMS algorithm against false positives.

We use the seismograph data in **Figure 5.9** again for the stress test. The subsequence length in this dataset is m=2,000.

We first test the sensitivity of MDMS over the length of missing blocks. Here we remove two missing blocks of length p, located at 7.5s and 15s respectively, from the data. In **Figure 5.14**, we show how the lower bound matrix profile varies as we increase p.

Note that the removed blocks are *not* within the two repeated earthquake patterns. As a result, the lower bound matrix profiles are the same as the oracle matrix profile in the vicinity of the two repeated patterns, while lower at other locations. In other words, it is easier to detect false positives with such setting.

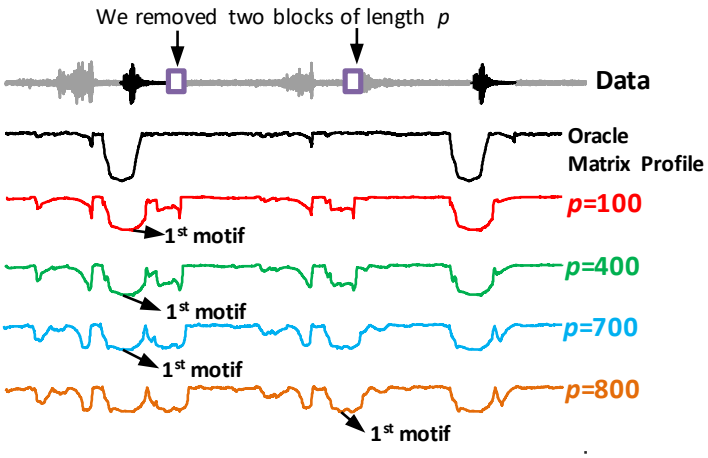

Figure 5.14. Lower bound matrix profiles corresponding to different missing block lengths. We removed 2 blocks of length *p* from the seismograph. The oracle matrix profile (black curve) finds the true motif. For *p*=100, *p*=400 and *p*=700, MDMS is able to find the true event as the 1st motif. When *p*=800, MDMS finds a false positive as the 1st motif.

We can see that when p=100 (5% of the subsequence length m), the side valleys in the oracle Matrix Profile become deeper, and two more side valleys show up in the vicinity of the removed blocks. As p increases, all the side valleys become deeper and deeper**.** For p=100, p=400 and p=700, we are able to find the true event as the 1st motif with MDMS. However, when p=800 (40% of m), the 1st motif (corresponding to the minimum point of the lower bound matrix profile) is no longer the true event. We show this false positive motif pair in **Figure 5.15**.

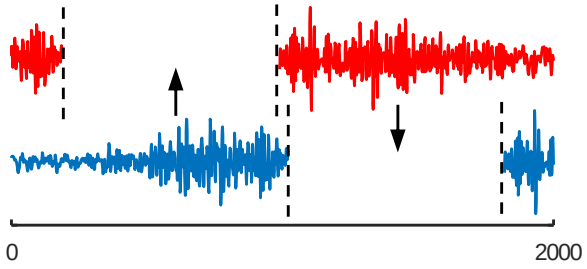

Figure 5.15. The 1st motif found by the MDMS algorithm when *p*=800.

The two subsequences are both in the vicinity of the second missing block in **Figure 5.14**. With a close inspection we can see why this pair is reported as the 1st motif by MDMS. If we fill in the missing part of each subsequence with their counterpart in the other subsequence (shown by the arrows in **Figure 5.15**), the two can be *very* similar to each other. Since MDMS does not allow false negatives, it will capture and report this possible matching pair.

Furthermore, **Figure 5.15** implies that p cannot be larger than 50% of the subsequence length, otherwise MDMS will be able to find a perfect match (with one subsequence missing the first 50% and the other missing the second 50%). When $p > 700$ (35% of the subsequence length), we are already very close to this limit. Therefore, in **Figure 5.15** we can see very deep side valleys in the vicinity of the missing blocks, and we are prone to detect false positives. When $p \leq 400$ (20% of the subsequence length), the two main valleys corresponding to the true events dominate, so we will not detect false positives.

We have demonstrated that MDMS is robust against discovering false positives when there are two missing blocks, and when the length of the two blocks are within a reasonable range. Next, we "stress test" the sensitivity of MDMS over the total number of missing values in the data. We again use the seismograph dataset in **Figure 5.9**, which consists of 50,000 data points. The lower bound matrix profile results are shown in **Figure 5.16**.

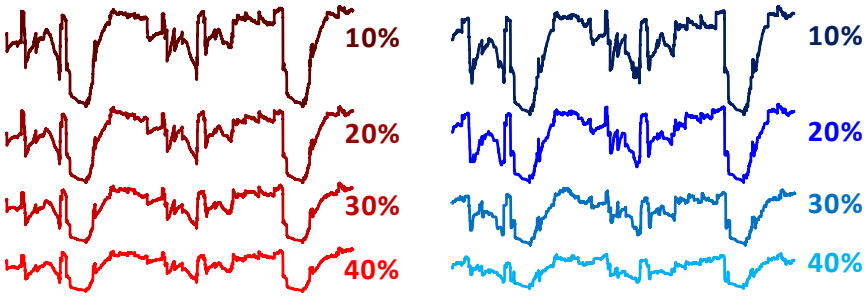

Figure 5.16. Lower bound matrix profiles corresponding to various percentage of data missing. *left)* random-missing data *right)* block-missing data.

In the first run, we randomly selected 5,000, 10,000, 15,000 and 20,000 points to remove from the data. From **Figure 5.16**.*left*, we can see that the scale of the matrix profile decreased as more points are missing. However, even when 40% of the data is missing, the two valleys corresponding to the true events (recall **Figure 5.9**) still dominate. In the second run, we removed blocks of length 200 from the data. The missing blocks were uniformly distributed, and the number of missing blocks increased from 25 to 100. **Figure 5.16**.*right* shows that even when 30% of the data is missing, the two main valleys still dominate. When 40% of the data is missing, the two main valleys are no longer apparent, but we can still find the true events as the 1st motif. The experiment demonstrates that MDMS is robust against detecting false positives even if a large percentage of data is missing.

5.4 Conclusions

We introduced what we believe to be the first time series motif discovery algorithm that can find motifs in the presence of missing data. The algorithm has the same time and space complexity as the fastest known algorithm for motif discovery, the STOMP

algorithm (Chapter 3). We formally proved the admissibility of our algorithm, it does not produce any false negatives. Experimental results show that our algorithm is also robust against false positives even when a large portion of the data is missing. Because our algorithm is based on creating a special version of the matrix profile, our work may have implications for other algorithms that can exploit the matrix profile, including discord discovery and time series joins. The lower bounds introduced can also be used to accelerate various length motif discovery. We leave such considerations to future work.

Chapter 6 Time Series Chains: A New Primitive for Time Series Data Mining

Time series motifs were introduced in 2002, and have since become a fundamental tool for time series analytics, finding diverse uses in dozens of domains. In this chapter we introduce Time Series Chains, which are related to, but distinct from, time series motifs. Informally, time series chains are a temporally ordered set of subsequence patterns, such that each pattern is similar to the pattern that preceded it, but the first and last patterns can be arbitrarily dissimilar. In the discrete space, this is similar to extracting the text chain “data, date, cate, cade, code” from text stream. The first and last words have nothing in common, yet they are connected by a chain of words with a small mutual difference. Time series chains can capture the evolution of systems, and help predict the future. As such, they potentially have implications for prognostics. We introduce two robust definitions of time series chains, and scalable algorithms that allow us to discover them in massive complex datasets.

6.1 On the Ubiquity of Time Series Chains

We have briefly and informally introduced time series chains in Section 1.2 (recall **Figure 1.2** and **Figure 1.3**). With a little introspection, it is easy to see that time series chains *should* exist in a host of diverse systems. Consider the following:

- **Human Heart**: An overcaffeinated heart can sporadically produce a pattern containing an extra beat, but over time the caffeine leaves the blood stream, and the pattern fades [40].
- **Distillation Process**: A distillation column is a ubiquitous industrial tool used to separate a mixture into its component parts. Ideally, most telemetry monitoring a distillation column should reflect a *repeating* process, over production cycles. However, most large distillation columns are open to the atmosphere, and the patterns observed may drift as the seasons change. In addition, a slowly clogging feed pipe can throttle the feed rate and force the patterns to drift until they become unacceptable and force maintenance.
- **Aggregate Human Behavior**: Human behavior is often unpredictable for *individuals*, but more structured in *aggregate*. For example, online shopping behaviors often shows conserved motifs, but these motifs may drift over time in response to advertising campaigns or cultural shifts. This has been noted in recent studies. For example, Krumme et al. [36] note that their attempts to model consumer e-commerce visitation patterns "*suggest the existence of a slow rate of environmental change or exploration that would slowly undermine the model's accuracy*."

- **Machines**: In general, most mechanical and electrical systems such as cars, motors, elevators, air conditioners, etc., are subject to gradual deterioration over time. This deterioration can be manifested in shorter or longer duty cycles, increased vibration, or some other gradually changing pattern. In the field of prognostics, the degree of deterioration is often called the State of Health (SoH) of the system. SoH is rarely directly measurable, and its estimation typically involves advanced modeling and estimation algorithms. Because a time series chain defines an implicit curve in some high-dimensional space, as shown in **Figure 1.2**, the natural coordinate along this curve can serve as a surrogate SoH measure. If high probability of failure can be associated reliably with a certain level of SoH, the discovered time series chain can be used successfully for prognostics and condition-based maintenance of machines.

As we will show in Section 6.4, once given the computational ability to find time series chains, we begin to find them everywhere, in datasets from ten seconds, to ten years in length.

6.2 Related Work and Background

Our review of related work is brief. To the best of our knowledge, there are simply no closely related ideas in the *time series* domain. However, there are very similar ideas in the *text* domain, even to the point of using similar language [102][9][94]. For example, Zhu and Oates discuss "*Finding Story **Chains** in Newswire Articles*" (analogous to our emphasis, [102]). Likewise, Bögel and Gertz argue for the need to go beyond finding

repeated variants of news articles (like *motifs*), to allowing "*Temporal Linking of News Stories*" (like *chains*, [9]). Beyond the difference in data type considered, this work is much more supervised. The user typically selects a particular news article, and asks "*what leads up to this?*" or "*what happened next?*". In contrast, because we are often exploring domains for which we have limited intuitions, we want to tell the algorithms nothing (except the desired length of patterns to consider) and have the algorithm find the natural chains in the data (if any).

There is a huge body of work in finding *periodicity* in time series [38]; however, this work is orthogonal to the task-at-hand. A time series can have perfect periodicity, but no chains (i.e. a pure sine wave), and a time series can have chains, but no appreciable periodicity (it is easy to construct artificial examples, for example by embedding increasing damped sine waves in random walk).

The notion of chains invokes the familiar idea of *concept drift* in [19], however, we are not starting with an explicit model to drift away from. Our starting point is a completely unannotated dataset.

Finally, time series chains are clearly related to time series motifs [56][96]. However, chains are neither a specialization nor a generalization of motifs. It is possible to have a rich set of motifs in a dataset, without having any chains (Our tilt-table example later in Section 6.4.1 illustrates that fact). Time series motifs have a rich and growing literature, we refer the reader to [96] and the references therein.

6.2.1 Developing Intuition for Time Series Chains

To help the reader understand the task at hand, and our contributions to it, we begin by considering a similar problem in a domain that better lends itself to discussion. In particular, it will be helpful to sharpen our intuitions on *strings*, the discrete analog to *time series*, and using the Hamming distance, the discrete analog to the Euclidean distance.

A *word ladder* is a classic puzzle used to challenge children to build their vocabulary [45]. The challenge is as follows: given two related words, such as "cat" and "dog", find a path between the words that consists of legal English words that differ only in one letter. For example, this instance is solved by {cat, cot, dot, dog}. By definition, each word is exactly a Hamming distance of 1 from both its neighbors. Let us consider variants of this problem. Suppose our words are subwords of length *m* in a longer, unpunctuated string *S*, of length *n*:

thecatsleepinginthecotwasawokenbydothedogwh…

Further suppose that we are challenged to find the longest ladder (or *chain*) of words in this string. We are told only that the words are of length 3, and that each word is *at most* a Hamming distance of 1 from both its neighbors. The problem is still tenable by eye, at least for this short string. However, the problem becomes significantly harder if the words are no longer constrained to be English words:

uifdbutmffqjohjouifdpuxbtbxplfoczepuifephxi…

This string is actually just the previous string Cesar-shifted by one letter, but without the intuition of meaningful words, the problem becomes much harder for the

human eye. Solving the problem with computers is also somewhat daunting. The obvious solution is depth-first-search, which only requires $O(n^2)$ space, but requires $O(n^n)$ time. If we constrain the subwords in a chain to have no overlap, the time complexity is slightly reduced to $O(n^{n/m})$.

Our consideration of strings allows further intuitive explanation of issues for the task at hand. Consider the following:

catauygfbatiuvheiucathoeircatiajesathfwecat…

Under the definition that each word is at most a Hamming distance of 1 from both its neighbors, this string has a chain of length six. However, this chain lacks *directionality*: the pattern is not drifting or evolving. Indeed, this "chain" might better be explained as multiple occurrences of a single prototypical pattern "cat", with some spelling errors. In the time series space, we already have a technique to find such patterns, *time series motifs* [56][96]. Thus, any definition we wish to formalize should guard against such pathological solutions.

Another important property that any definition of chains should have is *robustness*. Consider the following list of words that we will embed into a string {sad, had, ham, hag, rag}:

iwassadthatIhadahamsandwichwiththehaginrags…

Here we easily find the five-word chain. However, suppose we had a single letter misspelling in the string, for example:

iwassadthatIhadajamsandwichwiththehaginrags…

Because of this trivial single letter difference, we can only find two chains of length two, something that might easily have happened by chance. This brittleness of chains has been understood for centuries. Alexander Pope noted in 1733 "*From Nature's chain whatever link you strike, tenth or ten thousandth, breaks the chain alike*". Thus, when designing the definition of chains for the time series space, we want to make sure that our definition is robust to one or two links being missing in an otherwise long chain. This is especially important in the time series domain where we often encounter noisy/missing data.

In summary, considering a simpler but related problem, we can see that when designing a formal definition for our task at hand, we must strive (at a minimum) to make it *efficiently computable*, *directional*, and *robust*. In the next section, we will introduce a definition of time series chains that satisfy these requirements.

Finally, this is a good place to introduce some nomenclature. We plan to support two types of time series chains (here we show their analogs in a string):

- **Unanchored**: In this case we are interested in finding the unconditionally longest chain in the string. For example, considering *S*, the first string we introduced, FindChain(*S*, *m*, default) would find the longest chain (with $m = 3$) of length 4: {cat, cot, dot, dog}.
- **Anchored**: In this case we want to start the chain with a particular subsequence. For example, FindChain(*S*, *m*, 20) would find the longest chain (with $m = 3$) starting with the subword at index 20, which is {cot, dot, dog}.

Note that if we have discovered all the anchored chains, the unanchored chain is simply the longest one among them.

6.2.2 Time Series Notation

Before we formally define time series chains, we need to review some related time series definitions and notations, and create some new ones. Here we inherit the definitions and notations for time series (**Definition 2.1**), time series subsequence (**Definition 2.2**), distance profile (**Definition 2.3**), matrix profile (**Definition 2.4**) and matrix profile index (**Definition 2.5**) from Chapter 2.

Recall that the matrix profile is a data structure that stores nearest neighbor information for every subsequence in a time series, offering the solutions to many problems in time series data mining, including motif discovery and discord discovery [96]. We propose to leverage these ideas. However, it is useful for us to "re-factor" the computation into two halves, independently considering the nearest neighbor to the *left*, and the nearest neighbor to the *right*. Note that the total amount of computation we need to do is the same. **Figure 6.1** previews the two data structures: *left matrix profile* and *right matrix profile*. We could create the original *matrix profile* (**Definition 2.4**) by simply taking the minimum of the two.

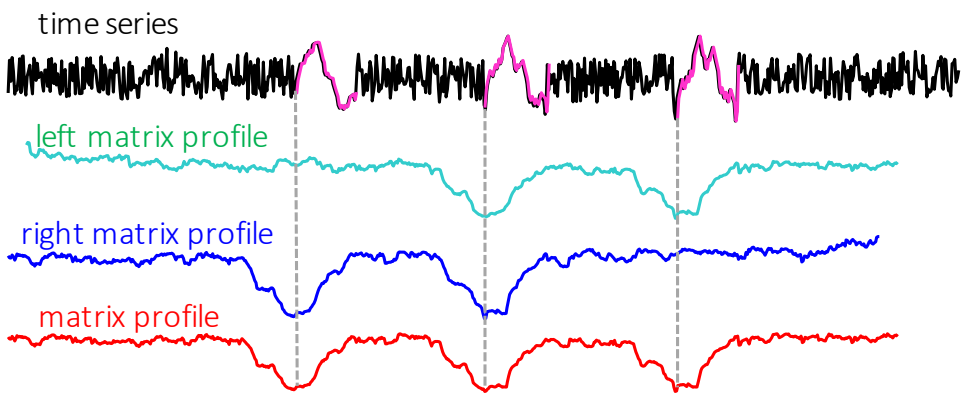

Figure 6.1. The left matrix profile, right matrix profile and matrix profile of a toy time series. The deep valleys within the (left/right) matrix profiles indicate that the corresponding subsequence has close (left/right) nearest neighbors. The matrix profile shows general nearest neighbor information.

Before introducing the *left matrix profile* and *right matrix profile*, we begin by showing that we can divide a distance profile ((**Definition 2.3**) into a *left distance profile* and *a right distance profile*.

Definition 5.1: A *left distance profile* DL_i of time series T is a vector of the Euclidean distances between a given query subsequence $T_{i,m}$ and each subsequence that appears before $T_{i,m}$ in time series T. Formally, $DL_i = [d_{i,1}, d_{i,2}, \ldots, d_{i,i-m/4}]$

Definition 5.2: A *right distance profile* DR_i of time series T is a vector of the Euclidean distances between a given query subsequence $T_{i,m}$ and each subsequence that appears after $T_{i,m}$ in time series T. Formally, $DR_i = [d_{i,\, i+m/4}, d_{i,\, i+m/4+1}, \ldots, d_{i,n-m+1}]$.

We can easily find the *left nearest neighbor* of a subsequence $T_{i,m}$ from the left distance profile, and the *right nearest neighbor* of $T_{i,m}$ from the right distance profile.

Definition 5.3: A *left nearest neighbor* of $T_{i,m}$, $LNN(T_{i,m})$ is a subsequence that appears before $T_{i,m}$ in time series T, and is most similar to $T_{i,m}$. Formally, $LNN(T_{i,m}) = T_{j,m}$ if $d_{i,j} = min(DL_i)$.

Definition 5.4: A *right nearest neighbor* of $T_{i,m}$, $RNN(T_{i,m})$ is a subsequence that appears after $T_{i,m}$ in time series T, and is most similar to $T_{i,m}$. Formally, $RNN(T_{i,m}) = T_{j,m}$ if $d_{i,j} = min(DR_i)$.

As shown in **Figure 6.1**, we use a vector called *left matrix profile* to represent the z-normalized Euclidean distances between all subsequences and their left nearest neighbors:

Definition 5.5: A *left matrix profile PL* of time series T is a vector of the z-normalized Euclidean distance between each subsequence $T_{i,m}$ and its left nearest neighbor in time series T. Formally, $PL = [\min(DL_1), \min(DL_2), \ldots, \min(DL_{n-m+1})]$, where DL_i ($1 \leq i \leq n-m+1$) is a left distance profile of time series T.

The i^{th} element in PL tells us the *distance* from subsequence $T_{i,m}$ to its left nearest neighbor in time series T. However, it does not tell *where* that left neighbor is located. This information is stored in a companion vector called the *left matrix profile index*.

Definition 5.6: A *left matrix profile index IL* of time series T is a vector of integers: $IL=[IL_1, IL_2, \ldots IL_{n-m+1}]$, where $IL_i=j$ if $LNN(T_{i,m}) = T_{j,m}$.

By storing the neighboring information this way, we can efficiently retrieve the left nearest neighbor of query $T_{i,m}$ by accessing the i^{th} element in the left matrix profile index.

Analogously, we define the *right matrix profile* (as shown in **Figure 6.1**) and the *right matrix profile index* as follows:

Definition 5.7: A *right matrix profile PR* of time series T is a vector of the Euclidean distances between each subsequence $T_{i,m}$ and its right nearest neighbor in time

series T. Formally, PR = [min(DR_1), min(DR_2), …, min(DR_{n-m+1})], where DR_i ($1 \leq i \leq n-m+1$) is a right distance profile of time series T.

Definition 5.8: A *right matrix profile index IR* of time series T is a vector of integers: IR=[IR_1, IR_2, … IR_{n-m+1}], where IR_i=j if $RNN(T_{i,m})$= $T_{j,m}$.

6.2.3 Formal Definitions of Time Series Chains

We are finally in the position to define time series chains. Before we do so, recall our guiding principle. We want something very like the definition of time series motifs [56][96], but with the additional property of *directionality*. For example, given a choice between the following:

{ ape → abe → ape → ape → abe → ape }

{ ape → apt → opt → oat → mat → man }

The latter is strongly preferred because the pattern is in some sense "evolving" or "drifting". We can now see this intuition in the *real-valued* space of interest. The definition below captures this spirit in the continuous case.

Definition 5.9: A time series chain of time series T is an ordered set of subsequences: TSC={$T_{C1,m}$, $T_{C2,m}$, … $T_{Ck,m}$} ($C1 \leq C2 \leq ... \leq Ck$), such that for any $1 \leq i \leq k-1$, we have $RNN(T_{Ci,m})$ = $T_{C(i+1),m}$, and $LNN(T_{C(i+1),m})$ = $T_{Ci,m}$. We denote k the *length* of the time series chain.

To help the reader better understand this definition, let us consider the following time series:

47, 32, 1, 22, 2, 58, 3, 36, 4, -5, 5, 40

Assume that the subsequence length is 1, and the distance between two subsequences is simply the absolute difference between them (to be clear, we are making these simple and pathological assumptions here just for the purposes of elucidation; we are actually targeting much longer subsequence lengths and using z-normalized Euclidean distance in our applications). According to **Definition 5.6** and **Definition 5.8**, we can store the left and right nearest neighbor information into the left and right matrix profile indices, as shown in **Figure 6.2**.

Index	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
Value	47	32	1	22	2	58	3	36	4	-5	5	40
IR	*12*	*8*	*5*	*8*	*7*	*12*	*9*	*12*	*11*	*11*	*12*	-
IL	-	*1*	*2*	*2*	*3*	*1*	*5*	*2*	*7*	*3*	*9*	*8*

Figure 6.2. The left nearest neighbor index and right nearest neighbor index of the toy example.

Here the Index vector shows the location of every subsequence in the time series, *IR* is the right matrix profile index and *IL* is the left matrix profile index. For example, *IR*[*2*] = 8 means the right nearest neighbor of 32 is 36; *IL*[*3*] = 2 means the left nearest neighbor of 1 is 32.

To better visualize the left/right matrix profile index, in **Figure 6.3** we use arrows to link every subsequence in the time series with its left and right nearest neighbors.

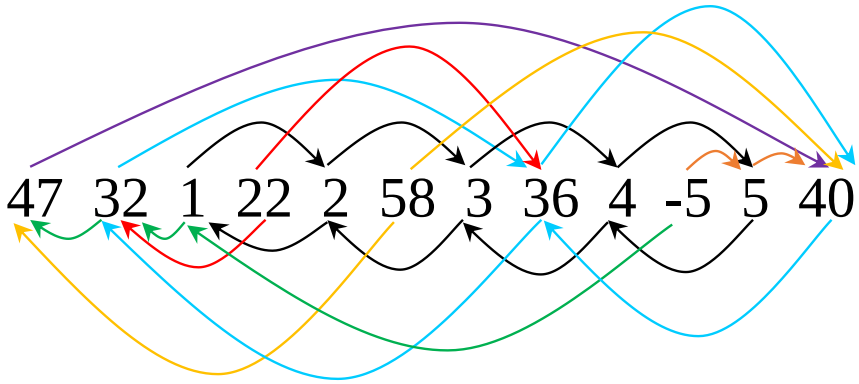

Figure 6.3. Visualizing left matrix profile index and right matrix profile index: every arrow above the time series points from a number to its right nearest neighbor; every arrow below the time series points from a number to its left nearest neighbor.

We call an arrow pointing from a number to its right nearest neighbor (arrows shown *above* the time series) a *forward arrow* (i.e. $x \rightarrow y$ means $RNN(x)=y$), and an arrow pointing from a number to its left nearest neighbor (arrows shown *below* the time series) a *backward arrow* (i.e. $x \leftarrow y$ means $LNN(y)=x$). **Definition 5.9** indicates that every pair of consecutive subsequences in a chain must be connected by both a forward arrow and a backward arrow. The diligent reader may quickly discover the longest time series chain in our toy example:

47, 32, 1, 22, 2, 58, 3, 36, 4, -5, 5, 40 (*Raw data*)

$1 \rightleftharpoons 2 \rightleftharpoons 3 \rightleftharpoons 4 \rightleftharpoons 5$ (*Extracted chain*)

We can see that this chain shows a gradual increasing trend of the data. Note that in this one-dimensional example, the elements of the chain can only drift by increasing or decreasing. In the more general case, the elements can drift in arbitrarily complex ways. Our claim is that our definition is also capable of discovering complex drifting patterns in high-dimensional space. For example, the reader can easily verify that the two-dimensional chain in **Figure 1.2**.*right*, a curvy evolving pattern, is captured by our

definition. The definition also works for a sin-wave drifting pattern, a zigzag, spirals, etc. We defer real-world examples in much higher dimensional spaces to Section 6.4.

However, to be clear, we are *not* claiming that we can discover *all* kinds of drifting; we are only targeting chains with *directionality* (the last item should be very different from the first item, as suggested previously). Therefore, full closed circles (i.e. {1, 3, 4, 5, 1}) are not captured by our definition. However, if needed, we can still potentially capture such topologies if we consider combining multiple chains. For example, in {1, 3, 4, 5, 1}, our definition captures two chains: {1, 1} and {3, 4, 5}. The circle is a combination of the two.

Beyond satisfyingly the *directionality* requirement, here we provide a simple sanity check of the *robustness* of our definition by *removing* a link from the chain. Imagine that in **Figure 6.3**, the number "3" is missing. In this case, $RNN(2)=4$, $LNN(4)=2$; we can still find the chain $1 \rightleftharpoons 2 \rightleftharpoons 4 \rightleftharpoons 5$. We defer a more "stressed" and quantified robustness test to Section 6.4.7.

The reader may wonder why we use a bidirectional chain definition here (i.e., using both *RNN* and *LNN*) instead of a unidirectional one (i.e., using only *RNN* or *LNN*). Consider the following example:

47, 11, 57, 12, 101, 13, 46, 14, 54, 15, 32, 1, 122, 2, 58, 3, 36, 4, -5, 5, 40

If we modify our chain definition such that every pair of consecutive subsequences are connected by only a *forward* arrow, then $11 \rightarrow 12 \rightarrow 13 \rightarrow 14 \rightarrow 15 \rightarrow 5$ would be valid. However, this sequence does not satisfy the directionality requirement of chains. Analogously, a chain definition based on only *backward* arrows would not work either:

$11 \leftarrow 1 \leftarrow 2 \leftarrow 3 \leftarrow 4 \leftarrow 5$ conforms to the definition, but violates directionality. In contrast, our bidirectional definition finds two chains: $11 \rightleftharpoons 12 \rightleftharpoons 13 \rightleftharpoons 14 \rightleftharpoons 15$ and $1 \rightleftharpoons 2 \rightleftharpoons 3 \rightleftharpoons 4 \rightleftharpoons 5$, both satisfying directionality.

Note that our bidirectional chain definition *can* find motifs that show no directionality, for example $1 \rightleftharpoons 1 \rightleftharpoons 1 \rightleftharpoons 1 \rightleftharpoons 1$. However, note that time series data normally contain some level of random noise, and motifs, though very similar to each other, are typically not *exactly* the same (especially in the high-dimensional space, see Section 6.4). Our bidirectional definition prevents motifs like 1.02, 1.01, 1.03, 0.98, 0.99 from being discovered as a chain.

As suggested in Section 6.2.1, we are especially interested in supporting two types of time series chains: anchored and unanchored chains. We formally define them as follows.

Definition 5.10: An *anchored* time series chain of time series T starting from subsequence $T_{j,m}$ is an ordered set of subsequences: $TSC_{j,m} = \{T_{C1,m}, T_{C2,m}, \ldots T_{Ck,m}\}$ $(C1 \leq C2 \leq ... \leq Ck, C1=j)$, such that for any $1 \leq i \leq k-1$, we have $RNN(T_{Ci,m})=T_{C(i+1),m}$, and $LNN(T_{C(i+1),m})=T_{Ci,m}$; for $T_{Ck,m}$, we have either $T_{Ck,m}$ is the last subsequence in T, *or* $LNN(RNN(T_{Ck,m})) \neq T_{Ck,m}$.

We can "grow" an anchored chain step-by-step as follows. Consider **Figure 6.3** as an example. If we start from 1, we find $RNN(1)=2$ and $LNN(2)=1$, so 2 can be added to the chain; since $RNN(2)=3$ and $LNN(3)=2$, 3 can also be added; this process continues until we reach 5. As $RNN(5)=40$ and $LNN(40) \neq 5$, the chain terminates, and finally we find the chain $1 \rightleftharpoons 2 \rightleftharpoons 3 \rightleftharpoons 4 \rightleftharpoons 5$ as the longest chain starting from 1.

Note that our definition produces one and only one anchored time series chain starting from any user-supplied subsequence $T_{j,m}$ ($1 \leq j \leq n-m+1$), as there is only one right (and also left) nearest neighbor for every subsequence in T. Based on this observation, we can find *all* the time series chains within T.

Definition 5.11: An all-chain set S_{TSC} of time series T is a set of all anchored time series chains within T that are not subsumed by another chain.

Here we are not simply finding all the anchored chains starting from all subsequences of T; S_{TSC} excludes those that are subsumed by another chain. For example, the all-chain set corresponding to **Figure 6.3** is S_{TSC} = {47, 32⇌36⇌40, 1⇌2⇌3⇌4⇌5, 22, 58, -5}. S_{TSC} does not contain the anchored chain 36⇌40, or 2⇌3⇌4⇌5, as they are both subsumed by longer chains.

Note that the all-chain set S_{TSC} has an important property: every subsequence of T appears exactly once in S_{TSC}. The all-chain set shows all possible evolving trends within the data.

We believe that of all the chains in S_{TSC}, the longest one should reflect the most general trend within the data. We call this chain the unanchored time series chain.

Definition 5.12: An *unanchored* time series chain of time series T is the longest time series chain within T.

Note that there can be more than one unanchored time series chain of time series T with the same maximum length. In case of such ties, we report the chain with minimum average distance between consecutive components. However, one might imagine other

tie-breaking criteria, such as choosing the chain with smaller variance of consecutive pairwise distances.

One can imagine in some situations that the chain of interest may not be the longest one. In the next section, we provide an algorithm to compute the all-chain set, which we use to easily find *any* anchored or unanchored chain, from the set.

6.3 Discovering Time Series Chains

To compute the time series chains, according to Definition 12, we first need to find the left/right nearest neighbor of every subsequence in the time series. Such information can be found from two vectors: left matrix profile index and right matrix profile index (**Definition 5.5** and **Definition 5.7**). The *LRSTOMP* algorithm is an (optimal) algorithm to efficiently compute these vectors.

6.3.1 LRSTOMP Algorithm

The STOMP algorithm introduced in Chapter 3 can efficiently compute matrix profile and matrix profile index in $O(n^2)$ time and $O(n)$ space. Here we briefly review how STOMP keeps track of the nearest neighbor of every subsequence: the algorithm computes distance profiles D_1, D_2, …, D_{n-m+1} (see **Definition 2.3**) in order. The matrix profile P is initialized as D_1 and the matrix profile index I is initialized as a vector of ones. As shown in **Figure 6.4**, once the computation of D_i is completed, we compare every element of D_i with its corresponding element in P: if $d_{i,j} < P_j$, we set $P_j = d_{i,j}$ and $I_j = i$. In this way, the matrix profile P and matrix profile index I keep track of the nearest neighbors of every subsequence in the time series.

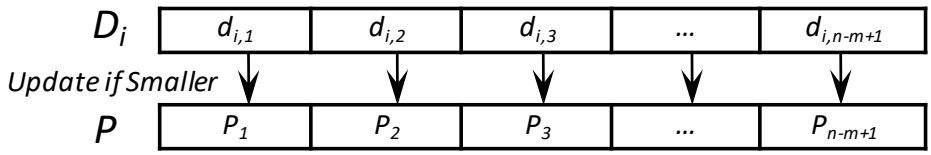

Figure 6.4. STOMP keeps track of the general nearest neighbor of every subsequence in the time series.

Instead of finding the general nearest neighbor information as in STOMP, to support chain discovery, we need to separately find the left and right nearest neighbors of each subsequence in the time series.

Leveraging off the insights of STOMP (Chapter 3), we call our algorithm LRSTOMP (Left-Right-STOMP). To initialize our four output vectors, we begin by setting both the left and right matrix profiles *PL* and *PR* as *Inf*s, and both the left and right matrix profile indices *IL* and *IR* as *zeros*. Then, using the technique in Section 3.2.1, we compute the distance profiles D_1, D_2, …, D_{n-m+1} (see **Definition 2.3**) in order. Note that the i^{th} subsequence can only be the right nearest neighbor of the 1^{st} to the $(i-m/4)^{th}$ subsequence in the time series, and the left nearest neighbor of the $(i+m/4)^{th}$ to the last subsequence in the time series. Therefore, as shown in **Figure 6.5**, after the i^{th} distance profile D_i is computed, we need to divide D_i into two halves. For $\forall\, j \in [1, i-m/4]$, if $d_{i,j} < PR_j$, we set $PR_j = d_{i,j}$ and $IR_j = i$. For $\forall\, j \in [i+m/4, n-m+1]$, if $d_{i,j} < PL_j$, we set $PL_j = d_{i,j}$ and $IL_j = i$.

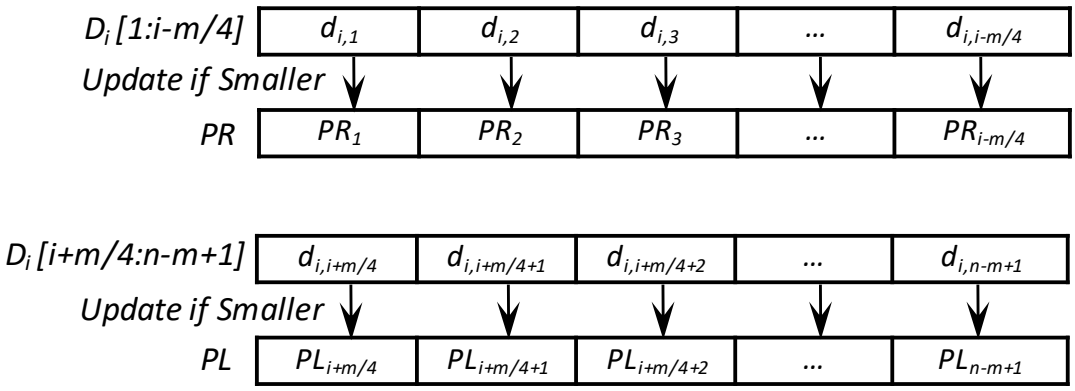

Figure 6.5. LRSTOMP keeps track of both the left and right nearest neighbors of every subsequence in the time series.

After evaluating all the distance profiles, we can obtain the final left and right matrix profiles (and matrix profile indices).

Note that switching the updating process from **Figure 6.4** to **Figure 6.5** does not affect the overall complexity of the algorithm. Therefore, the time complexity of LRSTOMP is $O(n^2)$ and the space complexity is $O(n)$, the same as STOMP (see Chapter 3).

6.3.2 Computing the Time Series Chains

Now we are in the position to compute the time series chains. We begin with the simpler variant, the algorithm (**Algorithm 8**) to compute the anchored time series chains (ATSC)**.**

The algorithm is straight-forward. We begin growing the chain from its user-specified anchor, the j^{th} subsequence. If the right nearest neighbor exists (if it does not exist, then $IR[j] = 0$; this indicates that we have reached the end of the time series) and

$LNN(RNN(T_{j,m})) = T_{j,m}$, then we set j as $RNN(T_{j,m})$ and add it to the back of the chain. The process iterates until nothing more can be added to the chain.

Algorithm 8: ATSC(*IL, IR, j*)

Input: The left matrix profile index ***IL*** and right matrix profile index ***IR*** generated by LRSTOMP(*T*, *m*), where *T* is the time series and m is the subsequence length; and ***j***, location of the anchor subsequence
Output: anchored time series chain ***C***, where $C[i] = j$ means the i^{th} element of the chain is the j^{th} subsequence in the time series

```
1  C ← [j]      // initialization
2  while IR[j]≠0 and IL[IR[j]] ==j do
3     j ← IR[j]
4     C ← [C, j]
5  end while
6  return C
```

The time and space overhead of the ATSC algorithm are both $O(n)$.

Given that we can efficiently compute the anchored time series chain starting from any subsequence, the all-chain set (ALLC) can also be computed. The (unanchored) time series chain is simply the longest chain in the all-chain set.

A simple approach to compute the all-chain set is enumerating all anchored chains starting from all subsequences, and removing those that are subsumed by longer chains. However, this brute-force approach would result in an undesirable $O(n^2)$ time complexity. Fortunately, as shown in **Algorithm 9**, by exploiting several properties of our definition of time series chains, we can reduce the time complexity of the ALLC algorithm to $O(n)$.

The vector L in line 1 is a vector of length $n-m+1$, the same length as the four meta time series. We use $L[i]$ to store the length of the anchored time series chain starting from $L[i]$, and initialize L with all ones (as the length of an anchored chain is at least 1). In lines 2 to 10, we iterate through all possible anchor points, and store in $L[i]$ the length of

the anchored chain starting from the i^{th} subsequence. We store all the chains found in S. In line 11, we find the unanchored time series chain corresponding to the maximum value in L.

Algorithm 9: ALLC(*IL*, *IR*)

Input: The left matrix profile index ***IL*** and right matrix profile index ***IR*** generated by LRSTOMP(*T*, *m*), where T is the time series and m is the subsequence length.
Output: The all-chain set ***S*** and the unanchored chain ***C***

```
1  L ← ones, S ← ∅                    //initialization
2  for i ← 1 to Length(IR) do
3      if L[i] ==1 do
4          j ← i, C ← [j]
5          while IR[j]≠0 and IL[IR[j]] ==j do
6              j ← IR[j], L[j] ← -1, L[i] ← L[i] + 1, C ← [C, j]
7          end while
8          S ← S ∪ C
9      end if
10 end for
11 C ← ATSC(IL, IR, MaxIndex(L))
12 return S, C
```

Note that in lines 5-7, as we grow an anchored chain from the i^{th} subsequence, we set $L[j]$ to *-1* for every subsequence j visited except the anchor subsequence. This helps us prune unnecessary computations, as there is only one anchored time series chain starting from any subsequence. Consider again the toy example in **Figure 6.3**: when i=3, we discover the chain $1 \rightleftharpoons 2 \rightleftharpoons 3 \rightleftharpoons 4 \rightleftharpoons 5$. By marking out the length of the anchored chain starting from 2, 3, 4 and 5 as -1s, we can avoid spending time on growing a chain like $2 \rightleftharpoons 3 \rightleftharpoons 4 \rightleftharpoons 5$, which is subsumed by $1 \rightleftharpoons 2 \rightleftharpoons 3 \rightleftharpoons 4 \rightleftharpoons 5$. With this technique, every subsequence in the time series is visited exactly once; therefore, the time complexity of the algorithm is $O(n)$, which is inconsequential compared to the $O(n^2)$ time (already demonstrated as ultra-fast in Chapter 3) to compute the left/right matrix profiles and matrix profile indices. This $O(n)$-complexity algorithm is the optimal algorithm to

compute the all-chain set under our definitions and assumptions, as we need to at least scan through the entire time series once.

Although we will mainly be showing the applications of the unanchored (or longest) time series chain, sometimes the chain of interest may not be the longest one. Based on domain knowledge, one may be interested in looking at the top-k chains, a chain starting from a specific location, a chain with less difference between the links, etc. All these tasks are trivial given the all-chain set *S*. Therefore, the ALLC algorithm can potentially help us discover any possible evolving trend within the time series. We reserve such considerations for future work.

6.3.3 Uniform Scaling Time Series Chains

In the previous sections, we have introduced the definitions and algorithms for chains with a fixed subsequence length. That is, we assumed that all the patterns (links) in a chain are of the same length, and that they evolve by changing the values of the patterns. The reader may have already speculated that there may exist other forms of chains which do not evolve by changing the *values* of patterns, but by changing their *length*, with the patterns either getting longer or shorter over time. **Figure 6.6** shows one such example. This idea is familiar in many human endeavors; in music the general term is called *tempo rubato*, with *ritardando* indicating a slowing down, and *accelerando* meaning a speed up. Changing tempo is also a key element of many types of dance, including the Sama (the "Whirling Dervish" dance)**.**

Many other domains may see uniform scaling time series chains. For example, electrical power demand for a house may show a seasonal effect as the occupants turn on the air conditioning earlier and earlier each day as the warmer summer looms.

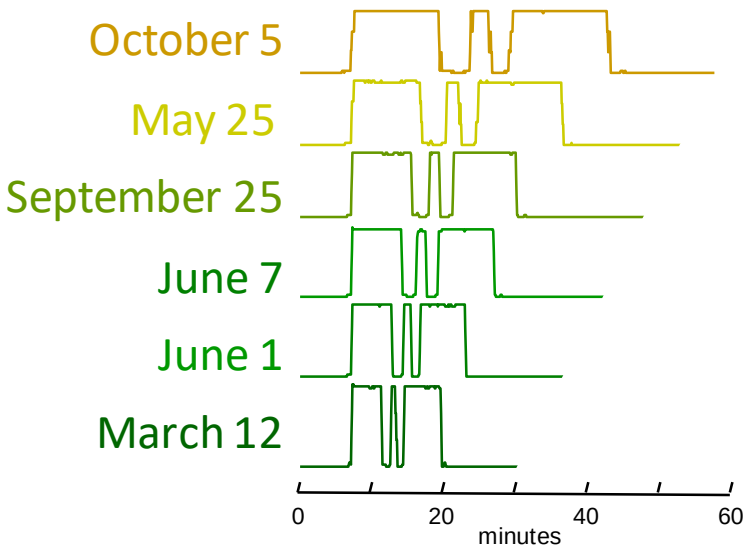

Figure 6.6. A uniform scaling time series chain we discovered in a household electrical demand time series [25]. Over twenty months the dishwasher cycle became progressively longer, perhaps as an inlet valve became progressively more clogged.

Note that our previous chain definition and algorithms do not apply to uniform scaling time series chains. To show this, in **Figure 6.7**.*top* we created a synthetic time series containing an embedded uniform scaling time series chain (the three patterns within the embedded chain are highlighted in red, located at 100, 200 and 300, with lengths 50, 25 and 18 respectively). We set a fixed subsequence length 50 and run the ALLC algorithm (**Algorithm 9)** to find the unanchored chain. From the results in **Figure 6.7**.*bottom* (highlighted in green) we can see the algorithm found another chain with evolving values instead of the embedded uniform scaling chain. The reason is that the embedded patterns are unable to locate each other as their left/right nearest neighbors

with a fixed subsequence length. For example, the first embedded pattern (located at 100) finds its right nearest neighbor at 299 instead of at 200; the second embedded pattern (located at 200) finds its left nearest neighbor at 43 instead of at 100, etc. Therefore, we need another approach to discover this special form of time series chains.

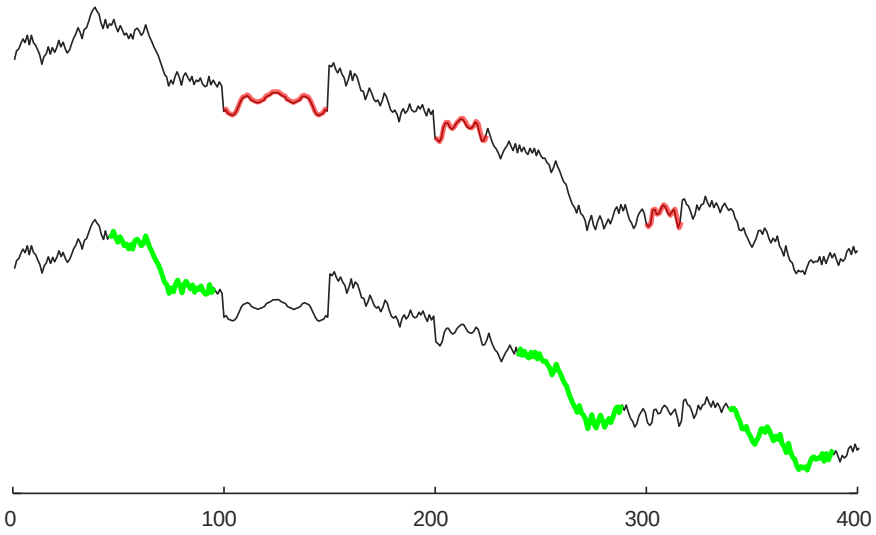

Figure 6.7. *top*) A time series containing a uniform scaling chain. *bottom*) the chain discovered with a fixed subsequence length 50.

Our approach is visualized in **Figure 6.8**. Note that the first embedded pattern (shown in red at **Figure 6.8**.*top*) is of length 50, twice the length of the second pattern (shown in pink at **Figure 6.8**.*top*). If we rescale the length of the original time series *T* by 200%, we can obtain a new time series *T'* (**Figure 6.8**.*bottom*): The red pattern in *T* will find the pink pattern as its nearest neighbor in *T'*, and vice versa. Similarly, if we create another time series *T''* by stretching the original time series to 300%, then the first pattern in *T* will find the third pattern (purple) as its nearest neighbor in *T''*, and vice versa.

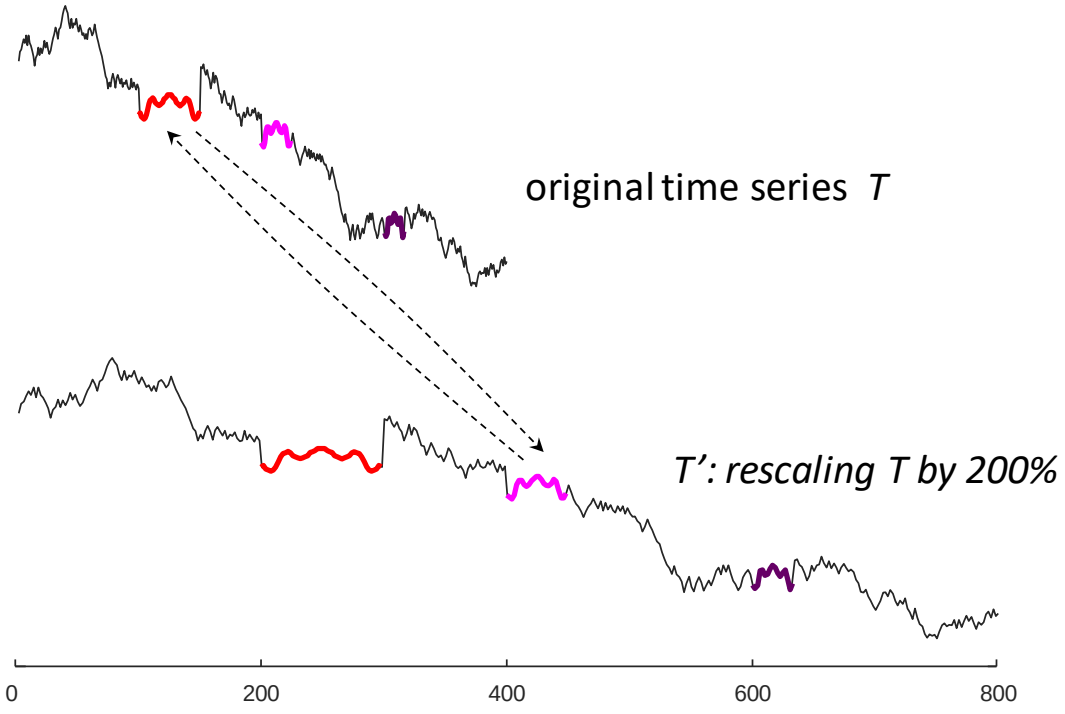

Figure 6.8. ***top*****) The original time series.** ***bottom*****) Rescaling the original time series by 200%. The first (red) pattern in the original time series matches very well with the second (pink) pattern in the rescaled time series.**

Based on this observation, we have created the *UniformScaleChain* Algorithm to discover the uniform scaling time series chains, as shown in **Algorithm 10**.

The algorithm requires the user to input the time series of interest *T*, a base subsequence length *m* and a number of *Scales* to explore. For the time series in **Figure 6.8**, *m* can be 50, and *Scales* can be [100%, 120%, 140%, …, 300%]. The step size between the scales can be larger or smaller. Lines 3-11 iterate through these scales. In line 4, we create a new time series *T'* by rescaling the original time series *T*. Line 5 evaluates the matrix profiles and matrix profile indices corresponding to the join of *T* and *T'* with the AB-join version of the STOMP algorithm (Chapter 3). *PI*[*j*] is the location of the nearest neighbor of $T_{j,m}$ in *T'* and *PI'*[*j*] is the location of the nearest neighbor of

$T'_{j,m}$ in T. Line 7 verifies whether the two arrows in **Figure 6.8** form a loop: if the red pattern and the pink pattern are links of a uniform scaling chain, then the pink pattern in T' must be the nearest neighbor of the red pattern in T and vice versa. In line 8, if $T_{j,m}$ finds such a match in T', then $I_{Valid}[j][i]$ stores the original location of the match in T; otherwise $I_{Valid}[j][i]$ is zero.

Algorithm 10: UniformScaleChain(*T, m, Scales*)

Input: time series ***T***, subsequence length ***m,*** a set of possible scales ***Scales***.
Output: uniform scaling time series chain ***C*** and a companion length vector ***S,*** where $C[i]=j$ means the i^{th} element of the chain is the j^{th} subsequence in the time series and $S[i]=k$ means the i^{th} element of the chain is of length $k \times m$

```
1  L ← |T|- m + 1
2  I_Valid ← Zeros(L,|Scales|)              //initialization
3  for i ← 1 to |Scales| do
4      T' ← T[1 : 1 / Scales[i] : end]      // rescale T
5      [P, PI, P', PI'] ← STOMP(T, T', m)   //  Chapter 3
6      for j ← 1 to L do
7          if PI'[PI[j]] == j do
8              I_Valid[j][i] ← PI[j] / Scales[i]
9          end
10     end
11 end
12 ALLC ← ∅, ALLS ← ∅                       // initialize all-chain set
13 for j ← 1 to L do
14     C ← j, S ← 1
15     [Val, Loc] ← NonzeroElements(I_Valid[j])
16     if Exist(Val) do
17         i ←1, C ← C ∪ Val[1], S ← S ∪ Scales[Loc[1]]
18         if Val[1] > j do                 //chain scale increasing
19             while i < |Val| and Val[i+1] > Val[i] do
20                 i ← i+1, C ← C ∪ Val[i], S ← S ∪ Scales[Loc[i]]
21             end
22         elseif Val[1]<j do               //chain scale decreasing
23             while i < |Val| and Val[i+1] < Val[i] do
24                 i ← i+1, C ← C ∪ Val[i], S ← S ∪ Scales[Loc[i]]
25             end
26         end
27     end
28     ALLC ← ALLC ∪ C, ALLS ← ALLS ∪ S
29 end
30 C, S ← Longest(ALLC, ALLS)
31 return C, S
```

In lines 13-29, we attempt to "grow" a uniform scaling chain from every possible anchor in the original time series *T*. In line 15, the non-zero elements in $I_{Valid}[j]$ indicate all the possible links that can be included in a chain anchored at *j*. Note that the length of each link in the uniform scaling chain can be either increasing over time, or decreasing over time; We handle the former case in lines 18-21, and the latter case in lines 22-25. In lines 18-21, we find the longest chain with increasing link lengths starting from *j*; in lines 22-25, we find the longest chain with decreasing link lengths ending in *j*. After finding all the uniform scaling chains starting from all anchors, we return the longest one and the corresponding scales of its links in line 30.

Finally, it is clearly possible that there may be chains that exhibit both shape and length evolution at the same time. Such patterns may best be discovered by a fusion algorithm. We leave such considerations to future work.

6.4 Empirical evaluation

"*You reasoned it out beautifully, it is so long a chain, and yet every link rings true*."
Sir Arthur Conan Doyle: Adventures of Sherlock Holmes, 1892.

We note in passing that all the experimental results in this paper are reproducible. To ensure this, we have created a website to archive all the datasets and code in perpetuity [76].

After an extensive literature search, we are convinced that there is no strawman algorithm to compare to. Moreover, unlike clustering or motif discovery, there is no formal metric to measure the quality of chains. In a sense, we are not the ideal group that should invent such a metric, as we could define one that tautologically rewards the

properties *we* have defined. However, in Section 6.4.7, we provide a pseudo measure of quality as the gold standard. We measure the length of a chain in a data set, then ask how robust would our chain discovery algorithm be if we distorted the data in various ways. Clearly a chain discovery algorithm would engender little confidence if minor changes to the data could prevent the discovery of the (same basic) chain.

Before this robustness test, we provide four case studies in which we applied our algorithm to various datasets. These case studies will help the reader gain an appreciation for the utility of chain discovery. These datasets are designed to span the diverse types of data encountered in time series data mining, some are stationary, some have trends, some are smooth, some are noisy, the shortest is ten seconds long, the longest is ten years, etc.

While we can obtain the all-chain set with the ALLC algorithm, in this section, we are mainly showing the application of the unanchored time series chain. Unless otherwise stated, in the rest of this section, we use the term "time series chain" to represent unanchored time series chain in **Definition 5.12**, rather than **Definition 5.9**.

6.4.1 Case Study: Hemodynamics

In November 2016, we briefed Dr. John Michael Criley, Professor Emeritus at the David Geffen School of Medicine at UCLA, and Dr. Gregory Mason of UCLA Medical Center, a noted expert on cardiac hemodynamics, on the capabilities of time series chain discovery. They suggested more than a dozen possible uses for it in various clinical and research scenarios in medicine. Here we consider one example they are interested in.

Syncope is the loss of consciousness caused by a fall in blood pressure. The tilt-table test (see **Figure 6.9**.*top*.*left*) is a simple, noninvasive, and informative test first described in 1986 as a diagnostic tool for patients with syncope of unknown origin [29].

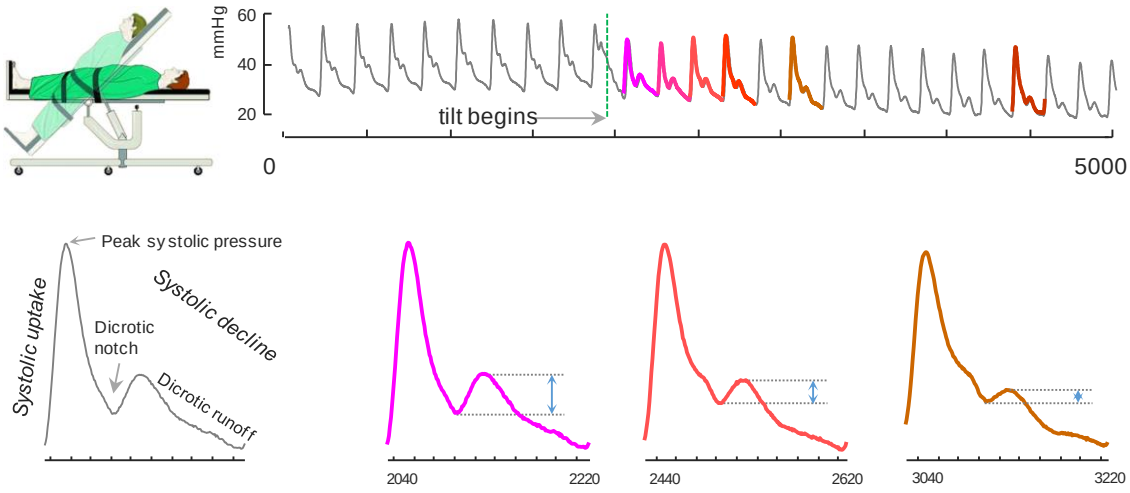

Figure 6.9. ***left-to-right*, *top-to-bottom*) A patient lying on a medical tilt table has his arterial blood pressure monitored. Nomenclature for a standard beat. The chain discovered in this dataset shows a decreasing height for the dicrotic notch.**

Beyond diagnosing the condition, the test may reveal the cause, neurological disorder, metabolic disorder, mechanical heart disease, cardiac arrhythmias, etc. [48].

In brief, the clinician will want to contrast any evolving patterns in the patient's arterial blood pressure (ABP) that are a response to changes in positon induced by a tilt table, with evolving patterns that are not associated with changes of posture. As hinted at in **Figure 6.9**, time series chains are an ideal way to find and summarize such patterns. Here we set m=200, as this is the typical length of an ABP signal (**Figure 6.9**.*bottom*.*left*).

Figure 6.9 shows just a snippet of the time series searched. We encourage the reader to see the full dataset/results at [76]. Nevertheless, even this snippet is visually

compelling. It shows that as the table is tilted, the height of the dicrotic notch steadily decreases. Per Dr. Mason, the change in orientation "*dramatically increases central venous filling and subsequent left ventricular end-diastolic volume, for several heart beats. Left ventricular stroke volume and effective cardiac output increase transiently*, (likely due to) *relative hyperemia, which is well-described during recovery from transient vascular occlusion*".

As noted above, **Figure 6.9** only shows a small section of the data we searched. In addition to finding meaningful chains, a good algorithm should avoid finding spurious chains, even if there are dense motifs (recall the distinction visualized in **Figure 1.2**). In **Figure 6.10** we show the prefix of the data we searched, but truncated out of **Figure 6.9**, gratifyingly, the chain we discovered has no element here, even though there are clearly dense motifs [96].

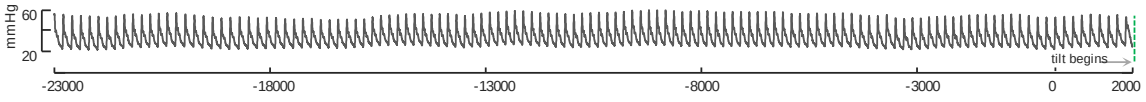

Figure 6.10. The prefix of the ABP data shown in Figure 6.9. There are no chain elements discovered in this region, although it is compressed of dense motifs.

6.4.2 Backtracing

We have shown in **Figure 6.9** that the unanchored chain reveals the gradual drifting process of a system. Sometimes it may be interesting to explore the data "backwards". That is to say, if we inspect an abnormal signal at the end of the drifting process, can we go backward to find *when* the system started to drift and possibly glean some insight as to what caused the drifting?

In contrast to the anchored chain definition in Section 6.2.3 which discovers a chain forward from an anchor, here we try to "grow" a chain backward from the end of a chain to find its origin. Note that our chain definition is *symmetric*: as shown in **Figure 6.3**, every pair of consecutive subsequences in a chain must be connected by both a forward arrow and a backward arrow. That is, if $T_{i,m}$ and $T_{j,m}$ are two consecutive subsequences in a chain, then $RNN(T_{i,m}) = T_{j,m}$ and $LNN(T_{j,m}) = T_{i,m}$. Therefore, if we grow a chain from its last link backward, we will find exactly the same chain as the one grown from its first link forward.

We call this process *backtracing*. The algorithm to discover a backtracing chain is just a simple modification of the ATSC algorithm (**Algorithm 8**): we begin growing the chain backward from the user-specified anchor (the last link), the j^{th} subsequence. If the left nearest neighbor of $T_{j,m}$ exists, and $RNN(LNN(T_{j,m}))=T_{j,m}$, then we set j as $LNN(T_{j,m})$ and add it to the front of the chain. The process iterates until nothing more can be added to the front of the chain.

We apply the backtracing algorithm to the ABP data from the last section. **Figure 6.11**.*top* shows an expanded view of the signal. At the end of the data, we find an abnormal signal (shown in red), indicating the system has drifted, and we would like to trace back to discover what causes the drifting. **Figure 6.11**.*middle* shows the backtracing chain discovered backward from the abnormal signal. The discovered chain indicates that the system starts drifting at around the 15,000th data point, which aligns very well with the ground truth when the bed starts to tilt.

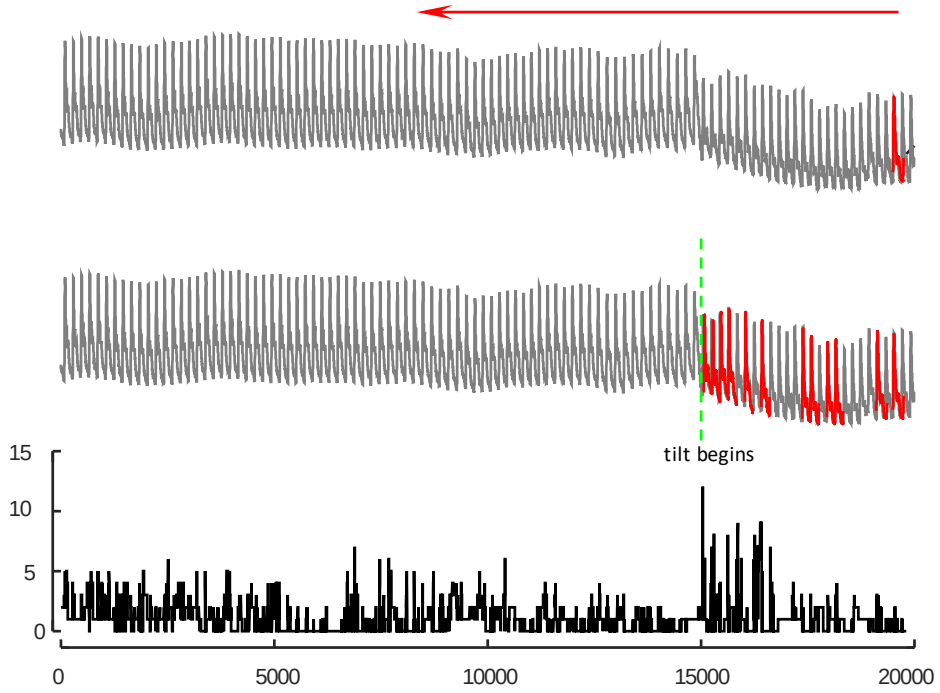

Figure 6.11. ***top)*** **An expanded view of the ABP data shown in Figure 6.9. We trace back from an abnormal pattern located at the end of the data.** ***middle)*** **The chain discovered.** ***bottom)*** **The length of chains starting from every anchor.**

In addition, we have also investigated the relationship between the all-chain set and the ground truth: **Figure 6.11**.*bottom* shows the length of chains starting from every anchor in the all-chain set, and we can see that the length of the chains becomes exceptionally large near the drifting point 15,000.

The results suggest that time series chains can automatically identify when the system starts to drift. This piece of information may be very helpful in prognostics applications, i.e., locating what causes the system to deteriorate, so that corresponding maintenance can be scheduled in time.

6.4.3 Case Study: Penguin Behavior

In this case study, we decided to explore a dataset for which we have no expertise, to see if we could find time series chains, which we could then show to an expert for independent evaluation of meaning and significance (if any).

To this end, we consider telemetry collected from a Magellanic penguin (*Spheniscus magellanicus*). The dataset was collected by attaching a small multi-channel data-logging device to the bird. The full data consist of 1,048,575 data points recorded at 40 Hz (about 7.2 hours). While a suite of measurements was recorded, for simplicity we focus on the X-axis acceleration (the direction of travel for a swimming bird). In **Figure 6.12** we show the snippet of the data in which we found a chain, with $m = 28$. This is about 0.7 seconds, and the approximate period of the data.

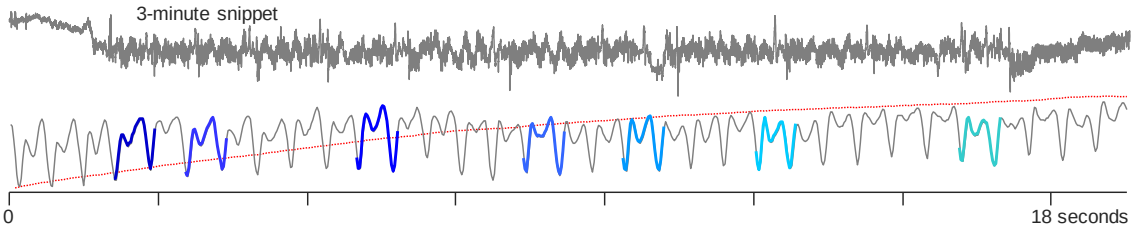

Figure 6.12. *top*) A random three-minute snippet of X-Axis acceleration of a Magellanic penguin (from a total of 7.2 hours). *bottom*) An eighteen-second long section containing the time series chain. In the background, the red time series records the depth, starting at sea-level and leveling off at 6.1 meters.

In fact, this chain *does* have a simple interpretation. Adult Magellanic penguins regularly dive to depths of up to 50 m to hunt prey, and may spend as long as fifteen minutes under water. One of our sensors measures pressure, which we showed in **Figure 6.12**.*bottom* as a fine/red line. This shows that the chain begins just after the bird begins its dive, and ends as it reached its maximum depth of 6.1 m. Magellanic penguins have

typical body densities for a bird at sea-level, but just before diving they take a very deep breath that makes them exceptionally buoyant [58]. This positive buoyancy is difficult to overcome near the surface, but at depth, the compression of water pressure cancels it, giving them a comfortable neutral buoyancy [58][93]. To get down to their hunting ground below sea level, it is clear that "(for penguins) *locomotory muscle workload, varies significantly at the beginning of dives*" [93]. The snippet of time series shown in **Figure 6.12** does not suggest much of a change in *stroke-rate*, however penguins are able to vary the thrust of their flapping by twisting their wings [93]. The chain we discovered shows this dramatic sprint downwards leveling off to a comfortable cruise. Fortunately, our data contain about a dozen major dives, allowing us to confirm our hypothesis about the meaning of this chain on more data.

Note that our chain does not include every stroke in the dive. Our data are undersampled (only 40Hz for a bird that can swim at 36kph) and these data are recorded in the wild, the bird may have changed directions to avoid flotsam or fellow penguins. However, this is a great strength of our algorithm: we do not need "perfect" data to find chains; we can find chains in real-world datasets. Also, from **Figure 6.12**.*bottom* we can see that m=28 is longer than the actual period of the data; our algorithm is not sensitive to this and still discovered a meaningful chain.

6.4.4 Case Study: Human Gait

In the experiments in the previous section we could be sure of the validity of the discovered chains, because we had access to some ground truth. In this section and the next, we show examples of chains we discovered in datasets for which we do not have an

obvious way to empirically verify. This demonstrates one use for chains, finding patterns that are interesting but speculative, and may warrant further investigation.

We first consider a snippet of a gait dataset recorded to test a hypothesis about biometric identification [30]. The dataset is shown in **Figure 6.13**.*top*. We set $m = 50$ here, as this is the approximate length of a period of the data.

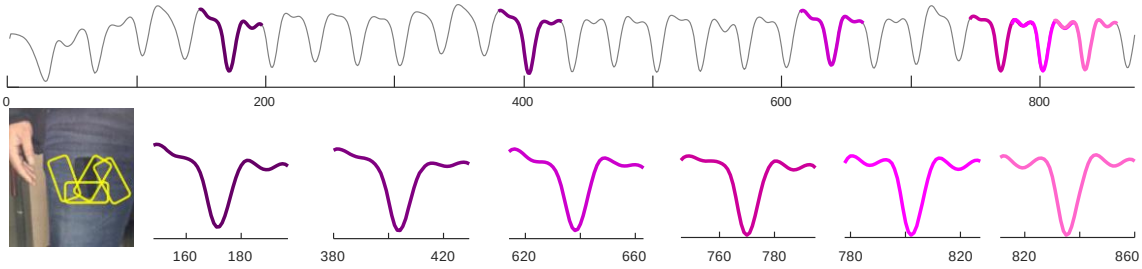

Figure 6.13. ***top*) A 30-second snippet of data from an accelerometer on a mobile phone. The phone was placed in the user's front pocket (*inset*). *bottom*) The extracted chain shows an evolution to a stable and symmetric gait.**

As hinted at in **Figure 6.13**.*inset* (taken from the original paper), the authors of the study were interested in "*the instability of the mobile in terms of its orientation and position when it is put freely in the pocket*" [30]. Given the experimental setup, we suspected that the gait pattern might start out as being unpredictable as the phone jostled about in the user's pocket, eventually settling down as the phone settled into place. This is exactly what we see in **Figure 6.13**.*top*. Note that the first few links are far apart and asymmetric, but the last few links are close together, and almost perfectly symmetric.

6.4.5 Case Study: Web Query Volume

In contrast to the smooth, stationary, oversampled accelerometer data considered in the last section, we next consider a dataset that is noisy, undersampled and has a growing trend. We examined a decade-long *GoogleTrend* query volume for the keyword *Kohl's*,

an American retail chain (data courtesy of [43]). As shown in **Figure 6.14**, the time series features a significant "bump" around the end-of-years holidays, unsurprising for a store known as a destination for gift buyers. Here we set m = 76 (the approximate length of a "bump").

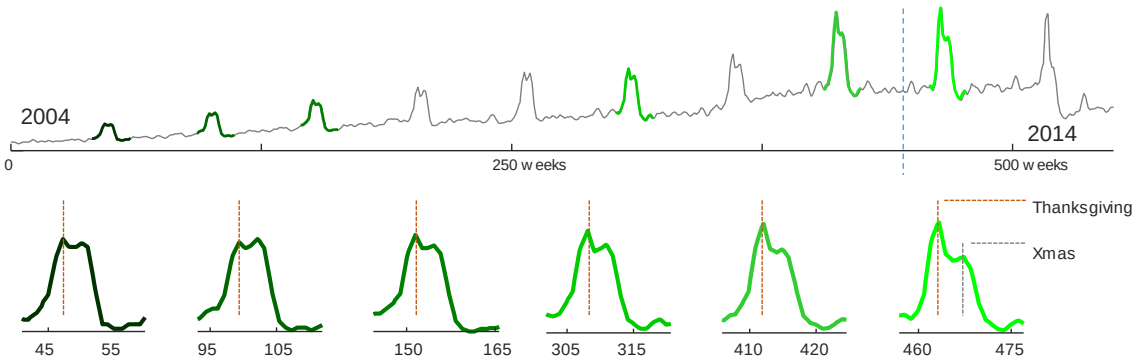

Figure 6.14. ***top*) Ten years of query volume for the keyword *Kohl's*. *bottom*) The z-normalized links of the time series chain discovered in the data hints at the growing importance of "*Cyber Monday*".**

The discovered chain shows that over the decade, the bump transitions from a smooth bump covering the period between Thanksgiving and Xmas, to a more sharply focused bump centered on Thanksgiving. This seems to reflect the growing importance of *Cyber Monday*, a marketing term for the Monday after Thanksgiving. The phrase was created by marketing companies to persuade people to shop online. The term made its debut on November 28th, 2005 in a press release entitled "*Cyber Monday Quickly Becoming One of the Biggest Online Shopping Days of the Year*" [72]. Note that this date coincides with the first glimpse of the sharping peak in our chain.

Here we seem to "miss" a few links in the chain. However, note that the data is noisy and coarsely sampled, and the "missed" bumps are too distorted to conform with the general evolving trend. This noisy example again illustrates the robustness of our

technique. As before, we note that we do not need "perfect" data to find meaningful chains. Even if some links are badly distorted, the discovered chain will still be able to include all the other evolving patterns.

Furthermore, consider the potential of using chains to predict the future. Assume that we are now at mid-2012 (the location of the blue line in **Figure 6.14**.*top*. We would like to forecast the shape *S* of the fist "bump" *after* the blue line, given the data *before* it.

In the data prior to mid-2012, we discovered a chain that consists of the first five links in **Figure 6.14**.*bottom* (call them S_1, S_2, S_3, S_4, S_5). Our assumption is that the difference between S_4 and S_5 is the same as the difference between S_5 and *S*. We compare our prediction result with a popular strawman in the literature, *persistence* prediction (i.e. which assumes $S = S_5$) [70], in **Figure 6.15**. Our simple, chain-based prediction method is more accurate (especially in the center part), as it captures the trend of the data.

Figure 6.15. ***left*) Our predicted shape (blue) is very similar to gound truth (red), with a Root Mean Squared Error (RMSE) of 0.17.** ***right*) Persistence prediction result (blue) is less similar to the ground truth (red), with a RMSE of 0.18.**

6.4.6 Parameter Setting

We have demonstrated the efficacy of our discovery algorithm, given a time series of interest and an appropriate subsequence length *m* to use. We do not consider *m* as a true parameter, as this is a user *choice*, and it is a required input for all existing motif

discovery algorithms (see [56][96] and the references therein). Nevertheless, the reader may still wonder how sensitive our chain discovery algorithm is to *m*.

To explore this, we again consider the Kohl's data in **Figure 6.14** (the original subsequence length is 76). Here we set *m* as 57 (25% shorter than the original subsequence length), 85 (25% longer than original), and 152 (100% longer than original), respectively, and compare the results of the chain discovery algorithm with that in **Figure 6.14**.

The result is shown in **Figure 6.16**. We can see that the discovered chain is basically the same as *m* varies (though the length of the links is different, and the total number of links can vary by +/- 1). The result indicates that *m* does not need to be precisely set; we can discover meaningful chains as long as *m* is in a reasonable range.

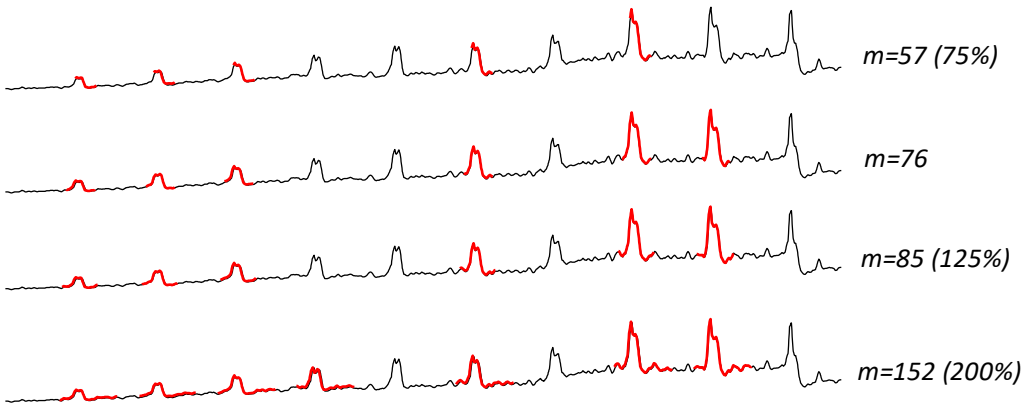

Figure 6.16. The chains discovered from the *Kohl's* data in Figure 6.14 as we vary the subsequence length *m*.

6.4.7 Quantifying the Robustness of Chains

In the previous sections, we showed the broad applicability of time series chains and implicitly showed the robustness of our algorithm/definitions; given that it can find meaningful chains even in real-world "non-perfect" datasets. To further demonstrate this

robustness, we need to provide a measure of the quality of time series chains that does not tautologically reward the properties we have *defined*, and can serve as a "gold standard" to compare the quality of chains before and after we have added some confounding factors.

To test the quality of our chain-discovery algorithm, we should consider two different scenarios: If the data include a long *intrinsic* chain, then a good algorithm should be able to discover (or "recover") a large portion of it. On the other hand, if the time series does *not* have any *intrinsic* evolving trend (for example, the data merely contain k repeated patterns), then we expect the length of the longest chain to be much shorter than k. We will test our algorithm in both scenarios.

Suppose we have a time series with an *intrinsic* chain of length k (that is to say, we know, possibly from external knowledge, that there should be exactly k evolving subsequences of length m in the time series, and we have a set L_{known}: $|L_{known}| = k \times m$ that shows the locations of all the data points within the embedded chain). Further suppose that, without knowing this, an algorithm discovers a time series chain of length $k_{discovered}$, and the locations of the $k_{discovered} \times m$ data points within the discovered chain is stored in the set $L_{discovered}$. Then we can define the *recall* of the chain as $R = \frac{|L_{discovered} \cap L_{known}|}{|L_{known}|}$ and the *precision* as $P = \frac{|L_{discovered} \cap L_{known}|}{|L_{discovered}|}$. For a robust chain, we expect $P \approx 1$. However, note that R does not necessarily need to be as large. Recall the example in **Figure 6.14**; although the discovered chain only covers around 60% of the "bumps", it still reflects the general trend of the data.

Therefore, once L_{known} is given, R and P are excellent measures of quality for the discovered chain. We propose to exploit this idea by building synthetic time series for which we know true chains (both length and locations), and distorting the data to "stress-test" the chain discovery algorithm.

Figure 6.17 shows an example of such a time series, with an embedded chain with k=5. Here the subsequences evolve gradually from a sine wave to a random-walk pattern, and in between the chain elements we inserted snippets of random noise.

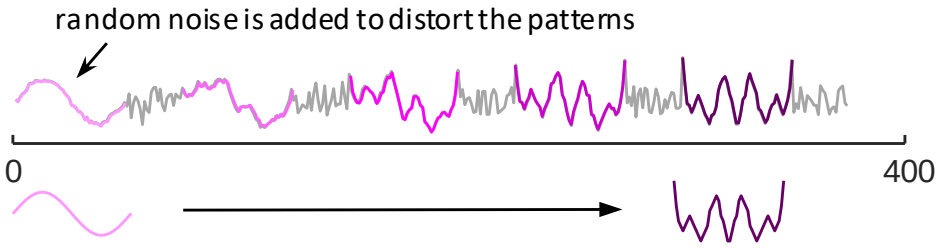

Figure 6.17. Synthetic time series embedded with a chain of five subsequences. The subsequences evolve from a sine-wave to a random-walk pattern.

We used 100 different random-walk patterns like the one in **Figure 6.17** to generate our benchmark time series. Each time series includes 20 subsequences of length 50 (*k=20, m=50*), evolving gradually from a sine wave to a random-walk pattern. **Figure 6.18**.*top* shows how the average results of R and P vary, over the 100 runs as we increase the noise level.

For a large amount of noise (1%~10% of the signal amplitude), we can successfully recover most of the embedded chain elements (more than 14 out of 20), with R > 70% and P > 95%. This demonstrates the robustness of our algorithm: though we missed a small number of embedded patterns, most of them are still recovered.

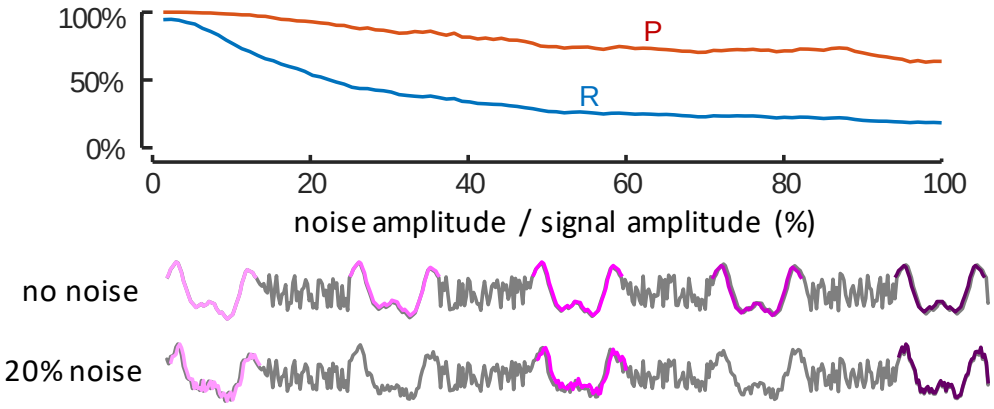

Figure 6.18. ***top*) *Recall* (*R*) and *Precision (P)* both decrease as the noise amplitude increases. *bottom*) A snippet of a "perfect" time series versus the same snippet with 20% noise added.**

However, when the noise amplitude gets over 20%, *R* becomes smaller than 50%. This is because the noise level becomes large enough to hide the evolving characteristics within some part of the data. To see this, in **Figure 6.18**.*bottom* we compared a snippet from a "perfect" benchmark time series without noise to the same snippet with 20% noise. The evolving trend is originally clear in the "perfect" time series; when the noise amplitude increases to 20%, the second and fourth patterns are heavily distorted, so they can no longer be included in the chain. According to **Figure 6.18**.*top*, though with 20% noise only about half of the embedded patterns (10 out of 20, with $R \approx 50\%$) are discovered, the precision *P* is still over 90%. Thus, the discovered chain can still reflect the general trend of the data. Moreover, note that in many cases we could "undo" much of the ill-effect of noise by simply smoothing the data, but that is orthogonal to the purpose of our demonstration.

We have demonstrated that our algorithm is robust in the face of (a reasonable amount of) noise, with a synthetic dataset that contains an *intrinsic* chain. Conversely, we

need to test if $k_{discovered}$ is small compared to k, when there is *no intrinsic* chain within the data, that is to say, are we robust to false positives?

To test this, as shown in **Figure 6.19**, we constructed a synthetic time series with k = 100 repeated random-walk patterns.

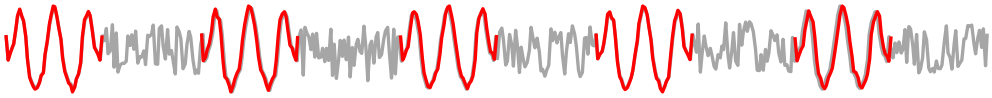

Figure 6.19. A snippet of a synthetic time series with 100 repeated patterns.

As before, we added random noise to all the repeated patterns, so they look slightly different from each other. Unlike the data in **Figure 6.17**, here the k patterns do *not* have an evolving trend. We constructed 100 such synthetic time series, and found that the average length of the discovered time series chain $k_{discovered}$ is 5.04, which is much smaller than k = 100. This result suggests that our algorithm is robust to discovering spurious chains, even in the face of frequent and dense motifs.

6.4.8 Finding Uniform Scaling Time Series Chains

The previous sections showed the efficacy and robustness of our algorithms in finding chains with a fixed subsequence m. Having demonstrated the existence of uniform scaling chains in **Figure 6.6**, here we content ourselves with demonstrating our ability to recover synthetically embedded chains in complex datasets. In **Figure 6.20**, we show a chain we discovered from a synthetic dataset with the UniformScaleChain Algorithm (**Algorithm 10**), which aligns perfectly with the embedded chain. Note that while the change of length appears gradual and subtle to human inspection, it is enough to confound simple motif discovery.

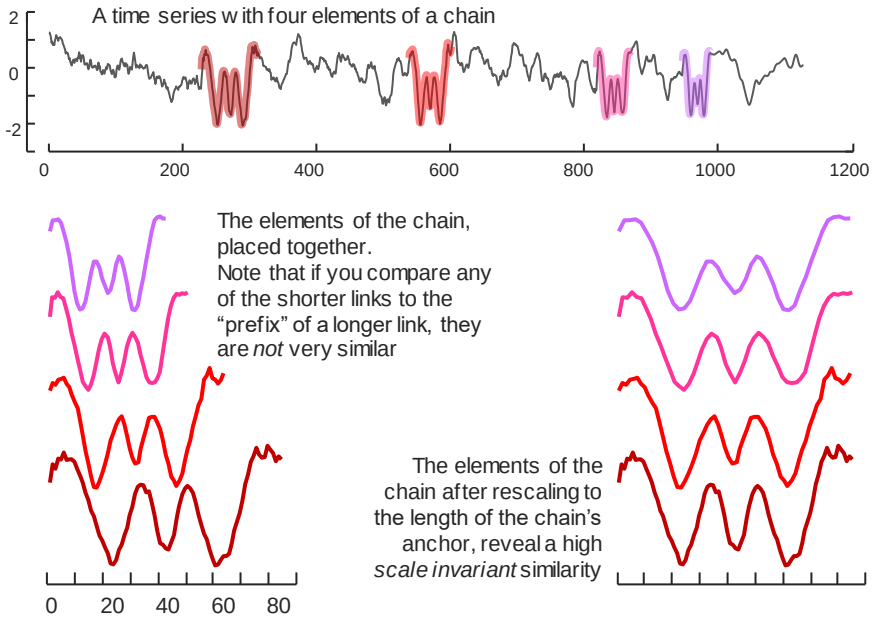

Figure 6.20. ***top*****) A random walk dataset into which we embedded a uniform scaling chain (highlighted). The UniformScaleChain algorithm recovers exactly the same chain.** ***bottom.left*****) the four elements of the chain. Note that we used the any element of the chain to do similarity search on the full time series, we find that it is not particularly similar to any other element under classic Euclidean distance.** ***bottom.right*****) However, rescaling the shorter links of the chain reveals the conserved structure.**

6.5 Conclusions and Future Work

We introduced *time series chains*, a new primitive for time series data mining. We have shown that chains can be efficiently and robustly discovered from noisy and complex datasets, to provide useful insights. We have placed all code and data online, to allow the community to confirm and extend our work. In future work, we plan to consider applications to several problems, especially problems in prognostics, where a chain may indicate a system devolving towards failure.

Chapter 7 Conclusions

In this dissertation we introduced the Matrix Profile, a general and versatile time series data mining tool which has implications for many time series data mining tasks, including motif discovery, discord discovery, shapelet discovery, etc. We presented a suite of algorithms that empower the Matrix Profile with computational efficiency, and introduced useful new primitives that can be extracted from it.

Our core contributions are as follows:

- We introduced a simple, ultra-fast, highly parallelizable and parameter-free batch algorithm to compute the Matrix Profile, and demonstrated that the algorithm incidentally provided the fastest known solution to the discovery of time series motifs, one of the most important time series primitives. When combined with the GPU framework, our algorithm can find the full set of exact motifs on a dataset with one hundred and forty-three million data points in just nine days. This is 143 times larger than the largest dataset ever mined for motifs and joins.
- We further expanded this scalability by introducing a novel fast-converging anytime algorithm to compute the Matrix Profile. For the first time, our algorithm allows the possibility of real-time *interactive* discovery of motifs in datasets of a few million data points long, using off-the-shelf consumer desktops.

- We introduced a novel lower-bound for the Matrix Profile, and proposed what we believe to be the first time series motif discovery algorithm that can find motifs in the presence of missing data without producing false negatives. The lower bound is later adopted and expanded in [39] to allow variable-length motif discovery.
- We introduced time series chains, a new primitive for time series data mining. Time series chains are built on top of the Matrix Profile; they can capture the evolution of systems, helping to predict the future. We demonstrated that time series chains can be efficiently and robustly discovered from noisy and complex datasets, to provide useful insights.

To date our published articles have been cited more than 50 times, and there have been numerous downloads of our software packages. A flurry of follow-up works adopted our algorithms and further applied the Matrix Profile on problems as diverse as guided motif search [16], multi-dimensional motif discovery [98], variable-length motif discovery [39], semantic segmentation [23], weakly labeled data analysis [97], music similarity search [69], etc. The research on the Matrix Profile is still ongoing. Future directions include, but are not limited to:

- Further improving the efficiency of existing algorithms. This includes pushing the computation into the cloud, or developing new lower-bounding techniques to accelerate the computation.
- Expanding the Matrix Profile to support Manhattan distance, dynamic time warping (DTW) distance, or other distance measure.

- Automatic discovery of a suitable subsequence length. This will eliminate the only parameter for our algorithms.
- Applying the Matrix Profile in higher-level time series data mining tasks, such as classification and clustering.
- Inventing new useful primitives for time series data mining.

We envision that the Matrix Profile will continue to play an important role in time series data mining research, and the highly scalable algorithms we have introduced will allow the community to find many uses of, or properties of, the Matrix Profile that did not occur to us.

Bibliography

[1] Aßfalg J, Kriegel HP, Kröger P et al (2009) Probabilistic similarity search for uncertain time series. In: SSDBM, pp. 435–443.

[2] Agrawal R, Faloutsos C, Swami A (1993) Efficient similarity search in sequence databases. Foundations of data organization and algorithmsm, 69–84.

[3] Allstadt K, Malone SD (2014) Swarms of repeating stick-slip icequakes triggered by snow loading at Mount Rainier volcano. Journal of Geophysical Research: Earth Surface, 119(5):1180–1203.

[4] Bailis P, Gan E, Rong K et al (2017) Prioritizing attention in fast data: principles and promise. In: CIDR.

[5] Baisch S, Bokelmann GH (1999) Spectral analysis with incomplete time series: an example from seismology. Computers and Geosciences, 25(7): 739–750

[6] Balasubramanian A, Wang J, Balakrishnan P (2016) Discovering multidimensional motifs in physiological signals for personalized healthcare. IEEE Journal of Selected Topics in Signal Processing 10(5):832–841

[7] Begum N, Keogh E (2014) Rare time series motif discovery from unbounded streams. Proceedings of the *VLDB Endowment* 8(2): 149-160

[8] Bertens R, Vreeken J, Siebes A (2016) Keeping it short and simple: summarising complex event sequences with multivariate patterns. In: ACM SIGKDD, pp 735–744

[9] Bögel T, Gertz M (2015) Time will tell: temporal linking of news stories. In: Proceedings of the 15th ACM/IEEE-CS Joint Conference on Digital Libraries, pp 195–204

[10] Brown AEX, Yemini EI, Grundy LJ et al (2013) A dictionary of behavioral motifs reveals clusters of genes affecting caenorhabditis elegans locomotion. Proceedings of the National Academy of Sciences, 110(2):791–796.

[11] Brown JR, Beroza GC, Shelly DR (2008) An autocorrelation method to detect low frequency earthquakes within tremor. Geophys Res Lett 35, L16305. https://doi.org/10.1029/2008GL034560

[12] Buza K, Schmidt-Thieme L (2009) Motif-based classification of time series with bayesian networks and svms. In: Advances in Data Analysis, Data Handling and Business Intelligence, pp 105–114.

[13] Chandola V, Banerjee A, Kumar V (2007) Anomaly detection: a survey. Technical Report, University of Minnesota.

[14] Chen Y, Keogh E, Hu B et al (2015) The UCR Time Series Classification Archive. URL: www.cs.ucr.edu/~eamonn/time_series_data/

[15] Chiu B, Keogh E, Lonardi S (2003) Probabilistic discovery of time series motifs. In: SIGKDD, pp 493–498.

[16] Dau HA, Keogh E (2017) Matrix Profile V: A Generic Technique to Incorporate Domain Knowledge into Motif Discovery. In: ACM SIGKDD, pp 125–134.

[17] Ding H, Trajcevski G, Scheuermann P et al (2008) Querying and mining of time series data: experimental comparison of representations and distance measures. *VLDB* 1(2): 1542–1552

[18] Faloutsos C, Ranganathan M, Manolopoulos Y (1994) Fast subsequence matching in time-series databases. In: SIGMOD, pp 419–429.

[19] Gama J, Žliobaitė I, Bifet A et al (2014) A survey on concept drift adaptation. *ACM Computing Surveys (CSUR)* 46(4): 1–37

[20] Gangadharan D, Phan LT, Chakraborty S, et al (2011) Video quality driven buffer sizing via frame drops. In: RTCSA 1, pp 319–328.

[21] Gao H, Bi S, Zhang R et al (2012) The design of a throwable two-wheeled reconnaissance robot: In IEEE Robotics and Biomimetics (ROBIO), pp 2150–2155

[22] Geller RJ, Mueller CS (1980) Four similar earthquakes in central California. Geophysical Research Letters, 7(10):821–824.

[23] Gharghabi S, Ding Y, Yeh CCM et al (2017) Matrix profile VIII: domain agnostic online semantic segmentation at superhuman performance levels. In: ICDM, pp 117–126.

[24] Gu Z, He L, Chang C et al (2017) Developing an efficient pattern discovery method for CPU utilizations of computers. International Journal of Parallel Programming, 45(4): 853–878

[25] Gupta S, Reynolds M, Patel S (2010) ElectriSense: single-point sensing using EMI for electrical event detection and classification in the home. In: Proceedings of the UbiComp 12th ACM International Conference on Ubiquitous Computing.

[26] Hao MC, Marwah M, Janetzko H et al (2012) Visual exploration of frequent patterns in multivariate time series. *Information Visualization* 11(1): 71–83

[27] Harris M (2007) Optimizing parallel reduction in CUDA. NVIDIA Developer Technology 2.4.

[28] Havskov J, Alguacil G (2004) Instrumentation in earthquake seismology. Vol. 358. Dordrecht: Springer

[29] Heldt T, Oefinger MB, Hoshiyama M et al (2003) Circulatory response to passive and active changes in posture. In: *Computers in Cardiology*, IEEE, pp 263–266

[30] Hoang T, Choi D, Nguyen T (2015) On the instability of sensor orientation in gait verification on mobile phone. In: 12th IEEE International Joint Conference on e-Business and Telecommunications (ICETE) 4: 148–159

[31] Honaker J, King G (2010) What to do about missing values in time-series cross-section data. American Journal of Political Science, 54(2): 561-581

[32] Huang T, Zhu Y, Mao Y et al (2016) Parallel discord discovery. In: PAKDD 2016, pp 233–244.

[33] Igarashi T, Matsuzawa T, Hasegawa A (2003) Repeating earthquakes and interplate aseismic slip in the northeastern Japan subduction zone. J Geophys Res 108, 2249. https://doi.org/10.1029/2002JB001920

[34] Iverson RM, Dzurisin D, Gardner CA et al (2006) Dynamics of seismogenic volcanic extrusion at Mount St. Helens in 2004-05. Nature 444(7118): 439–443

[35] Kolb I, Franzesi GT, Wang M et al (2018) Evidence for long-timescale patterns of synaptic inputs in CA1 of awake behaving mice. Journal of Neuroscience, 38(7): 1821–1834.

[36] Krumme C, Llorente A, Cebrian M et al (2013) The predictability of consumer visitation patterns. *Scientific Reports* 3:1645

[37] Li Y, U LH, Yiu ML et al (2015) Quick-motif: An efficient and scalable framework for exact motif discovery. In: ICDE, IEEE, pp 579–590

[38] Li Z, Han J, Ding B et al (2012) Mining periodic behaviors of object movements for animal and biological sustainability studies. *Data Mining and Knowledge Discovery* 24(2): 355–386

[39] Linardi M, Zhu Y, Palpanas T et al (2018) Matrix profile X: VALMOD-scalable discovery of variable-length motifs in data series. In: SIGMOD, pp 1053–1066.

[40] Lovallo WR, Wilson MF, Vincent AS et al (2004) Blood pressure response to caffeine shows incomplete tolerance after short-term regular consumption. *Hypertension* 43(4): 760–765

[41] Luo W, Tan H, Mao H et al (2012) Efficient similarity joins on massive high-dimensional datasets using mapreduce. In: MDM, IEEE, pp 1–10

[42] Makonin SV (2013) AMPds: a public dataset for load disaggregation and eco-feedback research. 2013 IEEE Electrical Power & Energy Conference (EPEC)

[43] Matsubara Y, Sakurai Y, Faloutsos C (2015) The web as a jungle: non-linear dynamical systems for co-evolving online activities. In: Proc' of the 24th WWW, pp 721–731

[44] McGovern A, Rosendahl DH, Brown RA et al (2011) Identifying predictive multi-dimensional time series motifs: an application to severe weather prediction. Data Mining and Knowledge Discovery, 22(1): 232–258

[45] McLoone J (2012) The longest word ladder puzzle ever. blog.wolfram.com/2012/01/11/the-longest-word-ladder-puzzle-ever. Retrieved 6 Sept 2016

[46] Meng X, Yu X, Peng Z et al (2012) Detecting earthquakes around salton sea following the 2010 mw7.2 El Mayor-Cucapah earthquake using GPU parallel computing. Procedia Computer Science 9:937–946.

[47] Minnen D, Isbell CL, Essa I et al (2007) Discovering Multivariate Motifs using Subsequence Density Estimation and Greedy Mixture Learning. In: AAAI, pp 615–620

[48] Moya A (2009) Tilt testing and neurally mediated syncope: too many protocols for one condition or specific protocols for different situations?. *Eur Heart J* 30(18): 2174-2176

[49] Mueen A, Keogh E, Zhu Q et al (2009) Exact discovery of time series motifs. In: SDM, pp 473–484.

[50] Mueen A, Keogh E, Young N (2011). Logical-shapelets: an expressive primitive for time series classification. In: SIGKDD, pp. 1154–1162.

[51] Mueen A, Zhu Y, Yeh M et al (2015) The fastest similarity search algorithm for time series subsequences under Euclidean distance. http://www.cs.unm.edu/~mueen/FastestSimilaritySearch.html. Retrieved 2 Feb 2017

[52] Murray D, Liao J, Stankovic L (2015) A data management platform for personalised real-time energy feedback. In: *EEDAL*

[53] Neocleous A, Petkov N, Schizas C (2013) Finding repeating Stanzas in monophonic folk songs of Cyprus. In: 6th Cyprus Workshop on Signal Processing and Informatics, pp 26

[54] NVIDIA CUDA C Programming Guide (2016) Version 7.5. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[55] NVIDIA CUFFT Library User's Guide (2016) Version 7.5. http://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf

[56] Patel P, Keogh E, Lin J et al (2002) Mining motifs in massive time series databases. In: ICDM, pp 370–377

[57] Patnaik D, Manish M, Sharma RK, Ramakrishnan N (2009) Sustainable operation and management of data center chillers using temporal data mining. In: SIGKDD, pp 1305–1314

[58] Ponganis PJ, St Leger J, Scadeng M (2015) Penguin lungs and air sacs: implications for baroprotection, oxygen stores and buoyancy. *Journal of Experimental Biology*: 720–730

[59] Project Website for Matrix Profile (2017) http://www.cs.ucr.edu/~eamonn/MatrixProfile.html

[60] Quick Motif (2015) http://degroup.cis.umac.mo/quickmotifs/

[61] Rakthanmanon T, Campana B, Mueen A et al (2013) Addressing big data time series: mining trillions of time series subsequences under dynamic time warping. TKDD 7(3):10

[62] REFIT: Smart Homes and Energy Demand Reduction. URL: www.refitsmarthomes.org/index.php/data. Accessed 01/21/2018

[63] Rong K, Bailis P (2017) ASAP: prioritizing attention via time series smoothing. *VLDB Endowment* 10(11):1358–1369

[64] Sarangi SR, Murthy K (2010) DUST: a generalized notion of similarity between uncertain time series. In: SIGKDD, pp 383–392

[65] Shelly DR, Beroza GC, Ide S (2017) Non-volcanic tremor and low-frequency earthquake swarms. Nature, 446(7133):305–307

[66] Shelly DR, Beroza GC, Ide S et al (2006) Low-frequency earthquakes in Shikoku, Japan, and their relationship to episodic tremor and slip. Nature 442(7099):188–191

[67] Shelly DR, Ellsworth WL, Ryberg T et al (2009) Precise location of San Andreas Fault tremors near Cholame, California using seismometer clusters: Slip on the deep extension of the fault? Geophys Res Lett 36, L01303. https://doi.org/10.1029/2008GL036367

[68] Shokoohi-Yekta M, Chen Y, Campana B et al (2015) Discovery of meaningful rules in time series. In: ACM SIGKDD, pp 1085-1094

[69] Silva D, Yeh CCM, Batista G et al (2016) SiMPle: Assessing Music Similarity Using Subsequences. In: ISMIR.

[70] Silver N (2012) The signal and the noise: the art and science of prediction. Penguin, London

[71] Simeone A, Wilson RP (2003) In-depth studies of Magellanic penguin (Spheniscus magellanicus) foraging: can we estimate prey consumption by perturbations in the dive profile? Marine Biology 143(4):825–831

[72] Smith J (2010) The accidentally-on-purpose history of cyber monday. www.esquire.com/news-politics/news/a23870/cyber-monday-online-shopping-4021548/. Retrieved 5 Feb 2017

[73] Song WZ, Huang R, Xu M et al (2009) Air-dropped sensor network for real-time high-fidelity volcano monitoring. In: Proceedings of the 7th international conference on Mobile systems, applications, and services, ACM, pp 305–318

[74] Sparks RSJ (2003) Forecasting volcanic eruptions. Earth and Planetary Science Letters 210(1):1–15

[75] Supporting Webpage for SCRIMP++ (2018): https://sites.google.com/site/scrimpplusplus/

[76] Supporting Webpage for Time Series Chains (2017) https://sites.google.com/site/timeserieschain/

[77] Supporting Webpage for Missing Data Motifs (2018) https://sites.google.com/site/motifmissingdata/.

[78] Syed Z, Stultz C, Kellis M et al (2010) Motif discovery in physiological datasets: a methodology for inferring predictive elements. *TKDD* 4(1): 2

[79] Szigeti B, Deogade A, Webb B (2015) Searching for motifs in the behaviour of larval Drosophila melanogaster and Caenorhabditis elegans reveals continuity between behavioural states. Journal of the Royal Society Interface 12(113): 20150899

[80] Tanaka Y, Iwamoto K, Uehara K (2005) Discovery of timeseries motif from multi-dimensional data based on MDL principle. Machine Learning, 58(2):269–300

[81] Torkamani S, Lohweg V (2017) Survey on time series motif discovery. Wiley Interdisciplinary Reviews: Data Min. Knowl. Discov. 7.2

[82] Truong CD and Anh DT (2015) An efficient method for motif and anomaly detection in time series based on clustering. *International Journal of Business Intelligence and Data Mining*, 10(4):356–377

[83] Tucker A, Liu X (2004) A bayesian network approach to explaining time series with changing structure. *Intelligent Data Analysis* 8(5):469–480

[84] Turkay C, Kaya E, Balcisoy S, Hauser H (2017) Designing progressive and interactive analytics processes for high-dimensional data analysis. IEEE Trans. Vis. Comput. Graph. 23(1): 131–140

[85] Vahdatpour A, Amini N, Sarrafzadeh M (2009) Toward unsupervised activity discovery using multi-dimensional motif detection in time series. In: IJCAI, Vol. 9, pp 1261–1266.

[86] Veeraraghavan A, Chellappa R, Roy-Chowdhury A (2006) The function space of an activity. Computer Vision and Pattern Recognition 1: 959-968

[87] Wang J, Cardell-Oliver R, Liu W (2016) An incremental algorithm for discovering routine behaviours from smart meter data. Knowledge-Based Systems 113: 61–74

[88] Wang L, Chng ES, Li H (2010) A tree-construction search approach for multivariate time series motifs discovery. Pattern Recognition Letters 31(9):869–875

[89] Wang X, Mueen A, Ding H et al (2013) Experimental comparison of representation methods and distance measures for time series data. Data Min. Knowl. Discov. 26(2): 275–309

[90] Weber M, Denk M (2010) Imputation of cross-country time series: techniques and evaluation. In: European Conference on Quality in Official Statistics

[91] Wei L (2006) SAX code for the N/n not equal an integer case. URL: www.cs.ucr.edu/~eamonn/SAX.htm. Accessed 01/21/2018

[92] Willett D, George J, Willett N et al (2016) Machine learning for characterization of insect vector feeding. PLoS computational biology, 12(11): e1005158

[93] Williams CL, Sato K, Shiomi K et al (2011) Muscle energy stores and stroke rates of emperor penguins: implications for muscle metabolism and dive performance. *Physiological and Biochemical Zoology* 85(2):120–133

[94] Yan R, Wan X, Otterbacher J et al (2011) Evolutionary timeline summarization: a balanced optimization framework via iterative substitution. In: Proc' of the 34th ACM SIGIR, pp 745–754

[95] Ye L, Keogh E (2009). Time series shapelets: a new primitive for data mining. In: ACM SIGKDD, pp 947–956.

[96] Yeh CCM, Zhu Y, Ulanova L et al (2016) Matrix profile I: all pairs similarity joins for rime series: a unifying view that includes motifs, discords and shapelets. In: IEEE ICDM, pp 1317–1322

[97] Yeh CCM, Kavantzas N, Keogh E (2017) Matrix profile IV: using weakly labeled time series to predict outcomes. *VLDB Endowment* 10(12): 1802–1812.

[98] Yeh CCM, Kavantzas N, Keogh E (2017) Matrix profile VI: meaningful multidimensional motif discovery. In: ICDM, pp 565–574.

[99] Yeh MY, Wu KL, Yu PS, and Chen MS (2009) PROUD: a probabilistic approach to processing similarity queries over uncertain data streams. In: EDBT, pp 684–695

[100] Yi X, Zheng Y, Zhang J, Li T (2016) ST-MVL: Filling missing values in geo-sensory time series data. In: IJCAI, pp 2704–2710

[101] Yoon CE, O'Reilly O, Bergen KJ et al (2015) Earthquake detection through computationally efficient similarity search. Science Advances 1(11):e1501057

[102] Zhu X, Oates T (2012) Finding story chains in newswire articles. In: IEEE IRI, pp 93–100