Mining Time Series for Maximal Coverage with Matrix Profiles and Constraint Optimization

by

Neeraj Nagar

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2025

© Neeraj Nagar 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Time Series Data (TSD) is essential in many areas of modern data analysis because they reflect how different processes change over time. Understanding these data can still be challenging. One of the main challenges is that the underlying states of a system are often hidden, making it harder to interpret patterns and draw reliable conclusions. This thesis addresses the critical task of mining recurrent patterns from systems whose internal states are neither directly observable nor controllable. It introduces a novel unsupervised approach explicitly designed to maximize coverage in TSD.

The research proposes a structured approach comprising three key steps: firstly, generating candidate patterns and their occurrences using an advanced Matrix Profile (MP) algorithm known for its efficiency and accuracy in detecting subtle recurrent patterns; secondly, translating these candidate patterns into a constraint-based model incorporating group constraints to enforce selection of all instances of the same pattern and exclusion constraints to prevent overlapping occurrences; and thirdly, selecting an optimal subset of core patterns using either a constraint solver to ensure optimal selection on shorter Time Series (TS), or scalable greedy heuristic methods that offer practical efficiency for larger or more complex datasets, thereby effectively balancing optimality with computational feasibility.

The effectiveness of the proposed approach is demonstrated through rigorous evaluations on real-world power consumption TSDs representing computer activities, alongside controlled synthetic datasets, using metrics such as coverage efficiency, computational time, and pattern compactness—striving to maximize data representation with minimal redundancy. Our experimental results show that the proposed method improves how efficiently data is covered, striking a practical balance between capturing key patterns and avoiding unnecessary repetition. This work contributes to the advancement in unsupervised pattern mining and has useful applications in areas such as forecasting, system health monitoring, anomaly detection, and policy verification.

Acknowledgements

I want to thank Professor Sebastian Fischmeister, my supervisor, sincerely for his constant encouragement and guidance throughout my academic career. His considerate feedback, motivation, and mentorship have significantly contributed to shaping my research and personal growth. I am greatly indebted to him for the privilege of learning from him.

I am immensely grateful to Arthur Grisel-Davy, a fellow PhD researcher in our group. His insightful guidance, patience, and open-mindedness in sharing ideas have been an invaluable support. Engaging discussions with him have deepened my understanding of our subject and enhanced the enjoyment and fulfillment of the research process. I also sincerely appreciate his help in this research, including his contribution to drawing several figures used in this work.

Their guidance has not only assisted in bringing this project to completion but has also challenged me to keep learning and pushing myself to grow. I am more than grateful for their trust in me and their support throughout this process.

Dedication

This thesis is dedicated to my mother, father, my entire family, a very special person in my life, and to everyone who supported me throughout my journey.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures x

List of Tables xiv

List of Abbreviations xv

List of Symbols xvi

List of Algorithms xvii

1 Introduction 1

1.1 Thesis Contribution 2

1.2 Thesis Outline 2

2 Literature Review 4

3 Problem Statement **8**

4 Methodology **10**

4.1 Dataset Description 10

4.2 Scalable Preprocessing 11

4.2.1 Scalable Chunking Strategy 12

4.2.2 Flat Region Detection and Masking 14

4.3 Motifs Detection Using Matrix Profile (MP) 14

4.4 Clustering Patterns with DBSCAN 19

4.5 Pattern Selection Approaches 21

4.5.1 MiniZinc-Based Constraints Formulation and Solving 21

4.5.2 Greedy Heuristic Approaches for Large-Scale Selection 23

4.6 Summary of Methodological Pipeline 24

5 Implementation **26**

5.1 Tools and Technologies 26

5.1.1 Software Libraries and Frameworks 27

5.1.2 Computational Resources 27

5.2 Implementation of Chunking and Parallelization 28

5.3 Flat Region Masking Prior to Matrix Profile (MP) Computation 29

5.4 Matrix Profiles (MPs) Computations 32

5.5 Motif Grouping Using DBSCAN 34

5.6 Implementation of Pattern Selection Methods 35

5.6.1 Constraint Solver 36

5.6.2 Greedy Heuristic Approaches 40

5.7 Summary 41

6 Experiments and Evaluation **44**
6.1 Datasets Used for Evaluation . . . 45
6.2 Impact of the k Parameter in Matrix Profile (MP) Computation . . . 48
6.2.1 Computational Challenges with Increasing k . . . 48
6.2.2 Optimization Using Threshold-Based Motif Discovery . . . 49
6.3 Effect of Flat Regions on Computational Performance . . . 53
6.3.1 Computational Overhead Introduced by Flat Regions . . . 53
6.3.2 Optimization via Preprocessing for Flat Region Mitigation . . . 54
6.4 Effect of Euclidean Distance Threshold on Computational Performance . . . 57
6.5 Effect of Window Size on Computational Performance . . . 59
6.6 Impact of Time Series (TS) Length on Computational Performance . . . 60
6.7 Summary of Experimental Findings . . . 62

7 Results **64**
7.1 Quantitative Evaluation of Greedy Algorithms . . . 64
7.1.1 Performance on Time Series TS-1 (Real-World Dataset) . . . 65
7.1.2 Performance on Time Series TS-2 (Real-World Dataset) . . . 71
7.2 Summary . . . 75

8 Discussion **76**
8.1 Impact of Hyper-parameter Choices . . . 76
8.1.1 Minimum Pattern Window Size . . . 76
8.1.2 Maximum Pattern Window Size . . . 77
8.1.3 Euclidean Distance Threshold . . . 77
8.1.4 Variance Threshold for Flat Region Masking . . . 77
8.1.5 Base Length Selection for Chunking . . . 78
8.1.6 Selection of k in STUMPY Motif Discovery . . . 78
8.2 Practical Use Cases . . . 79

8.2.1 System State Identification 79
8.2.2 Anomaly Detection and Diagnosis 79
8.3 Hardware Environment and Solver Selection 80

9 Conclusion **82**

References **84**

APPENDICES **87**

A Pseudocode for Subroutines **88**

B Time Series Plots **91**
B.1 Patterns Extracted from time series TS-3 91
B.2 Patterns Extracted from time series TS-4 93

C Quantitative Results **96**
C.0.1 Results on time series TS-2 96
C.0.2 Results on time series TS-3 97
C.0.3 Results on time series TS-4 98

Glossary **100**

List of Figures

4.1 Segmentation of a time series (TS) into overlapping chunks. Each chunk C_i overlaps with the next by $W_{\max}$ data points (shaded regions), ensuring motifs spanning chunk boundaries are fully captured. This strategy allows parallel motif discovery and prevents missed patterns at chunk edges. . . . 13

4.2 Structure of matrix profile (MP) outputs for different window sizes. The MP is calculated for each window size W from $W_{\min}$ to $W_{\max}$. For each W, the MP consists of $2k$ columns: the first k columns contain Euclidean Distance (EuDist) between reference subsequences and their top k nearest neighbors, and the next k columns store the corresponding indices in the TS. This allows the capture of patterns with different durations. 16

4.3 Illustration of matrix profile (MP) computation for multiple window sizes. Each MP block summarizes, for a specific window size, the top k EuDist and indices of the most similar subsequences in the TS. Colored arrows demonstrate how reference subsequences at $W_{\min}$ and $W_{\max}$ map to corresponding entries in the MP matrices. The final two columns of each matrix are omitted from further analysis. Dist(R_0,R_{30}) represent the EuDist between reference index i=0 and its similar subsequence j=30. 17

4.4 Pattern mining pipeline overview. Stage 1 detects motifs using matrix profile (MP) computation over multiple window sizes with a modified Stumpy and inclusion EuDist parameter, followed by DBSCAN grouping. Stage 2 applies group and exclusion constraints, and Stage 3 selects non-overlapping patterns maximizing coverage via exact optimization (MiniZinc) or greedy heuristics. **Note: Preprocessing steps chunking and flat region masking are not shown here for clarity; the full pipeline—including all preprocessing—is illustrated in Figure 5.2 and discussed in Section 4.2.** 24

5.1 Converting candidate patterns into a constraint model. Each colored box or line represents either a group constraint (same pattern across time) or an exclusion constraint (overlapping occurrences). This structured representation is then encoded into a constraint model interpretable by the MiniZinc framework. 37

5.2 Overview of the full pattern mining pipeline—input time series (TS) is chunked and masked, then processed using matrix profile (MP) computations to extract motifs. Masked regions (in red) are excluded. Motifs are globally clustered with Density-Based Spatial Clustering of Applications with Noise (DBSCAN), then filtered and selected using either constraint-based (MiniZinc) or heuristic (greedy) methods. 42

6.1 Normalized synthetic time series TS-0 containing periodic patterns with controlled noise, used for baseline validation. Sampling rate is 1 Hz (1 sample/second). 46

6.2 Normalized real-world time series TS-1 representing power consumption data collected over a 24-hour period. Sampling rate is 1 Hz. 46

6.3 Normalized real-world time series TS-2 generated by temporally stretching the first 6 hours of Figure 6.2. Used to evaluate the robustness of the algorithm. Sampling rate is 1 Hz. 47

6.4 Normalized real-world time series TS-3 characterized by frequent fluctuations and high-amplitude spikes. Used to test resilience under noisy and bursty conditions. Sampling rate: 1 Hz. 47

6.5 Normalized real-world time series TS-4 exhibiting bursty behavior and variability similar to Figure 6.4. Sampling rate: 1 Hz. 48

6.6 Comparison of Graphics Processing Unit (GPU) execution time (Execution Time (ET)) before and after threshold-based optimization for Figure 6.1, with fixed $W = 100$ and k ranging from 1 to 1000. 50

6.7 Comparison of GPU execution time (ET) before and after threshold-based optimization for Figure 6.2, with fixed $W = 100$ and k ranging from 1 to 1000. 51

6.8 Comparison of GPU execution time (ET) before and after threshold-based optimization for Figure 6.3, with fixed $W = 100$ and k ranging from 1 to 1000. 51

6.9 Comparison of GPU execution time (ET) before and after threshold-based optimization for Figure 6.4, with fixed $W = 100$ and k ranging from 1 to 1000. 52

6.10 Comparison of GPU execution time (ET) before and after threshold-based optimization for Figure 6.5, with fixed $W = 100$ and k ranging from 1 to 1000. 52

6.11 GPU execution time (ET) as a function of k (fixed $W = 100$) for Figure 6.1 under: (1) No optimization, (2) Threshold only, (3) Flat-region masking only, (4) Both optimizations. Combined gives best performance. 55

6.12 GPU execution time (ET) as a function of k (fixed $W = 100$) for Figure 6.2 under: (1) No optimization, (2) Threshold only, (3) Flat-region masking only, (4) Both optimizations. Combined gives best performance. 55

6.13 GPU execution time (ET) as a function of k (fixed $W = 100$) for Figure 6.3 under: (1) No optimization, (2) Threshold only, (3) Flat-region masking only, (4) Both optimizations. Combined gives best performance. 56

6.14 GPU execution time (ET) as a function of k (fixed $W = 100$) for Figure 6.4 under: (1) No optimization, (2) Threshold only, (3) Flat-region masking only, (4) Both optimizations. Combined gives best performance. 56

6.15 GPU execution time (ET) as a function of k (fixed $W = 100$) for Figure 6.5 under: (1) No optimization, (2) Threshold only, (3) Flat-region masking only, (4) Both optimizations. Combined gives best performance. 57

6.16 GPU execution time (ET) as a function of the Euclidean distance (EuDist) threshold (θ_{ed}) for five TSs, using the optimized Stumpy implementation with flat-region masking and threshold-based pruning. Window size fixed at $W = 100$. Lower θ_{ed} values reduce runtime by pruning the search space, while higher values converge toward the original stumpy implementation. . 58

6.17 GPU execution time (ET) vs. window size (W) for five TSs, using logarithmically spaced W values from 10 to $\frac{n}{2}$ and adaptive k_{opt}. Execution time decreases with W due to reduced subsequence comparisons, converging at large W. All experiments were repeated for five iterations, and the reported ET values represent the mean across iterations. 60

6.18 GPU execution time (ET) as a function of time series (TS) length across five different TS. Each TS is sampled at a rate of 1 data point per second and extended incrementally from 1 hour (3,600 points) to 24 hours (86,400 points). Parameters are fixed at window size $W = 100$ and number of nearest neighbors $k = 200$. The results show a near-linear increase in ET with respect to TS length. 61

7.1 IntervalTreeGreedy result on TS-1 with Number of Patterns (NoP) = 1 (Coverage = 39.19%) 67

7.2 IntervalTreeGreedy result on TS-1 with NoP = 2 (Coverage = 52.92%) . . 67

7.3 IntervalTreeGreedy result on TS-1 with NoP = 3 (Coverage = 57.52%) . . 67

7.4 IntervalTreeGreedy result on TS-1 with NoP = 4 (Coverage = 60.58%) . . 68

7.5 IntervalTreeGreedy result on TS-1 with NoP = 20 (Coverage = 70.11%) . . 68

7.6 IntervalTreeGreedy result on TS-1 with NoP = 100 (Coverage = 70.52%) . 68

7.7 Three pattern groups selected on TS-1 using IntervalTreeGreedy with NoP = 3. Each plot shows a representative group with aligned patterns across different window sizes (Coverage = 57.52%). 69

7.8 Four pattern groups selected on TS-1 using IntervalTreeGreedy with NoP = 4. Each plot shows a representative group with aligned patterns across different window sizes (Coverage = 60.58%). 70

7.9 IntervalTreeGreedy result on TS-2 with NoP = 1 (Coverage = 24.33%) . . 73

7.10 IntervalTreeGreedy result on TS-2 with NoP = 2 (Coverage = 31.99%) . . 73

7.11 IntervalTreeGreedy result on TS-2 with NoP = 3 (Coverage = 35.48%) . . 73

7.12 Three pattern groups selected on TS-2 using IntervalTreeGreedy with NoP = 3. Each plot shows a representative group with aligned patterns across different window sizes (Coverage = 35.48%). 74

List of Tables

7.1 Coverage (%) of greedy algorithms on time series TS-1 under varying NoP 65

7.2 Pattern set size (number of selected patterns) on time series TS-1 under varying NoP 65

7.3 Execution time ET (in seconds) of greedy algorithms on time series TS-1 under varying NoP 66

7.4 Coverage (%) of greedy algorithms on time series TS-2 under varying NoP 71

7.5 Pattern set size (number of selected patterns) on time series TS-2 under varying NoP 71

7.6 Execution time ET (in seconds) of greedy algorithms on time series TS-2 under varying NoP 72

8.1 Computational time comparison (in seconds) for Gecode and OR-Tools solvers on five representative time series (TS) datasets. Lower values indicate faster performance. 81

C.1 Coverage (%) of greedy algorithms on time series TS-2 under varying NoP 96

C.2 Execution time (in seconds) of greedy algorithms on time series TS-2 under varying NoP 97

C.3 Coverage (%) of greedy algorithms on time series TS-3 under varying NoP 97

C.4 Execution time (in seconds) of greedy algorithms on time series TS-3 under varying NoP 98

C.5 Coverage (%) of greedy algorithms on time series TS-4 under varying NoP 98

C.6 Execution time (s) of greedy algorithms on time series TS-4 under varying NoP 99

List of Abbreviations

CPU Central Processing Unit 13, 28, 30, 53, 80

CSP Constraint Satisfaction Problem 6, 80

DBSCAN Density-Based Spatial Clustering of Applications with Noise xi, xvii, 2, 19, 20, 27, 32, 34, 35, 41, 42, 62, 82, 83

ET Execution Time xi–xiv, 18, 44, 45, 49–64, 66, 72, 75, 77

EuDist Euclidean Distance x, xii, 5, 11, 15–20, 24, 27, 32–34, 44, 48–50, 57, 58, 62, 76, 77

GPU Graphics Processing Unit xi–xiii, 13, 18, 27–30, 41, 44, 49–63, 77, 79, 82, 83

MP Matrix Profile iii, vii, viii, x, xi, xvii, 2, 5, 6, 12–21, 24, 26–30, 32–34, 41, 42, 44, 48, 53, 54, 60–62, 64

NoP Number of Patterns xiii, xiv, 8–10, 21–23, 25, 38–40, 64–75, 82, 89, 90, 96–99

SA Simulated Annealing 24, 41, 64, 66, 72, 75

TS Time Series iii, viii–xiv, xvii, 1, 2, 4–8, 10–21, 23–36, 39–50, 53, 54, 58–75, 77, 78, 81–83, 89–91, 93, 96–99

TSA Time Series Analysis 4, 5, 19, 48, 53, 83

TSD Time Series Data iii, 1, 2, 4, 6, 10, 25, 26, 28, 32, 34, 44, 45, 53, 63, 64, 79, 80, 82

List of Symbols

k Number of top nearest neighbors considered for each subsequence x–xii, 14–19, 44, 48–54, 60, 62, 78, 79

L base length for a chunk 12, 28, 78

n Length of the time series 8, 12, 14

List of Algorithms

1 Chunking a Time Series (TS) into Overlapping Sub-Series 29
2 Flat Region Detection and Masking Based on Variance 31
3 Motif Detection for a chunk using Matrix Profile (MP) with Flat Region Masking and Grouping 33
4 Global Motif Grouping using DBSCAN 35
5 Pseudo-code for conversion into MiniZinc's constraint modelling language. 38
6 Naive Greedy Pattern Selection 89
7 Greedy Pattern Selection with Interval Tree 90

Chapter 1

Introduction

TSD have become increasingly central across numerous domains due to their inherent ability to capture temporal dynamics and evolving phenomena. Such data commonly serve as proxies for underlying system states or activities, including biological signals such as heartbeats in electrocardiograms (ECG) and financial indicators like stock prices. Gaining insights into these states is crucial for tasks such as prediction, anomaly detection, and system integrity assessment [7].

In situations where system states are directly observable and controllable, identifying key patterns and activities within TS can be relatively straightforward. For instance, recording ECG signals allows precise identification of cardiac activity patterns. However, in many real-world scenarios, systems generating TSD are neither controllable nor fully observable. Examples include weather systems, where myriad interdependent physical factors drive changes, or intricate embedded and production systems that preclude direct manipulation or clear observation.

Mining meaningful patterns from such unobservable and uncontrollable systems is challenging, primarily due to the absence of labeled data or predefined system states. Traditional pattern mining methods, which predominantly focus on identifying the most frequent or statistically significant patterns, often neglect comprehensive coverage, resulting in incomplete representations of the system's behavior [12, 21, 26]. Consequently, there is a critical need for novel methodologies capable of extracting representative, recurring patterns that maximize coverage and provide a comprehensive view of system dynamics.

This thesis addresses this gap by proposing a novel, coverage-driven approach to mining recurring patterns from univariate TSD without reliance on labeled instances. The proposed approach focuses on maximizing coverage and minimizing redundancy, providing

flexibility and adaptability to different application-specific requirements through multiple optimization scenarios. The effectiveness and practical utility of this approach are validated through extensive experiments on both real-world datasets (e.g., power consumption traces) and synthetic datasets. The results demonstrate the method's ability to extract meaningful and interpretable patterns, achieving high coverage while minimizing overlap. This research not only introduces a novel methodological framework but also has significant implications for applications in anomaly detection, system monitoring, and efficient summarization of complex TSD (see Section 8.2).

In this thesis, the terms motif and pattern are used interchangeably depending on the context. In general, motif refers to frequently occurring subsequences, as commonly used in TS literature. However, pattern is used more broadly to include any structurally meaningful subsequence exhibiting temporal variation that contributes to the coverage of the TS. This distinction allows flexibility in describing both recurring and informative segments, regardless of their frequency.

1.1 Thesis Contribution

The primary contributions of this thesis include the development of a novel, unsupervised pattern mining method for univariate TS aimed at maximizing coverage. Furthermore, the thesis introduces a two-fold optimization framework providing flexibility for varying application needs by balancing pattern coverage and simplicity. Extensive validation using both real-world and synthetic datasets demonstrates the practical efficacy and interpretability of the proposed approach. Additionally, the thesis presents a novel integration of MP-based techniques [14, 26] with constraint-based optimization [19, 2] for rigorous pattern selection.

1.2 Thesis Outline

This thesis is organized as follows: Chapter 2 presents a review of relevant literature and outlines the current limitations in pattern mining approaches. Chapter 3 formally introduces the problem statement and discusses the motivations and objectives behind the proposed work. Chapter 4 details the methodology, including MP-based detection, pattern filtering, clustering with DBSCAN, and constraint optimization. Chapter 5 describes the implementation aspects, including tools used, algorithmic integration, and system-level decisions. Chapter 6 presents the experiments and evaluation, including visualizations before and after optimization. Chapter 7 presents the results and detailed evaluation of the

proposed greedy selection algorithms, including quantitative benchmarking across multiple datasets and qualitative visualization of the extracted patterns to assess coverage, efficiency, and motif interpretability. Chapter 8 provides an in-depth discussion of hyper-parameters, use cases, and solver configurations. Chapter 9 concludes the thesis, summarizing the contributions.

Chapter 2

Literature Review

The identification and extraction of meaningful patterns from TSD have long been a pivotal area of research, driven by its broad applicability in diverse fields including finance, healthcare, cybersecurity, and environmental monitoring. Within these domains, TS typically serve as indicators or proxies that capture underlying system behaviors, enabling practitioners to extract valuable insights from continuous streams of numerical data. Practical applications range from forecasting market movements and diagnosing physiological conditions to detecting environmental anomalies and system faults. As datasets continue to expand in scale and complexity, there is an increasing demand for efficient, scalable, and interpretable methods for discovering and analyzing temporal patterns.

Recently, the domain of cybersecurity and software runtime analysis has increasingly leveraged Time Series Analysis (TSA) to identify and interpret subtle yet informative patterns embedded in sequential data. Side-channel analysis, for example, utilizes indirect observations such as power consumption traces to detect security breaches or anomalies [9, 5, 3]. Similarly, analyzing log data from software applications has become vital for identifying recurrent system faults or security threats. Notable approaches in log data analysis include frequent pattern mining [12], clustering-based methods like LogCluster [22], and sequence matching techniques exemplified by LogMine [10]. These methods adopt differing strategies—ranging from statistical frequency analysis and heuristic clustering to n-gram-based sequence matching—to detect recurring log structures or anomalies. While they demonstrate strong performance in recognizing commonly occurring patterns, they inherently prioritize frequency and clustering compactness over interpretability and completeness. In contrast, the approach proposed in this thesis emphasizes full coverage of the dataset, ensuring that all significant segments are represented by meaningful patterns. Here, coverage refers to the fraction of the TS spanned by occurrences of the selected pat-

terns. This distinction is especially critical in domains where unexplained intervals may indicate latent anomalies or overlooked system states.

The broader literature on TS pattern mining presents a rich diversity of methodologies, shaped by assumptions about data structure, noise, motif duration, and the nature of pattern recurrence. In a comprehensive survey, Fu [7] categorizes these techniques, noting that the majority fall into two dominant paradigms: clustering-based discovery and subsequence similarity matching.

A foundational contribution to motif discovery is the K-motif algorithm introduced by Lonardi and Keogh [17]. This method leverages Symbolic Aggregate approXimation (SAX)—a symbolic dimensionality reduction technique—to efficiently identify the top-K most frequently recurring non-trivial subsequences. The use of SAX enables linear time and space complexity and enhances interpretability by producing discrete motif representations. However, a key limitation of this method is its reliance on fixed-length windows, which constrains its ability to detect patterns of variable duration—an important characteristic in many real-world scenarios. Furthermore, the symbolic abstraction employed by SAX, while computationally advantageous, can lead to reduced accuracy when applied to noisy or structurally complex data. To address this, Tang and Liao [21] proposed an extension that allows motif detection over a predefined range of lengths. This enhancement significantly improves the completeness of motif extraction by accommodating the inherent variability of temporal structures.

A more recent and influential development in motif discovery is the MP algorithm, introduced by Yeh et al. [26]. This method efficiently computes the pairwise z-normalized EuDists between all fixed-length subsequences in a TS, producing a profile that identifies the nearest neighbor for each subsequence. Owing to its scalability, simplicity, and interpretability, the MP has become a foundational tool in modern TSA, particularly for motif and anomaly discovery. Yeh et al. later extended this idea to multivariate settings through the mSTAMP algorithm [25], enabling the detection of meaningful multidimensional motifs across multiple channels. Their work emphasized that straightforward extensions of MPs to multivariate data are insufficient and proposed efficient strategies to preserve temporal consistency across dimensions.

Further developments have explored domain-specific and structural extensions. Wankhedkar and Jain [24] applied MP techniques to ECG data, showcasing their effectiveness in capturing both recurrent cardiac patterns and anomalies. Linardi et al. [15] addressed the limitation of fixed-length motifs by introducing a variable-length motif and discord discovery method, significantly improving flexibility and speed. Gharghabi et al. [8] developed an online, domain-agnostic segmentation approach based on MPs, achieving real-time perfor-

mance and surpassing human performance in continuous monitoring tasks. Together, these works illustrate the versatility of the MP framework and its adaptability to diverse applications. In this thesis, we build upon this foundation by first employing matrix-profile-based candidate generation, followed by a constraint-driven pattern selection strategy designed to achieve high coverage of the TS using a minimal set of representative patterns.

Beyond frequency-based motifs, recent research has highlighted the importance of capturing contextual and temporal regularity in recurring patterns. For instance, Ho et al. [11] introduced the Seasonal Temporal Pattern Mining (STPM) framework, designed to identify motifs that recur periodically at consistent intervals. STPM addresses challenges such as the lack of antimonotonicity and the shortcomings of traditional support metrics by employing innovative pruning strategies and a mutual information-based approximation. This work aligns with the broader trend toward structure-aware and semantically rich pattern discovery methods.

Despite these advancements, a critical limitation remains across existing methods: the lack of emphasis on comprehensive coverage of the entire TS. Most approaches prioritize the identification of prominent motifs—those that are most frequent or most similar—without considering whether these motifs collectively explain the complete dataset. This can result in large portions of the TS being left unrepresented, particularly in complex or noisy domains where recurring patterns may be diverse and less frequent.

The approach proposed in this thesis directly addresses this gap by reframing the pattern selection problem as a Constraint Satisfaction Problem (CSP). Rather than focusing solely on motif frequency or density, this formulation prioritizes coverage while simultaneously enforcing important structural constraints—such as prohibiting overlaps between selected instances and ensuring group-level consistency. CSP models specify the desired properties of the solution in terms of constraints rather than explicit procedural logic, allowing for flexibility and generality in problem specification [2, 1]. To implement this framework, the MiniZinc modeling language [19] is used, which provides a high-level declarative interface to a variety of powerful backend solvers. This enables both exact optimization on smaller datasets and scalable approximation through greedy heuristics on larger or more complex ones.

In summary, while prior work has made significant strides in discovering representative patterns in TSD, most approaches have prioritized motif quality, recurrence, or structural compactness over completeness. By explicitly incorporating coverage into the discovery process and formulating the pattern selection task as a constraint-based optimization problem, this thesis introduces a novel and principled framework for extracting a minimal set of interpretable TS patterns that collectively explain the data. This approach not only fills

a critical gap in the existing literature but also enables new opportunities in downstream tasks such as anomaly detection, behavioral modeling, and efficient TS summarization

Chapter 3

Problem Statement

In this thesis, we consider a univariate TS $t_s = \{t_0, t_1, \ldots, t_{n-1}\}$, consisting of n observations collected sequentially over time. The core objective is to identify a set of patterns $P = \{p_1, p_2, \ldots, p_m\}$, where each pattern p_i is a unique, non-overlapping subsequence of the TS t_s, selected to maximize coverage. To ensure structural significance and prevent trivial matches, we require that each pattern occur at least twice within the TS. These patterns are intended to capture recurring or structurally important behaviors in the data.

The goal is to achieve high coverage of the TS using a compact and interpretable set of patterns. Depending on the application, this task can be formulated in more than one way, as described below.

Problem 1: Maximize Coverage under a Pattern Constraint

$$\begin{aligned} &\max_{P} \; Cov(P, t_s) \\ \text{s.t. } &P = \{p_1, \cdots, p_m\}, \;\; p_i \subseteq t_s, \;\; m \leq max_m \end{aligned} \tag{3.1}$$

Here, $P = \{p_1, p_2, \ldots, p_m\}$ denotes a set of unique and non-overlapping subsequences extracted from the TS t_s. The function $Cov(P, t_s) \in [0, 1]$ quantifies the proportion of the TS covered by all instances of the patterns in P. This value is normalized, with 1 indicating full coverage of the TS. While NoP $m = |P|$ is bounded above by the user-defined parameter max_m.

This formulation focuses on maximizing the total coverage of the TS using a limited NoP. By constraining the NoP to at most max_m, the approach ensures that the resulting solution remains compact and interpretable. This is particularly relevant in scenarios

where simplicity and computational efficiency are prioritized, as it keeps the NoP within a manageable limit.

Problem 2 (Alternative Formulation): Minimize Number of Patterns (NoP) under Coverage Threshold

An alternative formulation to Problem 3.1 is to minimize the NoP m subject to achieving a user-defined minimum coverage C. This approach is suitable for applications where a high level of coverage is essential. The goal is to achieve that coverage with minimal redundancy or pattern complexity.

$$\begin{aligned} &\min_{P} \; m \\ &\text{s.t. } \; P = \{p_1, \cdots, p_m\}, \;\; p_i \subseteq t_s, \;\; \text{Cov}(P, t_s) \geq C \end{aligned} \tag{3.2}$$

Although both Problems 3.1 and 3.2 could theoretically be combined into a single optimization, this would require arbitrary weighting and introduce dimensional inconsistencies, thereby reducing interpretability. To preserve interpretability and maintain the meaningfulness of the optimization process, it is more appropriate to fix one parameter—either by setting an upper or lower bound—while optimizing the other.

Equation (3.1) describes the problem where NoP m is bounded by an upper bound max_m, and the objective is to maximize coverage. Equation (3.2) describes the problem with the coverage bounded by a lower bound C, and the goal is to minimize the NoP m. The choice between these formulations depends on the specific application context. **In this thesis, we focus exclusively on Problem 3.1: maximizing coverage under a pattern constraint.**

In addition to the two optimization variables—(1) the total NoP m and (2) the coverage $\text{Cov}(P, t_s)$—the window size of each pattern serves as a crucial hyperparameter influencing both solution quality and computational cost. Since no strict constraints are imposed on the window size, the method evaluates all candidate subsequences within a user-defined window range. By default, we sample these window sizes exponentially (see Equation 4.1) to balance coverage and efficiency.

Extreme settings of $W_{\min}$ or $W_{\max}$ can result in *trivial solutions*—cases where the algorithm identifies patterns that technically satisfy the constraints but provide little practical insight. For example, setting $W_{\min}$ too low may yield very short, noise-like patterns, while setting $W_{\max}$ close to the full series length may produce a few overly long patterns that cover most of the data without capturing meaningful substructure. Guidance for selecting appropriate values for these parameters is provided in Sections 8.1.1 and 8.1.2.

Chapter 4

Methodology

This chapter presents a robust, scalable, and unsupervised methodology for mining recurring patterns from system TSD. The approach is specifically designed to maximize TS coverage while strictly limiting the NoP, and operates without reliance on labeled data. The methodology is organized into sequential stages, including scalable preprocessing, motif discovery, clustering of similar patterns, and selection of final, non-overlapping patterns via constraint optimization or greedy heuristics. Each stage is described in detail in the following sections. For short TS (up to approximately 20,000 data points), constraint solvers are employed because they guarantee an optimal solution; however, their exponential time complexity restricts their use to small datasets. For longer TS (millions of data points), greedy strategies are applied for efficiency, yielding practical and nearly optimal solutions. The TS is processed in parallelized segments to enable scalable analysis of large datasets. This approach is validated on both real-world and synthetic data, ensuring broad applicability. Overall, this methodology directly addresses the problem introduced in Chapter 3.

4.1 Dataset Description

This study employs both real-world and synthetic system power TSD to evaluate the proposed methodology. The real-world datasets are collected using Hall effect current sensors, which provide precise measurements of electrical current consumption in various systems. Each sensor records data at a high sampling rate of 10,000 samples per second (10 kHz), enabling fine-grained capture of both transient and steady-state behaviors within the monitored environment. All datasets are centrally maintained and accessed via a Minio object storage server, ensuring consistent data retrieval and reproducible experimentation

across multiple analysis runs. Synthetic datasets are constructed by embedding predefined patterns at controlled intervals, providing a clear ground truth for pattern extraction.

Each TS represents a sequence of continuous numeric readings, sampled at the specified rate. Although data are acquired at 10 kHz, all experiments reported here utilize a fixed sampling rate of 1 Hz to ensure comparability across datasets; this is achieved through downsampling during preprocessing. The framework, however, accommodates varying sampling rates as determined by dataset characteristics and user requirements. The datasets encompass diverse durations, typically ranging from a single day to several weeks or months, thereby supporting both short-term and long-term pattern analysis.

To prepare the data for analysis, a sequence of preprocessing steps is applied. First, a median filter is used for downsampling and smoothing, effectively reducing noise while preserving the essential structure of the signal. However, the degree of smoothing directly influences coverage: while moderate smoothing helps reduce noise and supports motif detection, excessive smoothing suppresses fine-grained variations and reduces the diversity of detectable patterns. Subsequently, all values are normalized to the range $[0, 10]$ using min-max scaling to ensure comparability across sessions and simplify the use of an absolute, user-defined EuDist threshold for motif discovery. However, this approach is sensitive to outliers: a single extreme value can distort the scaling and compress the rest of the data, which may affect the interpretation and effectiveness of a fixed threshold across different TSs. Users should consider this limitation when setting the EuDist threshold, and may need to use alternative normalization or adaptive thresholding for datasets with significant outliers (see Section 8.1.3 for details). Finally, missing or incomplete values are handled via interpolation to maintain continuity within each TS. Collectively, these preprocessing procedures ensure that the datasets are well-conditioned for subsequent motif discovery, clustering, and pattern selection.

4.2 Scalable Preprocessing

To efficiently mine patterns from long or high-resolution TS, this methodology applies two key preprocessing steps before motif detection: (i) chunking the TS into manageable, possibly overlapping segments, and (ii) masking extended flat regions that may otherwise inflate computation and obscure informative motifs. Both operations are performed independently for each chunk, forming the basis for all subsequent motif discovery and pattern selection. Implementation details and pseudocode for each preprocessing step are provided in Chapter 5, while their methodological importance is outlined below.

4.2.1 Scalable Chunking Strategy

Analyzing large-scale TS—particularly those spanning several days or weeks and comprising millions of data points—presents significant computational challenges due to the sheer volume of data involved. To address this, the TS is partitioned into smaller, more manageable segments referred to as chunks. Each chunk typically represents a fixed duration (e.g., 24 or 48 hours, or approximately 100,000 data points per chunk), which greatly facilitates parallel processing and significantly reduces memory and runtime requirements for MP computation.

A critical design aspect of this approach is the use of overlap between consecutive chunks. Specifically, each chunk is extended to overlap with its neighbor by a window of length equal to the maximum expected pattern size, denoted as $W_{\max}$. This overlap ensures that motifs located near chunk boundaries are fully captured and not fragmented or omitted during motif detection. Without such overlap, patterns spanning the borders of adjacent chunks may only be partially detected or missed entirely by the algorithm. This segmentation enables processing datasets with millions of points by leveraging parallelism, as emphasized in the thesis goals for real-world deployment.

It is important to note that chunking is an optional preprocessing step rather than a strict requirement of the framework. When computational resources and memory permit, the entire TS can be processed in a single pass without segmentation. However, chunking provides a practical solution for users facing memory limitations or substantial runtime concerns, as it allows the algorithm to be applied sequentially to each chunk, thereby avoiding memory overload and enabling the analysis of very long TS.

Let T denote the original TS, and let $C_1, C_2, \ldots, C_x$ represent the set of overlapping chunks, where each chunk C_i (except the last) includes an overlap of size $W_{\max}$ with its subsequent chunk. This process can be formalized as follows:

$$C_i = T[t_i : t_i + L + W_{\max} - 1], \quad \text{for } i = 1, 2, \ldots, x - 1$$

$$C_x = T[t_x : t_x + L - 1], \quad \text{for } i = x$$

where L denotes the base chunk length (excluding overlap), n is the total length of the TS, and x is the total number of chunks. The number of chunks is given by:

$$N_{\text{chunks}} = \left\lceil \frac{n}{L} \right\rceil$$

With the start and end indices of each chunk computed as:

$$\text{Start}_i = (i-1) \cdot L, \quad \text{End}_i = \min(\text{Start}_i + L + W_{\max},\ n)$$

This ensures that all chunks are properly aligned, incorporate the required overlap, and do not exceed the bounds of the original TS.

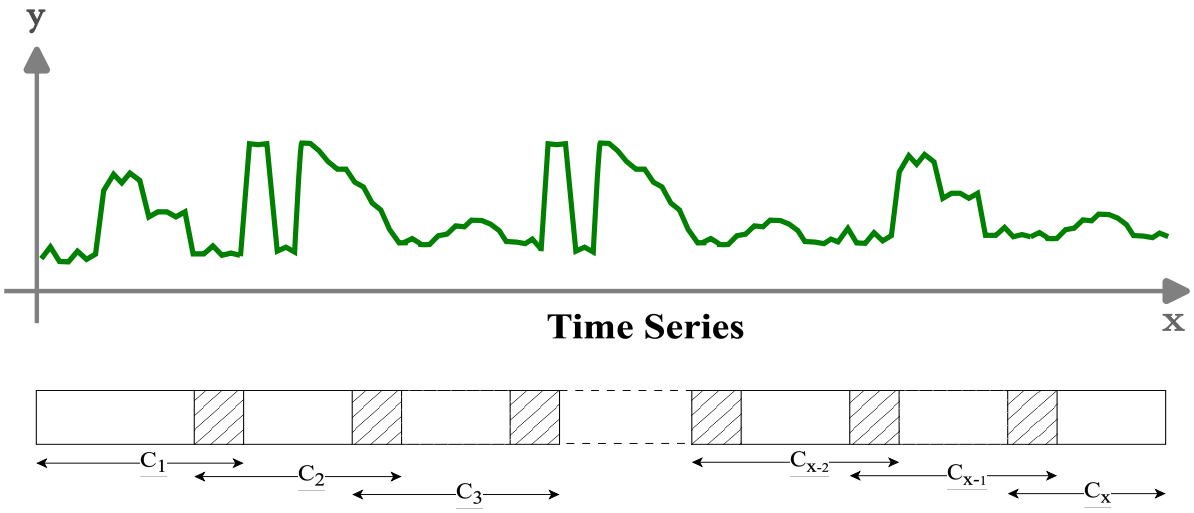


Figure 4.1: Segmentation of a time series (TS) into overlapping chunks. Each chunk C_i overlaps with the next by $W_{\max}$ data points (shaded regions), ensuring motifs spanning chunk boundaries are fully captured. This strategy allows parallel motif discovery and prevents missed patterns at chunk edges.

After segmentation, MPs are computed independently for each chunk using the selected range of window sizes. This allows for concurrent processing across multiple Central Processing Unit (CPU) threads or GPU cores, depending on the available computational resources. The motif candidates identified within each chunk are retained for later aggregation and post-processing (as described in Section 4.4).

Following motif extraction, all candidate patterns from the chunks are merged. Any duplicate or overlapping detections arising from the overlapping regions are resolved to avoid redundancy. This aggregation step ensures that the final set of patterns represents a global view of the entire TS, rather than treating each chunk in isolation.

In summary, chunking is essential for scaling the proposed algorithm to long TS without compromising accuracy. By incorporating overlapping regions of length $W_{\max}$, the

methodology ensures that motifs spanning across chunk boundaries are fully detected. This approach enables efficient parallelization while maintaining the integrity and completeness of motif discovery across large, multi-day, or multi-week datasets (millions of data points).

4.2.2 Flat Region Detection and Masking

To further improve the efficiency of motif discovery in long or high-resolution TS, a key preprocessing operation applied after chunking is flat region masking, which targets extended segments of low variance (flat regions) that commonly occur in real-world data such as power traces. These flat regions can obscure meaningful motifs and inflate computational cost if not handled explicitly.

For each chunk and motif window size W, the TS is scanned with a sliding window. If the variance of a window is below the threshold θ_v, the segment is marked as flat. Contiguous flat runs of at least $3W$ points are masked (excluding boundaries) by setting values to `NaN`, thereby excluding these uninformative regions from matrix profile computation. The impact of the variance threshold θ_v on motif discovery is discussed in Section 6.3. Full implementation details and all related pseudocode for flat region detection and masking are provided in Chapter 5, Section 5.3. Note that this methodological description focuses on the overall approach, while the detailed, step-by-step processing workflow (including chunking, masking, and matrix profile calculation) is presented in Chapter 5.

4.3 Motifs Detection Using Matrix Profile (MP)

To systematically identify recurring subsequences—also referred to as motifs—in the absence of labels, we apply the MP algorithm, implemented via the Stumpy Python library [14]. The MP is a data structure that, for every subsequence of a given window size within a TS, records the distances to its most similar subsequences, thereby enabling efficient motif discovery without prior labeling. This choice is aligned with the thesis objective of discovering patterns in the absence of labeled data, as MPs allow efficient unsupervised discovery of frequent subsequences.

For a TS of length n and a chosen window size W, the MP output for each window size is a matrix of dimensions $(n - W + 1) \times (2k + 2)$. Here, k denotes the number of top matches (nearest neighbors) considered for each subsequence. The reference subsequence is defined as the contiguous segment of length W starting at index i, where i ranges from

0 to $n - W$. Thus, there are a total of $n - W + 1$ reference subsequences for each window size (see Figure 4.2 and 4.3). The interpretation of the output columns is as follows:

- The first k columns `mp[:, :k]` contain the top-k smallest EuDists between the reference subsequence and its closest subsequences.
- The next k columns `mp[:, k:2k]` record the indices of these k matching subsequences within the TS.
- The final two columns `mp[:, 2k]` and `mp[:, 2k + 1]` (or, equivalently, `mp[:, -2]` and `mp[:, -1]`) correspond to the left and right MP indices for the top-1 matches. These columns are not used in our analysis.

Thus, the primary focus is on the first $2k$ columns, which together provide both the proximity of the match (distance) and its exact location (index) for each reference subsequence. When $k = 1$—the default setting in Stumpy—the first column holds the minimum EuDist for each reference subsequence, while the second column contains the index of its nearest neighbor. This format efficiently supports the identification of repeated patterns across the entire series.

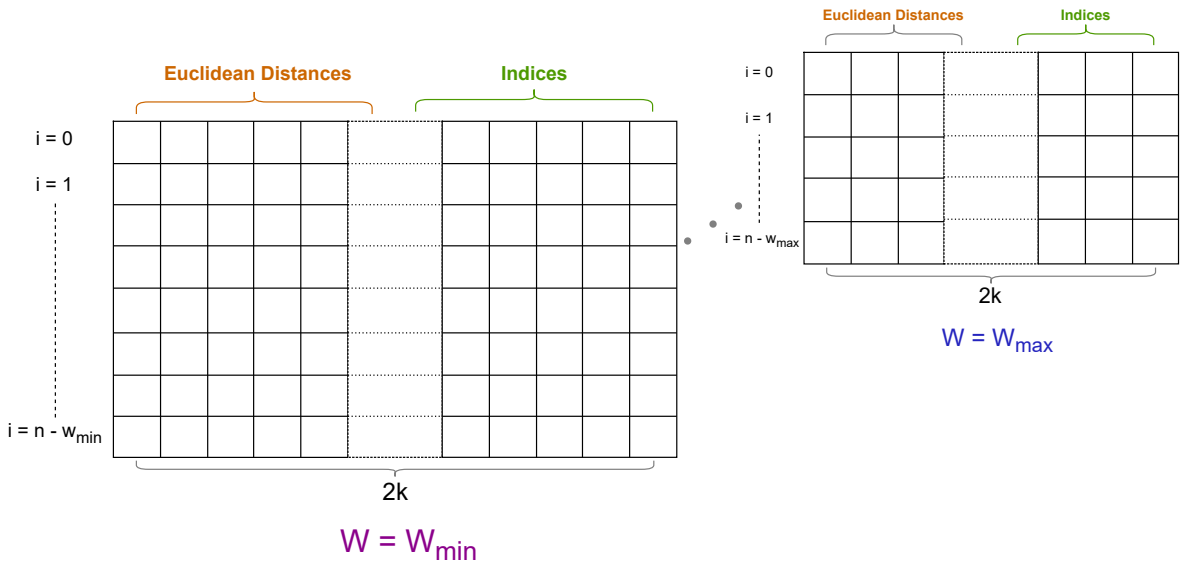


Figure 4.2: Structure of matrix profile (MP) outputs for different window sizes. The MP is calculated for each window size W from $W_{\min}$ to $W_{\max}$. For each W, the MP consists of $2k$ columns: the first k columns contain EuDist between reference subsequences and their top k nearest neighbors, and the next k columns store the corresponding indices in the TS. This allows the capture of patterns with different durations.

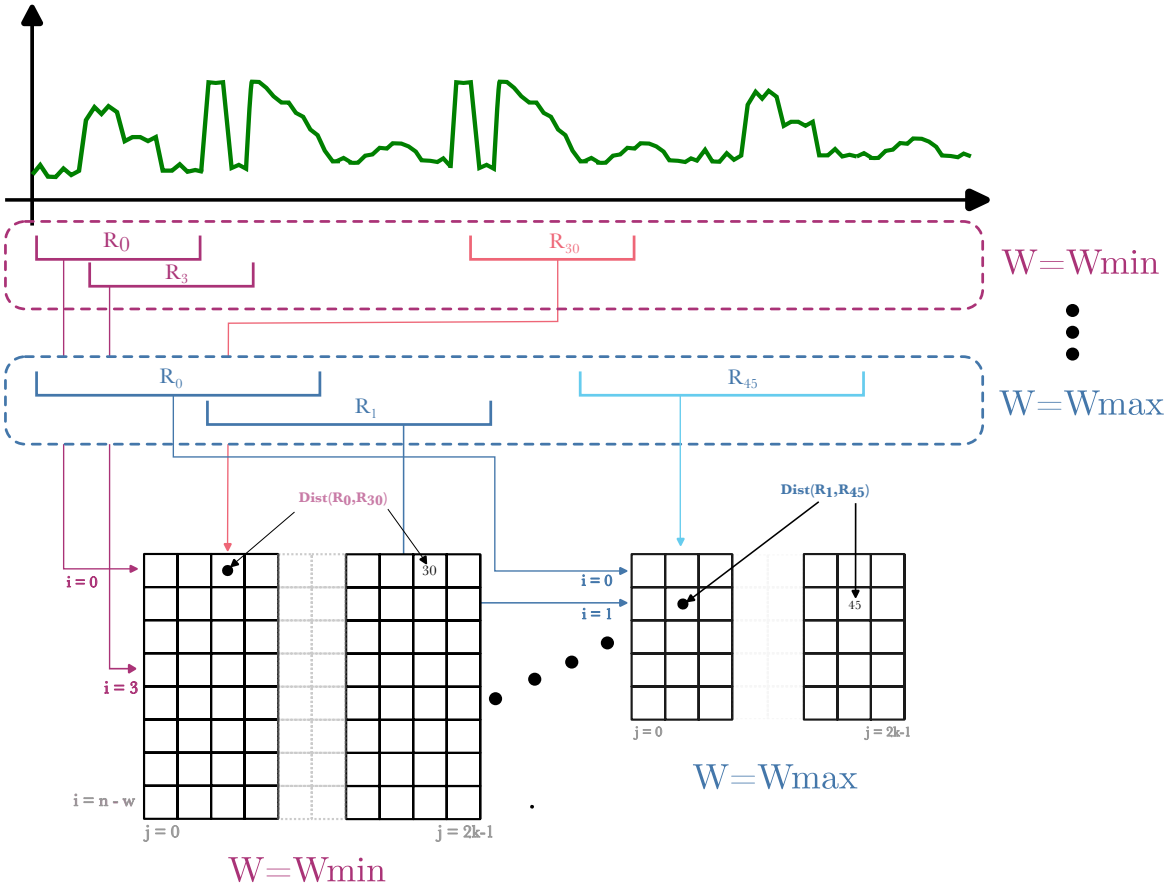


Figure 4.3: Illustration of matrix profile (MP) computation for multiple window sizes. Each MP block summarizes, for a specific window size, the top k EuDist and indices of the most similar subsequences in the TS. Colored arrows demonstrate how reference subsequences at $W_{\min}$ and $W_{\max}$ map to corresponding entries in the MP matrices. The final two columns of each matrix are omitted from further analysis. Dist(R_0,R_{30}) represent the EuDist between reference index i=0 and its similar subsequence j=30.

Two key challenges arise in this process. First, the optimal value of k—the number of nearest neighbors to consider is not known a priori. In the MP computation, motifs depend on the chosen k. To remove this sensitivity, we introduce a EuDist threshold θ_{ed} as described in Section 6.2.2 by modifying the Stumpy implementation: we initially set $k = n-W+1$ to allow all subsequences as neighbours, but subsequently adopt the memory-efficient strategy described in Section 8.1.6 and retain only those subsequences whose EuDist to the reference subsequence is $\leq \theta_{ed}$. The value of θ_{ed} governs the trade-off between

similarity strictness and motif diversity, with smaller values yielding fewer but more homogeneous motifs and larger values producing more diverse yet potentially less coherent motifs. Its practical impact on GPU ET is shown in Section 6.4, where smaller thresholds markedly reduce computation time through stricter early pruning. Consequently, the MP computation is governed by θ_{ed} rather than k, and our empirical study in Section 6.2 shows the effect of k on the computational time. This pruning does not compromise motif quality because θ_{ed} is the same criterion later used in post-processing to define motif; any true motif pair within θ_{ed} is preserved. Motivated by the observation in Section 6.2 that increasing k increases MP runtime, we integrated this exclusion threshold directly into the MP stage to perform early pruning—skipping subsequence comparisons once the EuDist exceeds θ_{ed}—and to avoid generating pairs that would be discarded during post-processing.

Second, motif discovery in long TS incurs substantial memory requirements, as the length of the TS increases, the memory requirements increase quadratically with length of TS. To address this challenge, the algorithm partitions long TS into smaller, manageable segments, such as daily or weekly chunks. This chunking approach significantly reduces memory requirements, allows motifs to be extracted in parallel across multiple segments, and improves scalability. The motifs from each chunk are subsequently integrated using a clustering strategy described in Section 4.4.

Within each segment, motif discovery is performed over a user-specified set of window sizes W_{List}. If W_{List} is not provided, we generate it automatically using an exponential schedule between $W_{\min}$ and $W_{\max}$ with growth factor $r > 1$ (see Chapter 3):

$$W_i = \left\lfloor W_{\min} \cdot r^i \right\rfloor, \quad \text{for } i = 0, 1, 2, \ldots, x \quad \text{such that } W_x \leq W_{\max},\ r > 1 \tag{4.1}$$

Here, $W_{\min}$ is the minimum window size, $W_{\max}$ is the maximum, and r is the growth rate (e.g., 1.1). This schedule reduces computation (fewer windows) and avoids closely spaced window sizes while still covering short- and long-duration motifs. Users may override the default by passing any custom W_{List} (e.g., $[W_1, W_2, W_3]$ or an application-specific set). All subsequent steps operate on candidates produced at the supplied window sizes. Given the user-defined W_{List} and EuDist threshold θ_{ed}, the selection stage maximizes coverage (Problem 3.1) under the pattern-count constraint ($m \leq max_m$)) using only these candidates.

Before introducing the EuDist threshold variable θ_{ed} directly into Stumpy, the pipeline followed a two-step process. First, the MPs were computed in full. Then, in a separate post-processing stage, subsequence pairs were grouped as motifs if their EuDist $\leq \theta_{ed}$. While this approach examined all possible subsequences in the TS and returned the top-k

closest matches based on EuDist, regardless of whether many of these pairs would later be discarded for exceeding the threshold. As a result, the computation time was unnecessarily high, since it included distance evaluations for subsequences that could never be part of the final motif set.

In the optimized pipeline, the same criterion θ_{ed} is enforced during MP computation by modifying Stumpy itself. This modification enables early pruning: any candidate pair with EuDist $> \theta_{ed}$ is skipped entirely, preventing further computation for that pair. Only pairs satisfying EuDist $\leq \theta_{ed}$ are retained, eliminating the need for a separate threshold-based filtering step and significantly reducing total computation time (as demonstrated in Section 6.2.2). The parameter k is set to its optimal value k_{opt}, as described in Section 8.1.6, to ensure balanced motif discovery. Finally, a post-processing stage ensures that, for each reference index in the MP, the retained matches are non-overlapping for the given window size.

Algorithm 3 performs threshold-pruned motif discovery in the optimized pipeline. For each user-specified window size $W \in W_{\text{list}}$, the MP is computed with the EuDist threshold θ_{ed} enforced during computation so only neighbor pairs with distance $\leq \theta_{ed}$ are produced. A post-processing step then retains only non-overlapping matches for each reference index.

4.4 Clustering Patterns with DBSCAN

Once candidate patterns have been extracted from each chunk of the TS, it is necessary to group similar motifs that may have been detected independently in multiple overlapping chunks. This consolidation is essential for forming global candidate pattern groups and ensuring that each unique pattern is represented only once in the final analysis. To achieve this, the DBSCAN algorithm [6] is employed to cluster similar subsequences based on their pairwise distances. This step ensures that each recurring pattern is represented only once in the candidate set, minimizing redundancy and improving downstream selection quality.

DBSCAN is particularly well suited to this context because it does not require the number of clusters to be specified in advance—a key advantage in TSA, where the true number and structure of motifs are often unknown. The algorithm uses a distance-based criterion to identify clusters, and any subsequence that does not belong to a sufficiently dense region is labeled as noise (which, in most DBSCAN implementations such as scikit-learn, is assigned the label -1 by convention). This feature allows DBSCAN to both form coherent candidate pattern groups and automatically filter out anomalous or outlier detections.

It is important to clarify that DBSCAN clustering is applied at two distinct stages in our pipeline. First, within each chunk (or the full TS if chunking is not used), after extracting non-overlapping candidate subsequences for each reference index, we use DBSCAN to address two key issues. The first is that, while a given reference subsequence i is similar (under the threshold) to both j and k, there is no guarantee that j and k are mutually similar. DBSCAN resolves this by forming clusters where all members are sufficiently close to one another, ensuring mutual coherence within a motif group. The second issue arises from the symmetric nature of nearest-neighbor relationships in the MP: if reference subsequence i is similar to j, then reference subsequence j will also be similar to i, leading to duplicate motif candidates across different reference indices. By clustering these reciprocal matches into a single group, DBSCAN eliminates redundancy and ensures that each motif is represented only once.

Second, after motif extraction from all chunks, we apply DBSCAN again to the aggregate set of motif representatives from each chunk. This global clustering step merges similar motifs across chunks, enabling the discovery of recurring patterns throughout the entire dataset. This two-stage DBSCAN approach ensures robust, consistent motif grouping at both local and global levels.

In this implementation, DBSCAN operates on subsequence vectors, which are either extracted directly from the raw TS or their lower-dimensional feature representations, depending on the computational and memory constraints of the dataset. EuDist is used as the similarity metric, maintaining consistency with the distance function applied during MP computation.

Formally, for a given subsequence p, the ε-neighborhood is defined as:

$$N_\varepsilon(p) = \{q \in D \mid \text{dist}(p, q) \leq \varepsilon\}$$

where D is the set of all subsequences under consideration, and $\text{dist}(p, q)$ denotes the EuDist between subsequences p and q. A subsequence p is considered a core point if:

$$|N_\varepsilon(p)| \geq \text{min_samples}$$

Here, ε is a user-defined radius that determines the density threshold for cluster formation, while min_samples specifies the minimum number of subsequences required to constitute a valid cluster. In this study, min_samples is set to 2 to ensure that each cluster contains at least two similar subsequences.

The output of the clustering process is a set of refined candidate pattern groups, each defined by a window size and a list of global start indices for motifs in the full TS. During post-processing, redundant detections—especially those arising from chunk overlap—are resolved, and subsequences labeled as noise are excluded from further consideration. This clustering step ensures that localized motif detections from chunks are consolidated into globally consistent candidate pattern groups, thereby reducing redundancy and enhancing the interpretability of the extracted patterns. By consolidating redundant motif instances across chunks, this clustering step directly supports our goal of maximizing overall coverage using a concise set of patterns.

4.5 Pattern Selection Approaches

The final stage of the proposed solution is the selection of a subset of candidate patterns that maximizes TS coverage while ensuring interpretability and minimizing redundancy. The set of candidate patterns obtained from the MP and subsequent grouping is reusable, allowing the pattern selection stage to target different coverage levels without recomputing the MP. For example, a solution that achieves 60% coverage with NoP = 10 can be extended to higher coverage by increasing the NoP value.

To achieve this, two complementary strategies are adopted: (1) an exact optimization approach using the MiniZinc constraint modeling framework, and (2) a suite of scalable greedy heuristic algorithms. The choice of approach is governed by the scale of the dataset: MiniZinc provides rigorous guarantees for small to medium-sized TS (up to approximately 20,000 data points), while heuristic methods are preferred for large-scale or complex datasets (with millions of data points) due to their practical computational efficiency.

4.5.1 MiniZinc-Based Constraints Formulation and Solving

MiniZinc is a high-level modeling language widely used for specifying and solving combinatorial constraint satisfaction and optimization problems [19]. In the context of this thesis, pattern selection is formulated as a constrained decision problem over binary variables. Each candidate pattern C_i is associated with a set of binary decision variables $O_{ij} \in 0, 1$, where $O_{ij} = 1$ indicates that the j-th occurrence of pattern C_i is selected. The model enforces two principal types of constraints:

- **Group Constraints**: Ensure that all or none of a pattern's occurrences are selected, thereby avoiding partial inclusion of motifs that are intended to appear together.

$$\forall C_i \in \mathcal{C} : \sum_{j=1}^{|C_i|} O_{ij} = 0 \quad \text{or} \quad \sum_{j=1}^{|C_i|} O_{ij} = |C_i| \tag{4.2}$$

  Equivalently, this can be expressed as $O_{ij} = y_i$ for all j, where $y_i \in 0, 1$ is a binary indicator variable for pattern C_i.

- **Exclusion Constraints**: Prevent overlapping patterns from being simultaneously selected. For any set of occurrences that share common time indices, the model restricts selection to at most one occurrence.

$$\sum_{(i,j)\in\mathcal{S}} O_{ij} \leq 1 \qquad \forall \mathcal{S} \text{ overlapping set} \tag{4.3}$$

The optimization objective is flexible and may be configured to either:

- **Maximize coverage**:

$$\max \quad \sum_{i=1}^{N_p} \sum_{j=1}^{n_i} O_{ij} \cdot \text{length}(C_i)$$

  where $\text{length}(C_i)$ denotes the number of data points covered by pattern C_i.

- **Minimize the NoP**:

$$\begin{aligned} \min \quad & \sum_{i=1}^{N_p} y_i \\ \text{s.t.} \quad & y_i \in \{0, 1\}, \\ & O_{ij} = y_i \quad \forall\, i, j, \\ & \sum_{(i,j)\in\mathcal{S}} O_{ij} \leq 1 \quad \forall\, \mathcal{S} \text{ (overlapping set)}, \\ & \text{Coverage}(P, t_s) \geq min_Cov \end{aligned} \tag{4.4}$$

In practice, one of these objectives is optimized while the other is constrained, following the problem statement formalized in Chapter 3. Consistent with Problem 3.1 defined in Chapter 3, this formulation aims to maximize coverage while bounding the NoP m.

The complete optimization problem can be summarized as:

$$\begin{aligned} \max \quad & \sum_{i=1}^{N_p} \sum_{j=1}^{n_i} O_{ij} \cdot \text{length}(C_i) \\ \text{s.t.} \quad & O_{ij} \in \{0,1\}, \quad y_i \in \{0,1\}, \\ & O_{ij} = y_i \quad \forall\, i,j, \\ & \sum_{i=1}^{N_p} y_i \leq m, \\ & \sum_{(i,j)\in\mathcal{S}} O_{ij} \leq 1 \quad \forall\, \mathcal{S} \text{ (overlapping set)} \end{aligned} \tag{4.5}$$

This formulation enables the efficient and interpretable selection of a maximal coverage pattern set while strictly enforcing group and exclusion constraints.

4.5.2 Greedy Heuristic Approaches for Large-Scale Selection

To address scalability challenges in large TS datasets, the study incorporates a suite of greedy heuristic algorithms that efficiently approximate the pattern selection task. These approaches operate under the same constraints as the MiniZinc model but use simplified, iterative decision-making to reduce computational overhead.

The following high-level heuristics were adopted:

- **Naive Greedy Selection**: A basic, iterative method that scans the candidate list and greedily adds patterns that provide the highest marginal coverage gain while avoiding overlaps.
- **Greedy Selection with Interval Tree Optimization**: An improved greedy method that uses an interval tree data structure to accelerate conflict checks. This substantially reduces redundant comparisons and enhances runtime efficiency [18, 4].

- **Simulated Annealing (SA)**: A metaheuristic strategy that begins from a greedy solution and iteratively perturbs the pattern set. It occasionally accepts worse solutions to escape local optima, gradually converging to a high-quality selection. The balanced variant used here applies a calibrated cooling schedule to trade off exploration and convergence [23, 13].

These greedy and metaheuristic methods offer practical solutions for large-scale, real-world TSs where exact constraint solvers become computationally infeasible. The overarching goal across all approaches is to select a subset of non-overlapping patterns that collectively maximize coverage of the TS, while adhering to predefined group and exclusion constraints. The rationale behind the selection and configuration of these methods is elaborated in Chapter 5, and their comparative performance—measured in terms of coverage, number of selected patterns, and runtime—is systematically evaluated in Chapters 6 and 7.

4.6 Summary of Methodological Pipeline

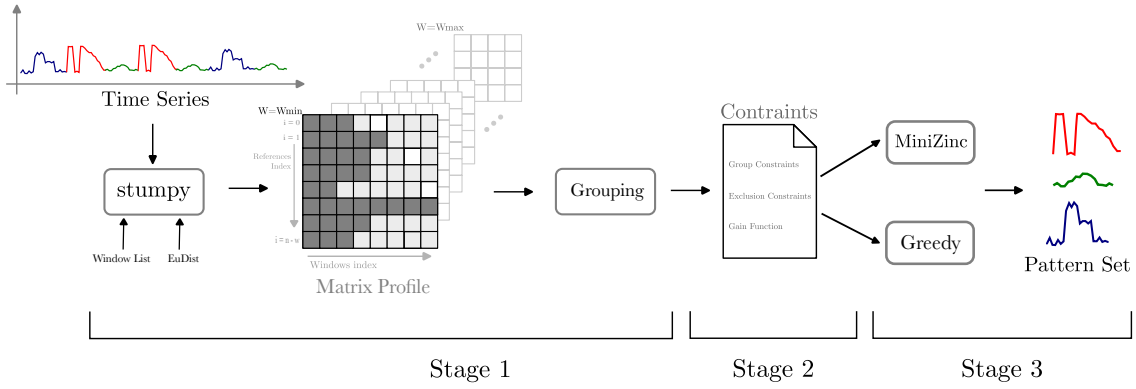


Figure 4.4: Pattern mining pipeline overview. Stage 1 detects motifs using matrix profile (MP) computation over multiple window sizes with a modified Stumpy and inclusion EuDist parameter, followed by DBSCAN grouping. Stage 2 applies group and exclusion constraints, and Stage 3 selects non-overlapping patterns maximizing coverage via exact optimization (MiniZinc) or greedy heuristics. **Note: Preprocessing steps chunking and flat region masking are not shown here for clarity; the full pipeline—including all preprocessing—is illustrated in Figure 5.2 and discussed in Section 4.2.**

In this chapter, we present a systematic and scalable methodology for unsupervised pattern mining in large-scale TSD, directly addressing the challenges identified in the problem statement. The structure of the proposed pipeline, as illustrated in Figure 4.4, consists of three principal stages: motif detection, constraint formulation, and pattern selection. For clarity, this figure omits certain preprocessing steps—specifically, chunking and flat region masking—which are essential for scalable and robust motif discovery in practice. The full implementation pipeline, including all preprocessing steps necessary for real-world datasets, is detailed in Chapter 5 (see Figure 5.2).

Altogether, this methodology introduces a robust and interpretable framework for pattern mining in univariate TSD. Unlike existing approaches that emphasize motif frequency or visual distinctiveness, this method emphasizes maximizing overall coverage under a strict constraint on the NoP, aligning with the optimization focus defined in Problem 3.1 of Chapter 3. By combining motif detection, scalable clustering, and constraint-driven selection, it ensures that each discovered pattern is both meaningful and non-redundant. The resulting non-overlapping pattern sets offer a compact yet expressive summary of system behavior, enabling downstream applications such as anomaly detection, system summarization, and policy verification. These capabilities are further demonstrated in Chapters 6 and 7. The next chapter presents implementation details, showcasing how this approach effectively addresses the coverage gap in TS pattern mining.

Chapter 5

Implementation

This chapter presents the implementation of the proposed unsupervised pipeline for mining recurring patterns in univariate TSD, with a particular emphasis on system power signals. Building upon the methodological foundation established in Chapter 4, the implementation aims to deliver a modular, scalable, and interpretable software solution capable of handling both short and long-duration datasets efficiently.

The implementation follows a three-stage architecture: (1) motif discovery using MP computations across multiple window sizes, (2) global grouping of structurally similar motifs using density-based clustering, and (3) constrained pattern selection through exact optimization solver or scalable greedy heuristics. Each stage has been carefully designed to ensure a balance between computational efficiency and analytical rigor, ensuring that the system remains adaptable, reproducible, and suitable for real-world TS applications. This chapter begins by outlining the tools, software libraries, and computational resources employed throughout the development and experimentation process.

5.1 Tools and Technologies

This section summarizes the principal software tools, libraries, and computational resources used to implement the proposed pipeline and execute the experiments.

5.1.1 Software Libraries and Frameworks

The implementation is primarily developed in Python and incorporates a range of open-source libraries supporting MP computation, clustering, constraint optimization, and data handling. The major components are described below, grouped by their functional roles:

- **Motif Discovery:** The Stumpy library (v1.13.0) [14] used for scalable computation of the MP [14]. It supports both z-normalized and raw EuDists. In this work, we employ the raw EuDist mode to work with motif extraction. The Stumpy library supports GPU-based acceleration via the `gpu_stump()` function, enabling efficient motif extraction on long TS.
- **Clustering of Candidate Motifs:** Structurally similar motif instances identified across different segments and window sizes are grouped using the DBSCAN clustering algorithm, as implemented in the scikit-learn library. DBSCAN is well-suited for TS motif grouping as it does not require prior knowledge of the number of clusters and can effectively identify noise points.
- **Constrained Pattern Selection:** To formulate and solve the constrained selection problem defined in Chapter 3, the MiniZinc (v2.6.4) modeling language is used [19]. It enables high-level specification of logical constraints, including group-level inclusion and overlap exclusion. For large-scale datasets, where exact constraint solving is computationally infeasible, a suite of greedy heuristics is implemented in Python, enabling fast, approximate selection of non-overlapping pattern sets with near-optimal coverage.
- **Supporting Libraries and Data Handling:** Core numerical operations and data handling are performed using NumPy and Pandas. NumPy provides fast matrix computations, array operations, and numerical routines essential for motif discovery and filtering. Pandas is used to structure, manipulate, and annotate the large volumes of intermediate and final results generated during the pipeline.

5.1.2 Computational Resources

All experiments were conducted on a dedicated remote server running Ubuntu 22.04.5 LTS. The hardware configuration included:

- **Three NVIDIA TITAN Xp GPUs** (each with 12 GB VRAM), used for accelerating MP computations and handling large-scale data in parallel.

- **Intel Core i7-5930K CPU @ 3.50GHz** with 12 threads, used for general-purpose computation and CPU-based preprocessing and clustering tasks.
- **32 GB RAM** to accommodate intermediate data structures during chunking, clustering, and constraint solving.
- **2 TB SSD storage** to manage large volumes of TSD and intermediate results.

The software environment was configured with `Python 3.13.5`, `NumPy 1.21.5`, and associated dependencies. GPU acceleration was enabled using CUDA 11.5 with NVIDIA driver version 570.153.02, ensuring compatibility with the GPU-accelerated Stumpy backend. This setup enabled parallelization of TS chunks and accelerated computation of MPs, both of which are critical to the scalability of the proposed pipeline. This high-performance setup ensures that the pipeline remains scalable and responsive, especially during MP computations and motif clustering, which are computationally intensive for long TS.

5.2 Implementation of Chunking and Parallelization

As previously introduced in Section 4.2.1, chunking is a fundamental design component that enables scalable motif discovery by partitioning long univariate TSs into smaller, overlapping segments. This section focuses on the implementation aspects of this strategy, detailing how the chunking process is operationalized to support efficient MP computation. Chunking not only facilitates memory-efficient processing but also enables the pipeline to scale effectively across large datasets, addressing the coverage maximization problem discussed in Chapter 3. Chunking is optional in the implementation; users may process the full TS in a single segment when memory allows.

The chunking procedure is implemented as a preprocessing step applied directly to the input TS, which is represented as a one-dimensional NumPy array. Each chunk is defined by a fixed base length L and is extended with an overlap region of size $W_{\max}$, corresponding to the maximum expected motif length. This overlap ensures that motifs spanning across segment boundaries are fully captured. The final chunk is adjusted to prevent index overflow when the TS length is not an exact multiple of the chunk size.

A loop iteratively computes the start and end indices for each chunk, ensuring the end index does not exceed array bounds by applying the `min()` function. Each chunk is extracted as a NumPy array and stored along with its corresponding metadata—specifically,

the start and end indices relative to the original TS. These entries are maintained in a list of tuples.

To enable efficient computation across long TSs, each chunk is treated as an independent unit of processing. In practice, MP computations are applied separately to each chunk to prevent memory saturation and enable scalable execution. On GPU-equipped systems, chunks are processed in sequential batches to optimize memory usage and maximize throughput.

A high-level pseudocode representation of the chunking logic is provided in Algorithm 1, as implemented in the pipeline. This implementation forms the foundation for scalable motif discovery, enabling MP computations to be performed independently across manageable data segments in the subsequent stages of the pipeline.

Algorithm 1 Chunking a Time Series (TS) into Overlapping Sub-Series

Require:

t_s: input Time Series (TS) (NumPy array) of length n
L: base chunk length (excluding overlap)
$W_{\max}$: maximum expected window size (used as overlap size)

Ensure: List of overlapping sub-series, each defined by `(start_idx, end_idx, segment)`

Initialize empty list `chunks`
$n \leftarrow \texttt{length}(t_s)$
$N \leftarrow \lceil \frac{n}{L} \rceil$ ▷ Total number of chunks
for $i = 0$ to $N - 1$ **do**
    `start` $\leftarrow i \cdot L$
    `end` $\leftarrow \min(start + L + W_{\max},\ n)$ ▷ Extend chunk by overlap
    `segment` $\leftarrow t_s[\texttt{start} : \texttt{end}]$ ▷ Extract overlapping chunk
    Append `(start, end, segment)` to `chunks`
end for
return `chunks`

5.3 Flat Region Masking Prior to Matrix Profile (MP) Computation

To reduce unnecessary computation during motif discovery via MP, this implementation incorporates a flat region masking step that is applied immediately after chunking and

before MP computation. The purpose of this step is to identify flat regions—defined as segments with negligible variance or minimal change over time—and exclude them from the MP analysis by marking them with `NaN` values. Since the Stumpy library automatically ignores any subsequence containing `NaN`, this pre-processing technique effectively prevents trivial patterns from participating in distance calculations.

Importantly, this masking is performed separately for each motif length. For each chunk generated during preprocessing, the algorithm iterates over a list of motif window sizes $W \in W_{\text{list}}$, and applies masking specific to each window length. This is necessary because flatness is a relative property that depends on the size of the subsequences under analysis. Consequently, the flat region detection and masking process must be repeated for each window to ensure consistency with motif length and to maintain tight integration with the overall discovery process.

The masking routine, detailed in Algorithm 2, proceeds as follows. For a given chunk and window size W, the TS is scanned using a sliding window of length W. Each windowed subsequence is evaluated to determine whether it qualifies as flat. A subsequence is considered flat if its variance falls below a predefined threshold θ_v, which is scaled with the window length using a sensitivity factor α. The result of this scan is a boolean array indicating which positions in the TS correspond to flat subsequences.

Once the flat subsequences are identified, the algorithm groups them into contiguous runs. Long flat runs—those spanning at least three consecutive window lengths ($3W$)—are treated as true flat regions, where W is the current window size from the user-defined W_{list}. For each W in W_{list}, this masking procedure is applied separately, ensuring that only regions flat for at least $3W$ samples are considered for masking. This minimum length criterion of $3W$ is a fixed design choice in our implementation and is not an additional user-defined hyperparameter. The first and last W points of each such region are always preserved to avoid masking potential motif boundaries, while only the central portion is masked out as uninformative.

This procedure is implemented using a combination of NumPy-based variance computations and index-based run detection. A copy of the chunk is modified in place to insert `NaN` values within masked segments, resulting in a masked TS subsequently used for MP computation via the Stumpy library. Masking low-variance flat regions ensures that only structurally informative motifs contribute to the final coverage optimization, avoiding skewed or redundant patterns. The implementation is efficient and compatible with both CPU and GPU backends of the Stumpy API.

A formal pseudocode description of the masking process is provided in Algorithm 2. This routine forms a critical step in the MP pipeline, enabling efficient and targeted exclu-

sion of uninformative subsequences while preserving the structural integrity of meaningful motifs.

Algorithm 2 Flat Region Detection and Masking Based on Variance

Require:
T_{chunk}: input chunk Time Series (TS) (NumPy array)
W: window size for flatness detection
α: flatness sensitivity factor

Ensure: Chunk TS T_{masked} with `NaN` values in flat zones (excluding boundaries)

$n \leftarrow \texttt{length}(T_{\text{chunk}})$
$\theta_v \leftarrow \alpha \cdot W$ ▷ – Threshold going to depend on window –
Initialize Boolean array `is_flat` of length $n - W + 1$ with `False`
for $i = 0$ to $n - W$ **do**
  if $\texttt{variance}(T_{\texttt{chunk}}[i : i + W]) \leq \theta_v$ **then**
    $\texttt{is_flat}[i] \leftarrow \texttt{True}$
  end if
end for
$T_{\text{masked}} \leftarrow \texttt{copy}(T_{\text{chunk}})$
$i \leftarrow 0$
while $i < \texttt{length}(is_flat)$ **do**
  if $\texttt{is_flat}[i]$ **then**
    $run_start \leftarrow i$ ▷ – Identify end of the current flat run –
    while $i < \texttt{length}(is_flat)$ **and** $\texttt{is_flat}[i] = \texttt{True}$ **do**
      $i \leftarrow i + 1$
    end while
    $run_end \leftarrow i$
    if $run_end - run_start \geq 3W$ **then**
      $mask_start \leftarrow \max(run_start + W,\ 0)$ ▷ – Excluding boundaries –
      $mask_end \leftarrow \min(run_end,\ \texttt{length}(T_{\text{masked}}))$
      $T_{\text{masked}}[mask_start : mask_end] \leftarrow \texttt{NaN}$
    end if
  else
    $i \leftarrow i + 1$
  end if
end while
return T_{masked}

Time Complexity. The flat region masking algorithm operates in $O(nW_{\max})$ time per chunk, where n is the length of the chunk and $W_{\max}$ is the largest window size in W_{list}. For each window size W, the algorithm performs a sliding window variance computation, which can be optimized to $O(n)$ using cumulative sums, followed by a linear pass for run detection and masking. Since masking is performed for each window size in W_{list}, the total complexity per chunk is $O(n \cdot |W_{\text{list}}|)$ in practice. As a result, the approach remains efficient and scalable, even for large TSD.

5.4 Matrix Profiles (MPs) Computations

Building upon the preprocessing stages of chunking and flat region masking (Sections 5.2 and 5.3), this section presents the implementation of the core motif discovery procedure based on MP computation. This step plays a pivotal role in identifying structurally similar, recurring subsequences within each chunk of the TS. The implementation leverages the Stumpy API, which provides optimized, scalable routines for computing the MP across a set of window sizes. This step directly operationalizes the motif discovery strategy described in Section 4.3, addressing the need for high-coverage candidate pattern extraction as formulated in Chapter 3.

For each chunk extracted from the original TS, the algorithm iterates over a user-specified list of window sizes $W \in W_{\text{list}}$, where each window size corresponds to a distinct motif length of interest. For each window size W, flat region masking is first applied to the chunk to eliminate subsequences with flat or low-variance regions, as described in Section 5.3. This preprocessing step ensures that only informative subsequences are considered during similarity calculations. Subsequently, the masked chunk is passed to an optimized MP computation function built upon the Stumpy API. This implementation has been modified to incorporate a distance-based threshold θ_{ed}, enabling the exclusion of weak matches early in the computation. For further implementation details of this optimization, refer to Section 6.2.2 for details on early pruning using EuDist.

After computing the MP, we apply DBSCAN clustering within each chunk to group similar motif candidates. While the MP efficiently identifies pairwise similar subsequences based on EuDist, it does not guarantee mutual similarity among all group members. By applying DBSCAN for each chunk, we ensure that only dense, mutually coherent groups are selected as motif clusters, with all motifs in a cluster being sufficiently close to each other according to the distance threshold. The final motif sets for each chunk are represented as tuples $(W, [l_1, l_2, \dots])$, where W is the window size and the sorted list contains the starting indices of the discovered motif instances. Since multiple motifs can be found for

the same window size, there may be several tuples sharing the same W. Each such list collectively represents a single, interpretable pattern of length W. This process is repeated independently for each chunk and each window size. The full logic of this implementation is summarized in Algorithm 3.

Algorithm 3 Motif Detection for a chunk using Matrix Profile (MP) with Flat Region Masking and Grouping

Require:

T_{chunk}: input time series TS chunk
W_{list}: user-defined list of window sizes
θ_{ed}: user-defined EuDist threshold
$f_{\min}$: minimum required occurrences per pattern (default: 2)
α: flatness sensitivity factor

Ensure: For each W: motif pattern groups as sets of non-overlapping start indices

$\mathcal{P} \leftarrow \emptyset$ ▷ Stores motif groups across all window sizes
for each window size W **in** W_{list} **do**
  $T_{\text{masked}} \leftarrow \text{MASKING}(T_{\text{chunk}}, W, \alpha)$ ▷ Mask flat regions using Algorithm 2
  $MP \leftarrow \text{COMPUTEMATRIXPROFILE}(T_{\text{masked}}, W, \theta_{ed})$
  $G \leftarrow \text{REMOVEOVERLAPS}(MP, W, f_{\min})$ ▷ Get non-overlapping motifs
  $G_W \leftarrow \text{DBSCAN_CLUSTER}(T_{\text{masked}}, G, \theta_{ed})$ ▷ Cluster motifs using DBSCAN
  $\mathcal{P} \leftarrow \mathcal{P} \cup G_W$ ▷ Append motif groups for current W
end for
return $\mathcal{P}$

Time Complexity. The time complexity of Algorithm 3 is dominated by $O(|W_{\text{list}}|\, n \log n)$ per chunk, where $|W_{\text{list}}|$ is the number of window sizes and n is the chunk length. This accounts for the FFT-based MP computations for each window size using the Stumpy API. Threshold-based early pruning via θ_{ed} eliminates the need to consider a fixed number of neighbors, so the cost of motif grouping and clustering is typically linear in the number of retained motif matches, which remains modest due to threshold filtering. Overall, the algorithm remains highly efficient and scalable for large TS.

The output of this stage is a structured set of candidate patterns $\mathcal{P}$, formally defined as:

$$\mathcal{P} = \{P_1 = (W_1, [l_{11}, l_{12}, \dots]),\ \dots,\ P_n = (W_n, [l_{n1}, l_{n2}, \dots])\} \tag{5.1}$$

Here, each pattern $P_i \in \mathcal{P}$ is characterized by a window size W_i and an associated list of non-overlapping starting indices $[l_{i1}, l_{i2}, \dots]$, corresponding to structurally similar

subsequences within a specific chunk. These candidate motifs are aggregated across all chunks to form a consolidated local pattern set $\mathcal{P}_{\text{local}}$. This set serves as the input to the subsequent global motif grouping stage, where patterns are clustered across chunk boundaries. These structured candidate groups provide interpretable building blocks that support the optimization tasks introduced in Section 4.5. Ultimately, this process lays the groundwork for downstream constraint-based selection and optimization to maximize TS coverage while minimizing redundancy.

5.5 Motif Grouping Using DBSCAN

Following the discovery of candidate motif groups within each chunk through flat region masking and MP computation (Section 5.4), the next stage involves aggregating these motifs at the global level. This step consolidates locally extracted motifs into coherent pattern groups that span chunk boundaries, preserving structural similarity throughout the entire TS. To achieve this, the DBSCAN algorithm is employed to cluster similar motifs based on their proximity in the subsequence space. This global grouping step is essential for eliminating redundancy across chunks and aligning motif instances that are structurally similar but discovered in separate segments.

Each candidate motif from the local set $\mathcal{P}_{\text{local}}$ is represented as a tuple $P_i = (W, [l_1, l_2, \dots])$, where W denotes the motif window size and $[l_1, l_2, \dots]$ represents the list of starting indices corresponding to structurally similar subsequences. For each motif group detected in a chunk, the starting indices—which are local to that chunk—are converted to global indices by adding the chunk's start offset. This ensures all motif locations are referenced to the original, full TS, resulting in a unified global representation $\mathcal{P}_{\text{global}}$ suitable for clustering across chunk boundaries.

We employ the DBSCAN algorithm due to its suitability for unsupervised learning scenarios, where neither the number of clusters nor their densities are known a priori. In particular, DBSCAN excels at identifying clusters of arbitrary shape and filtering out noise—features especially valuable in TSD, where motif frequency and distribution vary widely. It enables the discovery of recurring structures while discarding low-frequency, potentially spurious patterns.

In this implementation, DBSCAN is applied independently for each window size W. The clustering is performed in the embedding space using a EuDist metric consistent with the motif similarity criteria established during MP computation. We reuse the same distance threshold θ_{ed} as the DBSCAN `eps` parameter to maintain consistency. This procedure

effectively merges motif instances detected in different chunks into globally representative pattern groups, facilitating robust and interpretable motif discovery across the full TS, thus preparing the pattern set for the final constraint-based selection process outlined in Chapter 4.

Algorithm 4 Global Motif Grouping using DBSCAN

Require: $\mathcal{P}_{\text{local}}$: list of all motif groups across all chunks
Ensure: Global motif clusters $\mathcal{P}_{\text{global}}$
Initialize $\mathcal{P}_{\text{global}} \leftarrow \emptyset$
for each unique window size W in $\mathcal{P}_{\text{local}}$ **do**
    clusters $\leftarrow$ DBSCAN_Cluster$(t_s, \mathcal{P}_{\text{local}},\ \epsilon = \theta_{ed})$
    for each valid cluster c **do**
        Add $(W, \text{indices in } c)$ to $\mathcal{P}_{\text{global}}$
    end for
end for
return $\mathcal{P}_{\text{global}}$

Time Complexity. For each window size W, DBSCAN clusters N motif subsequences, requiring $O(N^2)$ time per W due to pairwise distance computations. In practice, N is moderate thanks to threshold pruning and chunking, ensuring that global motif grouping remains efficient within the overall pipeline.

The output of this stage is a globally clustered motif set $\mathcal{P}_{\text{global}}$, where each entry is a tuple $(W, [g_1, g_2, \dots])$ denoting the motif window size and the list of corresponding global start indices. These motifs serve as the input to the final selection phase, where constraint-based filtering is applied to maximize TS coverage under the constraints.

5.6 Implementation of Pattern Selection Methods

This section presents the practical implementation of the final phase of the pattern mining pipeline, where a subset of non-overlapping patterns is selected to maximize TS coverage under group and exclusion constraints. As discussed in Section 4.5, the choice between an exact constraint solver and scalable greedy algorithms depends on the number of data points in the input TS. For small to medium-sized TS, constraint solving offers optimal

results, while greedy heuristics are preferred for larger datasets due to computational efficiency.

5.6.1 Constraint Solver

Constraint Model Generation

To leverage the MiniZinc framework for solving the selection problem, it is first necessary to transform the input pattern data into a constraint-compatible format. This process involves converting the globally grouped candidate patterns, denoted as $\mathcal{P}_{\text{global}}$, into a `.dzn` data file that encodes the decision variables, objective function, and associated constraints.

Specifically, this transformation includes:

- Enumerating all binary decision variables corresponding to occurrences of candidate patterns.
- Defining *group constraints*, which ensure that all or none of the occurrences belonging to the same pattern group are selected.
- Encoding *exclusion constraints* to prevent the selection of overlapping pattern occurrences.
- Declaring a coverage expression to quantify the number of TS points covered by selected patterns.

Algorithm 5 presents the pseudo-code for converting candidate patterns into a MiniZinc-compatible constraint model. This includes the generation of binary decision variables, group-level selection constraints, exclusion rules, and the final optimization directive. The time complexity of the transformation is $\mathcal{O}(|C| \cdot k + n)$, where $|C|$ denotes the number of candidate patterns, k is the average number of occurrences per pattern, and n is the length of the TS. With optimized indexing for overlap detection, the exclusion constraint generation can be reduced to $\mathcal{O}(|C| \cdot k \cdot \log(|C| \cdot k))$, ensuring scalability for large datasets. Here, C corresponds to the global candidate pattern set $\mathcal{P}_{\text{global}}$ defined in Section 5.5.

Figure 5.1 provides a schematic representation of this process. At the top, a TS is shown, from which candidate patterns are extracted using the Stumpy library. Each horizontal line (A to E) corresponds to a group of similar pattern occurrences. Colored boxes represent group constraints, enforcing uniform selection across all occurrences of a group.

Red lines denote exclusion constraints, preventing the selection of temporally overlapping patterns. The entire structure is encoded into a constraint model, which is then passed to the MiniZinc solver for optimization.

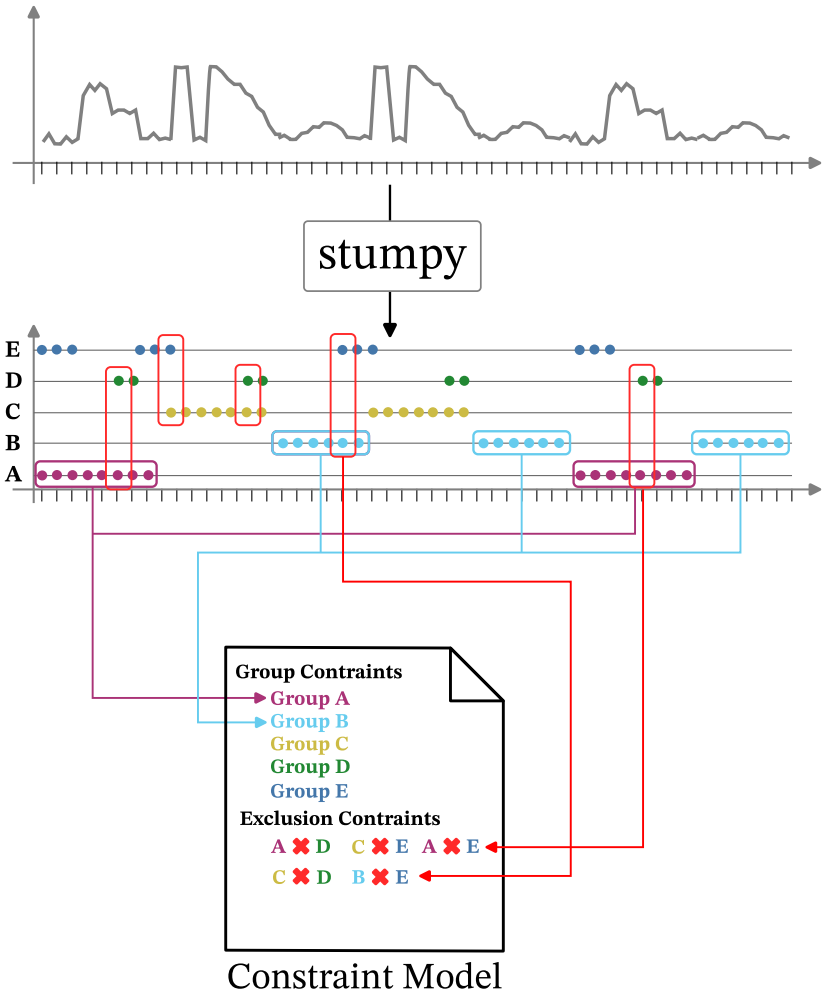


Figure 5.1: Converting candidate patterns into a constraint model. Each colored box or line represents either a group constraint (same pattern across time) or an exclusion constraint (overlapping occurrences). This structured representation is then encoded into a constraint model interpretable by the MiniZinc framework.

Algorithm 5 Pseudo-code for conversion into MiniZinc's constraint modelling language.

Require: Candidate patterns $C = \{(W_i, [l_{i1}, l_{i2}, \dots])\}$ from Algorithm 4
Ensure: Generates MiniZinc constraint model for pattern selection

$O \leftarrow$ empty list ▷ Occurrence-level decision variables
$Y \leftarrow$ empty list ▷ Pattern-level selection variables
for each pattern group $C_i \in C$ **do**
    $y_i \leftarrow$ new binary variable
    $Y \leftarrow Y \cup \{y_i\}$
    $O_i \leftarrow$ empty list
    for each occurrence $l_{ij} \in C_i$ **do**
        $O_{ij} \leftarrow$ new binary variable
        $O_i \leftarrow O_i \cup \{O_{ij}\}$
        write `var 0..1:` O_{ij} ;
        write `constraint` $O_{ij} = y_i$;
    end for
    $O \leftarrow O \cup O_i$
    write `var 0..1:` y_i ;
end for
write `var int: coverage;`
write `constraint coverage =` $\sum O_{ij}$;
write `constraint` $\sum y_i \leq m$; ▷ Max NoP
for each overlapping set $\mathcal{S}$ of occurrences **do**
    write `constraint` $\sum_{(i,j)\in\mathcal{S}} O_{ij} \leq 1$;
end for
write `solve maximize coverage;`

Two key constraint types are encoded into the model:

- **Group Constraints** (Equation 4.2): Ensure that if one occurrence of a pattern group is selected, all occurrences must be selected. This avoids partial selection of structurally related patterns.
- **Exclusion Constraints** (Equation 4.3): Prevent overlapping occurrences from being selected together. This is critical for avoiding temporal conflicts in the output.

Together, these constraints define a feasible solution space from which the solver selects the subset of patterns that optimizes coverage.

Constraint Solver - MiniZinc

Once the constraint model is generated, it is solved using the MiniZinc framework [19]. MiniZinc acts as a modeling layer that supports multiple back-end solvers. In this thesis, we utilize the Google OR-Tools solver, which was empirically found to outperform Gecode in terms of runtime efficiency (see Section 8.3). This choice aligns with the computational constraints and scale of real-world datasets introduced in Chapter 3, where exact solvers are viable for moderate-length TS with manageable candidate pattern sets. The construction of the model itself is informed by the constraint formulation detailed in Section 4.5.1.

The objective of the solver is to identify a valid subset of patterns that maximizes the total coverage of the TS, defined as the number of data points covered by the selected patterns. The optimization is subject to an upper bound m on the number of selected patterns, ensuring that the solution remains interpretable and compact.

The optimization problem is formally defined as:

$$\begin{aligned} \max \quad & \sum_{i=1}^{N_p} \sum_{j=1}^{n_i} O_{ij} \cdot \text{length}(C_i) \\ \text{s.t.} \quad & O_{ij} \in \{0,1\}, \quad y_i \in \{0,1\}, \\ & O_{ij} = y_i \quad \forall\, i,j, \\ & \sum_{i=1}^{N_p} y_i \leq m, \\ & \sum_{(i,j)\in\mathcal{S}} O_{ij} \leq 1 \quad \forall\, \mathcal{S} \text{ (overlapping set)} \end{aligned} \tag{5.2}$$

This formulation corresponds to Problem 3.1 from Chapter 3, enforcing group and exclusion constraints while optimizing coverage under a cardinality bound on the NoP m.

Here, O_{ij} is a binary variable indicating whether the j-th occurrence of pattern C_i is selected, and y_i is a binary indicator of whether the entire pattern C_i is included in the final output. The constraint $O_{ij} = y_i$ enforces the group constraint behavior, while the summation constraint over $\mathcal{S}$ ensures mutual exclusivity for overlapping patterns.

In some scenarios, a minimum target coverage may be specified instead of treating coverage as the optimization objective. Regardless of the objective formulation, the solver must respect the structural constraints imposed by the model.

While the solver provides optimal solutions for fixed upper bounds on the NoP, it does not simultaneously optimize for maximum coverage and minimum pattern count. This multi-objective formulation would require a well-defined gain function balancing both, but was intentionally avoided in this thesis to preserve model interpretability and consistency with Problem 3.1 as defined in Chapter 3. In this thesis, the focus is on maximizing coverage while bounding the NoP. Alternative gain functions, such as maximizing coverage per pattern (i.e., coverage divided by NoP), are possible but were not used due to reduced interpretability of results.

5.6.2 Greedy Heuristic Approaches

For large-scale TSs involving millions of data points, exact constraint-solving methods become computationally prohibitive. To address this scalability challenge, a suite of greedy heuristic strategies is employed to approximate high-quality pattern selections while adhering to group and exclusion constraints. These heuristics offer a trade-off between computational efficiency and solution quality, making them well-suited for practical deployment in real-world applications.

The following algorithms were implemented and evaluated in this study:

Naive Greedy Selection

Naive Greedy Selection provides a baseline: at each iteration, it selects the available candidate pattern that yields the largest increase in coverage, while avoiding overlaps. Although simple and intuitive, the method requires a full scan of the candidate set in each iteration, which leads to inefficiencies, especially for large datasets. Despite its computational limitations, this strategy is valuable due to its transparency, ease of implementation, and effectiveness in small-scale or illustrative scenarios.

Greedy Selection with Interval Tree Optimization

This algorithm improves upon the Naive Greedy approach by employing an interval tree data structure to efficiently handle overlap detection. Instead of brute-force comparisons, it uses spatial indexing to check for conflicts in logarithmic time. This optimization significantly reduces redundant checks and enables the algorithm to scale effectively with larger

datasets. The method preserves the core greedy principle of maximizing marginal coverage while introducing a structurally efficient mechanism for conflict resolution, making it practical for high-volume TS mining tasks [18, 4].

Simulated Annealing (SA) with Balanced Cooling

SA is a metaheuristic that explores the solution space beyond local optima. A balanced SA configuration is used, with the cooling schedule calibrated for a strong trade-off between exploration and convergence. The algorithm begins with a greedy solution and iteratively perturbs the pattern set, accepting both improving and, probabilistically, worsening moves to escape local maxima. Over time, the probability of accepting suboptimal changes decreases, guiding the solution toward a stable, high-quality state. The balanced variant used here ($T_0 = 1.0$, $\alpha = 0.90$, 1500 iterations) demonstrates strong empirical performance with moderate runtime, making it a robust addition to the heuristic suite [23, 13].

5.7 Summary

Figure 5.2 illustrates the implemented pattern mining pipeline. The process begins by partitioning the input TS into overlapping chunks to enable efficient, parallelizable analysis. Within each chunk, flat region masking is applied to exclude extended low-variance segments. Subsequently, MP computations are performed independently for each chunk across a predefined list of window sizes, using GPU-accelerated Stumpy. Local motif grouping is first achieved by post-processing the MP results to remove overlapping subsequences for each reference index. This is followed by DBSCAN clustering, which ensures that motifs within each group exhibit mutual structural similarity.

The locally discovered motifs from all chunks are then aligned in the global coordinate space and subjected to a second DBSCAN clustering stage. This global clustering step consolidates motifs that are structurally similar across chunks, thereby eliminating redundancy and improving the interpretability of the final pattern set. Each resulting cluster is treated as a candidate pattern group, and the aggregated global set of candidate patterns, denoted by $\mathcal{P}_{\text{global}}$, serves as input to a constraint-based model that explicitly enforces both group and exclusion constraints.

The final stage of the pipeline selects a concise set of non-overlapping patterns that collectively maximize TS coverage. This selection task is formulated as a constraint optimization problem, solved either through the MiniZinc solver for moderate-sized datasets or through scalable greedy heuristics for larger datasets.

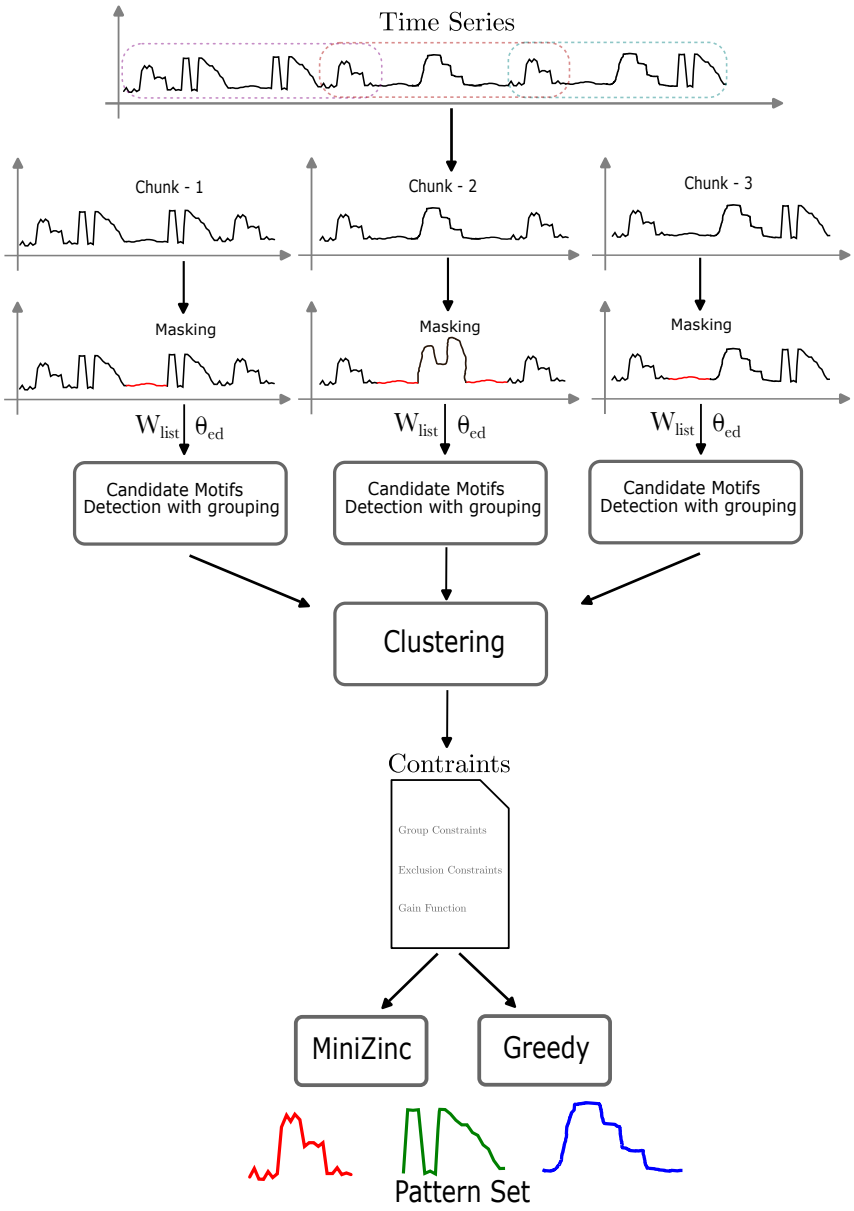


Figure 5.2: Overview of the full pattern mining pipeline—input time series (TS) is chunked and masked, then processed using matrix profile (MP) computations to extract motifs. Masked regions (in red) are excluded. Motifs are globally clustered with DBSCAN, then filtered and selected using either constraint-based (MiniZinc) or heuristic (greedy) methods.

The resulting output is an interpretable, non-overlapping set of patterns that effectively summarizes the TS while adhering to the defined constraints. This completes the pipeline described in Chapter 4 and 5 and sets the stage for empirical validation in Chapter 6.

Chapter 6

Experiments and Evaluation

This chapter presents an evaluation of the proposed TS pattern mining algorithm, with a focus on its computational performance, scalability, and robustness under varying conditions. The evaluation specifically examines the effects of the k parameter, the presence of long flat regions in TSDs, and the influence of other key factors—such as window size and TS length—on GPU ET and efficiency during motif discovery.

Pattern extraction in univariate TS plays a vital role in capturing representative system behaviors from sequential data. However, practical deployment of such algorithms must contend with challenges related to computational cost, data irregularities, and scalability. In this context, we systematically examine how changes in the number of nearest neighbors (k) affect both ET and the quality of the extracted motifs. To support this evaluation, we conducted our experiments using the Stumpy GPU API, which provides an efficient implementation of MP computation.

Additionally, we explore the effect of long flat regions—prolonged segments with constant or near-constant values—that are commonly encountered in real-world TSDs. While frequent, these regions typically lack informative structural content and may significantly inflate computation time without contributing to meaningful motif discovery. Consistent with our problem formulation, such regions are treated as uninformative and are excluded from the core analysis. We present and assess optimization strategies, such as flat-region masking and threshold-based optimization, to mitigate their impact on computational performance.

We also examine the EuDist threshold θ_{ed}, which directly controls the strictness of motif matching. By pruning candidate subsequences whose distances exceed θ_{ed}, the algorithm reduces computation while maintaining motif quality. This section investigates how θ_{ed}

interacts with data normalization, window size, and dataset characteristics, and proposes practical strategies for selecting its value to balance efficiency and inclusiveness.

The experiments further analyze the influence of varying window sizes and input TS lengths on motif discovery and ET. These analyses help clarify the relationship between algorithmic parameters and runtime behavior, contributing to a deeper understanding of how best to configure the system for different types of TSD.

Overall, this chapter aims to establish practical guidelines for parameter selection, identify performance bottlenecks, and validate the effectiveness of the proposed algorithm in improving the efficiency and reliability of large-scale TS pattern mining. All experiments and evaluations in this chapter are performed on 24-hour TSD rather than longer durations, as this provides better resolution and clarity for analysis.

6.1 Datasets Used for Evaluation

All experiments presented in this chapter were conducted using a consistent set of five univariate TSs: one synthetic and four real-world datasets. The synthetic dataset was deliberately constructed to include recurring motifs with controlled noise, providing a baseline for validating motif discovery performance under idealized conditions. The real-world datasets were derived from lab-recorded power consumption traces, chosen to reflect a variety of signal characteristics including smooth periodic behavior and bursty, noisy fluctuations.

Prior to analysis, each dataset was normalized and preprocessed to ensure consistency in scale and sampling rate. An overview of the five TS is provided in Figures 6.1 to 6.5. These datasets were used uniformly across all experiments reported in Sections 6.2, 6.3, 6.4, 6.5, and 6.6. All TSs were sampled at a fixed rate of 1 Hz (i.e., one observation per second), which was preserved throughout all preprocessing and analysis steps in this chapter.

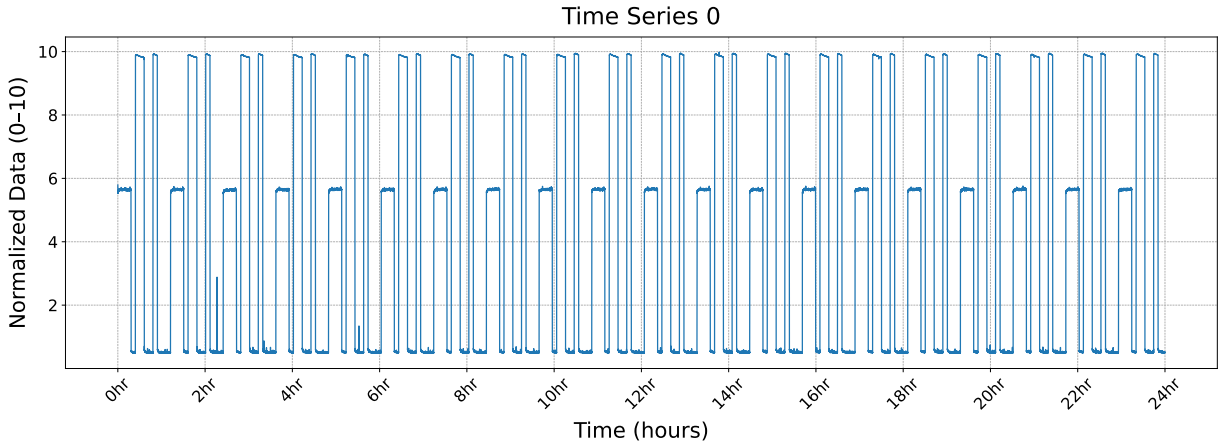


Figure 6.1: Normalized synthetic time series TS-0 containing periodic patterns with controlled noise, used for baseline validation. Sampling rate is 1 Hz (1 sample/second).

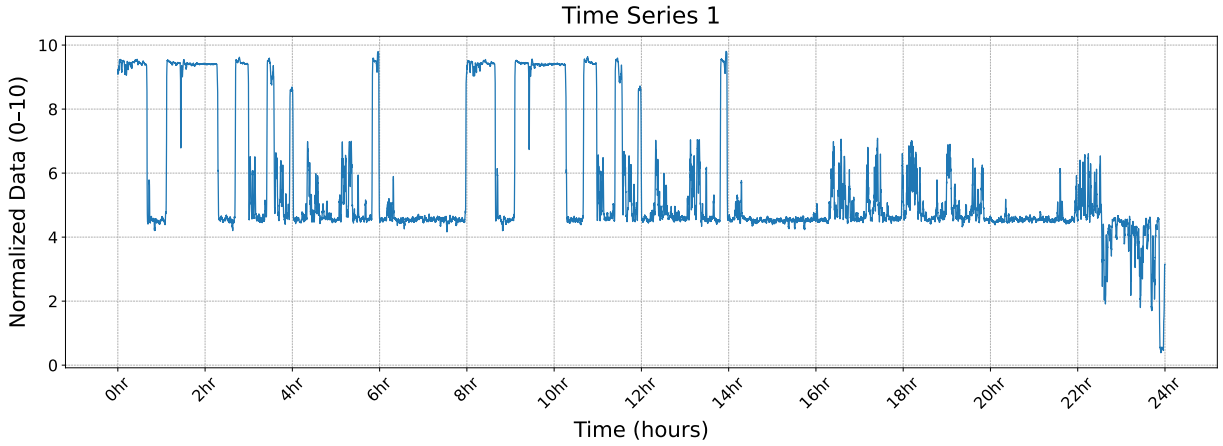


Figure 6.2: Normalized real-world time series TS-1 representing power consumption data collected over a 24-hour period. Sampling rate is 1 Hz.

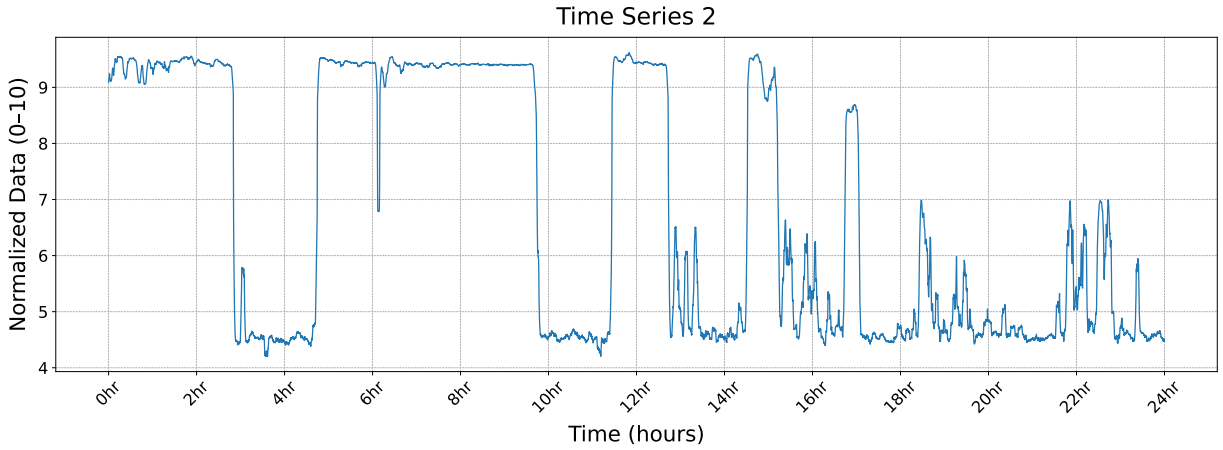


Figure 6.3: Normalized real-world time series TS-2 generated by temporally stretching the first 6 hours of Figure 6.2. Used to evaluate the robustness of the algorithm. Sampling rate is 1 Hz.

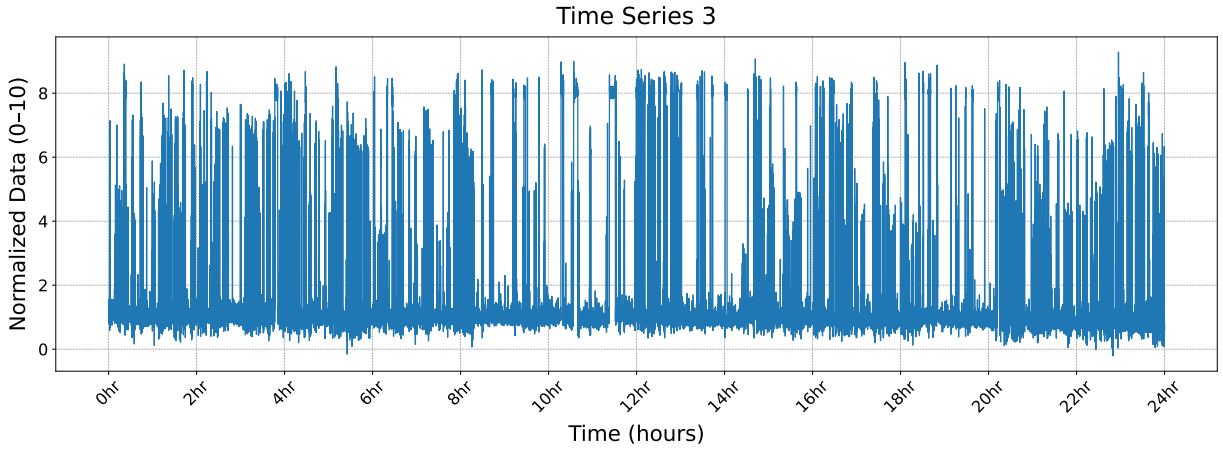


Figure 6.4: Normalized real-world time series TS-3 characterized by frequent fluctuations and high-amplitude spikes. Used to test resilience under noisy and bursty conditions. Sampling rate: 1 Hz.

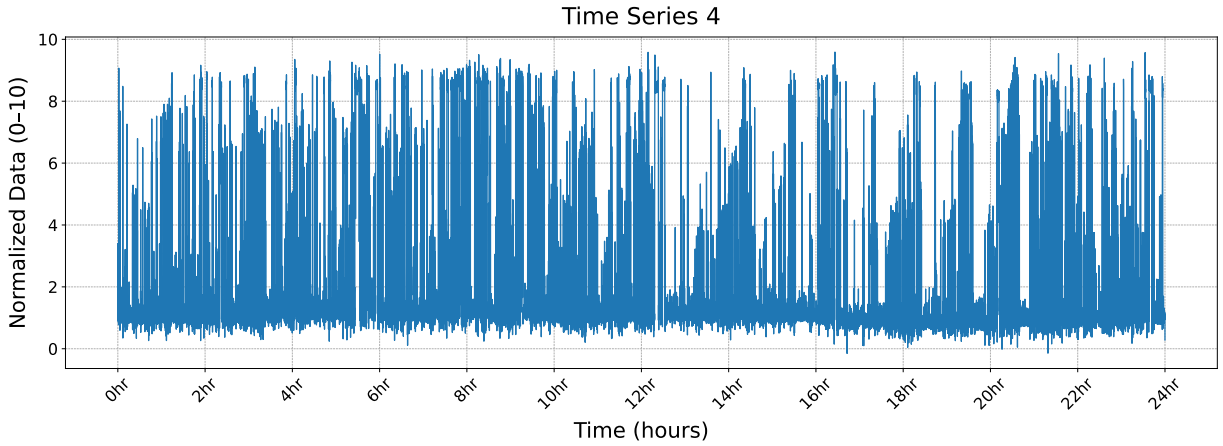


Figure 6.5: Normalized real-world time series TS-4 exhibiting bursty behavior and variability similar to Figure 6.4. Sampling rate: 1 Hz.

6.2 Impact of the k Parameter in Matrix Profile (MP) Computation

This section examines the influence of the parameter k on the computation performance of the Stumpy API, with a focus on MP-based motif discovery. The analysis covers the impact of varying k and evaluates optimization using a modified version of the library. All experiments presented here were conducted using the Stumpy GPU API, operating on a TS Figures 6.1 to 6.5. These datasets include both synthetic data for controlled evaluation and real-world datasets, each representing one full day of power consumption data sampled at 1 Hz, and serve as a standardized input for evaluating the algorithm's behavior across varying values of k.

6.2.1 Computational Challenges with Increasing k

In MP-based TSA, the parameter k determines the number of top-k nearest neighbors (closest subsequences) motifs—ranked by EuDist—that are returned for each reference subsequence. Selecting an appropriate k is inherently non-trivial, particularly in unsupervised settings where the optimal number of motifs is unknown a priori. Setting k too low risks missing relevant motifs, while excessively high values introduce unnecessary computational burden. While increasing k can improve motif detection by identifying a richer

set of similar patterns, it also substantially raises computational costs. Each increase in k requires additional pairwise distance computations and memory allocations, leading to longer ETs and greater resource consumption. Empirically, we observe that ET increases in a non-linear fashion with respect to k especially for longer TS. The elevated cost is attributed to the growing number of distance comparisons and memory operations required to maintain the k-nearest structure.

An additional challenge arises when the analysis requires only highly similar motifs, i.e., those within a specific EuDist threshold. The original Stumpy API does not natively support such distance-based filtering, necessitating post-processing to discard low-quality matches. Furthermore, the absence of an internal stopping criterion for automatically determining an optimal value of k poses a challenge. While increasing k may initially improve motif detection, the marginal benefit of including additional nearest neighbors decreases rapidly beyond a certain point, resulting in unnecessary computational overhead.

In summary, increasing the k parameter in the Stumpy API leads to higher computation time and memory usage, as the algorithm must identify and store more nearest neighbors for each subsequence. While larger k may initially enhance motif detection, the benefits plateau after a point, resulting in greater resource consumption without meaningful improvement. This motivates the need for an improved approach, where only motifs below a specific EuDist threshold are of interest; setting k excessively high proves both inefficient and unnecessary.

6.2.2 Optimization Using Threshold-Based Motif Discovery

To overcome the limitations associated with increasing k, we introduced an enhancement to the Stumpy to support an additional variable EuDist θ_{ed} threshold. This modification enables the algorithm to skip comparisons for subsequences whose EuDist exceeds a specified threshold, thereby reducing unnecessary computation.

For all experiments, the EuDist threshold was fixed at $\theta_{ed} = 0.25$, determined through empirical testing to balance motif quality across the evaluated TSs. As discussed in Section 8.1.3, the optimal threshold θ_{ed} may vary depending on dataset characteristics and specific application requirements.

Although a value for k must still be specified, we use the consistent window-aware rule from Section 8.1.6, setting $k = k_{opt} = \max\big(k_{\min}, \lceil n/W \rceil\big)$. This balances coverage with memory efficiency and avoids GPU out-of-memory errors, while ensuring all meaningful motif candidates are considered. With the inclusion of the threshold, the search space is effectively pruned, resulting in significant improvements in ET. This approach maintains

the quality of discovered motifs by excluding irrelevant subsequences early in the process and is especially beneficial when using high values of k (as defined above), where the risk of redundant or noisy matches is greater.

To validate the effectiveness of this optimization, we compared GPU ETs for the before (No Optimization) introducing the EuDist threshold variable θ_{ed} and after introducing (Threshold Optimization) theshold θ_{ed} in Stumpy across a wide range of k values. Figures 6.6 to 6.10 present the results for five TS, each evaluated with a fixed window size of $W = 100$ and k ranging from 1 to 1000. In every case, the Threshold optimized implementation significantly outperforms the original in terms of runtime.

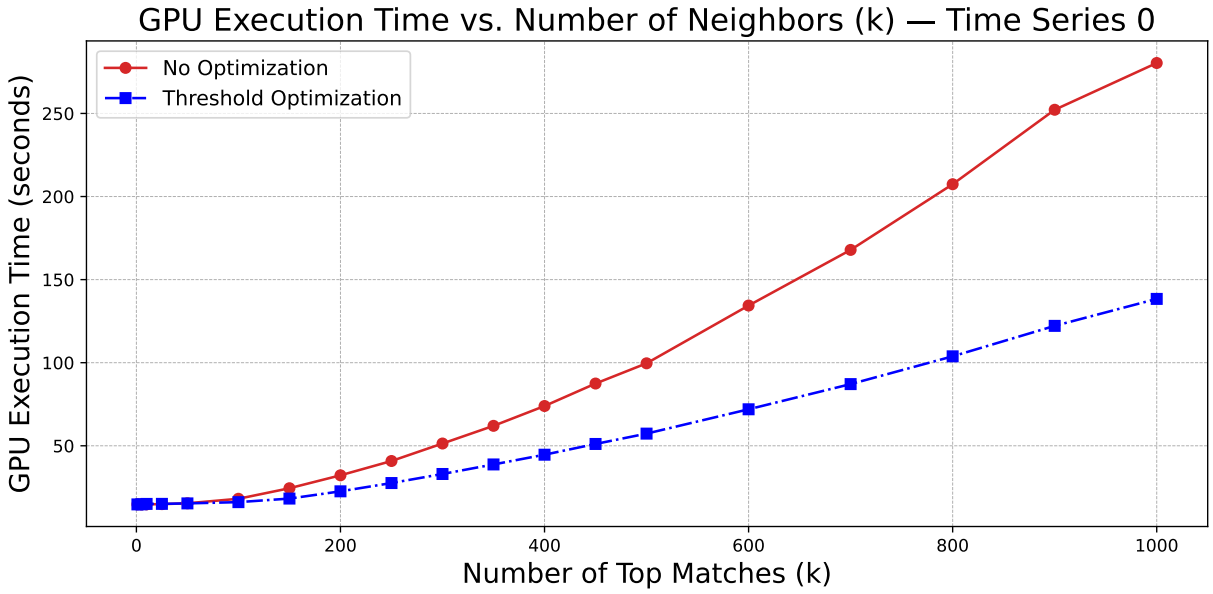


Figure 6.6: Comparison of GPU execution time (ET) before and after threshold-based optimization for Figure 6.1, with fixed $W = 100$ and k ranging from 1 to 1000.

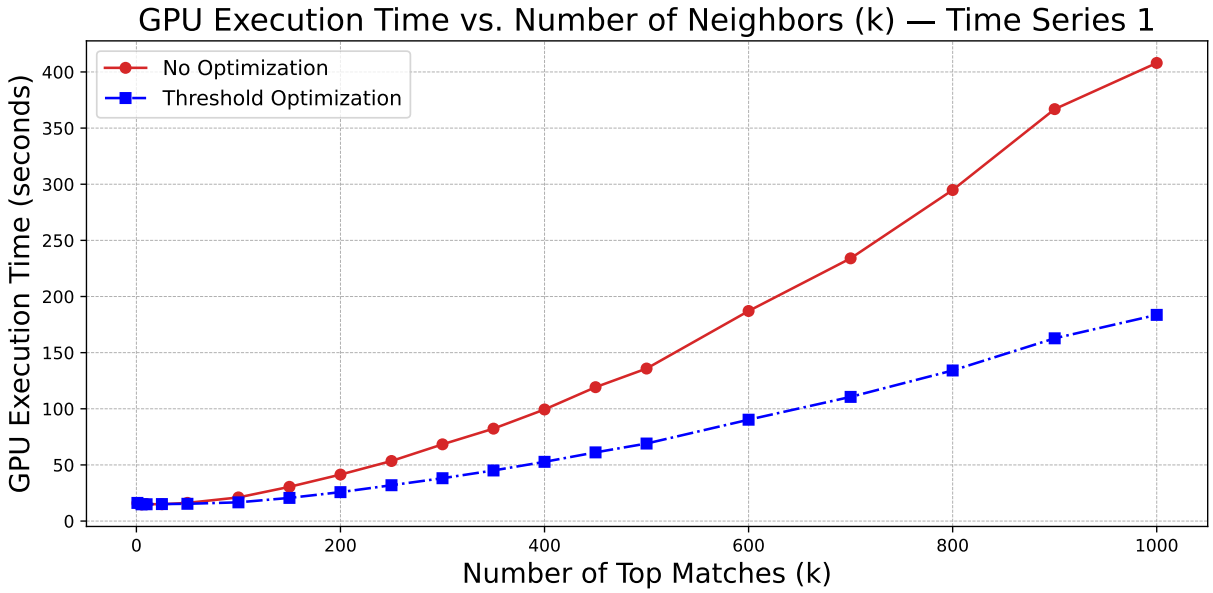


Figure 6.7: Comparison of GPU execution time (ET) before and after threshold-based optimization for Figure 6.2, with fixed $W = 100$ and k ranging from 1 to 1000.

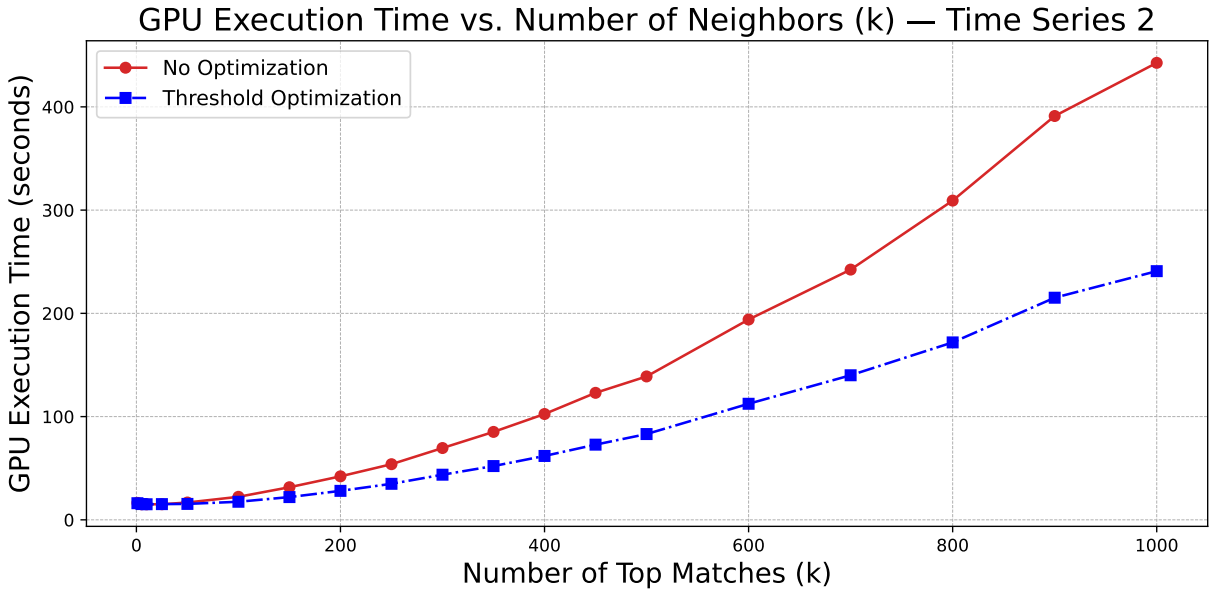


Figure 6.8: Comparison of GPU execution time (ET) before and after threshold-based optimization for Figure 6.3, with fixed $W = 100$ and k ranging from 1 to 1000.

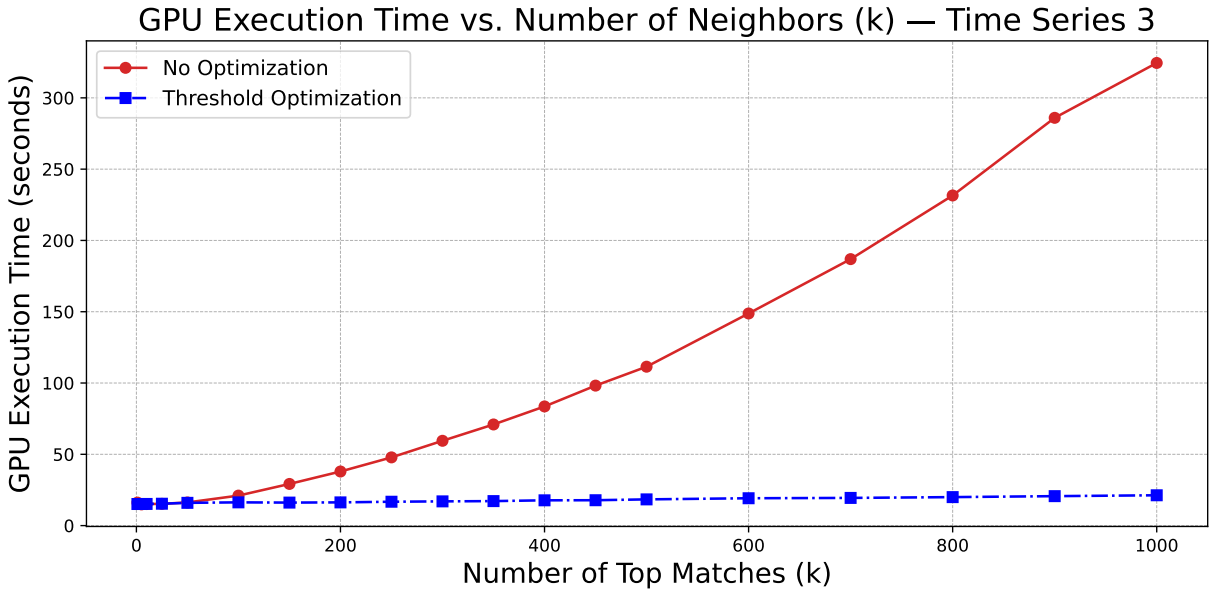


Figure 6.9: Comparison of GPU execution time (ET) before and after threshold-based optimization for Figure 6.4, with fixed $W = 100$ and k ranging from 1 to 1000.

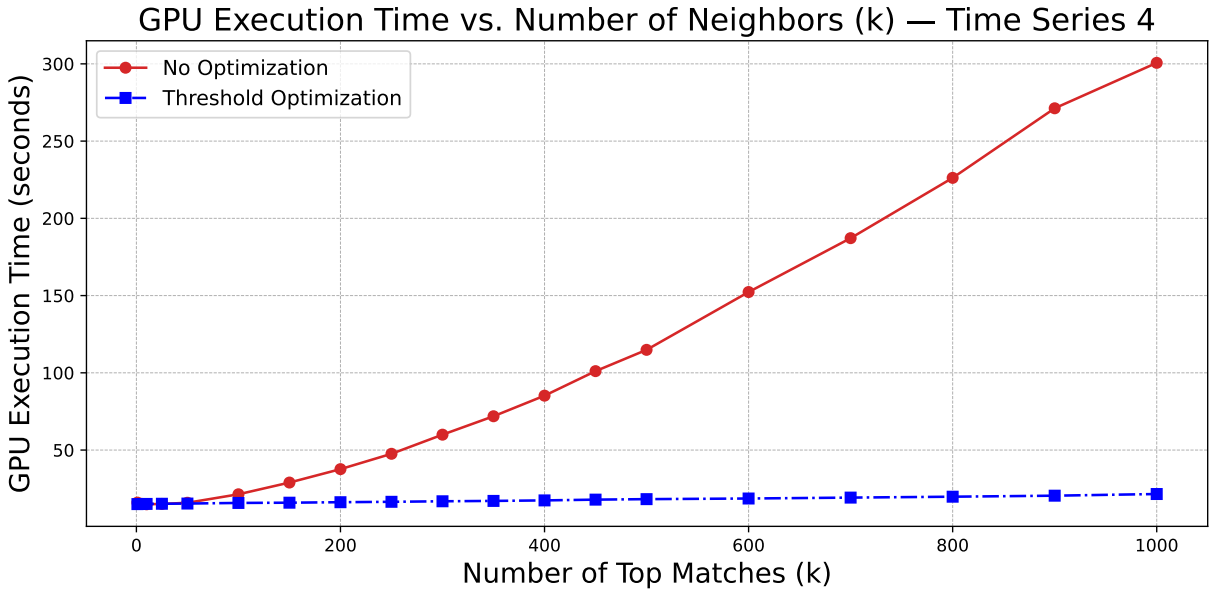


Figure 6.10: Comparison of GPU execution time (ET) before and after threshold-based optimization for Figure 6.5, with fixed $W = 100$ and k ranging from 1 to 1000.

Figure 6.6 shows that, as the number of nearest neighbors k increases, the ET for motif discovery grows sharply in the unoptimized pipeline. In contrast, applying threshold-based optimization keeps ET nearly linear and substantially lower, even for large k. Similar trends are observed for all real-world datasets (Figures 6.7 to 6.10), confirming that the optimization generalizes well.

Taken together, these results demonstrate that threshold-based optimization dramatically reduces computational time without degrading motif quality. This result justifies our use of threshold-based pruning in the pipeline described in Chapter 5 and all subsequent experiments.

6.3 Effect of Flat Regions on Computational Performance

Flat regions—long segments of near-constant value—are a frequent occurrence in real-world TS. If not handled appropriately, these regions can dominate computational resources and obscure the discovery of meaningful motifs. Evaluating the impact of flat-region masking is thus crucial for both efficiency and result quality in large-scale TSA.

Following the analysis of how the parameter k affects MP computation in Section 6.2, this section explores the computational impact of long flat regions in TSD. Although flat regions may appear structurally simple, they can significantly affect the runtime performance of both CPU and GPU-based MP calculations. We first outline the limitations introduced by such regions and then present a mitigation strategy that effectively reduces their computational overhead.

6.3.1 Computational Overhead Introduced by Flat Regions

In this thesis, our primary objective is to identify meaningful recurring patterns that represent underlying system behavior. However, many real-world TSs contain long flat regions—segments where the values have a low variation or are nearly constant over an extended period. These flat regions are independent of the window size and may span hundreds or thousands of data points.

While MP algorithms extract fixed-length subsequences from the TS, long flat regions naturally give rise to many such subsequences that are flat. These subsequences, referred to as flat patterns or flat motifs, are derived from overlapping windows entirely within a

flat region. Because they contain no meaningful variation, they rarely offer useful insight into the system's dynamics or state transitions.

Importantly, if flat regions are not explicitly excluded from analysis, Stumpy will process all resulting flat subsequences as valid inputs, even though they are later discarded in post-processing. This unnecessary computation inflates the MP size and leads to increased ET, especially for longer TS or when processing multiple window sizes. Although one could theoretically use flat regions to maximize coverage, doing so would undermine the interpretability of the results. For this reason, flat motifs are excluded from this thesis, and flat regions are treated as noise to be efficiently skipped.

6.3.2 Optimization via Preprocessing for Flat Region Mitigation

To avoid the overhead caused by flat regions, we introduce a lightweight preprocessing strategy that masks them prior to motif discovery. Specifically, we scan the input TS to identify long, flat regions—defined as segments where the values have a low variation, and replace their values with `NaN`. Since the Stumpy API is designed to ignore any subsequences containing `NaN`, this masking operation prevents flat regions from contributing to the MP computation.

To preserve potentially meaningful patterns near the start or end of each flat region, we skip masking for the first and last segments of length equal to the current window size. This ensures that any valid transitional motifs crossing into or out of flat regions are not inadvertently excluded.

This masking strategy significantly reduces the number of redundant comparisons and improves performance without altering the final motif set. As demonstrated Figures 6.11 to 6.15, we compare GPU ETs across four configurations derived from two key optimizations: threshold-based optimization and flat-region masking. Specifically, the configurations include: (1) No optimization (baseline), (2) Threshold-based optimization only, (3) Flat-region masking only, and (4) both optimizations applied together.Results show that flat region masking contributes to a noticeable reduction in ET and, when combined with threshold filtering, enables efficient motif discovery even on large datasets with high values of k.

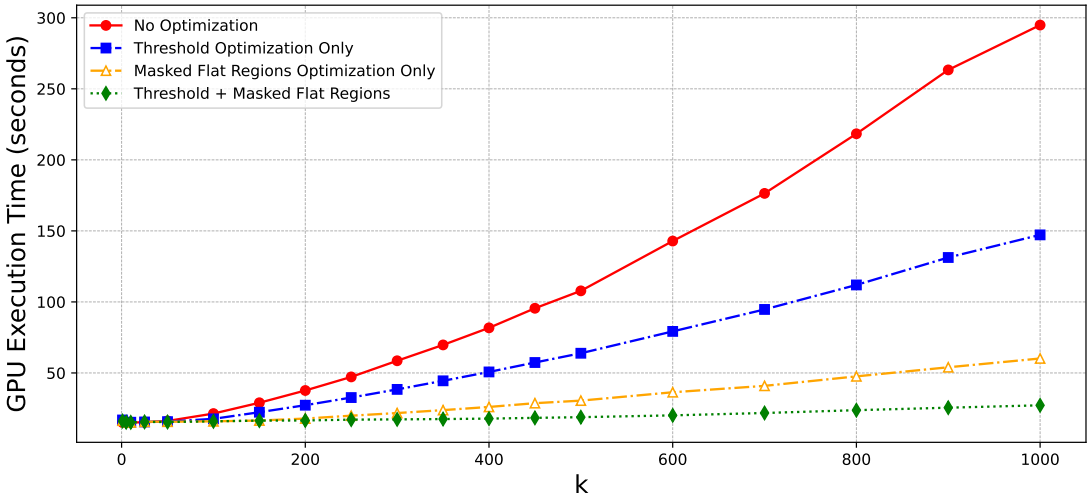


Figure 6.11: GPU execution time (ET) as a function of k (fixed $W = 100$) for Figure 6.1 under: (1) No optimization, (2) Threshold only, (3) Flat-region masking only, (4) Both optimizations. Combined gives best performance.

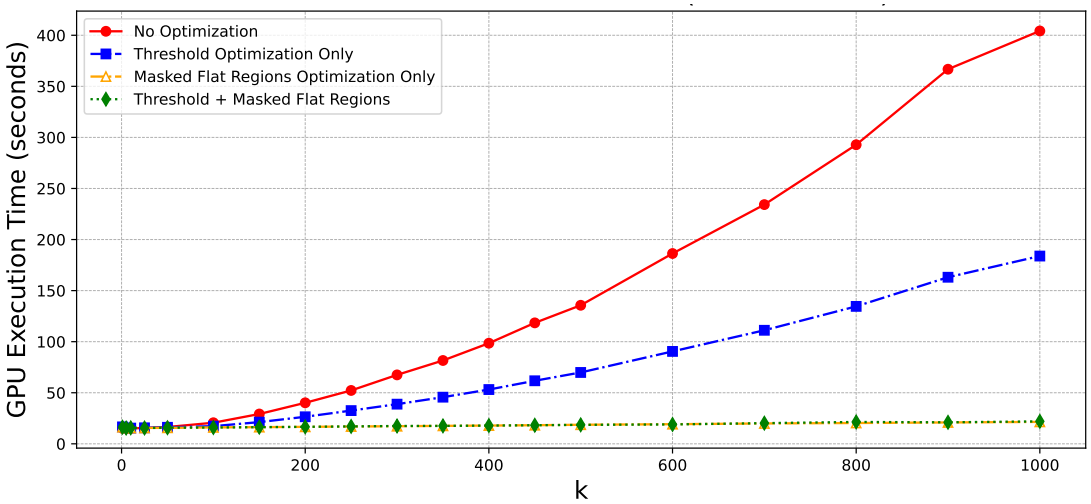


Figure 6.12: GPU execution time (ET) as a function of k (fixed $W = 100$) for Figure 6.2 under: (1) No optimization, (2) Threshold only, (3) Flat-region masking only, (4) Both optimizations. Combined gives best performance.

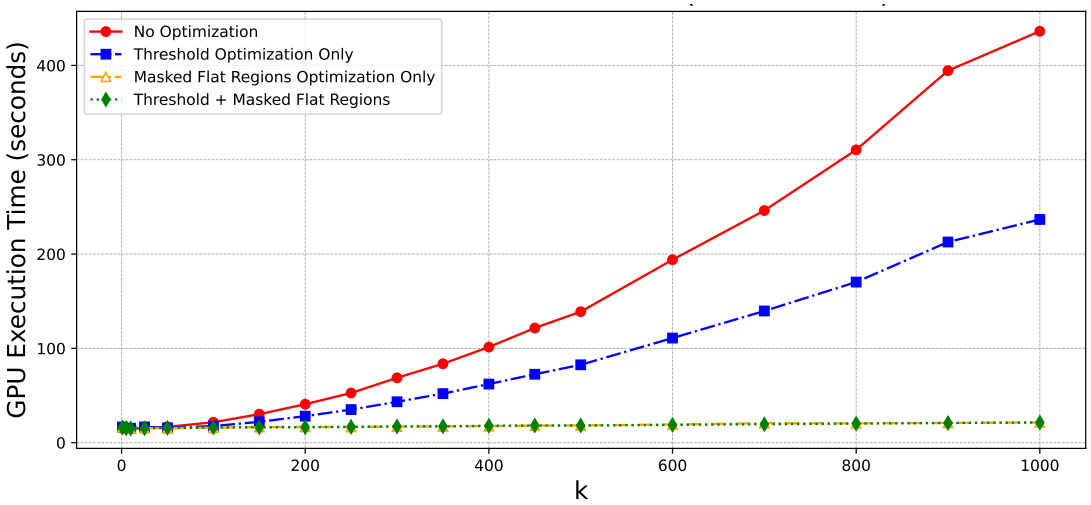


Figure 6.13: GPU execution time (ET) as a function of k (fixed $W = 100$) for Figure 6.3 under: (1) No optimization, (2) Threshold only, (3) Flat-region masking only, (4) Both optimizations. Combined gives best performance.

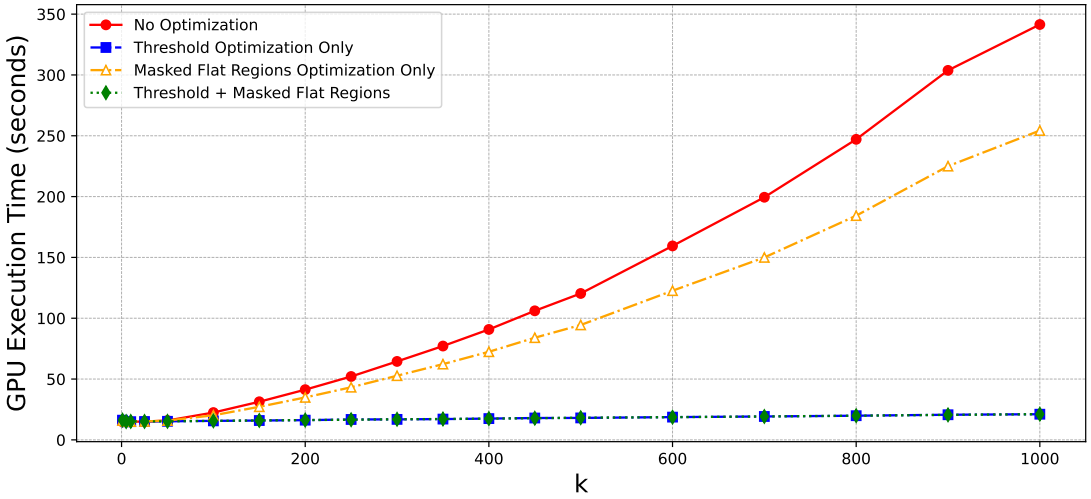


Figure 6.14: GPU execution time (ET) as a function of k (fixed $W = 100$) for Figure 6.4 under: (1) No optimization, (2) Threshold only, (3) Flat-region masking only, (4) Both optimizations. Combined gives best performance.

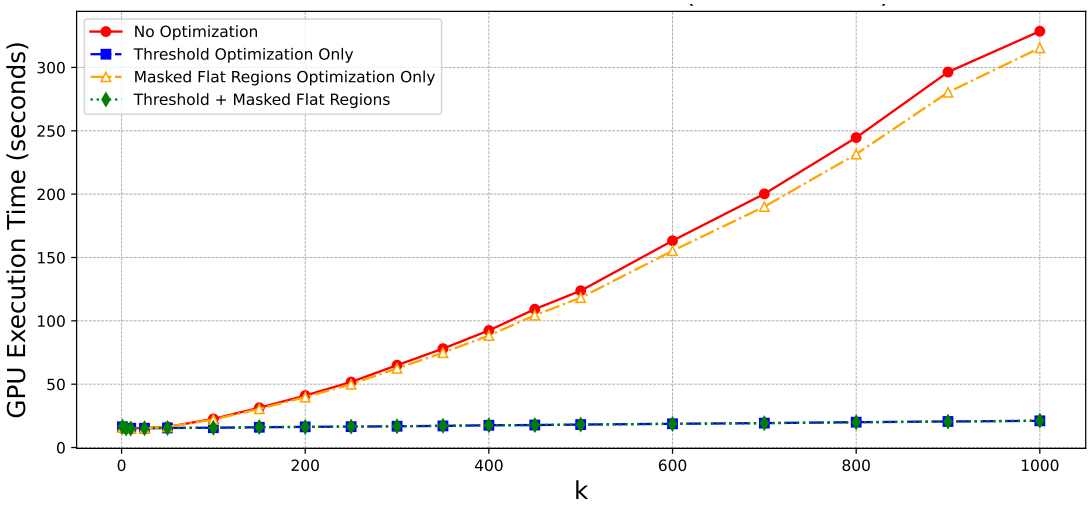


Figure 6.15: GPU execution time (ET) as a function of k (fixed $W = 100$) for Figure 6.5 under: (1) No optimization, (2) Threshold only, (3) Flat-region masking only, (4) Both optimizations. Combined gives best performance.

6.4 Effect of Euclidean Distance Threshold on Computational Performance

The EuDist threshold θ_{ed} defines the maximum allowable distance between a reference subsequence and a candidate match for them to be considered the same motif. Since EuDist depends on both data scale and subsequence length, θ_{ed} is inherently a *per-point* measure that scales with the window size W.

For data normalized to $[0, 10]$, the theoretical upper bound of θ_{ed} corresponds to the maximum possible pointwise difference of 10, with 0 as the minimum. This bound represents a case in which every point in two subsequences differs maximally. In practice, such high values admit entirely dissimilar patterns and are unsuitable for motif discovery. For meaningful structural similarity, the per-point threshold should typically be well below 1, ensuring that patterns remain consistent and interpretable. The optimal value is dataset- and application-specific, and should be adjusted according to W, as longer windows naturally yield higher accumulated distances. A practical strategy for selecting θ_{ed} is to begin with the smallest feasible value and gradually increase it in small increments, evaluating the resulting motifs at each step. This ensures that the chosen threshold preserves structural similarity while avoiding overly loose matches. In our experiments, an empirically

determined value of $\theta_{ed} = 0.25$ provided consistent, high-quality motifs across datasets, whereas starting with $\theta_{ed} \geq 0.8$ tended to produce highly dissimilar patterns, reducing the interpretability and structural coherence of the discovered motifs.

To evaluate its effect, GPU ET was measured across a range of θ_{ed} values for five TSs, with all other parameters fixed. Experiments were performed using the optimized Stumpy implementation with flat-region masking and threshold-based pruning enabled, and $W = 100$. As shown in Figure 6.16, lower θ_{ed} values significantly reduce runtime by pruning the search space, whereas higher values converge to the baseline with no threshold-based pruning. .

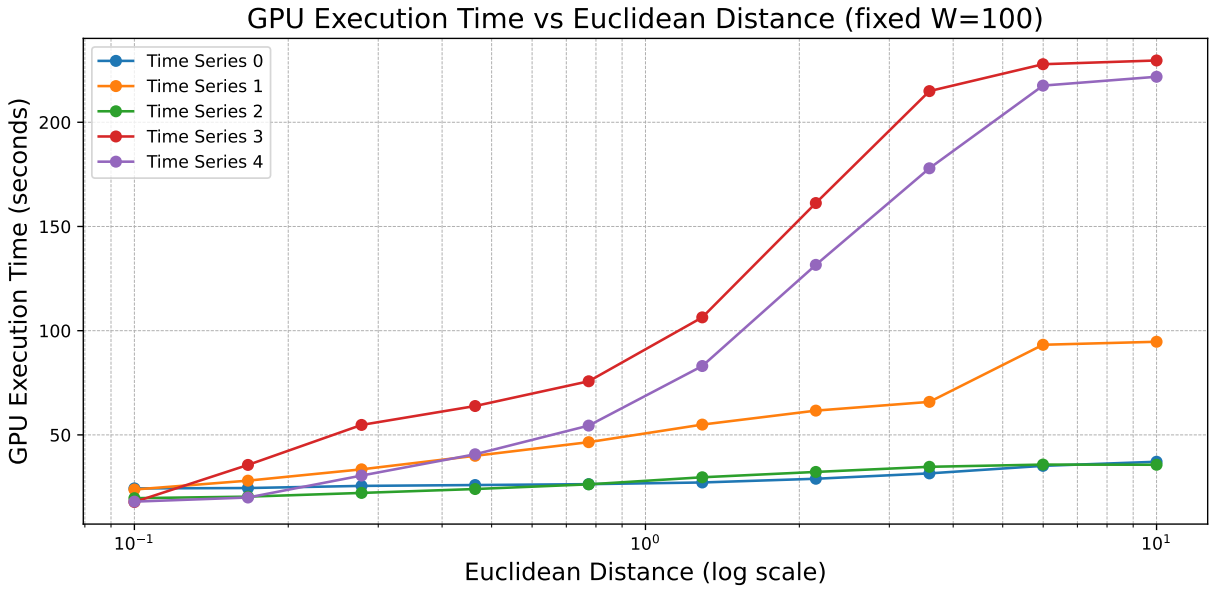


Figure 6.16: GPU execution time (ET) as a function of the Euclidean distance (EuDist) threshold (θ_{ed}) for five TSs, using the optimized Stumpy implementation with flat-region masking and threshold-based pruning. Window size fixed at $W = 100$. Lower θ_{ed} values reduce runtime by pruning the search space, while higher values converge toward the original stumpy implementation.

6.5 Effect of Window Size on Computational Performance

The choice of window size (W) plays a critical role in motif discovery, directly affecting both the types of patterns that can be detected and the computational cost of the analysis.

All experiments in this section are conducted using the optimized version of the Stumpy, which incorporates both threshold-based optimization and flat-region masking in the Stumpy GPU API. In this experiment, W was varied from 10 to $\frac{n}{2}$, where n is the length of the TS. Ten logarithmically spaced values were generated using `np.logspace`, producing

$$W \in \{10, 25, 64, 163, 413, 1046, 2653, 6724, 17043, 43200\}.$$

For each W, we fix the threshold parameter to 0.25, and the number of nearest neighbors k was adaptively set as $k = k_{opt} = \max\big(k_{\min}, \lceil n/W \rceil\big)$, ensuring a balanced trade-off between motif quality and computational efficiency

Figure 6.17 plots GPU ET (Y-axis, log scale) against window size W (X-axis, log scale) for TSs (Figure 6.1 to 6.5). Each curve represents one dataset, showing how runtime changes as W increases. The decreasing trend is consistent across datasets, with sharper drops observed for smaller W in more irregular TS-3 and TS-4.

The results reveal a clear trend: as W increases, ET decreases steadily. This is expected, as longer windows reduce the number of subsequences ($n - W + 1$) and therefore the total number of distance computations. At large W values, runtimes converge across datasets due to the significant reduction in candidate motifs. At smaller W, TSs with higher variability and fewer flat regions (e.g., glsts-3 and TS-4) exhibit substantially higher ET, driven by the greater number of structurally distinct subsequences requiring comparison.

Overall, these findings confirm that—under the proposed optimizations—GPU runtime decreases monotonically with increasing W.

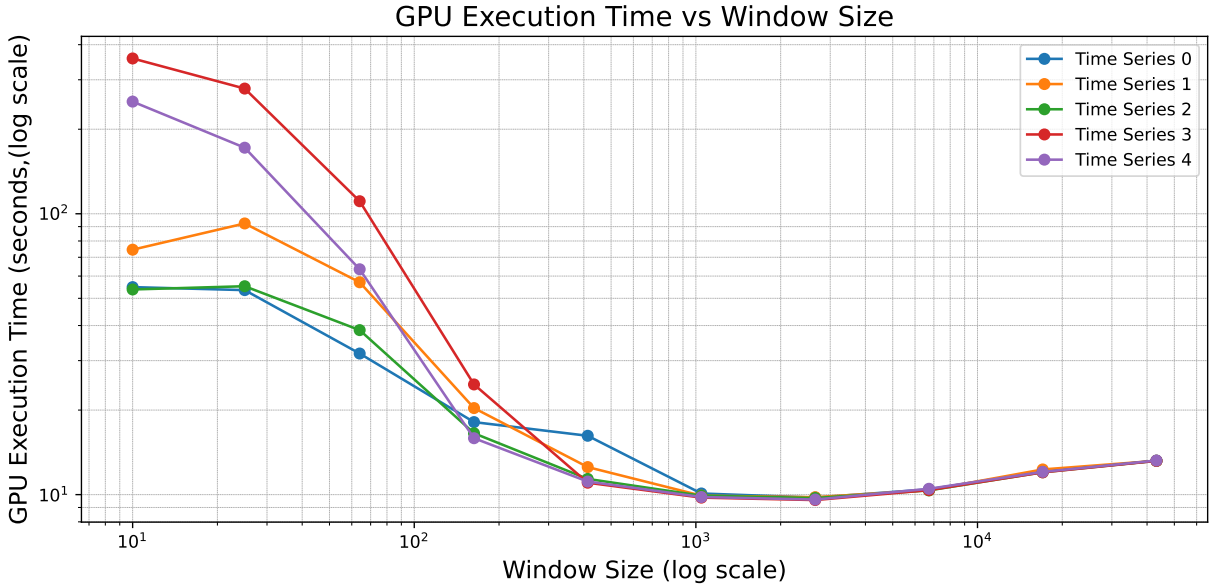


Figure 6.17: GPU execution time (ET) vs. window size (W) for five TSs, using logarithmically spaced W values from 10 to $\frac{n}{2}$ and adaptive k_{opt}. Execution time decreases with W due to reduced subsequence comparisons, converging at large W. All experiments were repeated for five iterations, and the reported ET values represent the mean across iterations.

6.6 Impact of Time Series (TS) Length on Computational Performance

As the length of the input TS increases, the computational and memory demands of motif discovery algorithms scale rapidly. Analyzing how ET and overall scalability are affected by data volume is essential for applying these methods to long-duration or high-resolution TS.

Building upon the previous analyses of how parameter choices such as k and W affect performance, this section investigates the influence of input TS length on computational cost. Specifically, we evaluate how the ET of the Stumpy GPU API scales with increasing TS duration under fixed W and k. Understanding this scaling behavior is critical when applying MP-based motif discovery to long-term monitoring datasets, where input size may range from thousands to millions of data points for TS.

To conduct this evaluation, we incrementally increase the length of each input TS from 1 hour to 24 hours in one-hour steps. Each hour comprises 3600 data points sampled at 1 Hz, esulting in a total range of 3,600 to 86,400 points. We used five different traces—both real-world and synthetic—as input, and fixed the $W = 100$ and $k = 200$ across all runs. These settings mirror those used in Sections 6.2 and 6.3, ensuring a consistent experimental framework.

To minimize measurement noise, each data point in the plot reflects the average of five repeated runs, following an initial warm-up execution to offset GPU initialization overhead. The results, shown in Figure 6.18, reveal a nearly linear increase in ET as the TS length grows. This behavior is consistent with the internal mechanics of MP computation, where the number of candidate subsequences and pairwise comparisons increases proportionally with input size.

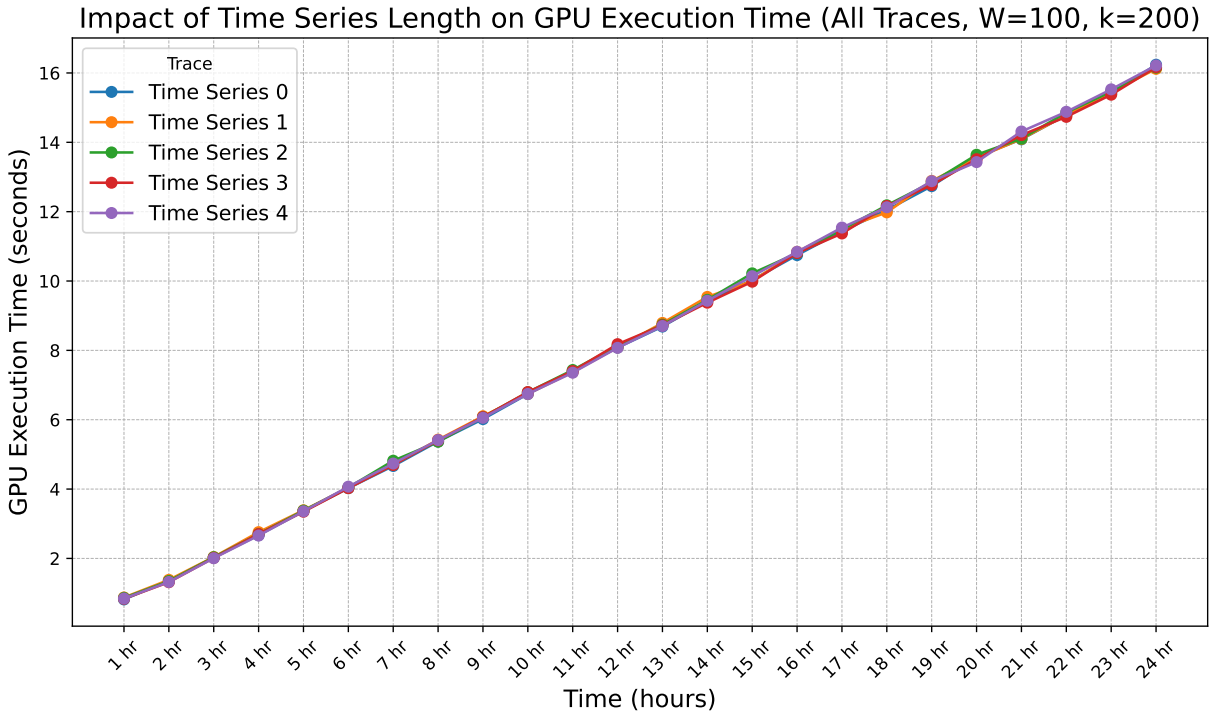


Figure 6.18: GPU execution time (ET) as a function of time series (TS) length across five different TS. Each TS is sampled at a rate of 1 data point per second and extended incrementally from 1 hour (3,600 points) to 24 hours (86,400 points). Parameters are fixed at window size $W = 100$ and number of nearest neighbors $k = 200$. The results show a near-linear increase in ET with respect to TS length.

These findings reaffirm that—even with optimizations such as threshold-based optimization and flat-region masking—the computational burden of motif discovery increases approximately linearly with input length. While the Stumpy GPU API demonstrates efficient execution for moderate-length TSs (e.g., up to one million data points, or approximately one to two weeks of data at 1 Hz), the runtime becomes prohibitive for significantly longer durations. To address this, our approach introduces a chunking strategy that processes each day's data independently in parallel. Subsequently, we apply DBSCAN to cluster patterns extracted from each chunk, thereby enabling scalable motif discovery across extended time periods without sacrificing performance or accuracy as discussed in Chapter 5.

6.7 Summary of Experimental Findings

This chapter evaluated the computational impact of key parameters in our motif mining pipeline, focusing on MP generation using the Stumpy API. Through controlled experiments, we examined how varying the number of neighbors k, the presence of flat regions, window size W, and TS length affects GPU ET and overall efficiency.

We observed that increasing k leads to a nonlinear rise in computation time, as more distances must be calculated and stored. However, beyond a certain point, the benefit of additional neighbors diminishes. To mitigate this, we introduced a threshold-based optimization, which skips comparisons unlikely to yield meaningful motifs. This significantly reduced ET while preserving output quality.

Long flat regions in the TS, which represent prolonged constant values, were also shown to inflate computation time unnecessarily. Although these segments do not contribute informative motifs in our context, they are still processed by Stumpy unless masked. By identifying and skipping flat regions using invalid entries (i.e., `NaN` values), we eliminated redundant computations and improved runtime performance.

Experiments on the EuDist threshold θ_{ed} confirmed it as a key determinant of both runtime and motif quality. Smaller θ_{ed} values markedly reduced ET by pruning subsequences with high pairwise EuDist, while larger values converged toward unoptimized performance. A practical selection strategy is to begin with the lowest feasible value and incrementally increase it while observing the resulting motif patterns.

The window size (W) demonstrated a clear inverse relationship with ET in our experiments. Using logarithmically spaced W values from 10 to $\frac{n}{2}$ and an adaptive nearest neighbor setting $k = k_{\text{opt}}$, we observed that GPU runtime consistently decreased as W

increased. This trend is driven by the reduction in subsequence count $(n - W + 1)$ at larger W, which lowers the number of required distance computations. While highly irregular TSs with small W values exhibited elevated ET due to numerous distinct, non-flat subsequences, runtimes converged across all datasets at large W.

Finally, we validated the scalability of our optimized pipeline by measuring GPU ET across TSs of increasing length. The results confirmed near-linear growth, reinforcing the effectiveness of our optimizations when applied to long-duration TSs.

In summary, the optimizations introduced in this chapter—threshold-based optimization and flat-region masking—substantially enhance computational efficiency without compromising motif quality. These findings support the broader goal of enabling scalable, interpretable, and efficient pattern mining from TSD in real-world applications.

Chapter 7

Results

This chapter presents an empirical evaluation of the proposed greedy-based pattern selection algorithm. The objective is to assess its effectiveness in terms of TS coverage, number of selected patterns, and ET (ET) under varying constraints on the number of patterns (NoP). The evaluation is conducted on a set of representative TSs.

7.1 Quantitative Evaluation of Greedy Algorithms

This section presents the performance evaluation of three greedy selection strategies—Naive Greedy, IntervalTreeGreedy, and Simulated Annealing (SA)—applied to grouped candidate patterns derived from MP-based motif discovery. The objective is to assess the algorithms' ability to select non-overlapping patterns that maximize TS coverage, under a fixed upper limit on the number of patterns (NoP). The evaluation spans five TSs, including one synthetic dataset (TS-0) and four real-world datasets (TS-1 to TS-4), as illustrated in Figures 6.1 to 6.5. Each algorithm was tested under varying NoP constraints, specifically for values 1, 2, 3, 4, 5, 10, 15, 20, 50, 100, and 500.

Performance was measured using three primary metrics: the percentage of TSD explained by the selected patterns (coverage), the number of selected patterns (pattern set size), and the execution time (ET) required to complete the selection. To maintain clarity and avoid redundancy, this section provides a detailed analysis for TS-1 and TS-2, which are representative of real-world scenarios. Full experimental results for TS-0, TS-3, and TS-4 are presented in Appendix C.

7.1.1 Performance on Time Series TS-1 (Real-World Dataset)

Table 7.1: Coverage (%) of greedy algorithms on time series TS-1 under varying NoP

NoP	Naive Greedy	IntervalTreeGreedy	SA
1	4.64	39.19	39.19
2	4.81	52.92	54.29
3	5.25	57.52	58.89
4	5.87	60.58	60.58
5	6.85	62.72	62.72
10	12.36	67.24	68.36
15	16.10	69.01	69.01
20	18.03	70.11	70.11
50	36.09	70.52	70.52
100	36.09	70.52	70.52
500	36.09	70.52	70.52

Table 7.2: Pattern set size (number of selected patterns) on time series TS-1 under varying NoP

NoP	Naive Greedy	IntervalTreeGreedy	SA
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
10	10	10	10
15	15	15	15
20	20	20	20
50	47	23	23
100	47	23	23
500	47	23	23

Table 7.3: Execution time ET (in seconds) of greedy algorithms on time series TS-1 under varying NoP

NoP	Naive Greedy	IntervalTreeGreedy	SA
1	1.15	4.05	10.22
2	1.18	4.65	9.91
3	1.21	4.76	10.79
4	1.16	5.23	10.90
5	2.27	5.21	10.46
10	1.19	5.25	10.16
15	1.44	5.67	10.80
20	1.88	5.65	12.05
50	0.48	5.49	11.18
100	0.85	6.51	11.28
500	0.50	6.38	11.49

Table 7.1 shows that IntervalTreeGreedy and SA consistently achieve higher coverage than Naive Greedy across all NoP values, with SA slightly outperforming at lower values (NoP = 2 and 3). Coverage plateaus around NoP = 50, indicating that the most relevant patterns are captured by this point.

Table 7.2 shows that all methods select the expected number of patterns up to NoP = 20. Beyond this point, Naive Greedy selects more patterns but does not fully utilize the higher NoP values, plateauing at 47. IntervalTreeGreedy and SA consistently select 23 patterns from NoP = 50 onward, indicating early convergence and better handling of overlap constraints.

Table 7.3 highlights that SA is the most computationally expensive. IntervalTreeGreedy achieves similar coverage with much lower runtime, especially beyond NoP = 10. Naive Greedy is the fastest but fails to deliver sufficient coverage, limiting its utility for quality pattern selection.

Overall, IntervalTreeGreedy provides the best balance between coverage and ET on real-world data. While SA achieves slightly higher coverage at low NoP values, its high runtime limits its practicality for large-scale use. Naive Greedy consistently performs worse in both coverage and pattern selection, showing that more advanced strategies are necessary for effective TS pattern mining.

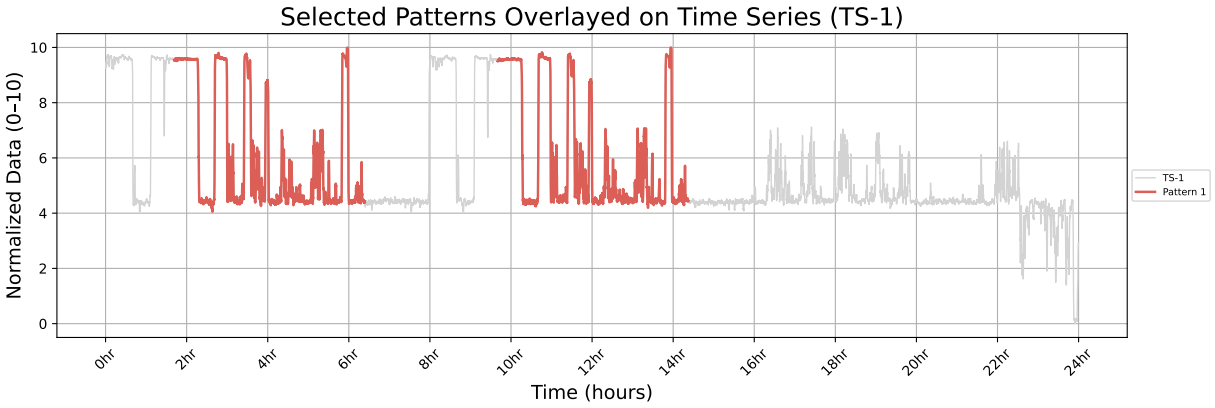


Figure 7.1: IntervalTreeGreedy result on TS-1 with NoP = 1 (Coverage = 39.19%)

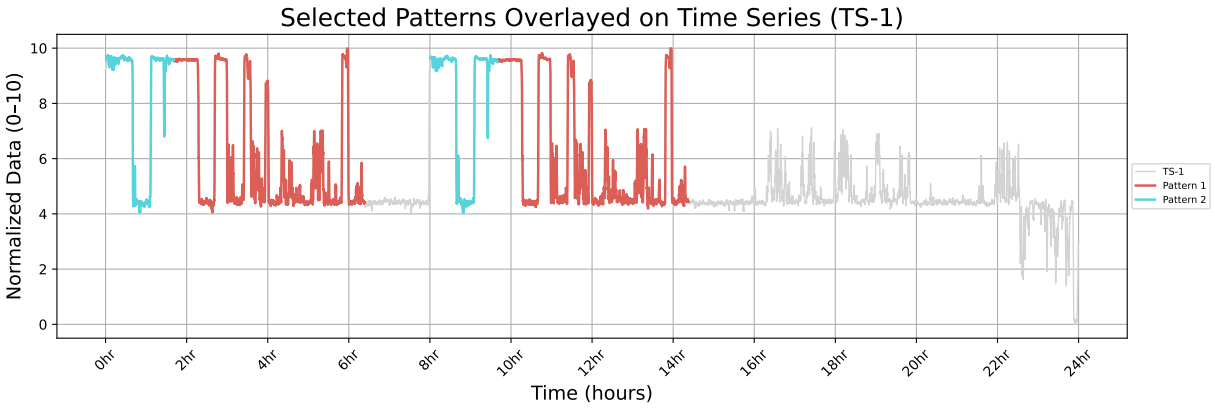


Figure 7.2: IntervalTreeGreedy result on TS-1 with NoP = 2 (Coverage = 52.92%)

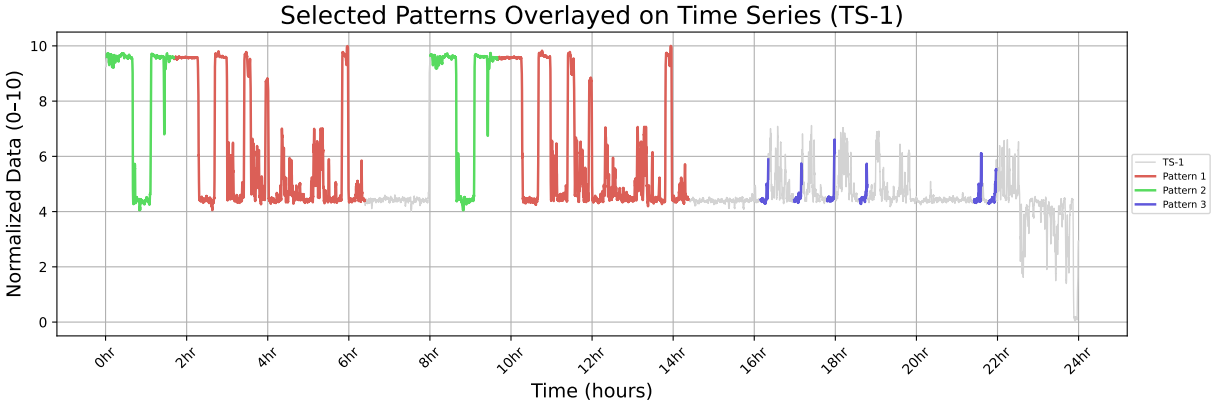


Figure 7.3: IntervalTreeGreedy result on TS-1 with NoP = 3 (Coverage = 57.52%)

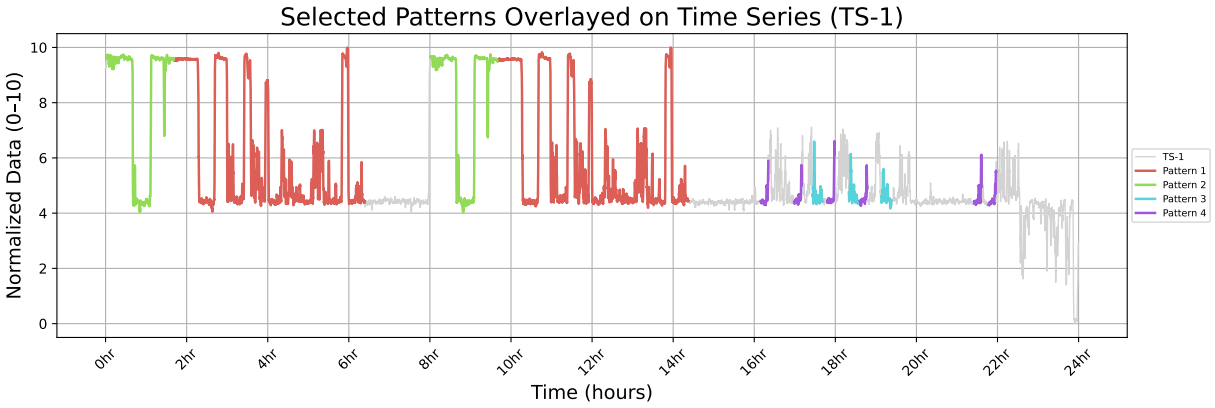


Figure 7.4: IntervalTreeGreedy result on TS-1 with NoP = 4 (Coverage = 60.58%)

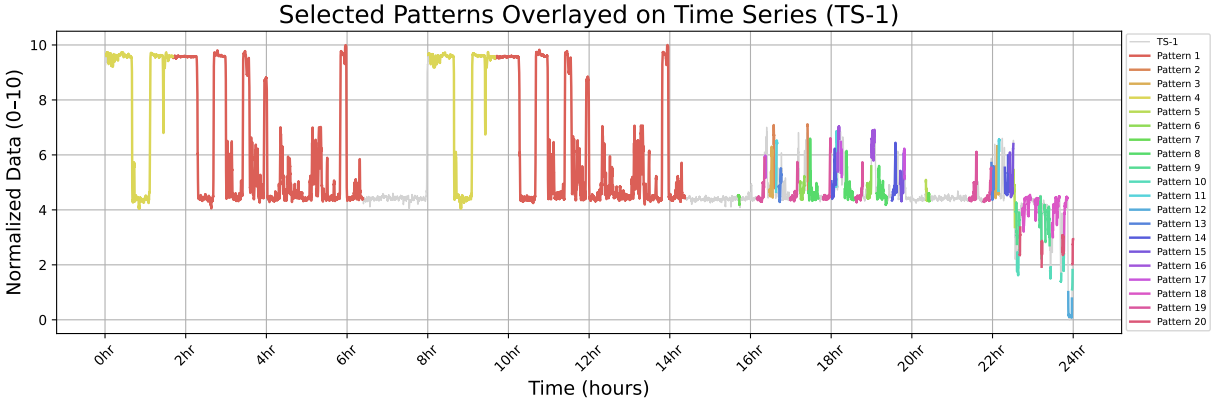


Figure 7.5: IntervalTreeGreedy result on TS-1 with NoP = 20 (Coverage = 70.11%)

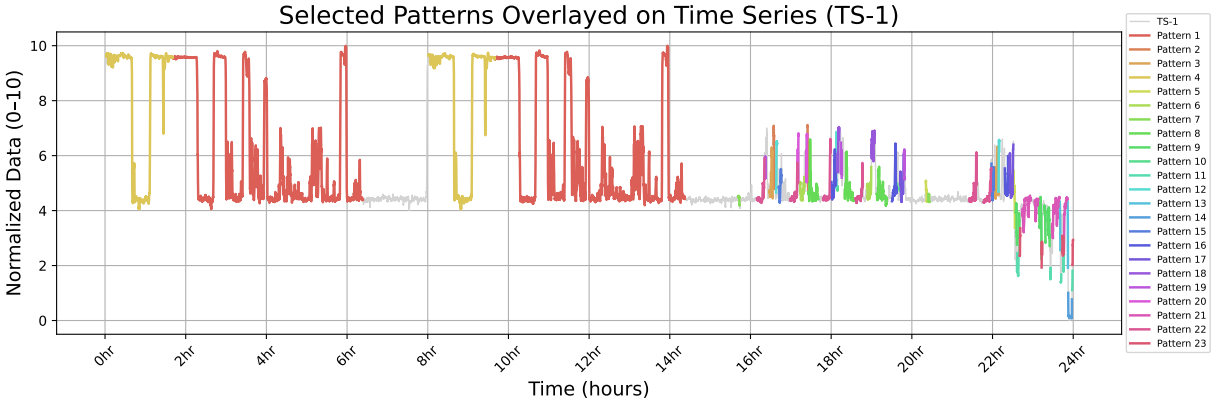


Figure 7.6: IntervalTreeGreedy result on TS-1 with NoP = 100 (Coverage = 70.52%)

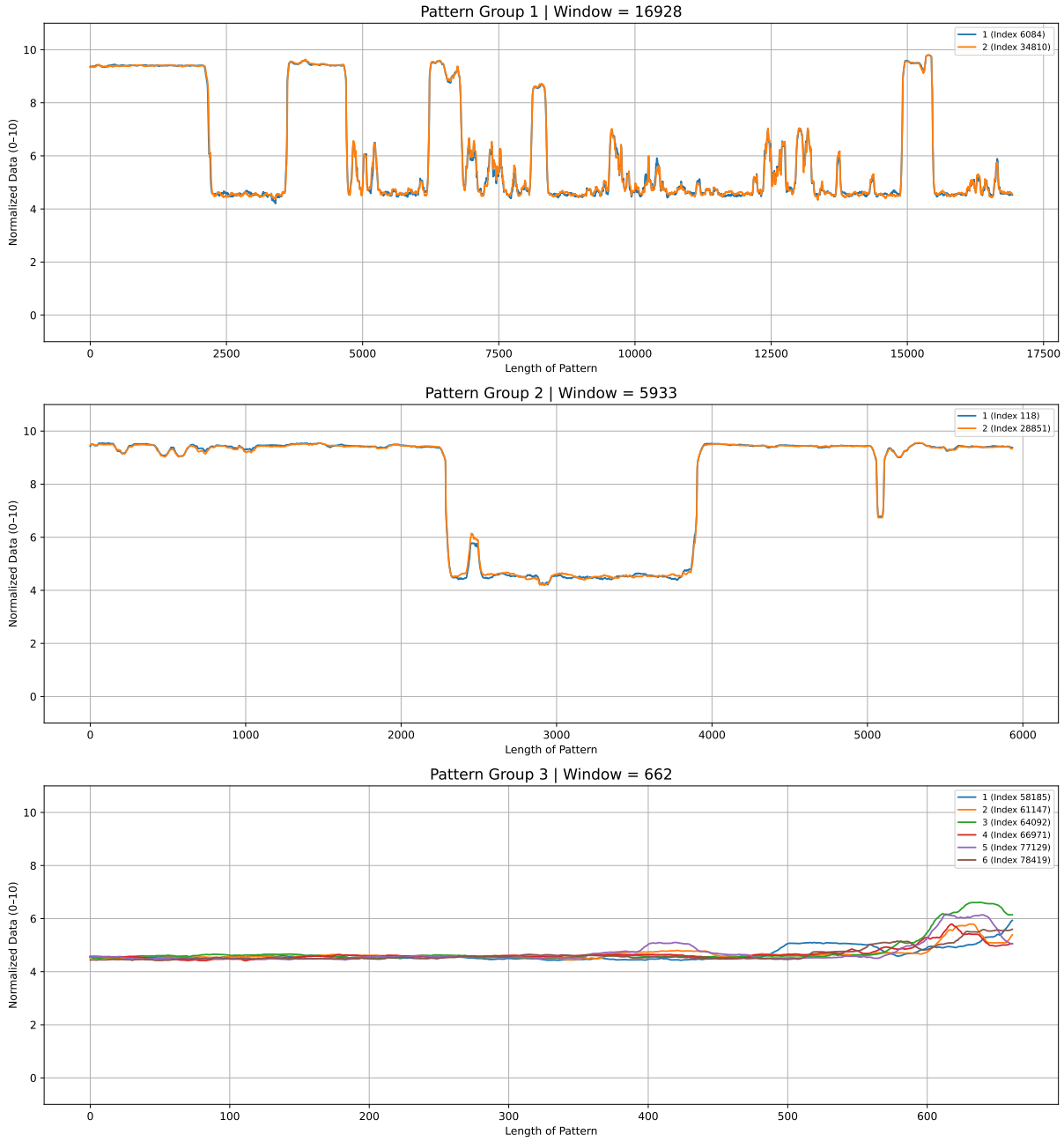


Figure 7.7: Three pattern groups selected on TS-1 using IntervalTreeGreedy with NoP = 3. Each plot shows a representative group with aligned patterns across different window sizes (Coverage = 57.52%).

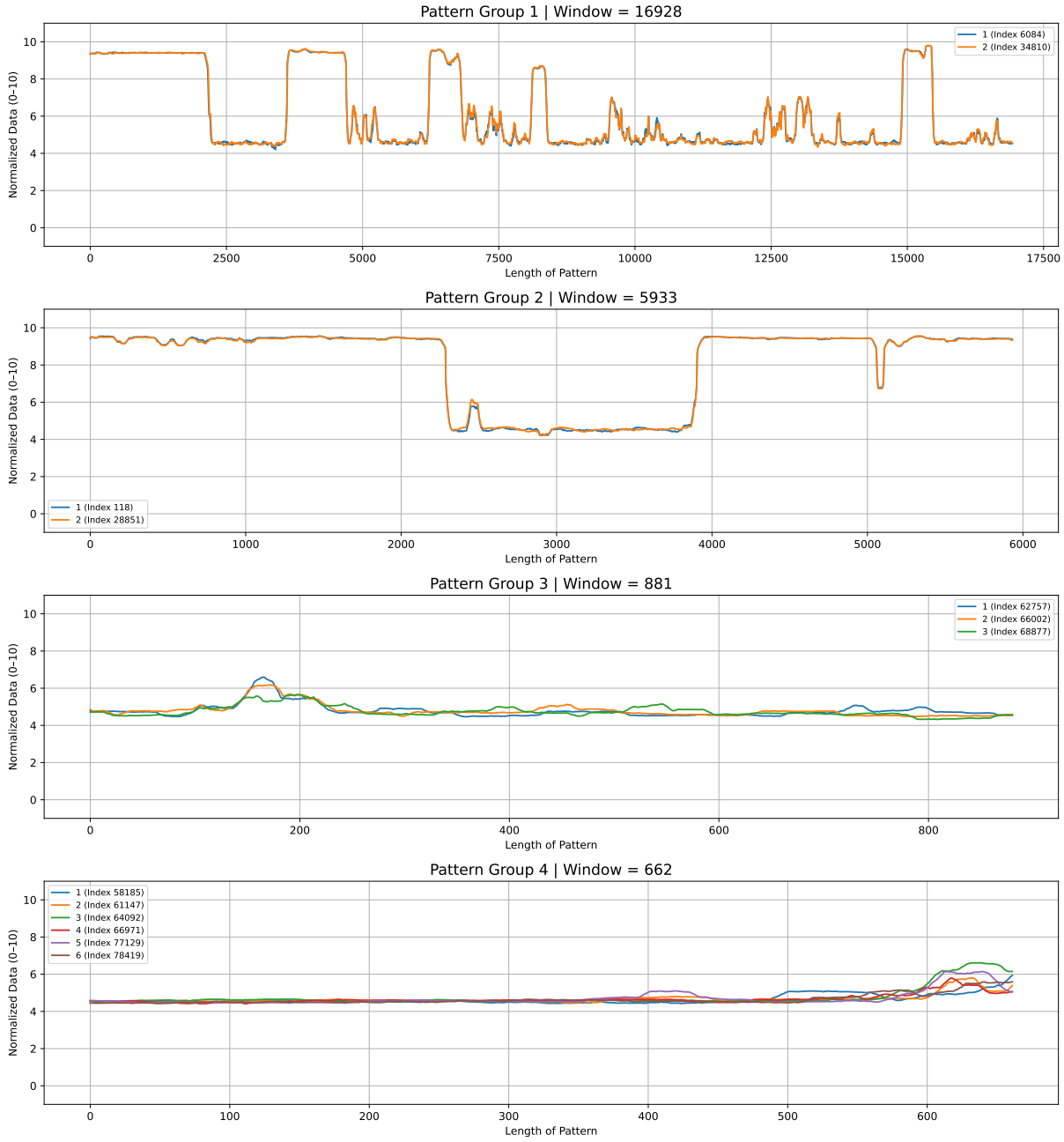


Figure 7.8: Four pattern groups selected on TS-1 using IntervalTreeGreedy with NoP = 4. Each plot shows a representative group with aligned patterns across different window sizes (Coverage = 60.58%).

7.1.2 Performance on Time Series TS-2 (Real-World Dataset)

Table 7.4: Coverage (%) of greedy algorithms on time series TS-2 under varying NoP

NoP	Naive Greedy	IntervalTreeGreedy	SA
1	2.25	24.33	24.33
2	8.65	31.99	31.99
3	8.94	35.48	35.48
4	9.21	38.00	38.00
5	9.73	40.04	40.04
10	11.04	46.81	46.81
15	12.24	50.66	50.66
20	13.93	52.49	52.49
50	20.18	53.13	53.13
100	20.18	53.13	53.13
500	20.18	53.13	53.13

Table 7.5: Pattern set size (number of selected patterns) on time series TS-2 under varying NoP

NoP	Naive Greedy	IntervalTreeGreedy	SA
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
10	10	10	10
15	15	15	15
20	20	20	20
50	34	24	24
100	34	24	24
500	34	24	24

Table 7.6: Execution time ET (in seconds) of greedy algorithms on time series TS-2 under varying NoP

NoP	Naive Greedy	IntervalTreeGreedy	SA
1	1.45	0.84	5.46
2	1.33	0.92	6.23
3	1.52	1.05	5.36
4	1.28	1.05	6.74
5	1.45	1.06	5.22
10	1.34	1.29	4.35
15	1.28	1.44	4.12
20	1.38	1.55	4.67
50	0.32	1.72	4.85
100	0.37	1.73	4.46
500	0.30	1.66	4.59

For TS-2, both IntervalTreeGreedy and SA significantly outperform Naive Greedy in coverage across all NoP values, with gains saturating around NoP = 50 (Table C.1). Pattern set sizes follow the NoP constraint up to 20, after which Naive Greedy selects more patterns than IntervalTreeGreedy and SA, but without meaningful coverage improvement (Table 7.5). ETs remain low for all methods, with IntervalTreeGreedy offering the best trade-off between efficiency and coverage, while SA remains the most computationally expensive (Table C.2).

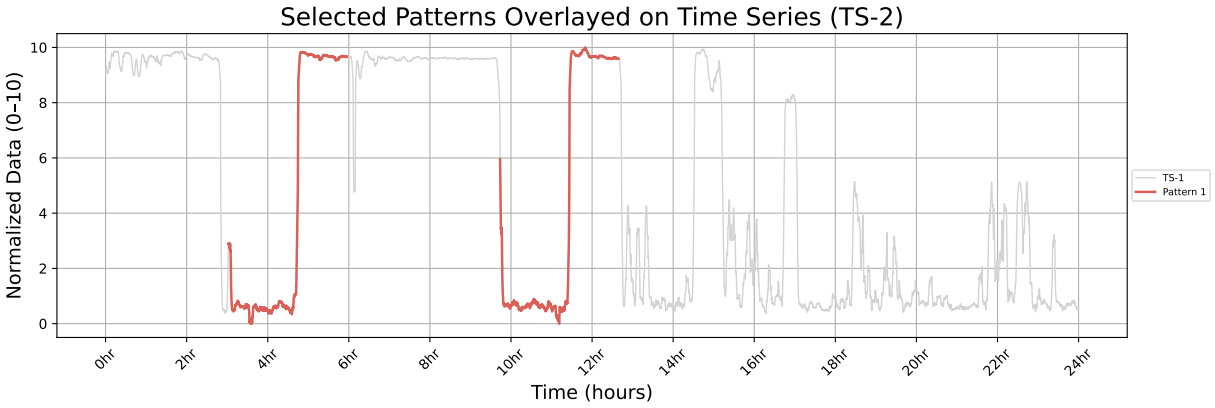


Figure 7.9: IntervalTreeGreedy result on TS-2 with NoP = 1 (Coverage = 24.33%)

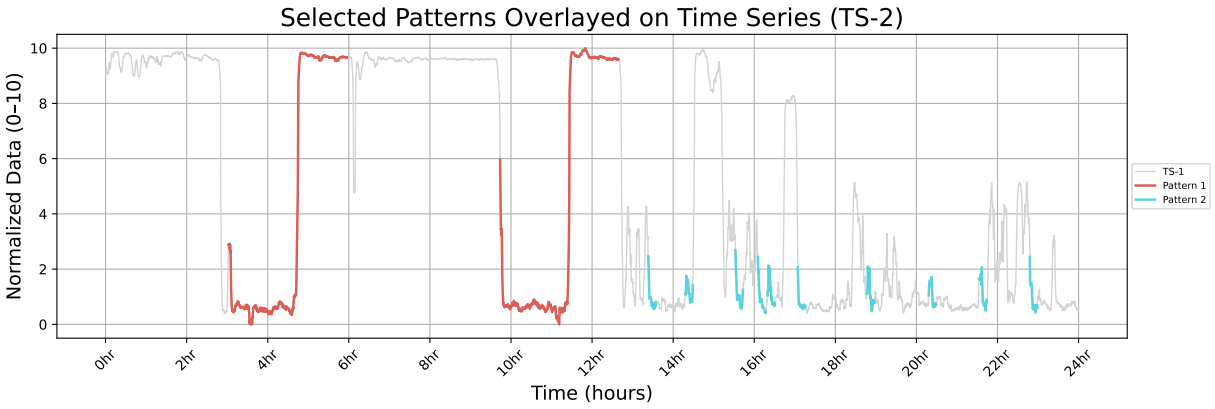


Figure 7.10: IntervalTreeGreedy result on TS-2 with NoP = 2 (Coverage = 31.99%)

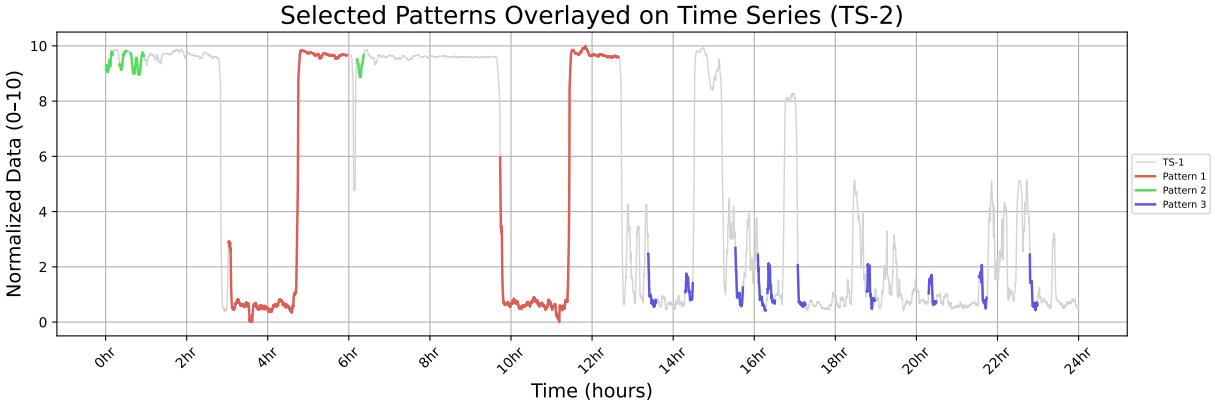


Figure 7.11: IntervalTreeGreedy result on TS-2 with NoP = 3 (Coverage = 35.48%)

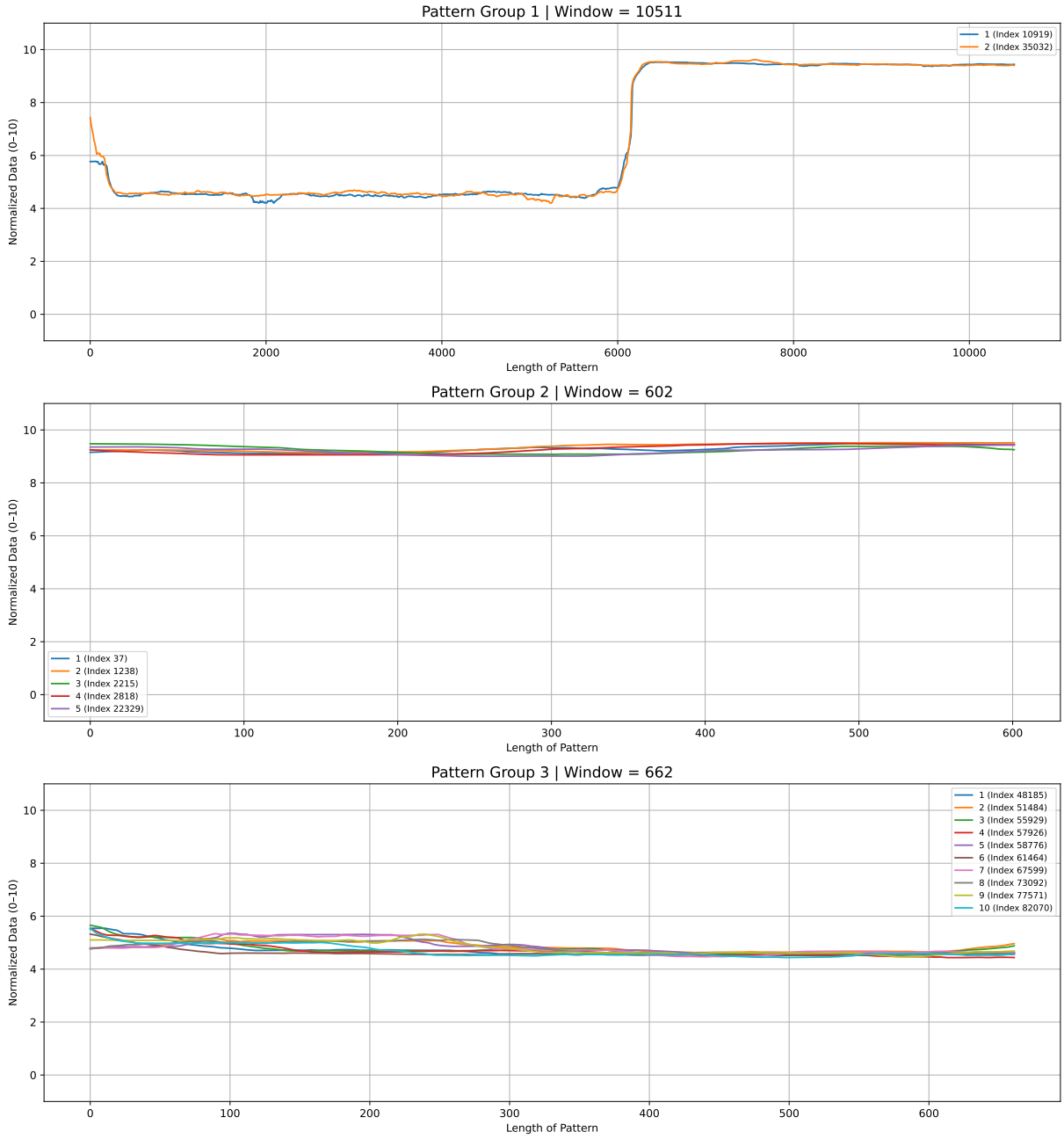


Figure 7.12: Three pattern groups selected on TS-2 using IntervalTreeGreedy with NoP = 3. Each plot shows a representative group with aligned patterns across different window sizes (Coverage = 35.48%).

7.2 Summary

This chapter evaluated the proposed greedy-based pattern selection methods across multiple TSs. IntervalTreeGreedy and SA consistently outperformed Naive Greedy in terms of coverage. While SA achieved slightly higher coverage at lower NoP values on TS-1, both methods showed similar performance on TS-2. IntervalTreeGreedy offered the best overall trade-off, achieving near-optimal coverage with significantly lower ET.

The results also indicated that coverage improvements tend to plateau beyond moderate NoP values in several cases, suggesting that a limited number of well-chosen patterns can often capture the key structural components of the data. Visual analyses further highlighted the ability of the methods—particularly IntervalTreeGreedy—to extract meaningful, diverse, and interpretable motifs, even under varying signal conditions.

Collectively, these findings confirm the effectiveness and efficiency of the proposed selection strategies, supporting their applicability in practical settings where both computational performance and result interpretability are essential.

Chapter 8

Discussion

This chapter analyzes the methodological choices, experimental outcomes, and practical implications of the proposed framework. It focuses on the impact of hyper-parameter selection, solver and hardware considerations, and discusses real-world use cases.

8.1 Impact of Hyper-parameter Choices

The proposed pattern mining framework depends on several critical hyper-parameters that influence both the structural quality of extracted motifs and the computational efficiency of the discovery process. Specifically, three primary parameters govern motif extraction: the minimum window size, the maximum window size, and the EuDist threshold. The significance of these parameters is empirically examined in Chapter 6, where their effects on motif quality and runtime performance are thoroughly evaluated.

8.1.1 Minimum Pattern Window Size

Setting $W_{\min}$ too low can lead to *trivial solutions*—cases where the algorithm selects patterns that technically satisfy the constraints but have little practical meaning. In this context, such solutions occur when patterns are so short they capture random noise rather than meaningful structure (e.g., $W_{\min} = 1$ or 2). In most cases, $W_{\min}$ should be no smaller than 3–5 data points as a practical starting point. A suitable value can be set to the approximate duration of the shortest recurring event of interest in the data, as informed

by domain expertise, visual inspection, or exploratory analysis. This ensures that detected patterns represent meaningful behaviors while avoiding the computational cost and redundancy associated with very short, uninformative matches.

8.1.2 Maximum Pattern Window Size

Setting $W_{\max}$ too high can also produce *trivial solutions*, where the algorithm returns patterns that formally meet the objectives but offer little insight into the system's structure. Here, this occurs when a small number of very long patterns cover most of the series without revealing meaningful substructure (e.g., $W_{\max}$ close to the series length). A practical choice is to set $W_{\max}$ to the approximate duration of the longest recurring event of interest, which can be estimated from domain expertise, visual inspection of the data, or basic analyses such as autocorrelation. This avoids capturing only broad trends while still allowing the detection of long but meaningful patterns.

8.1.3 Euclidean Distance Threshold

The EuDist threshold θ_{ed} defines the maximum allowable distance between subsequences for them to be considered the same motif and is inherently a *per-point* measure that scales with the window size W of the subsequences (Section 6.4). For data normalized to $[0, 10]$, the theoretical per-point maximum is 10, but practical values should be far lower to maintain physical feasibility in motif matching.

As shown in Section 6.4, smaller θ_{ed} values reduce GPU ET by aggressively pruning candidate subsequences, while larger values converge toward the runtime of the unoptimized implementation. A practical selection strategy is to start with the smallest feasible θ_{ed} and incrementally increase it, monitoring the resulting computation time. In this work, an empirically chosen $\theta_{ed} = 0.25$ provided consistent and efficient performance across datasets.

8.1.4 Variance Threshold for Flat Region Masking

The variance threshold used in flat region masking determines which segments of the TS are considered uninformative and excluded before motif discovery. Setting this threshold too low may result in retaining long stretches of trivial, near-constant values, which can lead to the extraction of redundant or uninformative motifs. Conversely, an excessively

high threshold may cause meaningful but low-variance patterns to be masked out, reducing motif diversity and coverage.

Importantly, this threshold is a user-defined parameter and should be selected based on the characteristics of the data, including its typical variability and the sampling rate or time scale. Careful adjustment allows users to filter out only those regions that do not contribute valuable structural information, while preserving subtle but important variations. Balancing this parameter is essential for achieving both computational efficiency and the interpretability of the resulting motifs.

8.1.5 Base Length Selection for Chunking

The base length for chunking, denoted as L, determines how the TS is divided for localized motif discovery. While chunking was not applied during the experiments (see Chapter 6) because the TS datasets were of manageable size and fit within available memory, the primary purpose of chunking in this framework is to address memory limitations when dealing with very long TS. By dividing the data into fixed-length segments, chunking enables processing without exceeding memory capacity and supports parallel computation. However, as shown in Chapter 6, computational time is linearly dependent on the length of the TS. Its role is thus limited to memory management rather than computational optimization.

8.1.6 Selection of k in STUMPY Motif Discovery

Throughout this thesis, the neighbour budget k used in Stumpy is

$$k = k_{opt} = \max\big(k_{\min},\ \lceil n/W \rceil\big), \qquad k_{\min} = \text{constant value},$$

where $k_{\min}$ is a fixed constant (e.g., $k_{\min} = 10, 50, 100$), n is the series length, W the window size, and $N = n - W + 1$ the number of subsequences.

Motivation.

- **Theoretical sufficiency:** The ratio $\lceil n/W \rceil$ is the maximum number of non-overlapping windows of length W that can fit in a series of length n; requesting more neighbours cannot reveal additional motifs.

- **Minimum budget:** For very large W the ratio $\lceil n/W \rceil$ can fall to very low value. A fixed floor of $k_{\min} = 100$ guarantees enough neighbours.
- **Memory safety:** Setting $k = N$ allocates an $N \times k$ matrix on the GPU; in the 86 400-sample trace in Section 6.2, this required more than 50 GiB of device memory and triggered out-of-memory errors. Under the window-aware rule, k grows only with $\lceil n/W \rceil$, so memory usage stays linear in N and fits comfortably on a GPU

This definition of k is used consistently throughout the thesis; later chapters cite Section 8.1.6 whenever the parameter is mentioned.

8.2 Practical Use Cases

The discovered motif patterns have practical utility in diverse application domains. Below, we illustrate how this framework can support tasks such as system monitoring, anomaly detection, and operational intelligence.

8.2.1 System State Identification

The proposed framework enables the extraction of characteristic motifs from TSD, which can be leveraged as input features for system state classification models. While the algorithm itself does not assign or interpret operational states, the motifs it identifies encapsulate recurring system behaviors that are informative for downstream state detection. In industrial settings, such features support the development of predictive maintenance systems and facilitate continuous monitoring to enhance operational reliability.

8.2.2 Anomaly Detection and Diagnosis

The framework provides a structured method to obtain representative motifs that reflect typical system behavior. These motifs can be used to define baselines for normal operations, enabling downstream anomaly detection algorithms to identify deviations indicative of faults, intrusions, or other irregularities. Although direct anomaly comparison or diagnosis is not performed within this framework, the extracted patterns serve as essential building blocks for advanced diagnostic and monitoring solutions in areas such as cybersecurity and healthcare.

8.3 Hardware Environment and Solver Selection

All experiments were conducted on a Linux system with dual Intel Xeon E5-2640 v2 CPUs (16 cores) and 64 GB of RAM, operating under Debian Linux 4.19.0-24-amd64.

For modeling and solving CSPs, we utilized MiniZinc, an open-source constraint modeling language widely recognized for its flexibility and broad solver support. MiniZinc interfaces with a variety of solvers, among which *Gecode* [20] and *OR-Tools* [16] are prominent. Each solver offers distinct advantages depending on the nature of the problem and optimization goals.

To determine the most suitable solver for our research, we conducted a comparative analysis of `Gecode 6.2.0` and `OR-Tools 9.1` by measuring their computation times across multiple TSDs (see Table 8.1). The evaluation revealed that OR-Tools consistently outperformed Gecode, achieving an average computation time of 27.15 seconds, compared to Gecode's 48.61 seconds. Based on these findings, we selected *OR-Tools 9.1* [16] for our experiments, as it demonstrated superior computational efficiency and faster solution times.

Table 8.1: Computational time comparison (in seconds) for Gecode and OR-Tools solvers on five representative time series (TS) datasets. Lower values indicate faster performance.

Time Series	Run	Gecode	OR-Tools
1	1	65.58 s	13.41 s
	2	86.84 s	13.26 s
	3	75.78 s	13.53 s
2	1	23.56 s	18.74 s
	2	28.93 s	18.68 s
	3	25.04 s	19.36 s
3	1	49.57 s	30.33 s
	2	56.47 s	30.05 s
	3	52.88 s	29.97 s
4	1	68.24 s	27.41 s
	2	58.89 s	26.09 s
	3	47.86 s	25.79 s
5	1	37.19 s	46.79 s
	2	31.16 s	46.71 s
	3	21.23 s	47.16 s
Average Time		**48.61 s**	**27.15 s**

Chapter 9

Conclusion

This thesis addressed the challenge of extracting interpretable, non-overlapping, and representative patterns from univariate TSD in a scalable and computationally efficient manner. Motivated by limitations in existing motif discovery methods, this work prioritized maximizing global TS coverage under explicit constraints on the NoP, thus promoting both comprehensiveness and interoperability.

As detailed in Chapter 3, although two problem formulations were presented, this thesis focused solely on maximizing coverage subject to a user-defined upper bound on the number of patterns (NoP), as formalized in Equation 3.1. All methods, implementation, and experimental results directly address this objective, providing a practical framework for extracting concise yet comprehensive pattern sets from real-world TSD.

The proposed solution is a modular framework that integrates GPU-accelerated motif discovery, threshold-based filtering, flat-region masking, and density-based clustering (DBSCAN). Pattern selection is formulated as a constraint programming problem and efficiently solved using Google OR-Tools, which, as shown in Section 8.3, outperformed alternative solvers in computation time. Implementation efforts focused on scalability, leveraging chunk-based processing and parallelism to handle large-scale datasets.

Empirical evaluation presented in Chapter 6 and Chapter 7 demonstrates the effectiveness and robustness of the proposed framework. Across both real-world and synthetic datasets, the system exhibits nearly linear scalability with respect to runtime, remains resilient to variations in motif length and signal complexity, and produces compact, high-coverage summaries. Notably, the introduction of a novel threshold variable within the Stumpy API optimization pipeline contributes to a substantial reduction in overall computation time.

The discussion chapter further examined the impact of hyper-parameters—such as window size, Euclidean threshold, and DBSCAN density—on both runtime and solution quality. Real-world use cases in anomaly detection and system state identification further illustrated the method's versatility and value across diverse domains, including industrial monitoring and cybersecurity.

The main contributions of this thesis are as follows. First, it introduces a novel formulation of TS pattern mining as a constrained optimization problem focused on maximizing global coverage while ensuring interpretability and practical utility. Second, the work presents a scalable methodology that integrates GPU-accelerated motif discovery, flat-region masking, and constraint-driven pattern selection to efficiently handle large datasets. Third, the framework features a custom extension of the Stumpy library, enabling efficient threshold-based pruning and significantly improving the scalability of motif discovery. Fourth, comprehensive empirical validation demonstrates the robustness and utility of the proposed approach, including benchmarking of solver performance, across both real-world and synthetic datasets. Finally, the practical value and versatility of the framework are established through detailed use case analyses, confirming its applicability in diverse real-world settings.

Future Work

While this thesis provides a solid foundation for interpretable and scalable TS pattern mining, several avenues remain for future exploration. First, extending the framework to multivariate TS would enhance applicability to complex, interdependent systems. Second, incorporating domain-specific constraints or supervised components could improve the relevance of discovered patterns in expert-guided settings. Third, adaptive thresholding—potentially informed by machine learning models—may enable dynamic tuning based on data characteristics. Finally, enabling real-time, streaming pattern mining with low-latency updates will be essential for applications such as financial TSA, industrial IoT monitoring, forecasting, and anomaly detection.

In summary, this research advances the state of the art in TS motif discovery by bridging interpretability, scalability, and global pattern coverage. The proposed framework not only addresses longstanding limitations in traditional approaches but also lays the groundwork for user-centric, intelligent data analysis tools across a wide range of real-world TS applications.

References

[1] Nicolas Beldiceanu and Mats Carlsson. Time-window based automata for time-series constraints. In *International Conference on Principles and Practice of Constraint Programming (CP)*, pages 177–193. Springer, 2016.

[2] S. C. Brailsford, C. N. Potts, and B. M. Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, 1999.

[3] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, W. Xu, and K. Fu. Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices. In *2013 USENIX Workshop on Health Information Technologies (HealthTech 13)*, 2013.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[5] F. Ding, H. Li, F. Luo, H. Hu, L. Cheng, H. Xiao, and R. Ge. Deeppower: Non-intrusive and deep learning-based detection of iot malware using power side channels. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 33–46, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[7] T.-C. Fu. A review on time series data mining. *Engineering Applications of Artificial Intelligence*, 24(1):164–181, 2011.

[8] Shaghayegh Gharghabi, Abdullah Mueen, Yan Zhu, Chen-Chia Michael Yeh, Liudmila Ulanova, and Eamonn Keogh. Matrix profile viii: Domain agnostic online semantic segmentation at superhuman performance levels. *IEEE ICDM*, 2018.

[9] A. Grisel-Davy and S. Fischmeister. Mad: One-shot machine activity detector for physics-based cyber security. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pages 684–695, 2023.

[10] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen. Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1573–1582, New York, NY, USA, 2016. Association for Computing Machinery.

[11] Van Long Ho, Nguyen Ho, and Torben Bach Pedersen. Mining seasonal temporal patterns in time series. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 2278–2287, New York, NY, USA, 2021. ACM.

[12] R. Iváncsy and I. Vajk. Frequent pattern mining in web log data. *Acta Polytechnica Hungarica*, 3(1):77–90, 2006.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[14] S. Law. Stumpy: A powerful and scalable python library for time series data mining. *Journal of Open Source Software*, 4:1504, July 2019.

[15] M. Linardi, Y. Zhu, T. Palpanas, and E. Keogh. Matrix profile goes mad: Variable-length motif and discord discovery in data series. *Data Mining and Knowledge Discovery*, 34:1022–1071, 2020.

[16] Google LLC. Google or-tools. https://developers.google.com/optimization/, 2023. Software library for combinatorial optimization.

[17] J. Lonardi and P. Patel. Finding motifs in time series. In *Proc. of the 2nd Workshop on Temporal Data Mining*, pages 53–68, 2002.

[18] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[19] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard cp modelling language. In C. Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer.

[20] C. Schulte, G. Tack, and M. Z. Lagerkvist. Case studies. In *Modeling and Programming with Gecode*. 2019. Corresponds to Gecode 6.2.0.

[21] H. Tang and S. S. Liao. Discovering original motifs with different lengths from time series. *Knowledge-Based Systems*, 21(7):666–671, 2008.

[22] R. Vaarandi and M. Pihelgas. Logcluster - a data clustering and pattern mining algorithm for event logs. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 1–7, 2015.

[23] P. J. M. van Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications*, volume 37 of *Mathematics and Its Applications*. Springer, 1987.

[24] R. Wankhedkar and S. K. Jain. Motif discovery and anomaly detection in an ecg using matrix profile. In *Progress in Advanced Computing and Intelligent Engineering: Proceedings of ICACIE 2019, Volume 1*, pages 88–95. Springer, 2021.

[25] C.-C. M. Yeh, N. Kavantzas, and E. Keogh. Matrix profile vi: Meaningful multidimensional motif discovery. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 565–574, 2017.

[26] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh. Matrix profile i: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 1317–1322, 2016.

APPENDICES

Appendix A

Appendix: Pseudocode for all Greedy approached with benchmarking

Algorithm 6 Naive Greedy Pattern Selection

Require:

L:	length of time series (TS)
$\mathcal{P}$:	list of candidate patterns
minCov:	minimum coverage threshold
maxNop:	maximum NoP to select

Ensure: A list of selected pattern IDs

Initialize $Selected \leftarrow \emptyset$, $Covered \leftarrow \emptyset$
while $|Selected| <$ maxNop and $|\text{Covered}|/L <$ minCov **do**
    $best \leftarrow$ None, $bestGain \leftarrow 0$
    for all $p \in \mathcal{P}$ not in $Selected$ **do**
        if p overlaps with any index in $Covered$ **then**
            continue
        end if
        $gain \leftarrow$ number of new indices covered by p
        if $gain > bestGain$ **then**
            $best \leftarrow p$, $bestGain \leftarrow gain$
        end if
    end for
    if $best =$ None **then**
        break
    end if
    Add $best.id$ to $Selected$
    Update $Covered \leftarrow Covered \cup best.coverage$
end while
return $Selected$

Algorithm 7 Greedy Pattern Selection with Interval Tree

Require:

L: length of time series (TS)
$\mathcal{P}$: list of candidate patterns
minCov: minimum coverage threshold
maxNop: maximum NoP

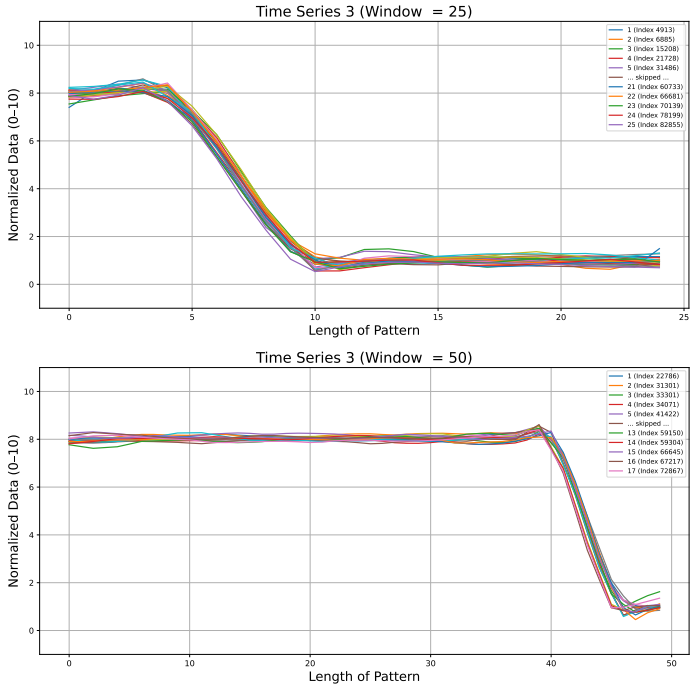
Ensure: A list of selected pattern IDs

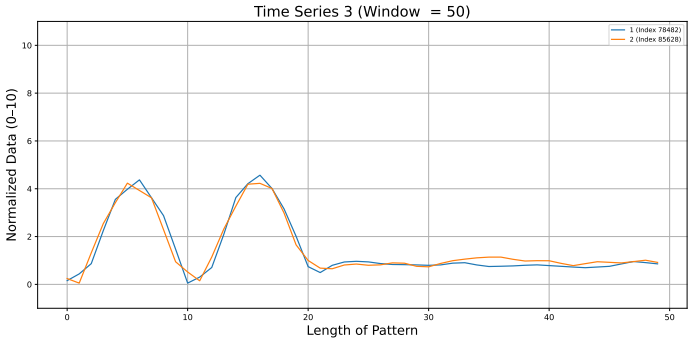
Initialize $Selected \leftarrow \emptyset$, $Covered \leftarrow \emptyset$
Initialize empty Interval Tree T
while $|Selected| <$ maxNop and $|\text{Covered}|/L <$ minCov **do**
    $best \leftarrow$ None, $bestGain \leftarrow 0$
    for all $p \in \mathcal{P}$ not in $Selected$ **do**
        if p conflicts with T **then**
            continue
        end if
        $gain \leftarrow |p.coverage \setminus Covered|$
        if $gain > bestGain$ **then**
            $best \leftarrow p$, $bestGain \leftarrow gain$
        end if
    end for
    if $best =$ None **then**
        break
    end if
    Add $best.id$ to $Selected$
    Add $best.coverage$ to $Covered$
    Insert $best.intervals$ into T
end while
return $Selected$

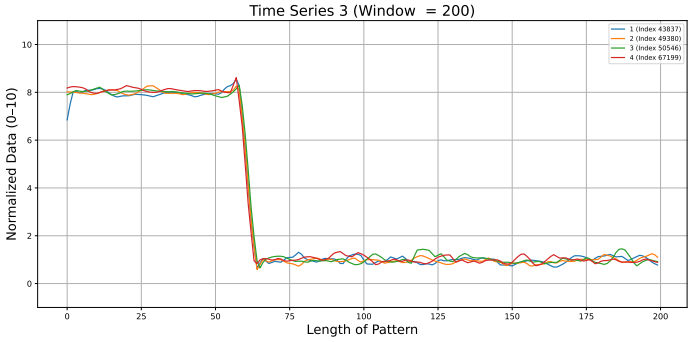
Appendix B

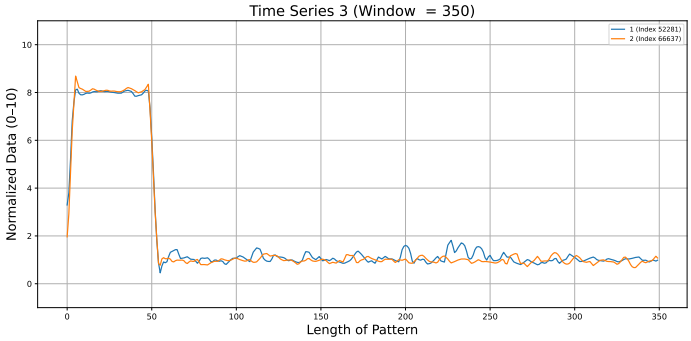
Time Series Plots

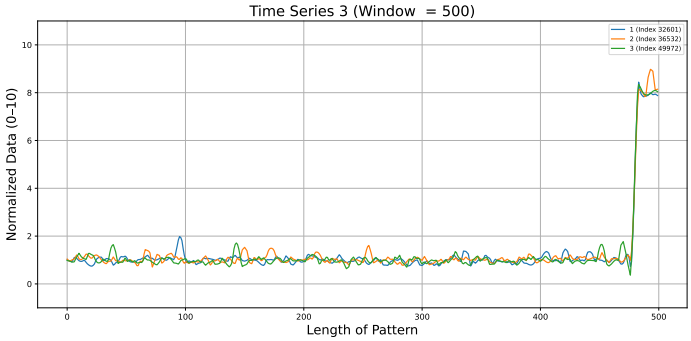
B.1 Patterns Extracted from time series TS-3

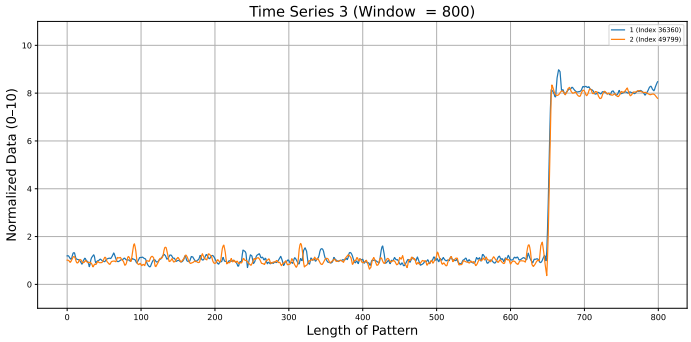


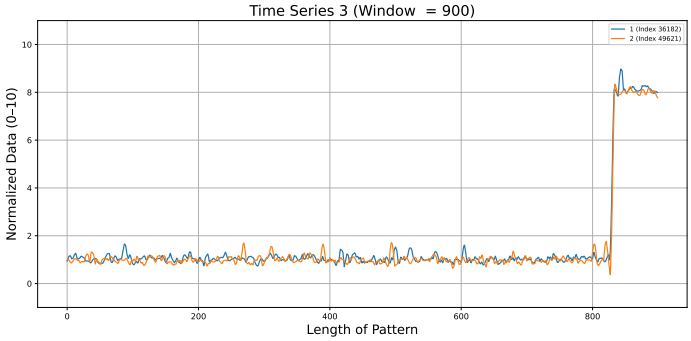

Time Series 3 (Window = 50)
1 (Index 78482)
2 (Index 85628)
Normalized Data (0–10)
Length of Pattern


Time Series 3 (Window = 200)
1 (Index 43837)
2 (Index 49380)
3 (Index 50546)
4 (Index 67199)
Normalized Data (0–10)
Length of Pattern

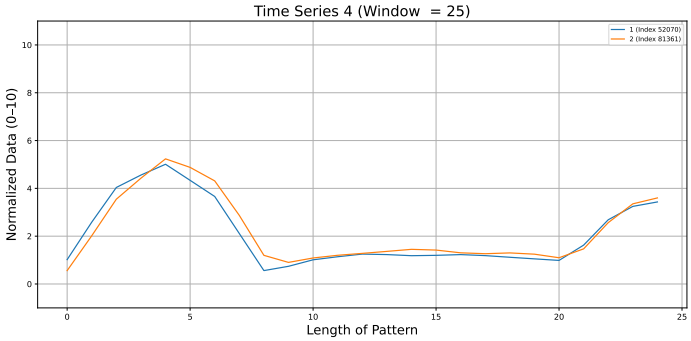

Time Series 3 (Window = 350)
1 (Index 52281)
2 (Index 66637)
Normalized Data (0–10)
Length of Pattern

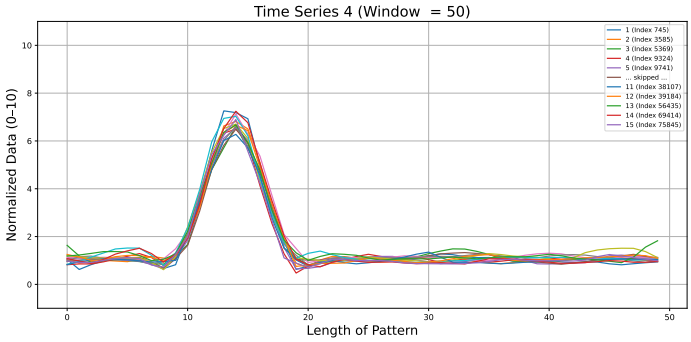

Time Series 3 (Window = 500)
1 (Index 32601)
2 (Index 36532)
3 (Index 49972)
Normalized Data (0–10)
Length of Pattern

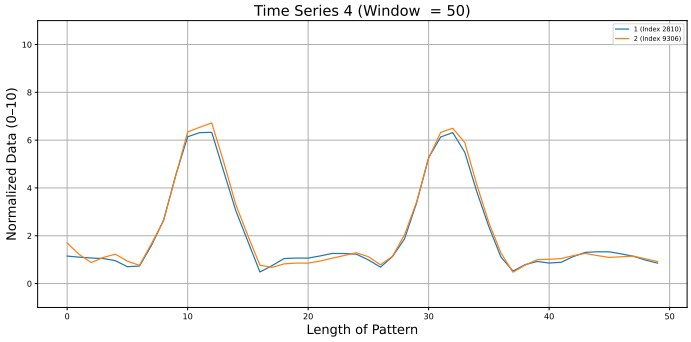


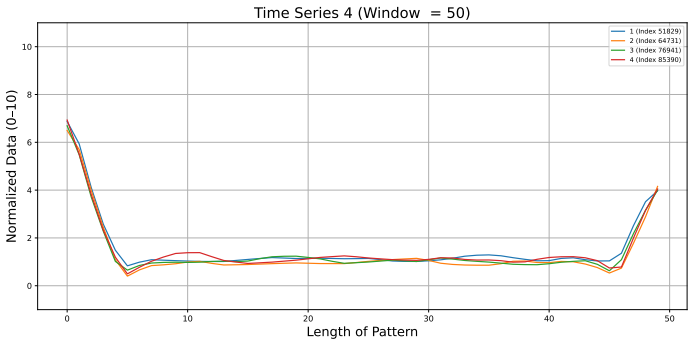


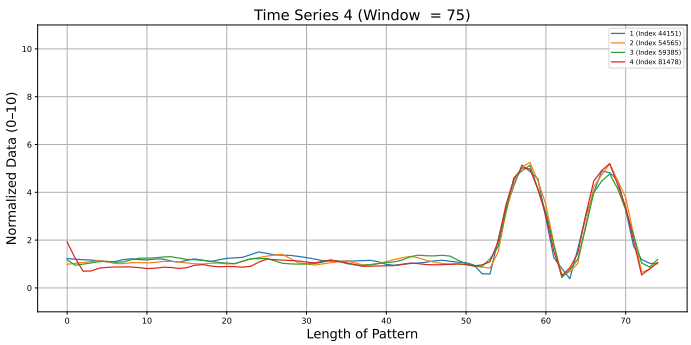
B.2 Patterns Extracted from time series TS-4

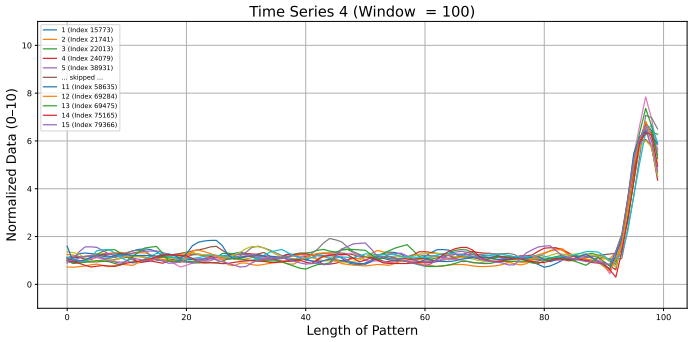


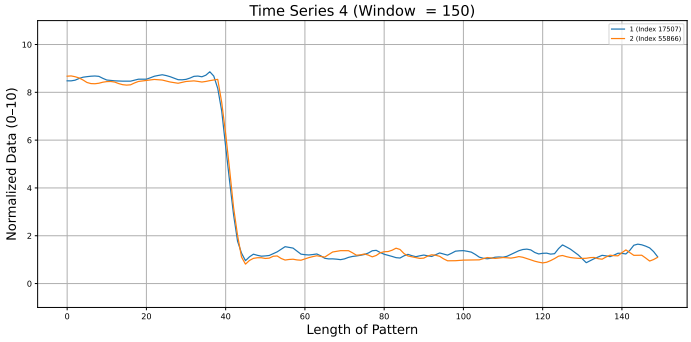

Time Series 4 (Window = 50)
Normalized Data (0–10)
Length of Pattern
1 (Index 745)
2 (Index 3585)
3 (Index 5369)
4 (Index 9324)
5 (Index 9741)
... skipped ...
11 (Index 38107)
12 (Index 39184)
13 (Index 56435)
14 (Index 69414)
15 (Index 75845)

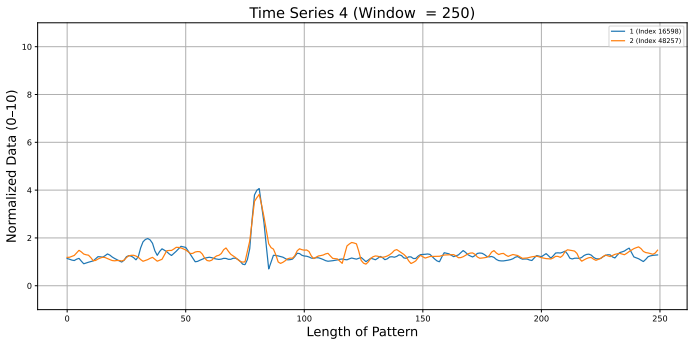

Time Series 4 (Window = 50)
Normalized Data (0–10)
Length of Pattern
1 (Index 2810)
2 (Index 9306)


Time Series 4 (Window = 50)
Normalized Data (0–10)
Length of Pattern
1 (Index 51829)
2 (Index 64731)
3 (Index 76941)
4 (Index 85390)


Time Series 4 (Window = 75)
Normalized Data (0–10)
Length of Pattern
1 (Index 44151)
2 (Index 54565)
3 (Index 59385)
4 (Index 81478)


Time Series 4 (Window = 100)
1 (Index 15773)
2 (Index 21741)
3 (Index 22013)
4 (Index 24079)
5 (Index 38931)
... skipped ...
11 (Index 58635)
12 (Index 69284)
13 (Index 69475)
14 (Index 75165)
15 (Index 79366)
Normalized Data (0–10)
Length of Pattern


Time Series 4 (Window = 150)
1 (Index 17507)
2 (Index 55866)
Normalized Data (0–10)
Length of Pattern


Time Series 4 (Window = 250)
1 (Index 16598)
2 (Index 48257)
Normalized Data (0–10)
Length of Pattern

Appendix C

Quantitative Results

C.0.1 Results on time series TS-2

Table C.1: Coverage (%) of greedy algorithms on time series TS-2 under varying NoP

NoP	Naive Greedy	IntervalTreeGreedy	SA
1	2.43	15.05	15.05
2	3.12	15.74	15.74
3	4.51	16.09	16.09
4	4.54	16.15	16.15
5	5.06	16.15	16.15
10	6.22	16.15	16.15
15	6.22	16.15	16.15
20	6.22	16.15	16.15
50	6.22	16.15	16.15
100	6.22	16.15	16.15
500	6.22	16.15	16.15

Table C.2: Execution time (in seconds) of greedy algorithms on time series TS-2 under varying NoP

NoP	Naive Greedy	IntervalTreeGreedy	SA
1	1.01	0.99	2.66
2	0.99	0.38	1.97
3	0.99	0.45	1.43
4	1.01	0.34	1.22
5	0.95	0.35	1.33
10	1.01	0.53	1.38
15	1.09	0.35	1.24
20	0.95	0.36	1.24
50	0.95	0.35	1.23
100	0.96	0.35	1.27
500	0.97	0.35	1.23

C.0.2 Results on time series TS-3

Table C.3: Coverage (%) of greedy algorithms on time series TS-3 under varying NoP

NoP	Naive Greedy	IntervalTreeGreedy	SA
1	0.03	58.47	58.47
2	0.10	61.34	61.34
3	0.12	61.63	61.63
4	0.17	61.81	61.81
5	0.49	61.97	61.97
10	1.74	62.35	62.35
15	4.28	62.63	62.63
20	4.77	62.86	62.86
50	10.50	63.62	63.62
100	18.66	64.07	64.07
500	35.35	64.07	64.07

Table C.4: Execution time (in seconds) of greedy algorithms on time series TS-3 under varying NoP

NoP	Naive Greedy	IntervalTreeGreedy	SA
1	0.96	0.31	2.98
2	0.97	0.29	3.18
3	0.95	0.29	3.28
4	0.99	0.30	3.57
5	1.04	0.30	3.74
10	1.08	0.31	3.89
15	1.01	0.32	3.86
20	1.02	0.34	3.14
50	1.01	0.41	2.58
100	0.96	0.49	4.08
500	0.35	0.48	4.02

C.0.3 Results on time series TS-4

Table C.5: Coverage (%) of greedy algorithms on time series TS-4 under varying NoP

NoP	Naive Greedy	IntervalTreeGreedy	SA
1	0.01	57.34	57.34
2	0.19	61.06	61.06
3	0.88	61.54	61.54
4	0.90	61.88	61.88
5	1.13	62.07	62.07
10	2.30	62.57	62.57
15	3.00	62.89	62.89
20	3.99	63.17	63.17
50	9.05	64.08	64.08
100	13.55	64.74	64.74
500	38.17	65.05	65.05

Table C.6: Execution time (s) of greedy algorithms on time series TS-4 under varying NoP

NoP	**Naive Greedy**	**IntervalTreeGreedy**	**SA**
1	0.98	0.25	3.12
2	1.05	0.26	3.16
3	1.05	0.25	3.32
4	1.01	0.26	3.21
5	1.10	0.26	3.86
10	1.01	0.27	3.01
15	1.71	0.31	2.88
20	1.04	0.29	4.12
50	0.98	0.43	3.23
100	0.99	0.51	2.86
500	0.44	0.53	4.38

Glossary

Stumpy An open-source Python library for scalable time series data mining, particularly for computing matrix profiles and discovering motifs and anomalies x, xii, 14, 15, 17–19, 24, 27, 28, 30, 36, 41, 49, 50, 54, 58, 59, 62, 78, 83

Stumpy API The Python programming interface provided by STUMPY for matrix profile computation and time series motif discovery 30, 32, 33, 48, 49, 54, 62, 82

Stumpy GPU API An informal reference to the GPU-accelerated functionality in the open-source STUMPY Python library, particularly the `gpu_stump()` function, which enables efficient matrix profile computation on CUDA-enabled GPUs 44, 48, 59, 60, 62