Investigation of Adaptive Piecewise Linear Approximation algorithms for Similarity Search on Big Time Series Data

Final Report

James Thompson
3rd Year University of Warwick
Supervisor: Dr. Weiren Yu

April 26, 2025

Abstract

As more data is collected in the world, databases of Time Series Data become larger and increase the strain on algorithms operating on the data. Compression techniques have been developed that maintain structural properties of the data to allow large datasets to be stored and processed efficiently. Adaptive Piecewise Linear Approximation is one compression technique that approximates continuous segments of data. However computing the optimal segments is challenging and multiple algorithms across multiple disciplines have been created.

Similarity Search is a fundamental problem in Time Series Analysis, for example used to detect heart beats and find stock market trends. This report provides a review and empirical comparison of current Adaptive PLA algorithms. A novel approach to extend Adaptive PLA to solve Similarity Search is presented, capable of using high compression techniques and offering unique advantages over other indexing schemes.

Keywords: Time Series, Compression, Similarity Search, Big Data, Approximation, Indexing Scheme, High Dimensional Data, GEMINI

Acknowledgements

Throughout the project, support was provided to me constantly from many. I would like to firstly like to thank my project supervisor Dr. Weiren Yu, for the time dedicated to helping, inspiring and supporting me with this project. There would be no research or report without him.

Content made freely available by the UCR Time Series Classification Archive greatly improved this project and enabled it to be comparable and accessible.

Finally, thanks to friends and family for helpfully reminding me there was still work to do to ensure it reached its conclusion.

Contents

1 Introduction **3**

2 Formal Definitions **5**
2.1 Time Series Data . . . 5
2.2 Comparing Time Series . . . 6
2.2.1 Fair comparisons . . . 7
2.3 Similarity Search and K Nearest Neighbours . . . 8
2.4 Dimension Reduction Techniques . . . 10
2.5 The Gemini Framework . . . 13

3 Methodology **15**
3.1 Objectives . . . 15
3.2 Research Methods . . . 15
3.3 Practical Methods . . . 16
3.3.1 Codebase Structure . . . 17
3.3.2 Software Used . . . 17

4 Current Literature **19**
4.1 Indexing Local DRTs . . . 19
4.2 Indexing Adaptive DRTs . . . 21
4.3 Exact Adaptive PLA . . . 22
4.4 Heuristic PLA . . . 22
4.5 Streaming Adaptive PLA . . . 23
4.6 Comparison of existing algorithms . . . 25

5 Generating Time Series Data **26**
5.1 Data Parsing . . . 26
5.2 Data Generation . . . 27

6 Exact Segment Algorithms **29**
6.1 Sliding Window Approaches . . . 29
6.2 Optimal Piecewise Linear Approximation . . . 34

7 Epsilon Precision Algorithms **37**
7.1 Top Down Algorithm . . . 37
7.2 Bottom Up Algorithm . . . 39
7.3 Sliding Window . . . 40
7.4 SWING Filters . . . 42
7.5 Correcting Dimension . . . 44

8 Priority Queue Optimisation **47**
8.1 Split and Merge . . . 47
8.2 Exact Segment Sliding Window . . . 47

9 Indexing Adaptive PLA **49**
9.1 Partition Cover Construction . . . 52
9.2 Merging Partition Covers . . . 53

10 Evaluations **56**
10.1 Exact Segment algorithms . . . 56
10.1.1 Observations . . . 61
10.2 Epsilon Precision Algorithms . . . 61
10.2.1 Observations . . . 64
10.2.2 Precision Methods with Exact . . . 64
10.3 Pruning Power . . . 67
10.3.1 Observations . . . 70
10.4 Similarity Search and K-NN . . . 70

11 Project Management **72**
11.1 Achieving Objectives . . . 72
11.2 Project Organisation . . . 73
11.2.1 Methodology Evaluation . . . 73
11.2.2 Considerations of time limitations . . . 76
11.3 Discussion of Ethical, Social and Legal considerations . . . 77

12 Conclusions **79**
12.1 Project Conclusions . . . 79
12.2 Future Work . . . 79
12.3 Closing Remarks . . . 80

Chapter 1

Introduction

Time series data is ubiquitous in the modern world, collected by sensors continuously on a increasing scale. Greater computing power and further interest in AI has both generated the production of huge amounts of data and motivated the development of tools to process and consume it. From 2013 to 2023, the amount of data collected in the world has increased more than tenfold[1] and even single source sensors from telescopes or hospital EEG scans can create datasets close to a trillion data points[2]. The sheer amount of data therefore poses a natural challenge as algorithms are tightly constrained by their running time and space requirements in order to execute in a timely manner.

Similarity Search is the problem of finding all similar subsequences in the dataset of distance ε from the target. K Nearest Neighbour (K-NN) finds the K closest subsequences to the query. Solving these problems is highly desirable in many applications such as disease detection, seismology, image recognition and structural engineering and they form a cornerstone of many specialised and complicated queries for data mining. Knowledge in advance that many queries of Similarity Search or K-NN may be performed on the dataset motivates optimisation of how the dataset is stored and accessed, to cull unnecessary computations and compute results quicker. In 1994 Falatsos et al. [3] proposed the Gemini Framework as a methodology to store a dataset in a manner enabling faster searches using spatial indexing structures. These provide a fast traversal of the higher dimensional space but are impacted by the curse of dimensionality - higher dimensional data is far sparser than lower dimensional data, making attempts to efficiently bound the space challenging. Hence methods to speed up Similarity Search and K-NN have focused on more accurate approximations and spatial indexing structures that perform better in higher dimensions.

Dimension Reduction Techniques (DRTs) are approximation methods that reduce the overall size (dimension) of the dataset whilst maintaining as much of the original information as possible. These are techniques that approximate the original signal via the use of other classes of functions that are simpler to express, and don't aim to compress the raw data like lossless methods such as Huffman's encoding. These approximations often have nicer properties due to the class of functions that they belong to, offering insights into the data without storing the original data in the main memory. Naturally, the closer the approximation to the original data source, the more information the insight provides and so research has focused on finding methods for determining accurate approximations of Time Series Datasets. When approximating one sequence with another, there is the opportunity to either

use one function to approximate the entire series (global approaches) or many functions to partition the larger sequence into smaller sequences in which more accurate estimates may be derived (local approaches).

Global optimising techniques approximate the whole time series as one continuous curve, often using some class of easy curves to form approximations such as Discrete Fourier Transforms or Chebyshev Polynomials. Local optimising techniques use classes of functions that can be expressed with few parameters, this enables them to approximate smaller segments of the data for the same cost of storage. Local optimising approximations can use constant or adaptive lengths. Constant length approximations have each function approximate a segment of the data of constant length, while adaptive length approximations allow functions to approximate segments of greater or smaller lengths to better fit the data. Adaptive length algorithms often form better approximations [4], [5]. A fast optimal algorithm for adaptive constant functions has been found but there is no equivalent for adaptive linear functions. Keogh et al.[6] in 2001 reviewed the three common techniques (Sliding Window, Bottom Up and Top Down) for approximating time series by linear segments and attempted to determine which of the three were 'most effective' but there have been developments since. An exact solution to adaptive PLA was found by Ljosa and Singh[7] but this algorithm is slow and therefore there remains the opportunity to make faster approximate algorithms.

This report investigates the approaches of recent adaptive algorithms and evaluates their design, implementation and effectiveness. It will also explore a new approach to indexing piecewise linear approximations, using methods developed by Chakrabarti et al.[4]. It will describe its design, implementation and proof of correctness, allowing analysis of raw compression of the time series datasets and solutions of Similarity Search and K-NN.

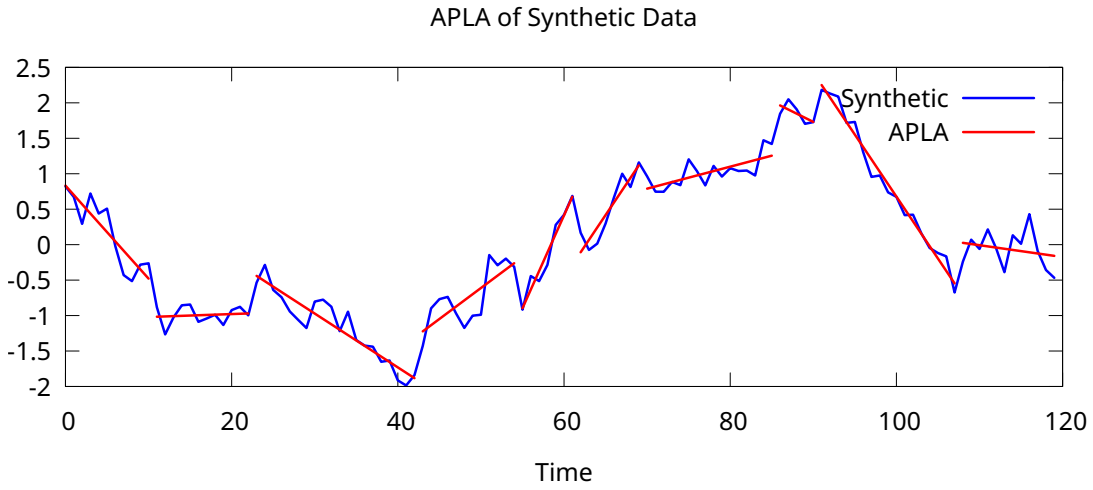


Figure 1.1: Adaptive Piecewise Linear Approximation (Red) of Time Series (Blue)

Chapter 2

Formal Definitions

2.1 Time Series Data

Time Series Data can be any data object indexed via time but the report will concentrate on the simpler case of one variable tracked over discrete, evenly spaced time intervals. This is called a *sequence* and also a *series* in Time Series Analysis literature and hence both terms are used interchangeably throughout the paper. All references to sequences or series can be assumed to be finite mathematical sequences, (or if preferred, elements of a vector space $\mathbb{R}^n$ for some $n \in \mathbb{N}$), and any summations (labelled *series* as in the case of infinite sums by mathematicians) used will be labelled clearly as summations.

A time series will be a sequence of numbers $\langle y_0, y_1, y_2, ... \rangle$ (indexed at 0 for coherence with our code) where y_i is the value at time $i \in \mathbb{N}$. Time series will be of finite length and the entire series will be denoted by $S[0..n] = \langle s_0, s_1, .., s_n \rangle$ indicating that the series S starts at index 0 and ends at index n. The length of S is $n + 1$, and S will be indexed by notations $S[i] = s_i$ for access to element i and $S[i : j] = s_i, s_{i+1}, .., s_j$ for contiguous subsequence access. An alternative view of such a time series is a vector, where a time series $S[n]$ is a vector in $\mathbb{R}^{n+1}$.

$$S = \langle 0, 2, 2, 4, 3, 1, -1, 2 \rangle \in \mathbb{R}^8$$
$$S[3] = 4, \quad S[0 : 4] = \langle 0, 2, 2, 4, 3 \rangle \in \mathbb{R}^5$$

Time Series Data of the format above can be represented as a line graph where element y_i is plotted as the point (i, y_i) and linearly interpolate between points to provide indication of trends.

As an example, consider the random walk generated via independent standard normal variables. The list below details from left to right the resulting values drawn from repeatedly adding an output of the random variable to a walk starting at the origin and is shown in Figure 2.1.

0.0	-0.549746	-1.95262	-0.369867	-1.41501	-1.15742
-3.11681	-4.62462	-4.93991	-4.08174	-4.01039	-6.3027
-7.71825	-6.83242	-6.20052	-6.16026	-7.06339	-6.88806
-6.69363	-7.22888	-6.45153	-6.28019	-6.72304	-5.01814
-4.09379	-5.39865	-5.77923	-6.52286	-6.95998	-7.38643

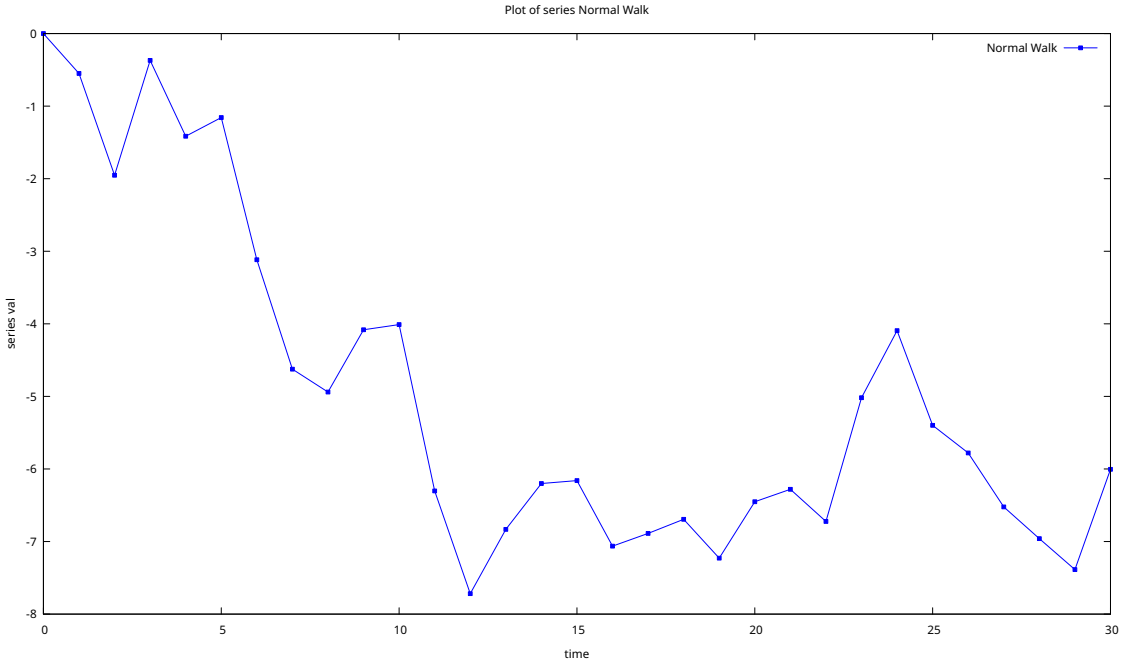


Figure 2.1: Random Walk Plot

2.2 Comparing Time Series

To compare our approximations to the original data, it is necessary to choose some means to measure the distance between the points. Here considering our data as vectors in some dimension $\mathbb{R}^n$ is advantageous as we can adopt the standard L_p norms defined on the space. These come with additional properties; as norms they have the following characteristics that make them highly suitable to measure differences.

1. $\forall x \in \mathbb{R}^n, \quad ||x||_{L_p} = 0 \iff x = 0$
2. $\forall x \in \mathbb{R}^n, \forall \alpha \in \mathbb{R}, \quad ||\alpha x||_{L_p} = |\alpha| \cdot ||x||_{L_p}$
3. $\forall x, y \in \mathbb{R}^n, \quad ||x + y||_{L_p} \leq ||x||_{L_p} + ||y||_{L_p}$

As these are the only norms for this paper L will be dropped and uses of $||S||_p$ refer to $||S||_{L_p}$. Other measurements of distance exist for Time Series Analysis.

To consider the similarity between two time series in $\mathbb{R}^n$ the norm of the difference between the two is used; $S, T \in \mathbb{R}^n, \text{dist}(S, T) = ||S - T||_{L_p}$.

The L_p norms are defined for all $p \in [1, \infty]$ and $S \in \mathbb{R}^n$ by;

$$||S||_p := (\sum_{i=0}^{n-1} |S[i]|^p)^{\frac{1}{p}}, p \in [1, \infty)$$

$$||S||_\infty = max_{i=0,..,n-1}|S[i]|$$

Two specific cases of this general definition will be used throughout the paper, the euclidean error (L2) as $||S||_2$ and the maximum deviation (MaxDev) as $||S||_\infty$.

As $x \to x^{\frac{1}{2}}$ is an increasing function it preserves inequalities. Therefore inequalities using the L2 distance can be replaced with the *Squared Error*; $\text{SE}(S) := (||S||_2)^2$. Although it doesn't maintain the scalar preserving property of a norm ($||\alpha S||_p \neq |\alpha| \cdot ||S||_p$), it is faster to compute.

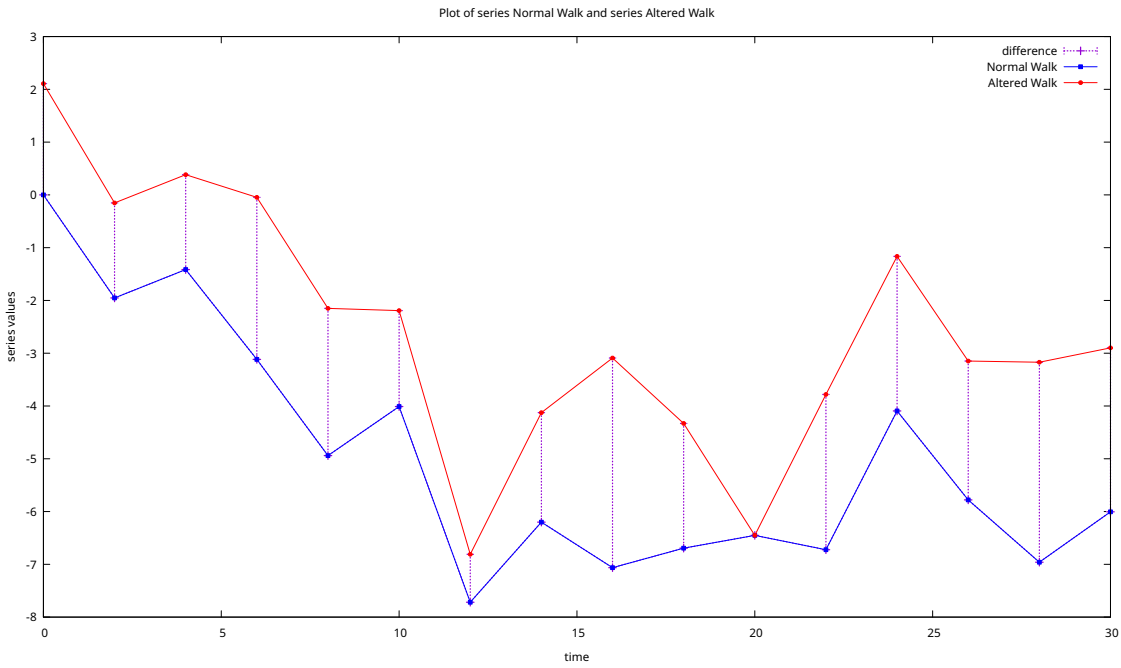


Figure 2.2: Comparison of Random Walk and Altered version

Figure 2.2 shows an example of two random walks in the same dimension with a visual indication of the distance between each point. The Squared Error of the distances would be the sum of the lengths of each distance squared, the Euclidean Distance would be the square root of the Squared Error and the Maximum Deviation would be the largest distance between two points.

2.2.1 Fair comparisons

Distance functions measure error between two time series but this error is proportional to the scale of these series. The second property of a norm is scalar preservation; scaling two time series by α would scale the error between them by α as well. For results on one dataset to be comparable to results on a different dataset, these sequences need to be the same scale.

This can be arranged via a step called *Z Normalisation*, that sets the mean value of the data to 0 and the variance to 1. This ensures that scale and bias won't be considered in results we derive in this paper. For a time series $S[0..n]$ we can determine the mean and variance as follows.

$$\mu_S = \frac{1}{n+1}\sum_{i=0}^{n} S[i] \quad \sigma_S^2 = \sum_{i=0}^{n} (S[i] - \mu_S)^2$$

Algorithm 2.1 Z Normalisation

Require: $S[0..n]$
1: **for** $i = 0, .., n$ **do**
2: $\quad S[i] \leftarrow \sigma_S^2 \cdot (S[i] - \mu_S)$
3: **end for**
4: **return** S ▷ Algorithm linear in time and constant in space

2.3 Similarity Search and K Nearest Neighbours

Similarity search finds all subsequences of a larger time series ε close to a query subsequence. With definitions of distance, Similarity Search and K-NN can be formalised as;

Similarity Search: Given time series dataset $S[0..n]$, query time series $Q[0..m]$ and error $\epsilon \geq 0$ with $m \leq n$, return all contiguous subsequences $S[i : j]$ such that $j - i = m - 1$ and $||S[i : j] - Q||_2 \leq \epsilon$.

K Nearest Neighbours: Given time series dataset $S[0..n]$, query time series $Q[0..m]$ and integer $k \in \mathbb{N}$ with $m \leq n$, return k contiguous subsequences $S[i_l : j_l], \quad l \in [k]$ such that $j_l - i_l = m - 1$ and $||S[i_l : j_l] - Q||_2 \leq \epsilon$.

The choice to use the euclidean norm is arbitrary and the definitions above could be described more generally on any norm.

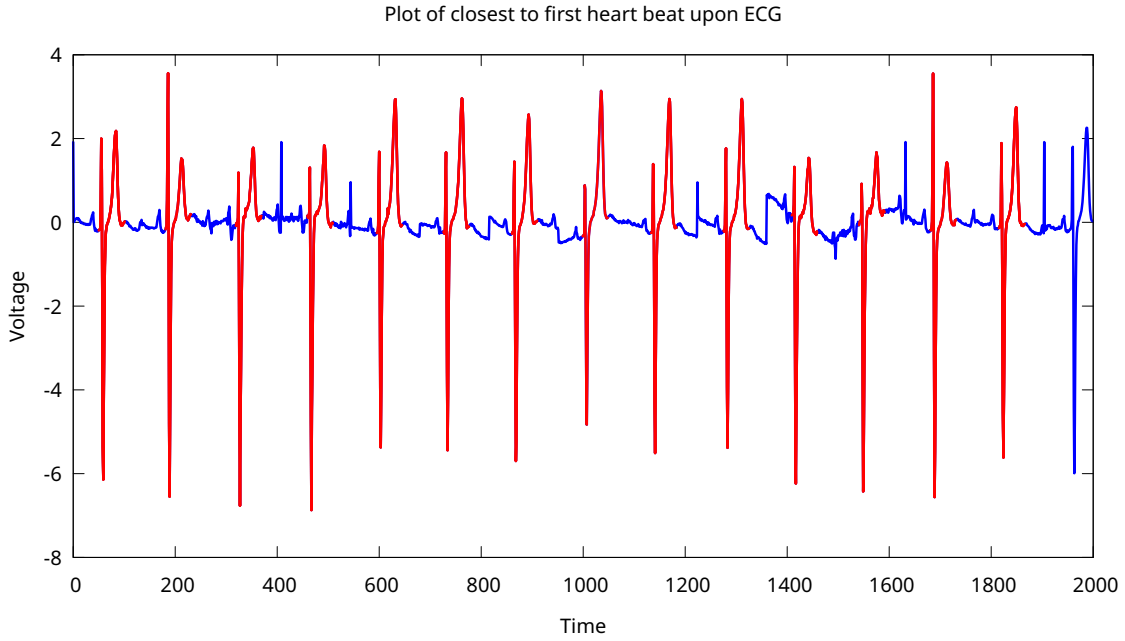


Figure 2.3: Example of Similarity Search employed to detect heart beats in an Electrocardiogram (ECG). Detected heart beats are shown in red whilst the series as a whole is drawn underneath in blue. Time here denotes the series index and not time in actual units. Query subsequence was the first heartbeat (index 51-101) and chosen epsilon was 7.0.

The naive approaches to these two problems are very similar but different in time complexities. The simple algorithm *Sequential Scan* passes a window of the same size as the query across the time series database and determines the euclidean distance between the query sequence and the target subsequence.

Sequence

0.1	0.6	1.1	0.4	-0.3	0.9	2.1	3.3	2.7	4.2	1.3	2.4

Query

0.8	2.4	3.3	2.9

Figure 2.4: Illustration of execution of sequential scan. Highlighted region represents the window passing over the larger time series.

In figure 2.4 the Euclidean Distance of the sequence within the window is:

$$\sqrt{(0.4-0.8)^2+(-0.3-2.4)^2+(0.9-3.3)^2+(2.1-2.9)^2}$$

To solve Similarity Search, only a single pass of the data is required, saving any subsequence that falls within ε of the query. This yields an $O(m \cdot n)$ solution as pseudocode will show and cannot be improved for a single query search. However, for a Time Series Database where many queries will be performed a faster algorithm is required. To guarantee that the sequential scan method finds the k closest subsequences, it can still do this in a single pass utilising a *priority queue* to maintain which subsequences were the closest.

Algorithm 2.2 Sequential Scan for Similarity Search

Require: $S[0..n], Q[0..m], \varepsilon \geq 0$
1: $i \leftarrow 0$
2: Q $\leftarrow \{\}$
3: **while** $i + m - 1 \leq n$ **do**
4: **if** $||S[i : i + m - 1] - Q||_2 \leq \varepsilon$ **then**
5: Q $\leftarrow$ Q$\cup\{i\}$
6: **end if**
7: $i \leftarrow i + 1$
8: **end while**
9: **return** Q

From Algorithm 2.2 it is apparent that with the right choice of data structure for Q, each element of S is accessed at most m times to determine the Euclidean Distance and each element of Q is accessed at most n times resulting in a time complexity of $O(m \cdot n)$.

Algorithm 2.3 Sequential Scan for K Nearest Neighbours

Require: $S[0..n], Q[0..m], k \in \mathbb{N}$
1: $i \leftarrow 0$
2: PriQ $\leftarrow \{\}$
3: **while** $i + m - 1 \leq n$ **do**
4: PriQ adds $(||S[i : i + m - 1] - Q||_2, i)$
5: $i \leftarrow i + 1$
6: **end while**
7: **return** PriQ$[0 : k - 1]$

In Algorithm 2.3 the analysis of the time complexity is similar to Similarity Search, but influenced by the data structure used to obtain the priority queue properties, as it is assumed that the k smallest elements lie at the beginning of the structure. The choice of a binary heap would grant a time complexity of $O(m \cdot n \log n)$

as it has a time complexity of $O(\log n)$ for insertion and $O(\log n)$ for retrieval of the top element.

2.4 Dimension Reduction Techniques

A Dimension Reduction Technique is a method to approximate a vector with a vector of a lower dimension and a method to convert that lower dimensional vector back to the higher dimension, preserving as much of the original point as possible. This looks like $f_{\text{reduce}} : \mathbb{R}^n \to \mathbb{R}^p, \quad x \mapsto \widehat{x}$ and $f_{\text{recover}} : \mathbb{R}^p \to \mathbb{R}^n, \quad \widehat{x} \mapsto \widetilde{x}$ such that x and $\widetilde{x} = f_{\text{recover}} \circ f_{\text{reduce}}(x)$ are close together.

For an object x in its original space $\mathbb{R}^n$ $\widehat{x}$ will denote its compressed form after application of a Dimension Reduction Technique and $\widetilde{x}$ denotes it after decompression. For example, taking the mean is a dimension reduction technique, $f_{\text{mean}}(S[0..n]) = \frac{1}{n+1} \sum_{i \in [n]} S[i]$. A sensible inverse for generating some approximation in $\mathbb{R}^n$ is $f_{\text{recover}}(\widehat{S}[0..0]) = \widehat{S}[0] \cdot \underline{1}$, where $\underline{1}$ denotes the vector where every entry is 1.

$$\langle 1, 2, 2, 3, 1 \rangle \longrightarrow \langle 1.8 \rangle \longrightarrow \langle 1.8, 1.8, 1.8, 1.8, 1.8 \rangle$$

Discretised versions of transforms (such as Fourier Transform) or functions that are *dense* in $\mathbb{R}^n$ (such as polynomials) form the basis of the Dimension Reduction Techniques. These can either be applied to approximate the whole function (global) or only to sections of the function (local).

The Discrete Fourier Transform is the classical Dimension Reduction Technique[3] and involves calculating the coefficients of the Fourier Transform to a certain point k. This can be accomplished in $O(n \log n)$ by the Fast Fourier Transform and has some advantageous properties in its compressed form that allow it to be used without using its inverse (these properties will be covered in section 2.5). Another global dimension reduction method is via the use of Chebyshev polynomials, which form a basis of functions in $\mathbb{R}^n$ and can be discretised to provide accurate approximations.

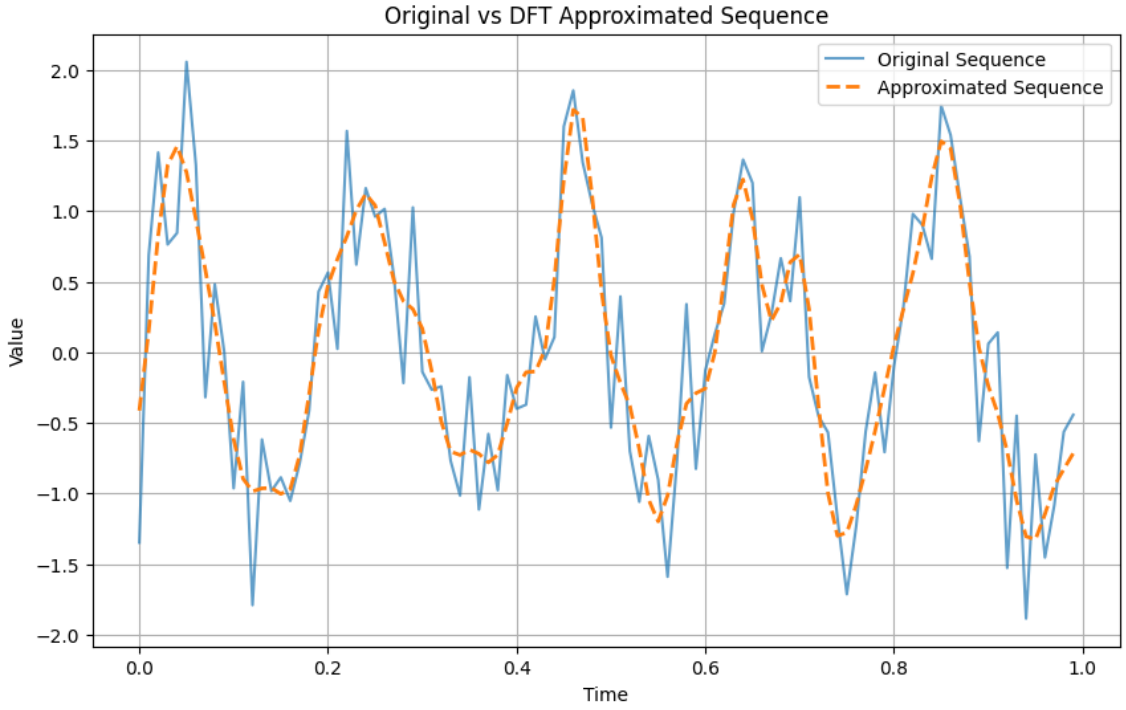


Figure 2.5: Approximation of an original sequence length 100 to a reduced dimension of 40 and then recovered to length 100 again. It closely matches the periodicity of the sampled function but struggles to fully capture the severity of the changes in the sequence.

Approximating Time Series Data with local adaptive methods often entails choosing classes of functions that require the fewest parameters to define. This is because given the same total number of parameters to use, a function that uses less parameters can be included more often. The two most cited uses of local DRTs are Piecewise Aggregate Approximation (PAA) and Piecewise Linear Approximation (PLA), where constant and linear functions are used respectively. Both algorithms assume a constant width of the intervals they approximate over.

Algorithm 2.4 Piecewise Aggregate Approximation

Require: $S[0..n], k \in \mathbb{N}$

1: $s \leftarrow \lceil \frac{n+1}{k} \rceil$ ▷ s is segment size
2: $\widehat{S} \leftarrow \{\}$
3: **for** $i \in \{s \cdot j | s \cdot j \leq n, j \in \mathbb{N}\}$ **do** ▷ Iterate on start positions of segments
4: $\quad \mu_i \leftarrow \frac{1}{s} \sum_{j=i}^{i+s-1} S[i]$ ▷ μ_i is mean of that segment
5: $\quad \widehat{S} \leftarrow \widehat{S} \cup \{\mu_i\}$
6: **end for**
7: **return** $\widehat{S}$

Algorithms 2.4 and 2.5 provide implementation details for the basic algorithms of local DRTs. As the sum and linear regression are both $O(n)$ operations, these algorithms only access each index of the data once. Both can be reconstructed into approximations of the original data via additionally storing the segment length. Detailed in algorithm 2.6 is the recovery of PLA, on line 4 DIV denotes division with the floor operation.

Algorithm 2.5 Piecewise Linear Approximation

Require: $S[0..n], k \in \mathbb{N}$

$s \leftarrow \lceil \frac{2 \cdot (n+1)}{k} \rceil$ ▷ Twice the parameters for function

$\widehat{S} \leftarrow \{\}$ ▷ $\implies$ double the segment length

for $i \in \{s \cdot j | s \cdot j \leq n, j \in \mathbb{N}\}$ **do** ▷ All multiples of s smaller than $n+1$

  $\alpha_i, \beta_i \leftarrow \text{regression}(S[i : i + s - 1])$ ▷ Find line of best fit

  $\widehat{S} \leftarrow \widehat{S} \cup \{(\alpha_i, \beta_i)\}$

end for

return $\widehat{S}$

Algorithm 2.6 Piecewise Linear Approximation Recovery

Require: $\widehat{S}[0..p] \in (\mathbb{R} \times \mathbb{R})^p, s \in \mathbb{N}, n \in \mathbb{N}$ ▷ s is segment length

$\widetilde{S} \leftarrow \{\}$

for $i = 0, .., n$ **do**

  $\alpha_j, \beta_j \leftarrow \widehat{S}[i \,\text{MOD}\, s]$ ▷ Find the segment i is in

  $\widetilde{S} \leftarrow \widetilde{S} \cup \{\alpha_j + \beta_j \cdot (i \,\text{DIV}\, s)\}$ ▷ Find the point along the line

end for

return $\widetilde{S}$

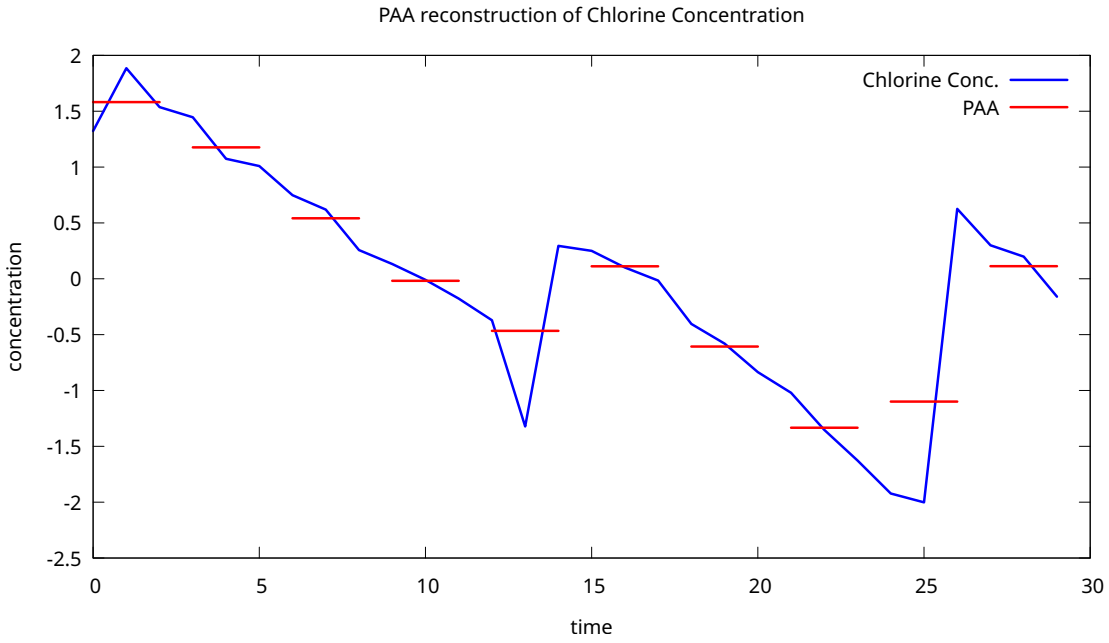


Figure 2.6: A time series generated by chlorine concentrations in a simulation and its approximation by PAA. The data holds strong trends that are not captured by the constant functions.

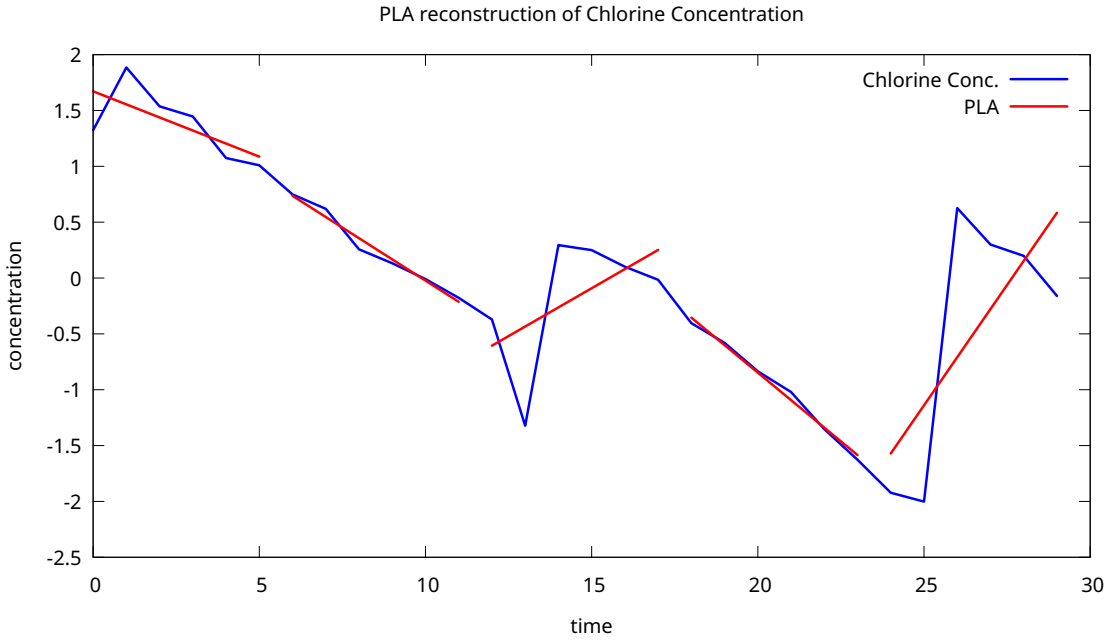


Figure 2.7: The time series generated by chlorine concentrations with an approximation instead by PLA. The data's trends are captured better but the requirement to have fixed length segments hampers the fit of many of the segments to the data.

2.5 The Gemini Framework

Faloutsos et al.[3] present a methodology for storing a time series database in an efficient manner to quickly compute many similarity search queries. It uses the assumption that there exists a minimum length for queries (denoted w). A sliding window is used across the time series data and upon each window the contiguous subsequence is mapped to a point in *feature space*. This is achieved using a Discrete Fourier Transform of each window and using only the first few coefficients. However, the authors note that any Dimension Reduction Technique that satisfies the *Lower Bound Lemma* could be used in similarity search. In this paper the Dimension Reduction Techniques will be Adaptive PLA algorithms that return a fixed number of segments.

The Lower Bound Lemma states that the distance function imposed on feature space should always lower bound the distance function in object space.

This means that for all series A and B of equal length it can be observed that:

$$\text{Dist}_{\text{DRT}}(\text{DRT}(A), \text{DRT}(B)) \leq \text{Dist}(A, B)$$

A *SubTrail-Index* was built that considered the DRT of each subsequence to be a point in feature space and found Minimum Bounding Rectangles (MBRs) to bound the areas of the sequence appropriately. These MBRs store contiguous sub-trails that are close in feature space and are inserted into a spatial indexing structure that allows for fast spatial queries.

Given a query Q of length w to find similar subsequences, it is converted to feature space and the spatial indexing structure returns the MBR's that are within ϵ of Q. The original sub-trails are retrieved and checked to be within ϵ of Q as the structure only guarantees *no false dismissals* and not *no false alarms*, meaning that

it will never miss a subsequence within ϵ of Q but may propose subsequences that aren't within ϵ of Q.

An inherent challenge of utilising local adaptive DRT methods within the Gemini Framework is the challenge of the feature space they map into. As an example, feature space of Adaptive Piecewise Constant Approximation (the adaptive equivalent of PAA) upon our random walk looks like the following.

$$-0.274873, 1, -1.22373, 5, -3.11681, 6, -4.41417, 10,$$
$$-6.71191, 22, -5.07245, 26, -6.71857, 30$$

Even indexes here hold the mean of the segment (the *value*), whilst odd indices hold the index of the last element in the segment. This asymmetry in feature space makes creating distance functions and applying local adaptive DRT's to the Gemini Framework a challenge as Minimum Bounding Rectangles over the *values* of the PCA make sense, but over the *indexes* is a greater challenge.

Chapter 3

Methodology

3.1 Objectives

The objectives are.

- Research novel PLA techniques to solve Similarity Search problem on Time Series Data.
- Research methods to improve PLA via optimisation techniques.
- Provide description and theoretical analysis of methods (time complexity or if time error bound guarantees).
- Design implementations of these methods.
- Test implementations on real and synthetic datasets. Provide reproducible benchmarks, test scalability and compare methods.
- Research indexing methods for adaptive PLA.
- Research and implement the GEMINI framework for PLA.

3.2 Research Methods

Algorithms for the problem of adaptive PLA were researched, and were evaluated on a diverse dataset of both real and synthetic sources, using a quantitative approach.

Algorithm Discovery was conducted by suggestions from my supervisor, previous reviews of algorithms from Keogh et al.[6], references from a tutorial at SIGMOD 2015[8] and using Google Scholar[9] and Semantic Scholar[10]. Research Databases were filtering to papers with references to Indexable PLA.

Optimisations Research Attempts were made to optimise the algorithms described in this report. Sometimes careful consideration of data structures led to improvements in time complexities, sometimes methods described in other papers could be used to create novel or hybrid techniques (for example, in the case of SAPLA [5]). However, testing also needed to be in place as theoretical time improvements didn't always lead to realised improvements (due to data locality and other underlying implementation details of data structures).

Evaluation Methodology is a purely quantitative approach, utilising the norms defined on vector spaces $\mathbb{R}^n$ to provide sensible measurements of distance and measurements like *pruning power* to measure its capabilities for indexing. There was an

emphasis on a diverse range of datasets to test the data on, due to earlier concerns about evaluations in Time Series Analysis by Keogh et al.[11]. Datasets used to test the algorithms were taken from The UCR Time Series Archive [12] and synthetic datasets generated from random walks. Figure 3.1 demonstrates the variety of datasets used from the UCR Times Series Archive, displaying periodic signals, signals with jumps and smooth continuous data.

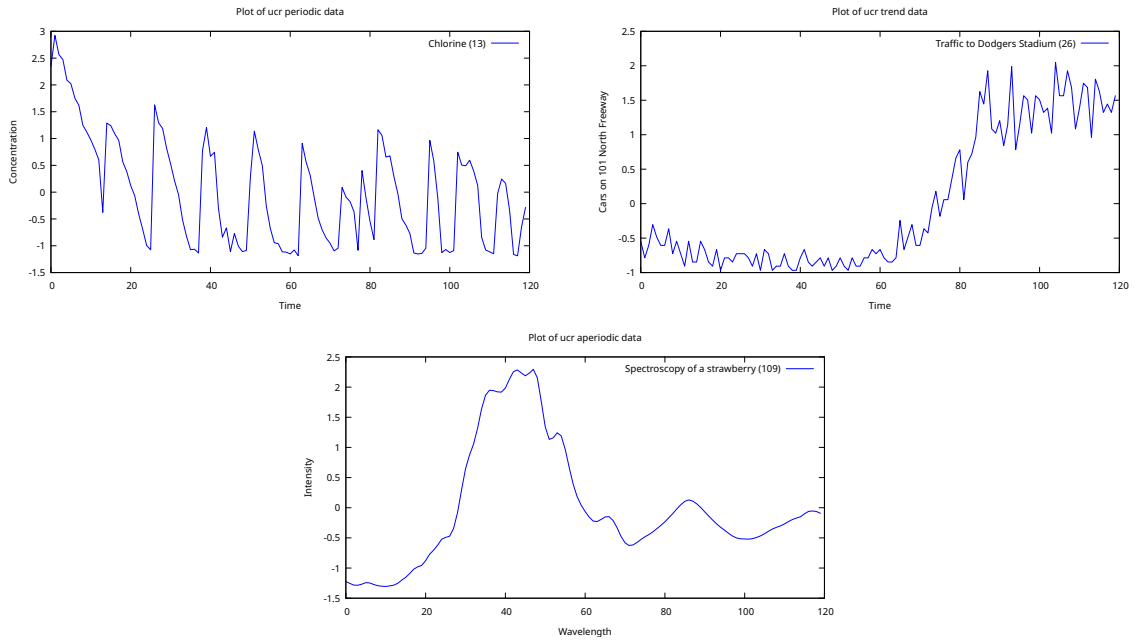


Figure 3.1: Plot of three datasets in the UCR collection (Top Left Chlorine Concentration, Top Right DodgerLoopDay, Bottom Strawberry).

3.3 Practical Methods

To ensure that components of this project like *data generation*, *Spatial Access Structures* and the GEMINI framework had adequate time to be built, an approach based off the Agile Values[13] was formed. Sprints of 1-2 weeks were allocated to a promising paper or algorithm to ensure strict deadlines and sufficient time for all aspects of the project. Focus was placed upon *Working Code*, *Frequent Additions*, *Attention to Detail* and *Simplicity of code*, to rapidly develop fast algorithms with clarity.

Therefore, the following process for researching and developing algorithms was used.

1. Research Paper.
2. Parse pseudocode.
3. Provide C++ implementation from my pseudocode.
4. Inspect implementation on small sections of data.
5. On failure, determine the underlying cause within my pseudocode.
6. On success perform a quick analysis of potential optimisations.
7. Create a quick demonstration of algorithm for discussion with supervisor.

3.3.1 Codebase Structure

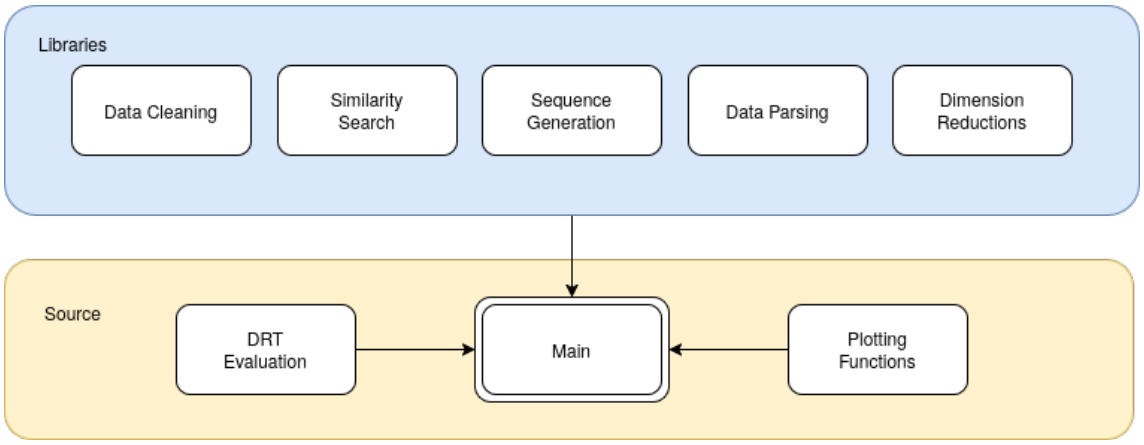


Figure 3.2: An overview of the codebase.

The project is primarily a comparison of algorithms and not meant to be a single application, so the codebase is designed to be a modular suite of tools for implementation and evaluation of algorithms. Libraries are used to keep code separated for clarity and modularity and encourage future adoption by others interested in PLA, see Figure 3.2.

3.3.2 Software Used

- **Nix:** Provides a declarative method to package management and enabled this project to quickly pivot libraries if needed without the risks of major downtime in installation. There is no need for another user to manually install the required libraries. Nix enables the following:
  - **Immediate Deployment:** With Nix and Flakes installed, running *nix develop* installs all requirements into a temporary shell without polluting your system. Without Nix and Flakes installed, the *flakes.nix* file still provides an immediate aid to what packages need to be installed.
  - **C++ Package Management:** C++ is renowned for difficulties in installing libraries and packages correctly. Nix provides a solution to this, allowing modification of a single file to offer libraries as if they were installed system wide.
  - **Cross Platform Support:** The package repository has variants of packages for many systems, enabling development on potentially many more machines.
  - **No Breaking Changes:** Nix Flakes offer the opportunity to lock the exact version of packages installed for the duration of the project and keep track of those editions in *flake.lock* ensuring myself and any future installations won't break due to updates to the libraries used.
- **CMake:** A software build system used primarily for C++, abstracted from direct compilation and Make, CMake allows for complex compilation processes to be defined tersely and support for mulitple builds to occur at once, powering the multi-library approach.

- **Git:** Version Control software allowing changes to be tracked and new experimental features to be tested in separate branches, where damaging changes can be reversed. The project codebase is also hosted on GitHub where others can go through the history of changes.
- **C++:** A language for programming fast algorithms with abilities to directly reference memory, allowing potentially slow algorithms on large datasets run quickly.

Chapter 4

Current Literature

The following sections describe papers with their key findings and contributions to the report.

4.1 Indexing Local DRTs

Indexable PAA 2001: Exploration into the approaches of local DRT's for the Similarity Search and K-NN problems only began after the seminal paper "Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases" by Keogh et al.[14] in 2001. This paper proved the lower bound existed for PAA representations of a time series, hence its suitability for the GEMINI framework and introduced a new value P that represented a view of the 'pruning power' of the Dimension Reduction Technique with indexing scheme. P represents the number of objects that need to be viewed to guarantee a 1-NN match to a query point.

$$P = \frac{\text{Number of points checked}}{\text{Number of points in database}}$$

This paper sparked interest in local dimension reduction techniques and provides the tools for evaluation of indexing schemes.

Summary of key developments:

- Presents the DRT Piecewise Aggregate Approximation.
- Presents optimisation of PAA to $O(Nm)$ time complexity.
- Proves Indexing Scheme for PAA representation.
- Provides pruning power measurement.
- Provides evaluation of PAA against other DRT methods.

Indexing scheme: For time series $S, T \in \mathbb{R}^n$ and number of segments $k \in \mathbb{N}$, with Dimension Reductions $S \to \widehat{S}, T \to \widehat{T}$ such that $\widehat{S}, \widehat{T} \in \mathbb{R}^k$.

$$D_{\text{PAA}}(\widehat{S}, \widehat{T}) := \sqrt{\frac{n}{k} \sum_{i=0}^{k-1} (\widehat{S}[i] - \widehat{T}[i])^2} \tag{4.1}$$

$$D_{\text{PAA}}(\widehat{S}, \widehat{T}) \leq ||S - T||_2 \tag{4.2}$$

Indexable PLA 2007: In "Indexable PLA for Efficient Similarity Search" Chen et al.[15] present a method to use PLA for indexing in the GEMINI framework. The paper presents a distance measure that satisfies the lower bounding lemma alongside optimised comparisons between compressed sequences and Minimum Bounding Rectangles.

Summary of key developments:

- Presents a distance measure for PLA space satisfying Lower Bounding Lemma.
- Proves Indexing Scheme for PLA representation.
- Presents tight distance measure between compressed sequence and MBR.
- Provides evaluation of PLA indexing scheme against APCA and Chebyshev Polynomials

Indexing scheme: For time series $S, T \in \mathbb{R}^n$ and number of segments $k \in \mathbb{N}$, with Dimension Reductions $S \to \widehat{S}, T \to \widehat{T}$ such that $\widehat{S}, \widehat{T} \in \mathbb{R}^{2k}$ and $\widehat{S} = \langle a_0, b_0, a_1, b_1, .., a_{k-1}, b_{k-1} \rangle, \widehat{T} = \langle a'_0, b'_0, a'_1, b'_1, .., a'_{k-1}, b'_{k-1} \rangle$. The length of a segment is $l = \lceil \frac{n}{k} \rceil$. The following was shown:

$$D_{\text{PLA}}(\widehat{S}, \widehat{T}) := \sqrt{\sum_{i=0}^{n-1} (a_{\lfloor \frac{i}{l} \rfloor} - a'_{\lfloor \frac{i}{l} \rfloor} + (i \,\text{MOD}\, l) \cdot (b_{\lfloor \frac{i}{l} \rfloor} - b'_{\lfloor \frac{i}{l} \rfloor}))^2} \tag{4.3}$$

$$D_{\text{PLA}}(\widehat{S}, \widehat{T}) \leq ||S - T||_2 \tag{4.4}$$

Both of the equations for distance in reduced space have intuitive underlying geometric interpretations. The two distances are the euclidean distance of the approximate sequences once uncompressed into $\mathbb{R}^n$ (ie. $||\widetilde{S} - \widetilde{T}||_2 \leq ||S - T||_2$). Figures 4.1 and 4.2 show the distance measures are the square root of the sum of the square distances.

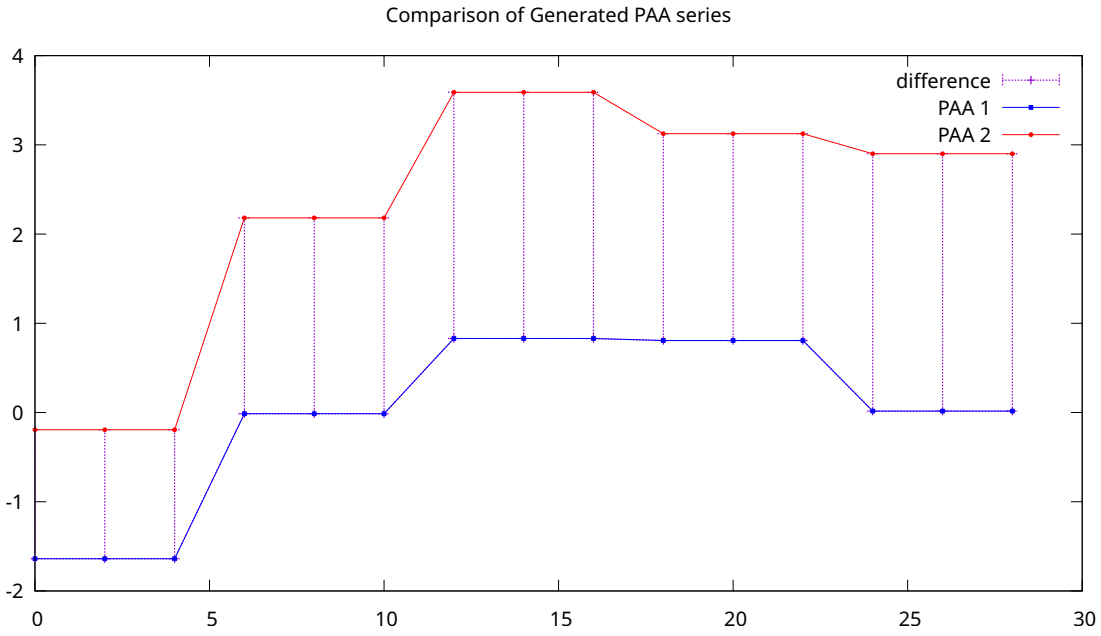


Figure 4.1: Comparison of generated PAA series with the distances between each observation of the sequences denoted by a dotted line.

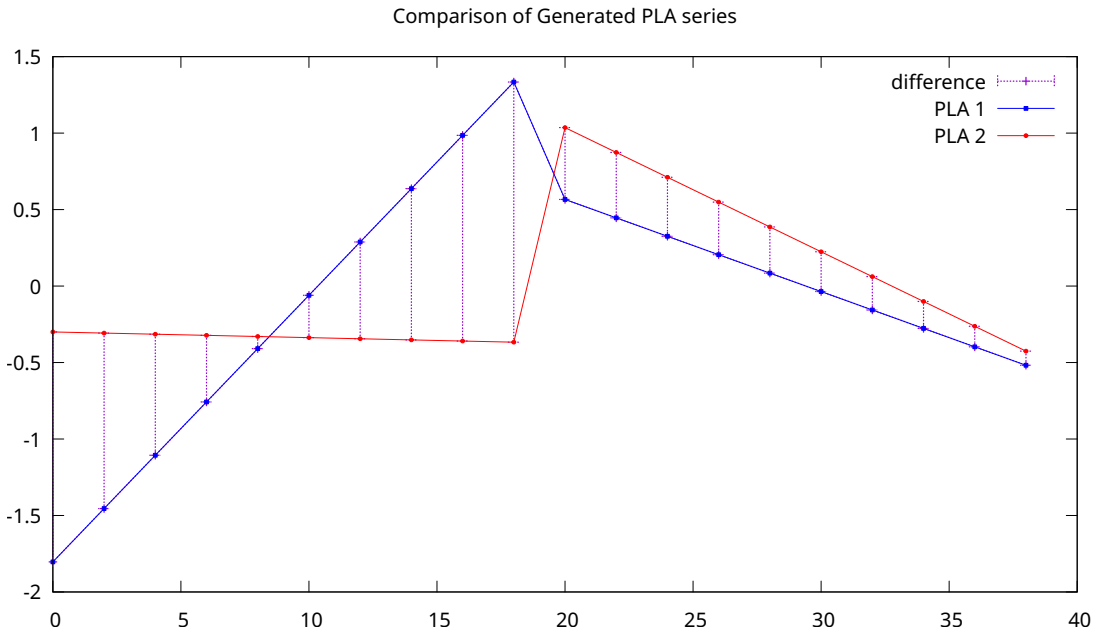


Figure 4.2: Comparison of generated PLA series with the distances between each observation of the sequences denoted by a dotted line.

4.2 Indexing Adaptive DRTs

The paper "Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases" by Chakrabarti et al.[4] presents Adaptive Piecewise Constant Approximation (APCA). This is an approximation method that uses Discrete Wavelet Transform with the Haar mother wavelet to approximate a sequence by linear step functions. Although the paper does not describe a distance measure that preserves the lower bounding property it still suggests a method for indexing via adaptive length algorithms. This is achieved via a weaker lower bounding property that still attains no false dismissals.

Weak Lower Bounding Property: For time series $S, T \in \mathbb{R}^n$ with $\widehat{S} \in \mathbb{R}^k$ being a compressed approximation of S, then we say distance measure $D : \mathbb{R}^k \times \mathbb{R}^n \to \mathbb{R}_{\geq 0}$ has the *Weak Lower Bounding Property* if $D(\widehat{S}, T) \leq ||S - T||_2$.

Summary of key developments:

- A new Dimension Reduction Technique, APCA.
- A distance measure from an uncompressed series to a compressed on or MBR.
- A methodology to merge compressed points into MBRs, maintaining the Weak Lower Bound.
- An indexing scheme for an adaptive segment length DRT.
- Evaluations against other methodologies in terms of pruning power and CPU time.

4.3 Exact Adaptive PLA

The paper "APLA: Indexing Arbitrary Probability Distributions" by Ljosa and Singh[7] provides the template for a dynamic programming approach to APLA that yields the exact answer via memoisation of a 2D-array. For time series $S \in \mathbb{R}^n$ and approximating by an adaptive PLA representation of k segments, it forms a $n \times k$ array where Table$[i, j]$ represents the best approximation of the sequence $S[0..i]$ using j segments. This is accomplished by having Table$[i, j]$ store the beginning index of the last linear segment and the total error of the approximation on $S[0..i]$.

APLA provides the following in its paper:

- A dynamic programming approach that can be modified to general APLA.
- A generalisation of Adaptive PLA for indexing that can handle probabilistic queries.

4.4 Heuristic PLA

PLA Investigation 2001: The first analysis of various methods for Piecewise Linear Approximation (or Piecewise Linear Representation in the paper) was conducted in "An Online Algorithm for Segmenting Time Series" by Keogh et al.[6]. This reviewed 3 previous algorithms, Sliding Window, Top Down and Bottom Up and introduced a fourth algorithm. Their analysis suggested that local methods like Sliding Window performed less favourably than methods that utilised the whole sequence at once. The fourth algorithm took a hybrid approach using the efficiency of sliding window with the accuracy of merging techniques.

Summary of key developments:

- Introduction and analysis of Sliding Window, Top-Down and Bottom-Up.
- Introduction of new algorithm.
- Evaluation of all 4 methods for both speed and accuracy.

SAPLA 2022: "An Indexable Time Series Dimensionality Reduction Method for Maximum Deviation Reduction and Similarity Search" by Xue et al.[5] presents a more complex algorithm based on multiple passes of the data to improve an approximation. The algorithm SAPLA adopts three passes; *Initialisation*, where a methodology such as Sliding Windows is used to estimate good segments for linear approximation; *Split and Merge*, to convert the approximation to the right dimensions required, and finally *Segment Endpoint Movement Iteration* which shift the endpoints of the segments until a minima of error is reached. The algorithm obtains a competitive $O(n \cdot (k + \log n)$ run-time and introduces a distance measure that obeys the lower bounding lemma. However the distance measure for two objects in the lower dimension *feature space* fragments the Adaptive PLA representations and therefore requires access to the original sequences.

Summary of key developments:

- The adoption of a error based approach refined by split and merge.
- The presentation of optimisation methods for existing Adaptive PLA methods.
- An algorithm with comparable results to *OPTIMAL* from APLA.

- A distance measure that obtains the *Lower Bounding Property*.
- A data structure capable of solving Similarity Search.

4.5 Streaming Adaptive PLA

Situations when data isn't fixed and more can be appended over time are called *streams*. This requires either recomputing the entire approximation, or utilising an approach that only considers modifying some small fraction of the approximation to accommodate the new points. Whilst Top Down and Bottom Up approaches accommodate new points, Sliding Window algorithms have the advantage that the window can be restarted at the point when it first includes new data, maintaining all the results before it.

"Online Piece-wise Linear Approximation of Numerical Streams with Precision Guarantees" by Elmeleegy et al.[16] describes algorithms based on Sliding Window approaches for streams. Precision Guarantees refer to the property that all points are within ε of the approximation.

The paper proposes two kinds of *filters*, that pass along the dataset, adding points to a segment until adding the next would violate the error guarantee, instantiating a new segment at that point. The filters maintain a set of viable lines that stay within ε of all of the points so far and attempt to add the subsequent point to the current segment by re-evaluating the set of viable lines. When this set is empty it indicates that adding this point would form a segment that violates the guarantee.

The first filter proposed is *SWING*, that maintains that line segments share endpoints and together form a continuous line of linear segments.

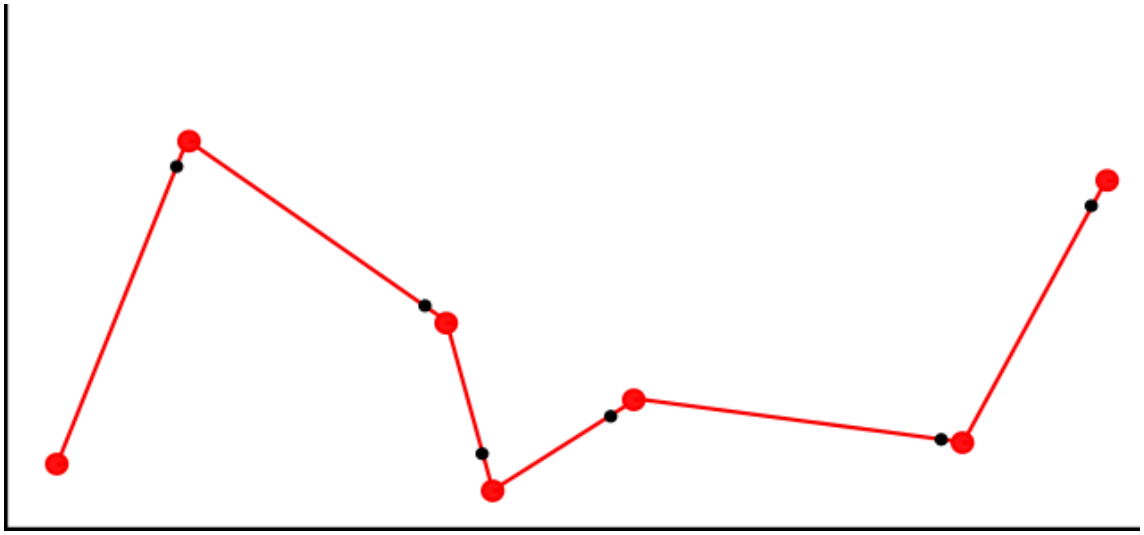


Figure 4.3: Illustrates of how segments 'share' endpoints.

In Figure 4.3 the red line shows the shape of the approximation as an unbroken line, the red dots denote the start and end of each linear segment and the black dots denote the endpoints according to the traditional Adaptive PLA format. The black dots occur one index before the next red dot.

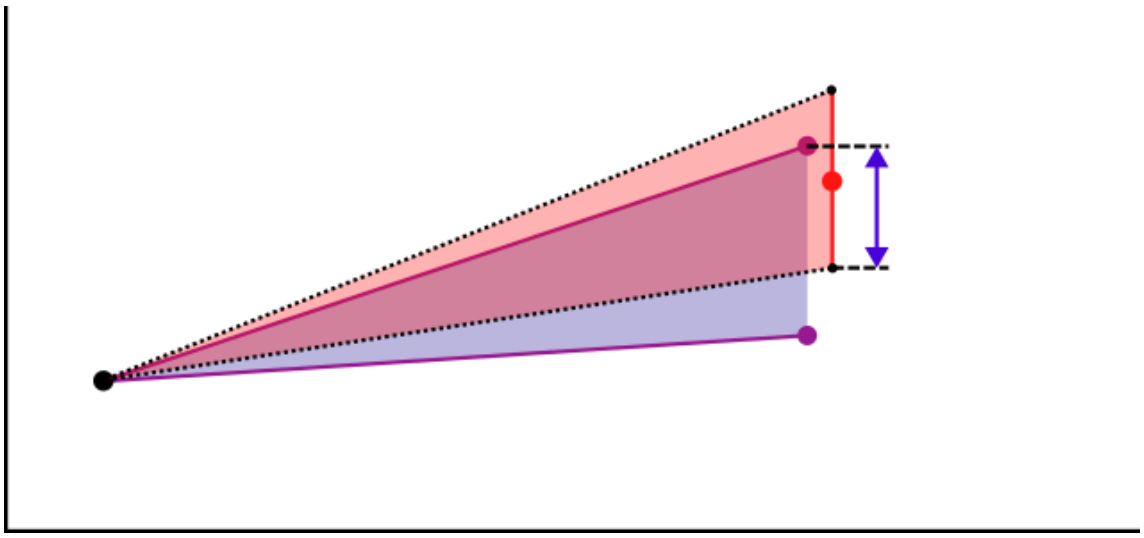


Figure 4.4: Illustrates how the interval is updated for new points.

SWING maintains a set of lines going through a start point (the left black dot of Figure 4.5) that are all within ε of any point in the segment (denoted by the blue area). To include the next data point (the red point of Figure 4.5), the set of all lines starting at the same start point and within ε of the next point are intersected with the previous set of lines. This intersection (the blue arrow) contains all lines within ε of all previous points and within ε of the next point. If this set is non-empty then the next point is included into the segment, otherwise a new segment is created.

The second filter SLIDE is more complex, as it relaxes the constraint that the lines must be contiguous (start at the previous lines endpoint) and therefore maintains a larger set of potential lines. The next data point is again added to the segment if the set of lines within ε of the next point and all prior is not empty.

Summary of key developments:

- Two algorithms based on Sliding Window approaches forming Adaptive PLA representations.
- ε precision guarantees even for streaming scenarios.
- Evaluations against the Sliding Window algorithm of "An Online Algorithm for Segmenting Time Series"[6].

4.6 Comparison of existing algorithms

	Advantages	Disadvantages
Piecewise Aggregate Approximation	Less storage per segment allows more segments	Models trends poorly as cannot capture gradients
	Simple to implement and fast execution	Models data with large jumps poorly
	Indexable	
Piecewise Linear Approximation	Linear approximations often approximate much better constant functions	Fixed length segments mean it models data with jumps poorly
	Simple to implement and fast execution	
	Indexable	
Adaptive Piecewise Constant Approximation	Adaptive segments approximate time series better	Models trends poorly as cannot capture gradients
	Indexable	
Exact Adaptive Piecewise Linear Approximation	Often much more accurate than non Adaptive PLA methods	Steep time and space complexity hurt for high dimensional data
Top Down	Simple algorithm to implement	Can use too many segments under certain conditions
	Can be extended to parallel computing	
Bottom Up	Algorithm often uses few segments for approximation	Requires data locality making implementation challenging
	Implementation with Priority Queue improves time complexity	
Sliding Window	Simple to implement and fast execution	Can use too many segments for data that isn't smooth
SAPLA	Accurate Adaptive PLA algorithm similar results to Exact	Requires choice of epsilon which may vary accuracy of approximation
	Allows use of best precision guarantee algorithms	
SLIDE Filter	Improves Sliding Window	Calculation of Convex Hull and lines makes implementation a challenge
	Implementation with Convex Hull improves speed	
SWING Filter	Improves Sliding Window	Fixed start point hinders approximation
	Fast to calculate for a time series	

Figure 4.5: Advantages and Disadvantages of algorithms discussed in Current Literature.

Chapter 5

Generating Time Series Data

This chapter offers explanations and analysis of the methods used and implemented within the *Data Parsing*, *Data Cleaning* and *Sequence Generation* libraries of the project. This includes the methods to parse the data within the UCR Time Series Archive, data structures used in implementation, data cleaning processes and synthetic data generation.

Sequences were chosen to be stored as dynamic array's of double precision floating point numbers, in C++ denoted as `std::vector<double>`. It was realised that Double precision was necessary for some some arithmetic algorithms (those in Chapter 9) and C++'s vectors offer several valuable properties, including iterators, integration with the rest of the standard library and contiguous memory usage allowing for direct memory access.

5.1 Data Parsing

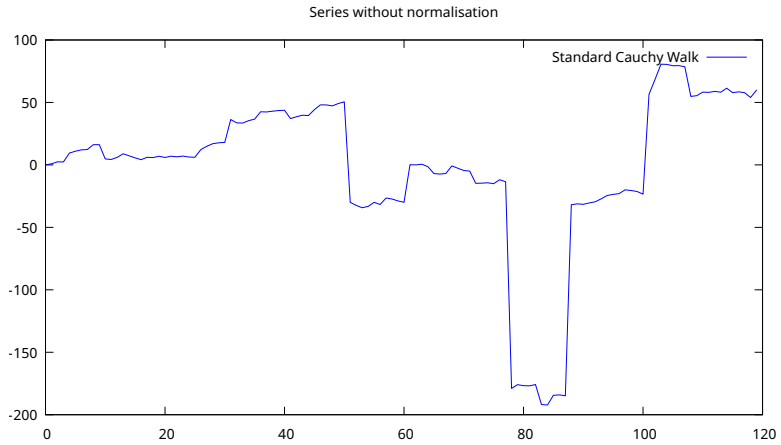
The UCR Time Series Archive provides time series data for analysis and classification. Data is separated into two files TRAIN and TEST for classification purposes. Data is stored in a Tab Separated Value[1] format, where series values are separated by tabs or new-lines. Every line starts with a value representing the current class the line of values belongs to. Some files contain NaN values, which were removed as approximation algorithms are designed to work on general time series data. Algorithm 5.1 shows pseudocode of the parsing process to access the UCR Time Series Archive.

Algorithm 5.1 TSV Parsing

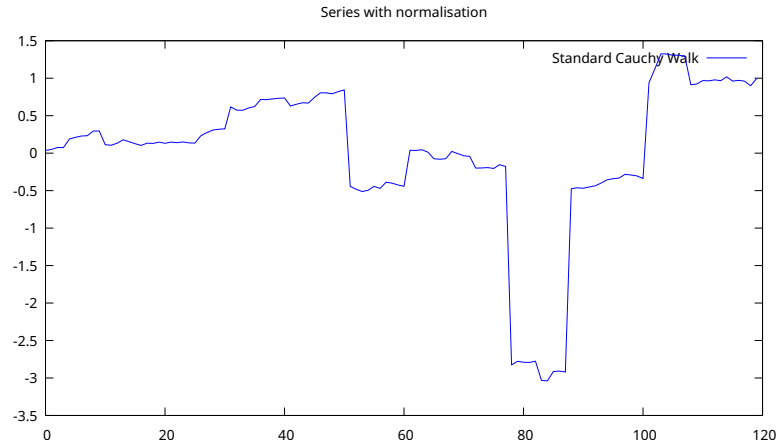
Require: FILE
1: seq ← ∅
2: **for** LINE in FILE **do**
3: VALS ← split(LINE, "\t") ▷ split function turns string into list
4: **for** s in VALS[1:] **do** ▷ ignore the first value
5: **if** s ≠ "" **and** s ≠ "NaN" **then** ▷ ignore NaN values
6: seq ← seq∪{float(s)}
7: **end if**
8: **end for**
9: **end for**
10: **return** seq

[1] https://en.wikipedia.org/wiki/Tab-separated_values

Z-Normalisation is a crucial component to ensure that evaluations made by this project remain fair and comparable with other published results[11]. All datasets have been z-normalised and the process is described in Fair Comparisons. Figure 5.1 shows the effects of the process. The data is translated such that its mean lies at 0 (image (b) is subtly shifted from starting at 0, demonstrating this) and scaled such that the variance of the sequence is 1, so around 60% lies in the range $[-1, 1]$. In the case of the Cauchy walk, its fat tails result in significant scaling observable by the scale of the y-axis.



(a) Random walk without z-normalisation.



(b) Random walk with z-normalisation.

Figure 5.1: Random walk generated from repeated addition of Random Variables distributed by Cauchy Distributions, $a = 0.0, b = 1.0$ (with seed value of 7, `std::cauchy_distribution`). The fat-tails of the Cauchy distribution cause noticeable "jumps" in random walks.

5.2 Data Generation

Data generation was achieved by simulating random walks. This is a stochastic process where a summation of random variables is considered by its partial sums. The classic example of a random walk is the repeated addition of variables from the Radermacher distribution, that takes value +1 with probability 0.5 and -1 with probability 0.5, independent of all other random variables. The probability mass

function is given below.

$$p(x) = \begin{cases} \frac{1}{2} & x = -1 \\ \frac{1}{2} & x = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{Given} \quad X_1, X_2, .., X_n \overset{\text{iid}}{\sim} \text{RaderMacher}$$

$$W_0 = 0, \quad W_k = \sum_{i=1}^{k} X_i$$

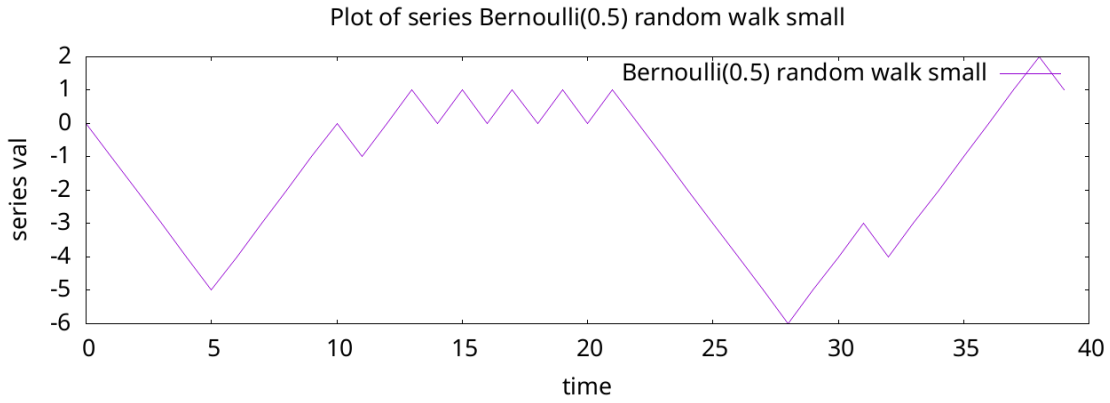


Figure 5.2: Plot of small random walk using RaderMacher variables.

Random walks were implemented to be used with any provided random variable using *Functors*, classes in C++ that overwrite usage of () operator, allowing instances of them to be run as if they were functions, whilst maintaining state. Functors neatly describe repeated trials of a random experiment and can be configured to store the seed value for a pseudo random generator in their state. This seed value can be provided in the report allowing exact reproduction of the trials. Functors have been implemented for RaderMacher, Normal and Cauchy distributions (with full customisation of parameters and seed values) as each offer valuable random walks.

RaderMacher random walks are archetypal and at larger values of n simulate a normal random walk, due to the Central Limit Theorem (and being *sub-Gaussian* although this won't be discussed in this report).

Normal random walks are useful for modelling *typical* conditions of stock markets and hence are used regularly for Time Series Analysis. Qualitative studies have shown that industry and research professionals cannot tell the difference between actual stock movements and simulations ([11]), justifying their use.

Cauchy random walks see modelling uses due to the property of *fat tails*, meaning that a sizeable proportion of probability mass lies in the tail of the distribution and it doesn't decay exponentially to 0. This lends itself to modelling extreme values and areas of high volatility, and was show in Figure 5.1, where despite typically drawing values in range $[-1, 1]$, the distribution is still likely to sample values in hundreds or even thousands.

Chapter 6

Exact Segment Algorithms

The majority of algorithms for Adaptive PLA split the data into linear segments with some precision guarantee (ie. all segments have any estimate within ε of the actual value), instead of being constrained by the number of segments [17],[18] [6],[19],[16] [5]. Only APLA[7] offered a method constrained only by the number of segments allowed, using Dynamic Programming to obtain an exact solution.

6.1 Sliding Window Approaches

The project saw some attempts to create some new approximations via a desired number of segments. The approaches relied primarily on Sliding Window methods. This involved a Double Sliding Window (a window of two parts that traverse the data together), evaluating values based on the linear approximation they formed on that window of the segment. A score is assigned to every double window, a larger score indicates a poorer approximation of the full double window by linear interpolation.

Figure 6.1: Illustrates the Double Sliding Window approach. The blue and red highlighted areas indicate the two windows traversing the sequence together.

To assess how the windows would form a linear approximation, the slope (difference between the current value and the previous) was calculated for each value in the sequence. This sequence of value changes is ΔS, indexable (for $S[0:n]$) by $\Delta S[1:n]$. Slope is a simplification of gradient where it is assumed that each point lies one unit of time after the previous.

$$\Delta S[i] = S[i] - S[i-1]$$

From the sequence S in Figure 6.1, the slope reductions are ΔS, with the values in the blue sliding window $\langle 0.1, 1.1, -3.2 \rangle$ and values in the red sliding window $\langle 1.5, 2.1, 5.6 \rangle$.

$$S = \langle 0.0, 0.1, 1.2, -2.0, -0.5, 1.6, 7.2, 8.9, -1.2, -9.0 \rangle$$
$$\Delta S = \langle 0.1, 1.1, -3.2, 1.5, 2.1, 5.6, 1.7, -10.1, -7.8 \rangle$$

The algorithms developed find partitions of ΔS that ensure that the values of each segment are as similar as possible. Performing this with the Double Sliding

Window was achieved by Interval Projection and Average Value to score the 'similarity' between the two windows.

- **Interval Projection** scores a double window by the difference in maximum range of slopes in the full double window and the maximum range of slopes in the first window.
- **Average Value** scores a double window by the difference in the average value of slopes in the second window and the average value of slopes in the first window.

Comparisons between Interval Projection and Average Value showed Average Value performed better on the majority of datasets but both performed poorly compared to fixed segment length algorithms (Piecewise Aggregate Approximation and Piecewise Linear Approximation). Hence weighted average values were considered to allow further modifications with alternative hypotheses to test for improvements.

- **Weighted Average Value** is a generalisation of Average Value, that weighted every element of the window separately.
  - **Increasing Weights**. Values could be weighted differently based on how close they were to the split. This would skew the score so values close to the potential split have more impact.
  - **Skipped Second Window Slope**. 0 is assigned to the first value of the second window as upon being split, it doesn't measure the gradient of the linear approximation of the window.

Algorithm 6.1 Sliding Window Algorithm

Require: $lw \in \mathbb{N}, rw \in \mathbb{N}$ ▷ lw and rw are size of left and right window

function SlidingWindow($S[0 : n-1]$, $N \in \mathbb{N}$)
  segments $\leftarrow \{[0 : n-1]\}$
  while size(segments) $< N$ **do**
    for segment $[i : j]$ in segments **do**
      Find $y_{[i:j]}$ in segment $[i : j]$ maximising
        mergeCost($\Delta S[y_{[i:j]} : y_{[i:j]} + lw - 1]$
          $, \Delta S[y_{[i:j]} + lw : y_{[i:j]} + lw + rw - 1]$)
      if $\nexists y \in [i : j] : y + lw + rw - 1 \leq j$ **then**
        $y_{[i:j]} \leftarrow -1$
      end if
    end for
    let $y_{[k:l]} \leftarrow max_{[i:j] \in \text{segments}}(y_{[i:j]})$
    if $y_{[k:l]} = -1$ **then**
      return segments
    end if
    segments $\leftarrow$ segments $-\{[k : l]\}$
    segments $\leftarrow$ segments $\cup\{[k : y_{[i:j]} + lw - 1], [y_{[i:j]} + lw, l]\}$
  end while
  return segments
end function

Algorithm 6.1 shows a higher level pseudocode of the operation of the sliding window algorithm. The algorithm uses a general function *mergeCost* that uses either **Interval Projection** or **Average Value** to assign a *score* to merging the two segments defined within the window ranges.

Halting Algorithm 6.1 will halt.

Either the input is valid (meaning at no point do all intervals get assigned merge score -1) or the input is invalid (meaning there will be a point where all intervals get assigned merge score -1) .

If the maximum merge cost is always ≥ 0, one segment in *segments* gets split in two, increasing the size of *segments* by 1. This halts when *size(segments)* $\geq N$. The *for* loop on line 5 and the search (*Find ... maximising*) on line 6 operate on a finite set so the algorithm halts.

When the maximum merge cost is not well defined, the algorithm immediately halts and returns the current state of segments.

Correctness With a valid input, Algorithm 6.1 halts with an approximation of N segments.

If the maximum merge cost is always ≥ 0 then one segment in *segments* gets split in two until the size of *segments* equals N. The algorithm halts with an approximation of N segments.

Complexity Analysis The algorithm has a time complexity of $O(n \cdot N)$.

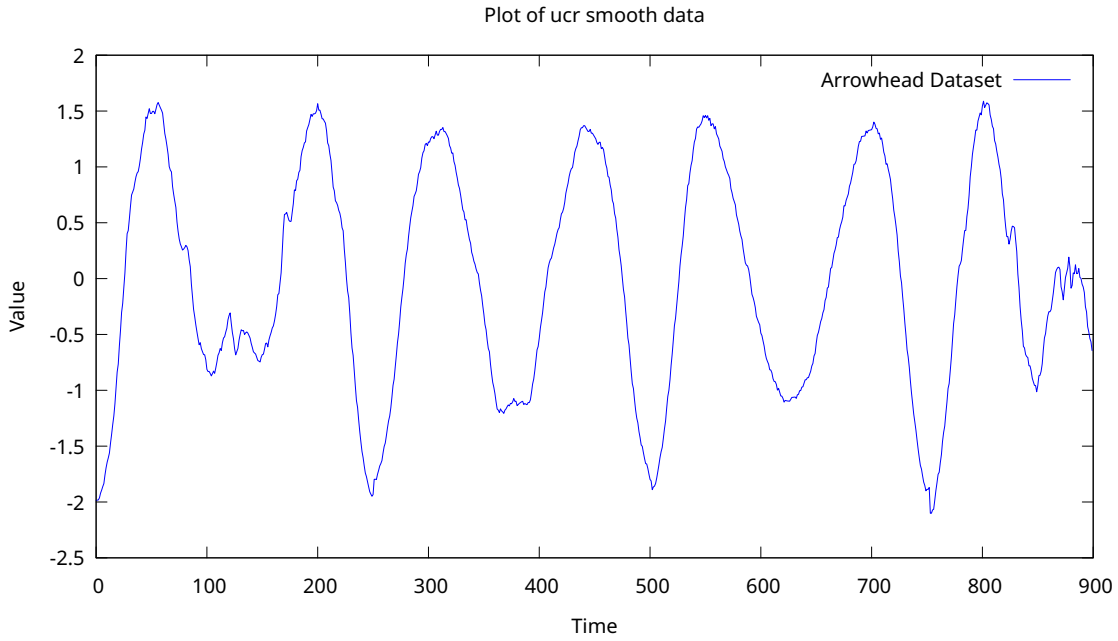
The *while* loop on line 3 iterates N times. On every iteration every entry in the sequence is accessed $lw + rw$ times (constant in terms of inputs) to find the best split index ($y_{[i:j]}$) of a segment. The algorithm runs in time $O(N \cdot n) = O(nN)$.

This time complexity is *tight*; by requesting $\frac{n}{2}$ segments with $lw = rw = 1$, causing the algorithm to terminate with a sequence broken into pairs. This requires $\frac{n}{2}$ iterations of scanning at least $\frac{n}{2}$ elements in the sequence, yielding an upper bound of at least $\frac{n}{4}$ memory read calls.

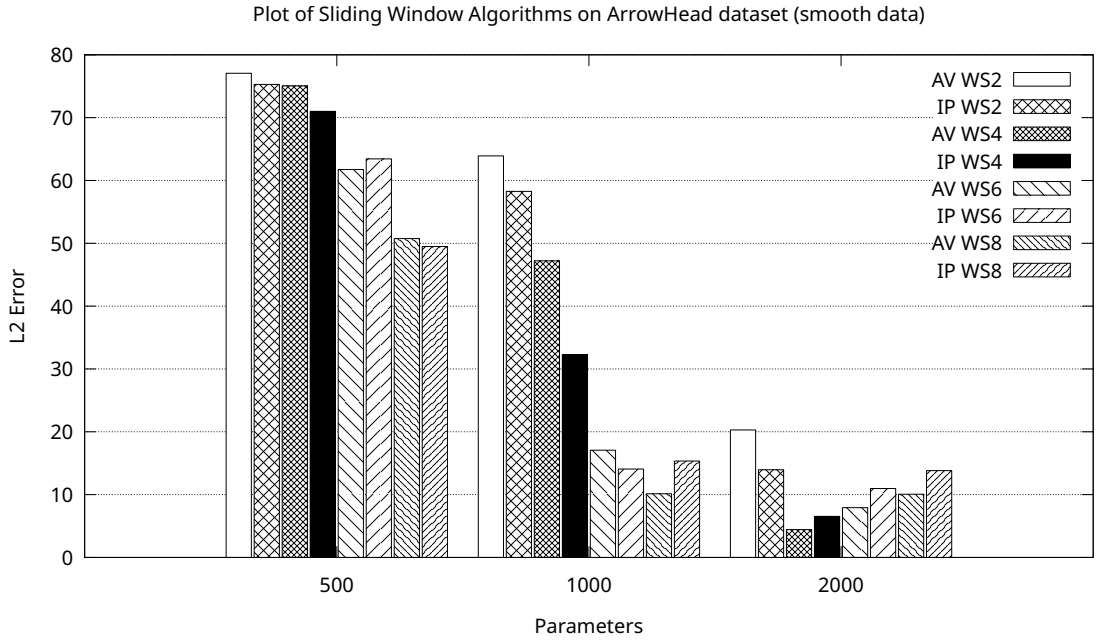
The algorithm has a spatial complexity of $O(n)$.

The algorithms uses two structures that aren't constant space. The first, *segments*, holds the information about the current partition and holds at most $O(n)$ values as at maximum size, every part of a partition must still hold a unique point. The second is ΔS, but this is notational convenience for the pseudocode, calculating a value of ΔS from S is constant time, so in implementation, ΔS does no need to be calculated beforehand. This yields the spatial complexity of $O(n)$.

Figure 6.2 shows larger window sizes improves approximation. Interval Projection outperforms Average Value, bars for Interval Projection (IP) are lower than Average Value (AV) showing that it outperforms on smooth data. For rough data, shown in Figure 6.3, larger window sizes have little impact on approximation. Average Value outperforms Interval Projection as bars for Average Value (AV) are lower than Interval Projection (IP) showing that Average Value outperforms on rough data.

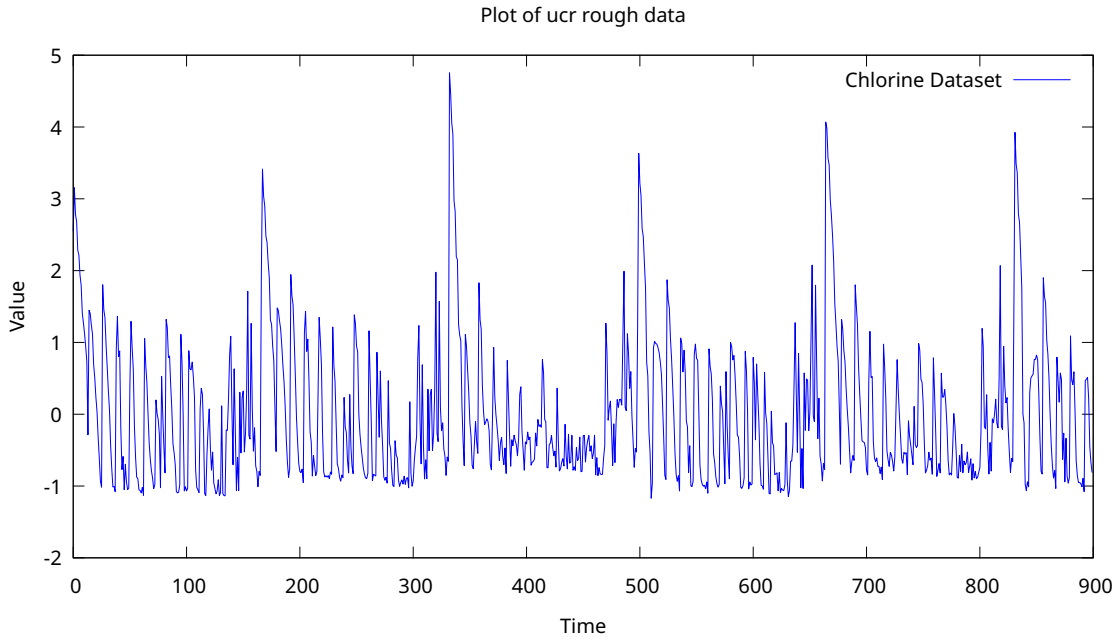


(a) First 900 entries in Arrowhead dataset, displaying smooth properties.

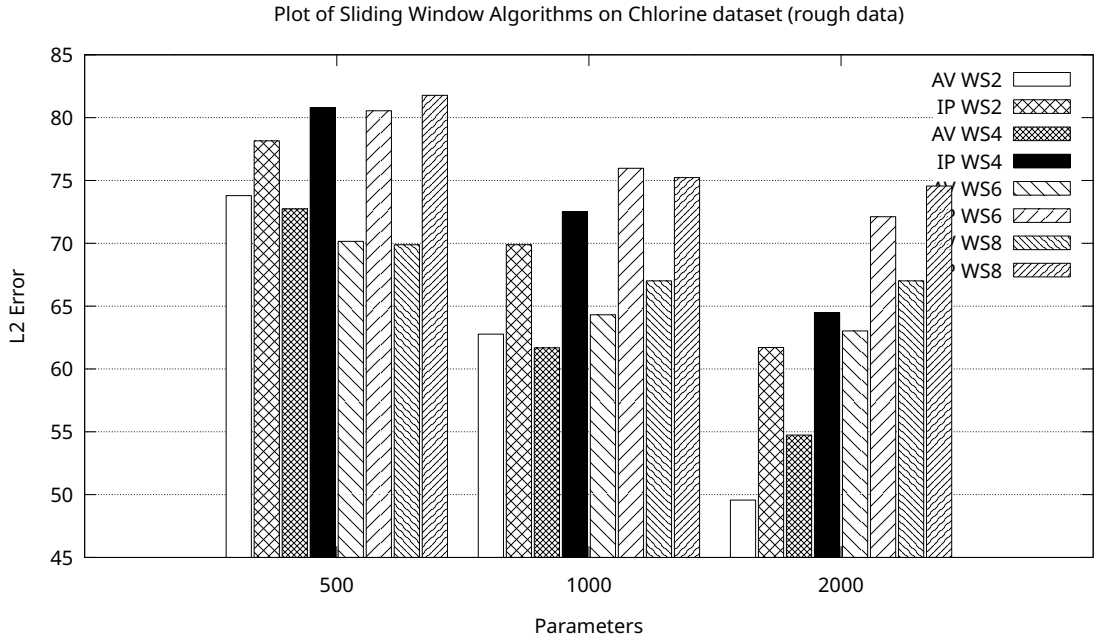


(b) Bar chart of euclidean error for Adaptive PLA upon Arrowhead dataset, reduced to size 500, 1000, 2000 respectively.

Figure 6.2: Bar Chart of Adaptive PLA methods on a smooth dataset of size 9000, AV = Average Value, IP = Interval Projection and WS = window size.



(a) First 900 entries in Chlorine dataset, displaying rough properties.



(b) Bar chart of euclidean error for Adaptive PLA upon Chlorine dataset, reduced to size 500, 1000, 2000 respectively.

Figure 6.3: Bar Chart of Adaptive PLA methods on a rough dataset of size 9000, AV = Average Value, IP = Interval Projection and WS = window size.

6.2 Optimal Piecewise Linear Approximation

For any time series there exists a best partition of the sequence for pieces to be approximated by linear functions. Ljosa and Singh[7] describe an algorithm using Dynamic Programming that calculates the best partition in polynomial time. Their algorithm required segments to have matching endpoints and therefore their algorithm has been modified to remove this requirement.

Given a data sequence $S[0:n-1]$ and a desired number of segments m, a $n \times m$ table T is constructed. The cell in row i column j contains sufficient information about the best approximation of $S[0:i]$ into j segments. Utilising dynamic programming, to calculate the best value for $T[i][j]$ (cell of row i column j) requires only observation of cells $S[k-1][j-1], \forall k \in [i]$.

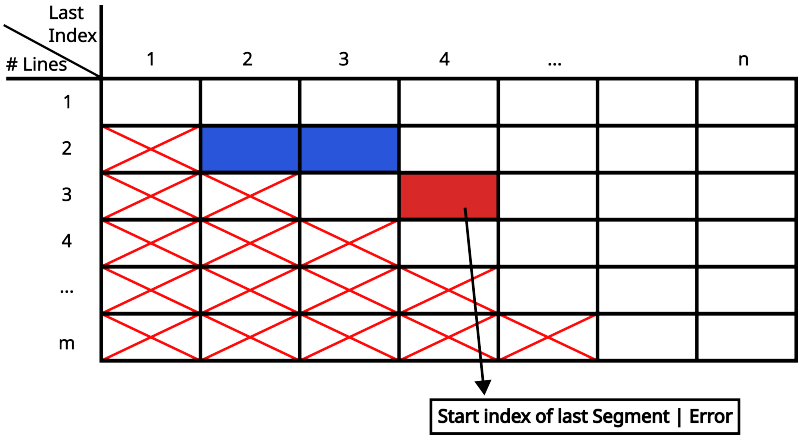


Figure 6.4: Visualisation of 2D table T to determine the optimal segment.

In Figure 6.4 each cell stores the start index of the last segment and the error of the total approximation. Cells marked with a cross aren't valid approximations because the sequence is shorter than the number of segments to split it into (each segment needs at least one element). The blue cells are the only two cells necessary for the red cell to determine its value.

To calculate the optimal segmentation $S[0:i]$ into j segments (cell $T[i][j]$):

$$\text{startIndex}[i][j] = \text{argmin}_{k \in [i]}\{\text{error}[k-1][j-1] + \text{errorOf}(\text{regress}(S[k:i]))\} \quad (6.1)$$

$$\text{error}[i][j] = \text{error}[\text{startIndex}[i][j]-1][j-1] + \text{errorOf}(\text{regress}(S[\text{startIndex}[i][j]:i])) \quad (6.2)$$

As the values of cells that split a sequence into $j-1$ segments are determined before the values of cells that split a sequence into j segments, all cells accessed by cell $T[i][j]$ (cell in row i, column j) are already defined.

Algorithm 6.2 APLA Algorithm

```
1: function APLA(S[0 : n − 1], N ∈ ℕ)
2:     error ← ARRAY[0 : n − 1][1 : N]
3:     startIndex ← ARRAY[0 : n − 1][1 : N]
4:     for i ∈ [n] do
5:         startIndex[i − 1][1] ← 0
6:         error[i − 1][1] ← errorOf(regress(S[0 : i − 1]))
7:     end for
8:     for j ∈ {2, .., N} do
9:         for i ∈ {j − 1, .., n − 1} do
10:            for a ∈ {j − 2, .., i − 1} do
11:                E ← error[a][j − 1] + errorOf(regress(S[a + 1 : i]))
12:                if E < error[i][j] then
13:                    error[i][j] ← E
14:                    startIndex[i][j] = a + 1
15:                end if
16:            end for
17:        end for
18:    end for
19:    intervals ← {}
20:    j ← N
21:    i ← n − 1
22:    while j > 0 do
23:        segments ← segments ∪{[startIndex[i][j] : i]
24:        i ← startIndex[i][j] − 1
25:        j ← j − 1
26:    end while
27:    return segments
28: end function
```

Halting Algorithm 6.2 will halt.

Each *for* loops operate on a finite set of values, with content inside the loops that are guaranteed to terminate. The *while* loop on line 22 terminates as we decrement j from $N > 0$ to 0.

Correctness The algorithm will halt with the optimal partition of the sequence for the best linear approximation on segments.

Claim: The cells hold the correct values of the minimum error and the start index of the last segment for the optimal partition.

This claim proves the algorithm is correct, as the final cell in the table is the optimal partition of the full sequence into N segments.

Proof of Claim:

This will be proven via induction on N, for an input sequence of any size.

Base Case is that $N = 1$. Here lines 8-18 are not executed and all subsequences are approximated by a single segment. The error for each subsequence is accurate as it is the error of the single linear segment and the start index is always 0.

Induction Hypothesis is that at step k, all cells in column $k - 1$ have the values matching an optimal partition of the subsequence into $k-1$ segments. They have the same start index of the last segment and the same total error of the approximation.

Induction Step at step k computes *startIndex* and *error* (referenced in Equations 6.1) of cell $T[i][k]$. This requires access only of the cells $T[\iota - 1][k - 1], \forall \iota \in [i]$. By

the induction hypothesis the *startIndex* and *error* are defined for the cells accessed by $T[i][k]$ and therefore $T[i][k]$ correctly formulates startIndex$[i][k]$ and error$[i][k]$.

Therefore the cells hold the correct error and start index of the minimum error approximation, so the *while* loop constructs the minimum error PLA approximation as it navigates the indexes of the segments for minimum error.

Complexity Analysis The time complexity of Algorithm 6.2 is $O(n^3 \cdot N)$.

The *for* loop on line 4 iterates over n values and calculates the linear interpolation of each subsequence. As linear interpolation is $O(n)$, lines 4-7 has a time complexity of $O(n^2)$. The loops on lines 8-10 iterates over N, $O(n)$ and $O(n)$ values respectively and inside perform a calculation of linear interpolation from values $a+1$ to i meaning the section of code of lines 8-18 have a time complexity of $O(N \cdot n \cdot n \cdot n)$. Lines 22 - 26 loop over N and inside perform only constant time operations. The time complexity of Algorithm 6.2 is $O(n) + O(n^3 \cdot N) + O(N) = O(n^3 \cdot N)$.

The Space complexity of the algorithm is $O(n \cdot N)$ via the creation of the 2D array necessary for Dynamic Programming, declared on lines 2 and 3.

The steep time and space complexity of this algorithm make it difficult to use on very large sequences.

Chapter 7

Epsilon Precision Algorithms

This chapter discusses Epsilon Precision Algorithms, algorithms that segment a time series optimal for piecewise linear representations via a constraint parameter ε. The algorithms provide the guarantee (or some variation under a different norm) that the Maximum Deviation (l_∞ norm) is smaller than ε. For maximum deviation, this means that for sequence $S[0:n]$ and its reconstituted approximation $\widetilde{S}[0:n]$, $\forall i \in \{0,..,n\}, |S[i] - \widetilde{S}[i]| < \varepsilon$.

Precision guarantee algorithms are better tools for data compression than algorithms that require exact segmentation. These algorithms have no size guarantee and are not directly suitable for the GEMINI Framework so this chapter will discuss a method to overcome this.

7.1 Top Down Algorithm

The Top Down algorithm (or Ramer Douglas Peucker algorithm) takes a Divide and Conquer approach to forming an approximation with the ε guarantee. The algorithm starts by approximating the entire sequence by one segment, splitting it into smaller segments at the point furthest from the approximation (until all points are within ε of the approximation).

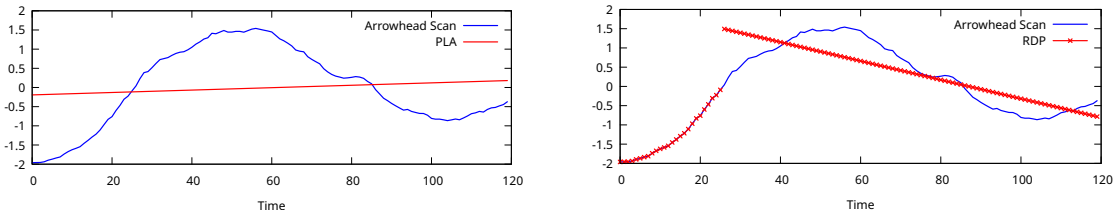


Figure 7.1: Two plots of a Top Down approximation on ArrowHead dataset.

Figure 7.1 shows a potential failue of the Top Down algorithm. The first plot shows a loose approximation that only required one segment to be within ε of all points and the second plot shows a tighter ε. In execution of Top Down, the furthest point was the first point, so it was split, and then the second, and so forth, resulting in the first 20 segments lying exactly on the time series. This is less optimal as these 20 segments could have been approximated correctly by one line segment and still held the ε guarantee.

Top Down is an accurate approximation method for precision (better than other algorithms for map simplification in cartography [20]). A Divide and Conquer ap-

proach allows parallel computing to increase performance by allocating different segments to different threads.

Algorithm 7.1 RDP Algorithm

```
1: function RDP(S[0 : n − 1], ε ∈ ℝ>0)
2:     segments ← {[0 : n − 1]}
3:     accurateSegments ← {}
4:     while |segments| > 0 do
5:         [i : j] ← segments.pop()
6:         maxIndex ← argmax_{k∈[i:j]}(|S[k] − regress(S[i : j])[k]|)
7:         maxError ← |S[maxIndex] − regress(S[i : j)[maxIndex]|
8:         if maxError < ε then
9:             accurateSegments ← accurateSegments ∪ {[i : j]}
10:        else
11:            if maxIndex = i then
12:                segments ← segments ∪ {[i : i + 1], [i + 2 : j]}
13:            else if maxIndex = j then
14:                segments ← segments ∪ {[i : j − 2], [j − 1 : j]}
15:            else
16:                errorLeft ← |S[i : maxIndex] − regress(S[i : maxIndex])| +
   |S[maxIndex +1 : j] − regress(S[maxIndex +1 : j])|
17:                errorRight ← |S[i : maxIndex −1]−regress(S[i : maxIndex −1])|+
   |S[maxIndex : j] − regress(S[maxIndex : j])|
18:                if errorLeft > errorRight then
19:                    segments ← segments ∪ {[i : maxIndex], [maxIndex +1 : j]}
20:                else
21:                    segments ← segments ∪ {[i : maxIndex −1], [maxIndex : j]}
22:                end if
23:            end if
24:        end if
25:    end while
26:    return accurateSegments
27: end function
```

Halting Algorithm 7.1 halts.

Once a segment is moved to *accurateSegments*, its indices never appear in *segments* again. This is because either the segment is moved to *accurateSegments*, or it is split in two and both sub-segments added to *segments*. Once moved, indices of that segment no longer appear in *segments*. Also, the size of segments added back to *segments* are always reduced as the original segment has been split in two. If a segment is size 1 or 2 it must be within ε of its linear approximation (because the error of the approximation is 0). Any segment will be reduced to sub-segments of size 1 or 2 (where all points lie within the ε distance). Therefore the algorithm will halt.

Correctness Algorithm 7.1 halts with a partition of the sequence with the ε precision guarantee.

For a segment to be moved to *accurateSegments*, it must have the ε guarantee, and eventually all segments end in *accurateSegments* (as the only way they leave *segments* is via moving to *accurateSegments*). Hence *accurateSegments* is a partition of [0:n-1] and each segment obeys the precision guarantee.

Complexity Analysis The time complexity of the algorithm is $O(n^2)$.

A pass of segments can be viewed as one iteration through the current partition of $[0 : n-1]$, at each iteration fragmenting pieces that are too inaccurate. In each of these iterations, each segment requires computation of its linear approximation ($O(|\,\text{segment}\,|)$), so it requires $O(n)$ time to complete an iteration. The algorithm starts with one segment and upon each iteration splits any segment that isn't within ε of its approximation in two. This bounds the number of iterations by n so the time complexity is $O(n^2)$.

The space complexity of the algorithm is $O(n)$.

The pseudocode shows the only other data structure used other than input S are *segments* and *accurateSegments*. Every part must contain at least one unique point as *segments* is a partition of S so the maximum size of *segments* is $O(n)$. *accurateSegments* is also a partition so its size is also $O(n)$.

7.2 Bottom Up Algorithm

Bottom Up is another algorithm reviewed by Keogh et al. [6] and is the complement of Top Down. Instead of starting with approximating it as one segment and splitting, it starts by approximating it as n segments of size 1 and merges until it would violate the ε constraint.

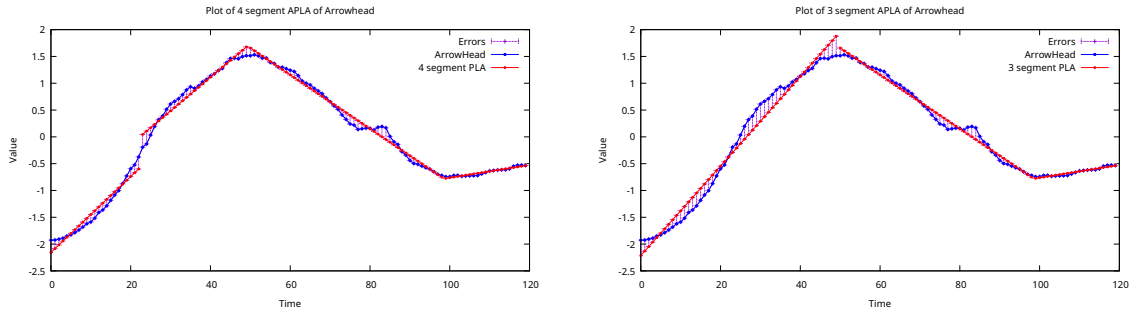


Figure 7.2: Two plots showing a step in the Bottom Up approximation on ArrowHead dataset.

The plots in Figure 7.2 show the execution of one step in the Bottom Up algorithm. The first plot is before the step and the second plot is afterwards. The algorithm identifies that the first two segments incur the least error when merged and merges them together to be approximate by one segment instead.

There is no simple parallel equivalent for Bottom Up, as two threads can merge the same segment simultaneously. Keogh found it performed better than Top Down and the algorithm was adapted to be *online* with the implementation of a buffer that slides across the data.

Algorithm 7.2 Bottom Up Algorithm

```
1: function BottomUp(S[0 : n − 1], ε ∈ ℝ>0)
2:     segments ← {[i : i] : i ∈ [0 : n − 1]}          ▷ segments is a self-sorting array
3:     minMergeError ← 0
4:     minMergeIndex ← 0
5:     while minMergeError ≤ ε & # segments > 1 do
6:         [i : j] ← segments[minMergeIndex]
7:         [k : l] ← segments[minMergeIndex +1]
8:         segments ← segments −{[i : j], [k : l]}
9:         segments ← segments ∪{[i : l]}
10:        minMergeIndex ← argmin_{i∈0,..,# segments −2}(
11:            max_{α∈segments[i]∪segments[i+1]}(
12:                |S[α] − regress(S[segments[i] ∪ segments[i + 1]])[α]|
13:            )                                  ▷ Max Dev of merging adjacent segments
14:        )
15:        minMergeError ← min_{i∈0,..,# segments −2}(
16:            max_{α∈segments[i]∪segments[i+1]}(
17:                |S[α] − regress(S[segments[i] ∪ segments[i + 1]])[α]|
18:            )
19:        )
20:    end while
21:    return segments
22: end function
```

Halting Algorithm 7.2 halts.

At each iteration, the size of segments decreases by one and the loop halts when there is only one segment.

Correctness Algorithm 7.2 halts with a partition of the sequence with the ε precision guarantee.

It starts with all segments within ε and won't merge two segments that would violate the guarantee.

Complexity Analysis The time complexity of the algorithm is $O(n^2)$.

On each iteration it computes linear regression over every value in the sequence. After n iterations every segment will be merged into one and the algorithm will halt. Therefore the time complexity is $O(n^2)$.

The space complexity of the algorithm is $O(n)$.

The pseudocode shows the only other data structure used other than input S is *segments*. Every part must contain at least one unique point as *segments* is a partition of S so the maximum size of *segments* is $O(n)$.

7.3 Sliding Window

Sliding Window has different names in the literature [21][16] and is considered less good for data compression than Top Down and Bottom Up. However as it is *online* it is valuable for forecasting in databases that have active additions[21].

The algorithm forms a piecewise approximation by passing along the sequence, adding points to the current piece until the error would exceed ε. As Sliding Window can decide the current and future pieces using only the points of the current piece

and those unseen, it can accommodate new data by being ran from the previously last piece.

Algorithm 7.3 Sliding Window Algorithm

```
1:  function SlidingWindow(S[0 : n − 1], ε ∈ ℝ>0)
2:      segments ← {}
3:      i0 ← 0
4:      i1 ← 0
5:      while i1 < n do
6:          l ← regress(S[i0 : i1])
7:          d ← 0
8:          for j ∈ [i0 : i1] do
9:              d ← max(d, l(j − i0))
10:         end for
11:         if d ≥ ε then
12:             segments ← segments ∪ {[i0 : i1 − 1]}
13:             i0 ← i1
14:         else
15:             i1 ← i1 + 1
16:         end if
17:     end while
18:     if segments.last().end() ≠ i1 − 1 then
19:         segments ← segments ∪ {[i0 : i1 − 1]}
20:     end if
21:     return segments
22: end function
```

Halting The algorithm will halt.

The outer *while* loop of line 5 halts as i_1 starts at 0 and will always either increment by 1 or increment by 1 on the next iteration (as a linear approximation of a single value is exact). The inner *for* loop of line 8 operates on a finite set and performs a constant time operation, so halts. The value i_1 will reach n and the algorithm will halt.

Correctness Algorithm 7.3 halts with a partition of the sequence with the ε precision guarantee.

Algorithm 7.3 increments i_1 if the segment has the guarantee and splits before adding a point that would violate it. The *if statement* on line 18 ensures that the segment returned is a complete partition of $[0 : n - 1]$ as required.

Complexity Analysis The time complexity of the algorithm is $O(n^2)$.

To consider the next point for the current piece, linear interpolation is ran on the current piece. The size of a piece is bounded by n and linear interpolation is ran on a piece for every element of the sequence. Therefore the algorithm runs in $O(n^2)$.

The space complexity of the algorithm is $O(n)$.

The pseudocode shows the only other data structure used other than input S is *segments*. Every part must contain at least one unique point as *segments* is a partition of S so the maximum size of *segments* is $O(n)$.

7.4 SWING Filters

SWING filters improve the accuracy of the Sliding Window algorithm by accepting the next point if there exists a line on that segment obeying the ε guarantee. When there is no line that can be matched to the segment maintaining the ε precision guarantee, a new segment is created.

For SWING, it is assumed that any approximation goes through the endpoint of the previous segment, and therefore lines on this segment only need to store their gradient (as the y-intercept is fixed). Furthermore, the set of valid lines (those obeying the ε guarantee) have their gradients form an interval in $\mathbb{R}$. To represent this interval, only the maximum and minimum gradient are necessary.

When considering adding the next point to the current piece, SWING redetermines the set of valid lines by finding the new maximum and minimum gradient. The new maximum gradient is chosen from the smaller of two options; the gradient to the largest positive deviation from the new point or the largest gradient already within ε of previous points. The new minimum gradient is chosen from the larger of two options; the gradient to the largest negative deviation from the new point or the smallest gradient already within ε of previous points.

If the set of valid lines is empty, (meaning there doesn't exist a line on the segment obeying the ε guarantee) then the interval will be poorly defined. This occurs when the maximum gradient is smaller than the minimum gradient; if this occurs, a new segment is created and a linear approximation of the previous segment is calculated. The new segment has its start location be the endpoint of the previous segment.

Algorithm 7.4 Swing Filters Algorithm

1: **function** SwingFilter($S[0 : n-1], n \geq 3, \varepsilon \in \mathbb{R}_{>0}$)
2: segments $\leftarrow \{\}$
3: $i_0 \leftarrow 0$
4: $i_1 \leftarrow 2$
5: $\Delta_{\max} \leftarrow S[1] + \varepsilon - S[0]$
6: $\Delta_{\min} \leftarrow S[1] - \varepsilon - S[0]$
7: $s \leftarrow S[0]$
8: **while** $i_1 < n$ **do**
9: $\Delta_{\max} \leftarrow \min\{\Delta_{\max}, (S[i_1] + \varepsilon - S[i_0])/(i_1 - i_0)\}$
10: $\Delta_{\min} \leftarrow \max\{\Delta_{\min}, (S[i_1] - \varepsilon - S[i_0])/(i_1 - i_0)\}$
11: **if** $\Delta_{\max} \geq \Delta_{\min}$ **then**
12: $i_1 \leftarrow i_1 + 1$
13: **else**
14: $l \leftarrow$ regressThroughStartpoint($S[i_0 : i_1 - 1], s$)
15: **if** # segments $= 0$ **then** ▷ First segment doesn't share start index
16: segments $\leftarrow$ segments $\cup \{[i_0 : i_1 - 1]\}$
17: **else**
18: segments $\leftarrow$ segments $\cup \{[i_0 + 1 : i_1 - 1]\}$
19: **end if**
20: $s \leftarrow l(i_1 - i_0 - 1)$ ▷ Next start position of line found
21: $\Delta_{\max} \leftarrow S[i_1] + \varepsilon - S[i_1 - 1]$
22: $\Delta_{\min} \leftarrow S[i_1] - \varepsilon - S[i_1 - 1]$
23: $i_0 \leftarrow i_1 - 1$
24: $i_1 \leftarrow i_1 + 1$
25: **end if**
26: **end while**
27: **if** segments.last().end() $\neq i_1 - 1$ **then**
28: segments $\leftarrow$ segments $\cup \{[i_0 : i_1 - 1]\}$
29: **end if**
30: **return** segments
31: **end function**

The function *regressThroughStartpoint* on line 14 in Algorithm 7.4 takes a sequence $S[i : j]$ and value s and computes the best linear approximation (through minimising least squares) of a sequence starting with s and followed by $S[i+1 : j]$ that starts at s. Equation 7.1, translates every point downwards such that s lies on the origin and computes the gradient of the line of best fit that goes through the origin.

$$\widehat{\beta} = \frac{\sum_{k=1}^{j-i-1} k \cdot (S[k+i+1] - s)}{\sum_{k=1}^{j-i-1} k^2} \tag{7.1}$$

Halting Algorithm 7.4 halts.

The loop terminates upon $i_1 \geq n$, i_1 starts at 2 and upon every iteration, the only modification made to i_1 is to increment it by 1.

Correctness Algorithm 7.4 halts with a partition of the sequence with the ε precision guarantee.

The proof of this is provided by Elmeleegy et al.[16].

Complexity Analysis The algorithm runs in $O(n)$ time.

For any iteration of the *while* loop on line 8, this iteration either causes the creation of a new segment or only increments i_1. If it only increments i_1 then only constant time operations are performed. If it creates a new segment then linear interpolation is ran on the previous segment, so every element is accessed once for linear interpolation. The time complexity is $O(n) + O(n) = O(n)$.

The space complexity of the algorithm is $O(n)$.

The pseudocode shows the only other data structure used other than input S is *segments*. Every part must contain at least one unique point as *segments* is a partition of S so the maximum size of *segments* is $O(n)$.

7.5 Correcting Dimension

Algorithms that use parameter ε to split a series until all linear approximation of segments are within ε of the points don't guarantee to use the same number of intervals. A time series where every point lies on a line only requires one segment, whilst a time series of random points may need more. These algorithms cannot be integrated into the GEMINI Framework as all approximations need to have the same dimension.

The Split and Merge algorithm, presented in SAPLA[5], takes an approximation in an arbitrary dimension and converts it to an approximation in specified different dimension. This occurs by splitting segments if there are too few and merging adjacent segments if there are too many, until the correct number of segments are present. Splitting also requires finding the best place to split in a segment, achieved by considering a split at every possible index and choosing the index that minimises error.

$$\text{Split: } \langle 1.2, 0.0, 2, 1.5, 0.3, 5\rangle \to \langle 0.7, 0.0, 1, 0.7, 0.3, 2, 1.5, 0.3, 5\rangle$$

$$\text{Merge: } \langle 0.7, 0.0, 1, 0.7, 0.3, 2, 1.5, 0.3, 5\rangle \to \langle 0.7, 0.0, 1, 1.0, 0.3, 5\rangle$$

In the split example above, the first segment (denoted by the first three numbers) is split into two intervals, the second original segment is preserved. In the merge example above the second and third segments are merged together, preserving instead the first segment. Segments to merge or split are chosen in a *Greedy* approach, the segment that reduces the error the most by being split (or merged with its neighbour) is chosen to be split (or merged) and the process repeats until the time series is the correct size.

Algorithm 7.5 Split Algorithm

```
1: function SPLIT(S[0 : n − 1], m < n, segments)
2:     while # segments < m do
3:         s_e ← 0                                  ▷ Maximum reduction of error by split
4:         [s_0, s_1] ← [−1, −1]                    ▷ Segment whose splitting results in least error
5:         s_i ← −1                                 ▷ Index in S of element to split at
6:         for [i, j] ∈ segments do
7:             d ← 0                                ▷ Maximum distance from approximation
8:             i_[i,j] ← −1                         ▷ Index of maximum distance from approximation
9:             l ← regress(S[i : j])                ▷ Linear Approximation of segment
10:            for k ∈ {i, .., j − 1} do
11:                d_k ← |S[k] − l(k − i)|
12:                if d_k < d then
13:                    d ← d_k
14:                    i_[i,j] ← k
15:                end if
16:            end for
17:            e ← error(S[i : j], regress(S[i : j]))
18:                − error(S[i : i_[i,j]], regress(S[i : i_[i,j]]))
19:                − error(S[i_[i,j] + 1 : j], regress(S[i_[i,j] + 1 : j]))
20:            if e < s_e then
21:                s_e ← e
22:                [s_0, s_1] ← [i, j]
23:                s_i ← i_[i,j]
24:            end if
25:        end for
26:        segments ← segments −{[s_0, s_1]}
27:        segments ← segments ∪{[s_0, s_i], [s_i + 1, s_1]}
28:    end while
29:    return segments
30: end function
```

Halting Algorithm 7.5 halts.

Either the number of segments is larger than or equal to m, or the number of segments is smaller. If the number of segments is larger, then the algorithm halts. If the number of segments is smaller, a segment is split in two and the number of segments increases by 1. This is repeated until the number of segments is larger than or equal to m and the algorithm halts. A segment must always include at least one point so $m < n$ is necessary.

Complexity Analysis Algorithm 7.5 has a time complexity of $O(n^2)$.

For every iteration of the *while* loop the entire array of S[0:m-1] is scanned via segments. Every segment runs a linear time algorithm (to find the index furthest from the linear approximation) so has a time complexity of $(O(|\,\text{segment}\,|))$. The run-time of an iteration is $O(n)$ and the list of segments can only be split a maximum of n times,so the time complexity is $O(n^2)$.

The space complexity of the algorithm is $O(n)$.

The pseudocode shows the only other data structure used other than input S is *segments*. Every part must contain at least one unique point as *segments* is a partition of S so the maximum size of *segments* is $O(n)$.

Algorithm 7.6 Merge Algorithm

```
1: function MERGE(S[0 : n − 1], m ∈ ℕ, segments)
2:     while # segments > m do
3:         s_e ← ∞                                  ▷ Minimum increase in error of a merge
4:         [s_0, s_1, s_2] ← [0, 0, 0]      ▷ Segments whose merging results in least error
5:         for [i, j], [j + 1, k] ∈ segments do          ▷ Grab adjacent segments
6:             e ← error(S[i : k], regress(S[i : k]))
7:                 − error(S[i : j], regress(S[i : j]))
8:                 − error(S[j + 1 : k], regress(S[j + 1 : k]))
9:             if e < s_e then
10:                s_e ← e
11:                [s_0, s_1, s_2] ← [i, j, k]
12:            end if
13:        end for
14:        segments ← segments −{[s_0, s_1], [s_1 + 1, s_2]}
15:        segments ← segments ∪{[s_0, s_2]}
16:    end while
17:    return segments
18: end function
```

Halting Algorithm 7.6 halts.

Either the number of segments is smaller than or equal to m, or the number of segments is larger. If the number of segments is smaller, then the algorithm halts. If the number of segments is larger, a segment is merged with the next and the number of segments decreases by 1. This is repeated until the number of segments is smaller than or equal to m and the algorithm halts.

Complexity Analysis Algorithm 7.6 has a time complexity of $O(n^2)$.

For each iteration of the *while* loop the entire array of S[0:m-1] is scanned via segments.

For each segment, a linear time algorithm is run on elements of this segment and the next segment. Each element is accessed twice, once from the previous segment and once from the current segment. The list of segments can only be merged a maximum of n times and every element inside the array is accessed a constant number of times, the time complexity $O(n^2)$.

The space complexity of the algorithm is $O(n)$.

The pseudocode shows the only other data structure used other than input S is *segments*. Every part must contain at least one unique point as *segments* is a partition of S so the maximum size of *segments* is $O(n)$.

Chapter 8

Priority Queue Optimisation

Algorithms that repeatedly search the same space often generate redundancy by calculating the same results again and again. Instead, results be cached and retrieved rather than repeated, to speed up the algorithm. This chapter discusses the Priority Queue data structure within Split and Merge and the Exact Segment Sliding Window algorithm (Algorithm 6.1) to improve run-times.

8.1 Split and Merge

Both Split and Merge search the set of segments for a target segment whose modification by splitting it in two (Split) or merging it with the next segment (Merge) reduces the error of the approximation. This is achieved by searching the set of segments one after another, storing the current minimum error of modifying the segment as it passes through all the segments.

By inserting all segments into a priority queue ranking lower error segments as higher priority, segments to modify can be accessed quickly and redundant computation has been eliminated. Splitting a segment doesn't effect the cost of splitting any other segment, so only the costs of splitting the new segments need to be inserted into the queue. Merging two segments together (first and second) changes the costs of merging the segment before the first segment, the cost of merging the first segment and removes the cost of merging the second segment. With an implementation of a priority queue via a Red Black tree, insertion, deletion and search are all $O(\log n)$ operations. Forming a priority queue takes $O(n \log n)$ time and one iteration of Split or Merge requires one search, one (two for merge) deletion and two insertions into the priority queue. The time complexity is therefore $O(n \log n + n \cdot \log n) = O(n \log n)$ instead of $O(n^2)$.

8.2 Exact Segment Sliding Window

Priority Queues can also optimise Algorithm 6.1 in a similar manner. The double window traverses the time series, storing the index of it's window in a priority queue that ranks the highest difference between the two windows as the highest priority. At each iteration, the highest priority window is sought and the segment containing that index is split in two.

A window which considered values that overlap with the split created by the highest priority window is now invalidated. In the original algorithm, this window would not have been evaluated in the next iteration as the double window is only

evaluated upon whole segments. To mimic the execution of the original algorithm, this window needs to be removed.

If window sizes are lw and rw, then no more than $lw + rw$ windows could be invalidated by the split, a constant value in the analysis of the algorithm. One iteration performs a search and $lw + rw + 1$ deletions from the priority queue, so the time complexity of the algorithm is reduced to $O(n \log n)$.

Chapter 9

Indexing Adaptive PLA

Indexing Adaptive PLA creates a scheme to lower bound the distance between two sequences by the distance between approximations of the sequences .

$$\text{dist}_p(\widehat{S}, \widehat{T}) \leq \text{dist}_n(S, T) \tag{9.1}$$

If the distance function *dist* is a metric then for any set $X \subset \mathbb{R}^p$ a new function can be defined:

$$\text{dist}_{\text{shape}}(X, \widehat{T}) = \begin{cases} 0 & \widehat{T} \in X \\ \inf_{x \in X}\{\text{dist}(x, \widehat{T})\} & \text{otherwise} \end{cases}$$

Theorem 9.1 *Suppose we have metrics* dist_p *and* dist_n *on* $\mathbb{R}^p$ *and* $\mathbb{R}^n$ *respectively and Dimension Reduction Technique* $f_{reduce} : \mathbb{R}^n \to \mathbb{R}^p$, $f_{approx} : \mathbb{R}^p \to \mathbb{R}^n$ *such that together they obey the lower bounding lemma shown in Equation 9.1. Then for the* dist_{shape} *defined as above, we have that* $\forall T \in \mathbb{R}^n$, $\forall X \subset \mathbb{R}^p$*:*

$$\text{dist}_{shape}(X, f_{reduce}(T)) \leq \text{dist}_n(f_{approx}(x), T), \forall x \in X$$

Proof:

All that needs to be shown is that $\text{dist}_{\text{shape}}(X, \widehat{T})$ lower bounds the distance $\text{dist}_p(x, \widehat{T})$ for all $x \in X$, as the lower bounding lemma immediately implies the conclusion.

Suppose there exists $y \in X$ such that $\text{dist}_p(y, \widehat{T}) < \text{dist}_{\text{shape}}(X, \widehat{T})$. If $\widehat{T} \in X$ this is impossible, as a distance function maps to $\mathbb{R}_{\geq 0}$ implying that $\text{dist}_p(y, \widehat{T}) < 0$. This implies that $\widehat{T} \notin X$ and using that $\inf_{x \in X}\{\text{dist}(x, \widehat{T})\} \leq \text{dist}_p(y, \widehat{T})$ by properties of the infimum, this completes the contradiction. ■

With Theorem 9.1, GEMINI performs range queries (searches for all elements within ε distance of a query sequence) on subsequences by recursively generating rectangles that bound other rectangles or points. This tree pattern can be traversed by looking at a rectangle and calculating if the query lies inside the rectangle or how far away it is from the boundary (both fast operations). If the distance from the query to the boundary is larger than ε then Theorem 9.1 guarantees that no element of the rectangle will be within ε of the query. Therefore rectangles covered by a larger rectangle should be considered only if the query is within ε of the larger rectangle.

SAPLA by Xue et al.[5] found a lower bounding distance function for Adaptive PLA approximations but this was an *impure* function (in this case requiring access to the original sequence).

Chakrabarti et al.[4] worked on a similar case for Adaptive Piecewise Constant Approximation (APCA) where the lengths of segments can vary but are approximated by a constant function on that interval. They found no pure distance function for APCA, instead presenting the weaker lower bounding property.

$$\text{dist}_p(\widehat{S}, T) \leq \text{dist}_n(S, T) \tag{9.2}$$

Here $\text{dist}_p : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}_{\geq 0}$ instead of $\text{dist}_p : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}_{\geq 0}$ in Equation 9.1. A function for distance to a set can still be defined $\forall X \subset \mathbb{R}^p$:

$$\text{dist}_{\text{shape}}(X, T) = \begin{cases} 0 & \widehat{T} \in X \\ \inf_{x \in X}\{\text{dist}(x, T)\} & \text{otherwise} \end{cases}$$

Theorem 9.2 *Suppose we have* $\text{dist}_p : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}_{\geq 0}$ *and metric* $\text{dist}_n : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_{\geq 0}$ *and Dimension Reduction Technique* $f_{reduce} : \mathbb{R}^n \to \mathbb{R}^p$, $f_{approx} : \mathbb{R}^p \to \mathbb{R}^n$ *such that together they obey the weak lower bounding lemma shown in Equation 9.2. Then for the* dist_{shape} *defined as above, we have that* $\forall T \in \mathbb{R}^n$, $\forall X \subset \mathbb{R}^p$*:*

$$\text{dist}_{shape}(X, T) \leq \text{dist}_n(f_{approx}(x), T), \forall x \in X$$

Proof:

Again it suffices to show that $\text{dist}_{\text{shape}}(X, T)$ lower bounds the distance $\text{dist}_p(x, T)$ for all $x \in X$, as the weak lower bounding lemma immediately implies the conclusion.

Suppose there exists $y \in X$ such that $\text{dist}_p(y, T) < \text{dist}_{\text{shape}}(X, T)$. If $\widehat{T} \in X$ this is impossible, as a distance function maps to $\mathbb{R}_{\geq 0}$ implying that $\text{dist}_p(y, T) < 0$. This implies that $\widehat{T} \notin X$ and using that $\inf_{x \in X}\{\text{dist}(x, T)\} \leq \text{dist}_p(y, T)$ by properties of the infimum, this completes the contradiction. ■

Chakrabarti et al.[4] use Theorem 9.2 to index APCA by using a tree of bounding rectangles (as these are designed to be as tight as possible, they are refer to as Minimum Bounding Rectangles or MBRs). The GEMINI Framework can work with the weaker lower bounding property as there is no requirement for the query subsequence to be reduced, allowing range queries to be performed.

The following implementation of $\text{dist}_p(\widehat{S}, T) : \mathbb{R}^{3 \cdot k} \times \mathbb{R}^n \to \mathbb{R}_{\geq 0}$ is well defined for an Adaptive PLA $\widehat{S}$ where the first index indicates the y-intercept of the linear function, the second the gradient and the third the end of the segment.

$$\text{dist}_p(\widehat{S}, T) = D_{\text{APLA}}(\widehat{S}, f_{\widehat{S}}(T))$$

Here $f_{\widehat{S}}$ is a dimension reduction technique that takes T and maps it to $\mathbb{R}^{3 \cdot k}$ by taking the exact same segments as $\widehat{S}$ (and hence every third index will be identical to $\widehat{S}$ as these denote the end of the segments) and computing linear regression upon each segment of values from T.

$$D_{\text{APLA}}(\widehat{S}, \widehat{T}) := \sqrt{\sum_{i=0}^{k-1}\Big(\sum_{j=0}^{\widehat{S}[3i+2]-\widehat{S}[3i-1]} (\widehat{S}[3i] - \widehat{T}[3i] + (\widehat{S}[3i+1] - \widehat{T}[3i+1]) \cdot j)^2\Big)} \tag{9.3}$$

The function defined in Equation 9.3 is a generalisation of D_{PLA} from Equation 4.3 and requires proof that together it satisfies the weaker lower bounding property.

This follows from the proof by Chen et al.[15] in Theorem 3.1 of "Indexable PLA for Efficient Similarity Search" where they prove that any two sequences of the same length are further apart than their linear interpolations.

Equation 9.3 sums over the segments and finds the distance between the linear interpolations on each segment. As the distance between each segment lower bounds the actual distance on each segment, $D_{\text{APLA}}(\widehat{S},\widehat{T})^2 \leq (||S-T||_2)^2$ and as square root is an increasing function, the weak lower bounding property is satisfied.

This may appear to be a valid indexing scheme but this necessitates the linear approximation of the segment to be the line that minimises the least squares, not any linear approximation. Merging Adaptive PLA sequences into Minimum Bounding Rectangles (MBRs) may violate this property so a different approach needs to be taken.

Instead $\text{dist}_{PC}(R,T)$ was constructed, taking a bounding 'rectangle' R and an uncompressed time series T and returning a distance ($d \in \mathbb{R}_{\geq 0}$) that lower bounds the distance between T and any time series S whose approximation lies in R. Rectangles can be defined by two points, denoted U and L (for Upper and Lower) such that $\forall i \in [k], L[i] \leq U[i]$. The 'rectangles' used by the distance function are not strictly rectangles as there exists no sensible definition of $\widehat{U}$ and $\widehat{L}$ maintaining those properties. Nonetheless, these 'rectangles' belong in $\mathbb{R}^{6k}$ where they can be viewed as a sequence of quadrilaterals in the plane, every 6 entries denoting a cover of a segment and will be called *partition covers* by this paper (so dist_{PC} accepts valid partition covers, not just rectangles).

Partition Cover Property: A key property of our definition of a partition cover is that on any segment, the lower line segment evaluated at endpoints is smaller than or equal to the upper line segment evaluated at endpoints. By the intermediate value theorem, this property maintains that the upper line never crosses the lower line.

A partition cover in $\mathbb{R}^{6k}$ is a shape that covers a series $S \in \mathbb{R}^n$ that has been partitioned into k segments. Every 6 entries $la_i, lb_i, ss_i, ua_i, ub_i, se_i$ form a quadrilateral that covers x values ss_i to se_i (segment start to segment end) defined by two lines, a lower line $l(x) = la_i + lb_i \cdot x$ and an upper line $u(x) = ua_i + ub_i \cdot x$.

For partition covers, segments can overlap (meaning the end of the previous segment occurs after the beginning of the next) so function $\text{segments}(R,i)$ is defined to take an integer $i \in [n]$ and return a set of indices of segments that i belongs to in the partition cover R. As notational convenience indexing of R will be treated as per segment; $R[i] = \langle la_i, lb_i, ss_i, ua_i, ub_i, se_i \rangle, \quad \forall i \in [k]$.

$$\text{dist}_{PC}(R,T) := \sqrt{\sum_{i=0}^{n-1} \min_{s \in \text{segments}(R,i)} \text{distSqr}_{PC}(R[s], T[i], i)} \tag{9.4}$$

$$\text{distSqr}_{PC}(R[s],T[i],i) := \min \begin{cases} \infty \cdot \mathbf{1}(\\ \quad la_s - lb_s \cdot (i - ss_s) \\ \quad \leq T[i] \leq \\ \quad ua_s - ub_s \cdot (i - ss_s) \\ ) \\ (T[i] - ua_s - ub_s \cdot (i - ss_s))^2 \\ (T[i] - la_s - lb_s \cdot (i - ss_s))^2 \end{cases} \tag{9.5}$$

The indicator function in distSqr_{PC} specifies that if a point of T lies within the area defined by the segment (in between the upper and lower lines) then its distance is 0. A time series $S \in \mathbb{R}^n$ is defined as *inside* partition cover R when for every point $S[i]$, there exists segment $s \in [k]$ such that $i \in \{ss_s, .., se_s\}$ and $(i - ss_s, S[i])$ lies between the upper and lower lines of $R[s]$.

Claim: $\text{dist}_{PC}(R, T) \leq ||T - S||_2$ for all S *inside* R.

Proof: For any index $i \in [n]$, either $(i - ss_j, T[i])$ lies inside some segment $R[j]$ or it doesn't. If it lies inside a segment then its distance to $S[i]$ is 0. If it lies outside all segments then either $(i, T[i])$ lies above $(i, S[i])$ or below it as points of S lie inside the region specified by R.

If $(i, T[i])$ lies above $(i, S[i])$ then any segment j that i belongs to must have an upper line that evaluates at $i - ss_j$ to be a number x between $T[i]$ and $S[i]$. As $T[i] > x \geq S[i] \implies (T[i]-x)^2 > (T[i]-S[i])^2$, the minimum value over all segments of $\text{distSqr}_{PC}(R[j], T[i], i)$ lower bounds $(T[i] - S[i])^2$. By a similar argument, the same holds true if $(i, T[i])$ lies below $(i, S[i])$ and with the property of a square root to preserve inequalities, the claim is reached. ■

By constructing a partition cover R around a point $S \in \mathbb{R}^n$, and merging partition covers such that all points of S lie in the merge of R with any other cover, range queries can be performed. At this point, only a suitable construction of this partition cover for a sequence $S \in \mathbb{R}^n$ and operation of merging is required to define an indexing scheme for Adaptive PLA.

9.1 Partition Cover Construction

Given a sequence $S \in \mathbb{R}^n$, a partition cover R is constructed such that all points of S lie inside R.

1. An Adaptive PLA algorithm is used to find a piecewise linear approximation of S.

2. For each segment j, two more lines are created, identical to the linear approximation on the segment. One is shifted up and the other down until together all points of the segment lie in between them.

3. These upper and lower lines form the upper and lower lines of segment j of R.

4. The start and end of segment j are recorded in ss_j and se_j.

Claim: This is a Partition Cover (it satisfies the Partition Cover Property).

Proof: The lines cannot intersect as they are parallel. The upper line is either equal to the lower line or has both endpoints above it.

Claim: S is *inside* the partition cover R.

Proof: Segments of R are disjoint so for an index $i \in [n]$ only a single segment$j$ will contain a bound for it. For every $i \in [n]$, the upper line is constructed to be above $(i, S[i])$ when evaluated at $i - ss_j$ and every lower line is constructed to be below.

9.2 Merging Partition Covers

To merge partition covers R' and R'' into one partition R, it must completely overlap both covers. This can be achieved naively by taking the maximum or minimum of parameters for each segment j after extending upper and lower lines to operate on the same line segment.

$$\begin{aligned} ss_j &\leftarrow \min(ss'_j, ss''_j) & se_j &\leftarrow \max(se'_j, se''_j) \\ la'_j &\leftarrow \text{extend}(la'_j, ss_j) & la''_j &\leftarrow \text{extend}(la''_j, ss_j) \\ ua'_j &\leftarrow \text{extend}(ua'_j, ss_j) & ua''_j &\leftarrow \text{extend}(ua''_j, ss_j) \\ la_j &\leftarrow \min(la'_j, la''_j) & lb_j &\leftarrow \min(lb'_j, lb''_j) \\ ua_j &\leftarrow \max(ua'_j, ua''_j) & ub_j &\leftarrow \max(ub'_j, ub''_j) \end{aligned}$$

The naive merge retains being a Partition Cover and any sequence that was inside R' or R'' is still inside R making it a valid solution, but offers distance approximations when used in dist_{PC}. There are partition covers that occupy less area of $\mathbb{R}^n$ than naive and keep all sequences inside R.

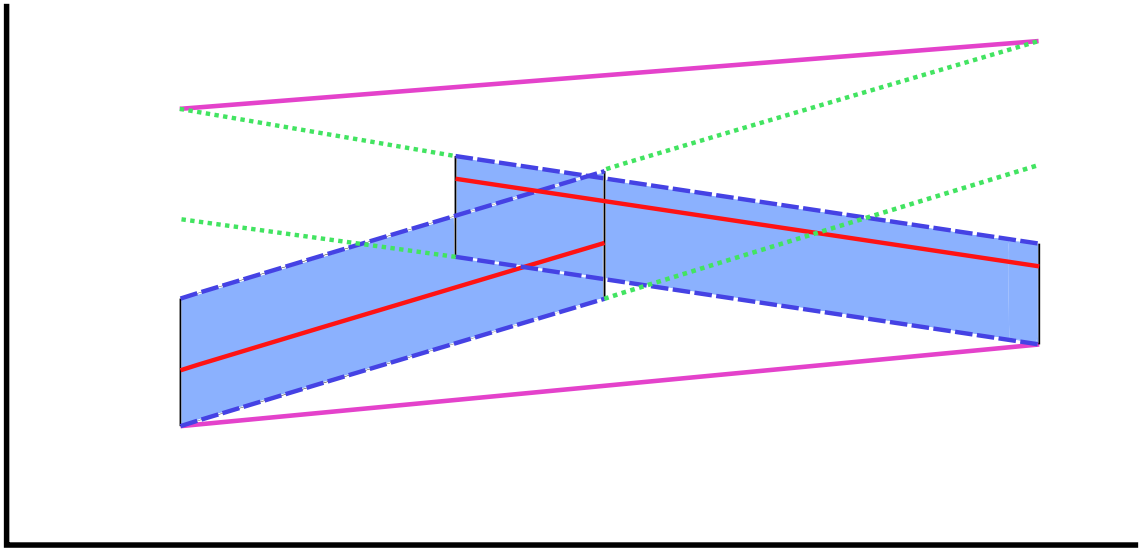


Figure 9.1: Illustrates an improved method to merge the i-th segment of two partition covers.

To retain the same dimension, rectangles R' and R'' need to attempt to merge each segment individually. Figure 9.1 shows how the i-th segment of R' and R'' is merged in a tighter merge operation. The segments, shown as the two blue parallelograms with their upper and lower lines (the blue dashed lines) are being merged. The green dashed lines show the extension of the upper and lower endpoints of both segments being extended to the new segment start and segment end indexes. The purple lines connect the greatest left endpoint with the greatest right endpoint and the smallest left endpoint with the smallest right endpoint. The purple lines bound the segments in the figure, showing that a new segment formed from the purple lines would keep any sequence that was inside R' or R'' inside the merged R. The merged segment will have the partition cover property by construction as the upper and lower lines are defined by the upper endpoints being greater than the lower endpoints.

If the naive approach was taken upon the segments of Figure 9.1 then the intercepts of the upper and lower lines would be the same but the gradients further apart. The upper line would have the gradient of the left segment and the lower line would have the gradient of the right segment. Together this segment would span a much larger area of R^n and would therefore be a *looser* approximation of distance for a sequence inside one of the segments.

Algorithm 9.1 Partition Cover Merge Algorithm

function PCMERGE($R', R'' \in \mathbb{R}^{6k}$)
  $R \leftarrow \underline{0} \in \mathbb{R}^{6k}$
  for $j \in \{0, .., k-1\}$ **do**
    $\langle la'_j, lb'_j, ss'_j, ua'_j, ub'_j, se'_j \rangle \leftarrow R'[j]$
    $\langle la''_j, lb''_j, ss''_j, ua''_i, ub''_j, se''_j \rangle \leftarrow R''[j]$
    ▷ Extend segment to encompass both prior segments
    $ss_j \leftarrow \min(ss'_j, ss''_j)$
    $se_j \leftarrow \max(se'_j, se''_j)$
    ▷ Upper and lower left endpoints after extending
    $ela'_j \leftarrow la'_j + lb'_j \cdot (ss_j - ss'_j)$
    $ela''_j \leftarrow la''_j + lb''_j \cdot (ss_j - ss''_j)$

    $eua'_j \leftarrow ua'_j + ub'_j \cdot (ss_j - ss''_j)$
    $eua''_j \leftarrow ua''_j + ub''_j \cdot (ss_j - ss''_j)$

    $la_j \leftarrow \min(ela'_j, ela''_j, eua'_j, eua''_j)$
    $ua_j \leftarrow \max(ela'_j, ela''_j, eua'_j, eua''_j)$
    ▷ Upper and lower right endpoints after extending
    $lr'_j \leftarrow la'_j + lb'_j \cdot (se_j + ss_j - ss'_j)$
    $lr''_j \leftarrow la''_j + lb''_j \cdot (se_j + ss_j - ss''_j)$

    $ur'_j \leftarrow ua'_j + ub'_j \cdot (se_j + ss_j - ss'_j)$
    $ur''_j \leftarrow ua''_j + ub''_j \cdot (se_j + ss_j - ss''_j)$

    $lr_j \leftarrow \min(lr'_j, lr''_j, ur'_j, ur''_j)$
    $ur_j \leftarrow \max(lr'_j, lr''_j, ur'_j, ur''_j)$
    ▷ Calculate gradients from the endpoints
    $lb_j \leftarrow (lr_j - la_j)/(se_j - ss_j)$
    $ub_j \leftarrow (ur_j - ua_j)/(se_j - ss_j)$

    $R[j] = \langle la_j, lb_j, ss_j, ua_j, ub_j, se_j \rangle$
  end for
  return R
end function

Correctness: Algorithm 9.1 halts with a Partition Cover that contains the inputted Partition Covers.

Proof:

Pairs la_j, lr_j and ua_j, ur_j are the endpoints of the upper and lower lines of R by construction. As the maximum of a set is always greater than or equal to the minimum of the same set, R has the Partition Cover Property.

For any segment, the endpoints of the merged upper line lie above the endpoints

of the extended upper line of R' by construction and the endpoints of the lower line of R lie below the extended lower line of R'. As these are lines, having both endpoints of line l_1 above another line l_2 means that upon that segment any evaluation of l_1 lies above the evaluation of l_2 with the same input. The same holds true for l_1 having endpoints lie below l_2, l_1 lies totally below l_2 on that interval. Given $(i, y) \in [n] \times \mathbb{R}$, if (i, y) was inside segment j of R' then it lies in segment j for R and if it lay outside for R' then it lies closer to the upper or lower line of R than R' (or inside R). This implies that:

$$\begin{aligned} &\text{distSqr}_{PC}(R[j], T[i], i) \leq \min\{\text{distSqr}_{PC}(R'[j], T[i], i), \text{distSqr}_{PC}(R''[j], T[i], i)\} \\ &\implies \text{dist}_{PC}(R, T) \leq \min\{\text{dist}_{PC}(R', T), \text{dist}_{PC}(R'', T)\}\blacksquare \end{aligned}$$

Complexity Analysis The merge operation can be completed in $O(k)$ time, where k is the number of segments in the Partition Covers to merge. It uses $O(n)$ space as the object it returns is the size of the objects inputted and only a constant number of variables are defined inside the *for* loop on line 3.

Chapter 10

Evaluations

The project has created implementations of 10 algorithms to compare accuracy and efficiency; PAA, APCA, PLA, Interval Projection, Average Value, Weighted Average Value, Top Down, Bottom Up, Sliding Window and SWING Filters. A new indexing scheme has also been proposed.

Datasets

5 datasets were chosen to display results in this report, 4 from the UCR Time Series Archive and 1 generated by a random walk. The 4 UCR datasets are labelled *Arrowhead*, *ChlorineConcentration*, *DodgerLoopDay* and *Strawberry* and are chosen for their dissimilarity. Figure 3.1 shows *Chlorine Concentration*, *DodgerLoopDay* and *Strawberry* and Figure 6.2(a) displays the *Arrowhead* dataset.

The random walk uses independent identically distributed Normal random variables with mean 0 and variance 1. They are drawn from `std::normal_distribution<>` using a `std::mt19937` pseudo random number generator with seed 1.

Hardware All results were produced on a *Lenovo ThinkPad P14s* with an *Intel i7-1165G7 @ 2.80GHz* processor and 16GB of main memory.

10.1 Exact Segment algorithms

Euclidean Distance and Maximum Deviation

PLA algorithms are assessed by the Euclidean Distance from the approximation to the original and the Maximum Deviation of the approximation from the original. Euclidean Distance is a sensible choice as it represents the distance of a 'straight line path' from the approximation to the original, weighting extreme distances heavily. The Euclidean Distance provides a good analogy of the typical error of every point when divided by the size of the sequence.

Maximum Deviation gives an upper bound on the error any point can have from its approximation, this quantifies the 'tightness' of an approximation.

Parameters

Evaluation of algorithms will vary by the number of segments. Different methods may need different numbers of parameters to express a segment, so to evaluate accuracy fairly, each algorithm splits the sequence such that their approximation can lie as a point within the space $\mathbb{R}^m$ for some $m \in \mathbb{N}$. This removes biases as all algorithms reduce the sequence to the same lower dimension.

Figures 10.1, 10.2 and 10.3 were run on each dataset of size 900. This dataset size was chosen for the feasibility of testing APLA. The Double Sliding Window algorithms had window sizes of 5. Figures 6.2 and 6.3 show that the approximation benefits from a larger window size of 6 to 8 on smooth data and a smaller window size of 4 to 6 on rough data. A window size of 5 was chosen to be a compromise of these observations. *SkipWAV PLA* and *IncrWAV PLA* were programmed with the priority queue optimisation of Chapter 8.

PAA	Piecewise Aggregate Approximation
APCA	Adaptive Piecewise Constant Approximation
PLA	Piecewise Linear Approximation
AV PLA:	Average Value Double Window
IP PLA:	Interval Projection Double Window
SkipWAV PLA:	Weighted Average Value with Skip First Hypothesis
IncrWAV PLA:	Weighted Average Value with Increasing Value Hypothesis
APLA	Adaptive Piecewise Linear Approximation

Table 10.1: Key for Figures 10.1, 10.2 and 10.3.

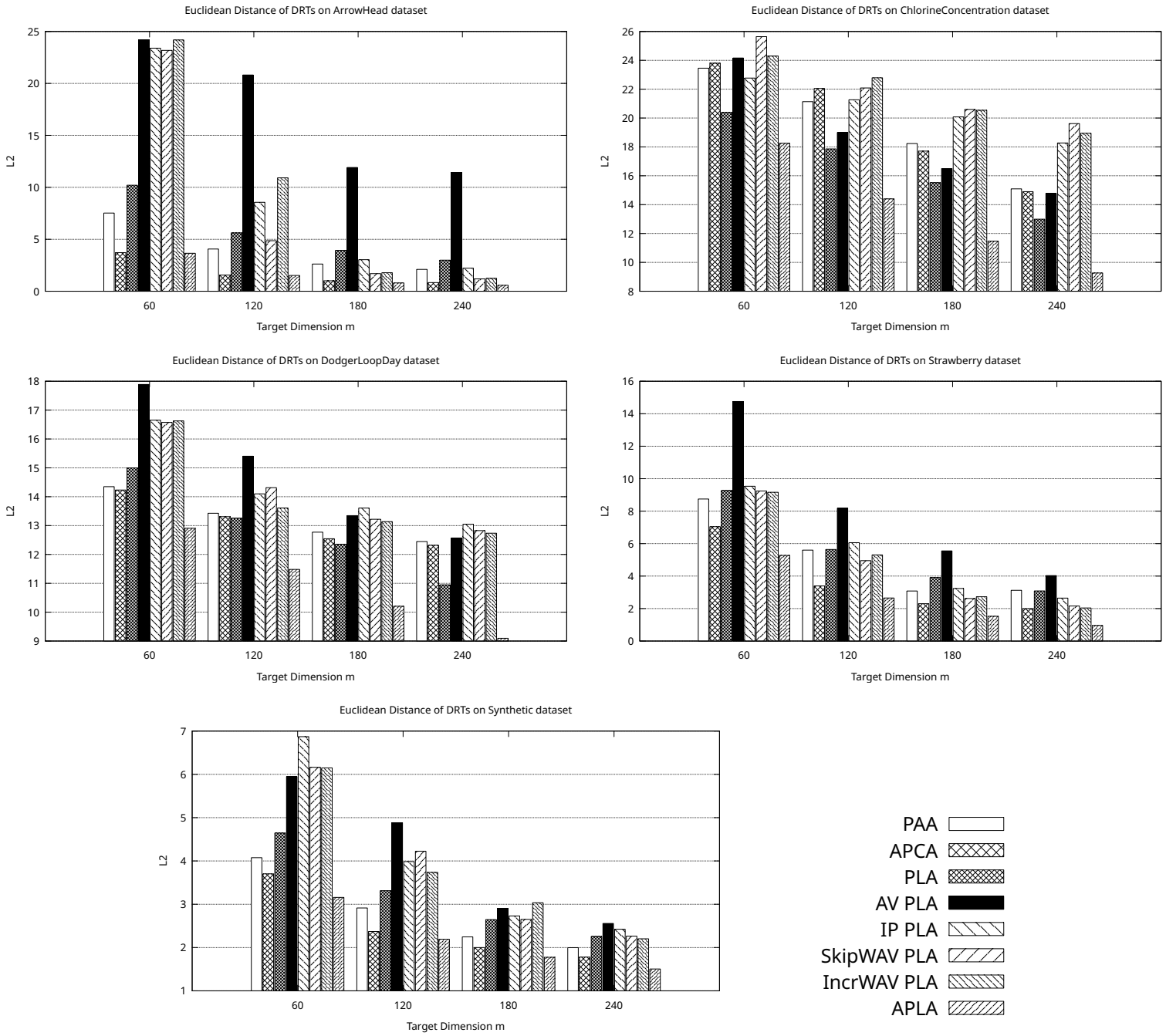


Figure 10.1: Euclidean Error of algorithms on the 5 datasets.

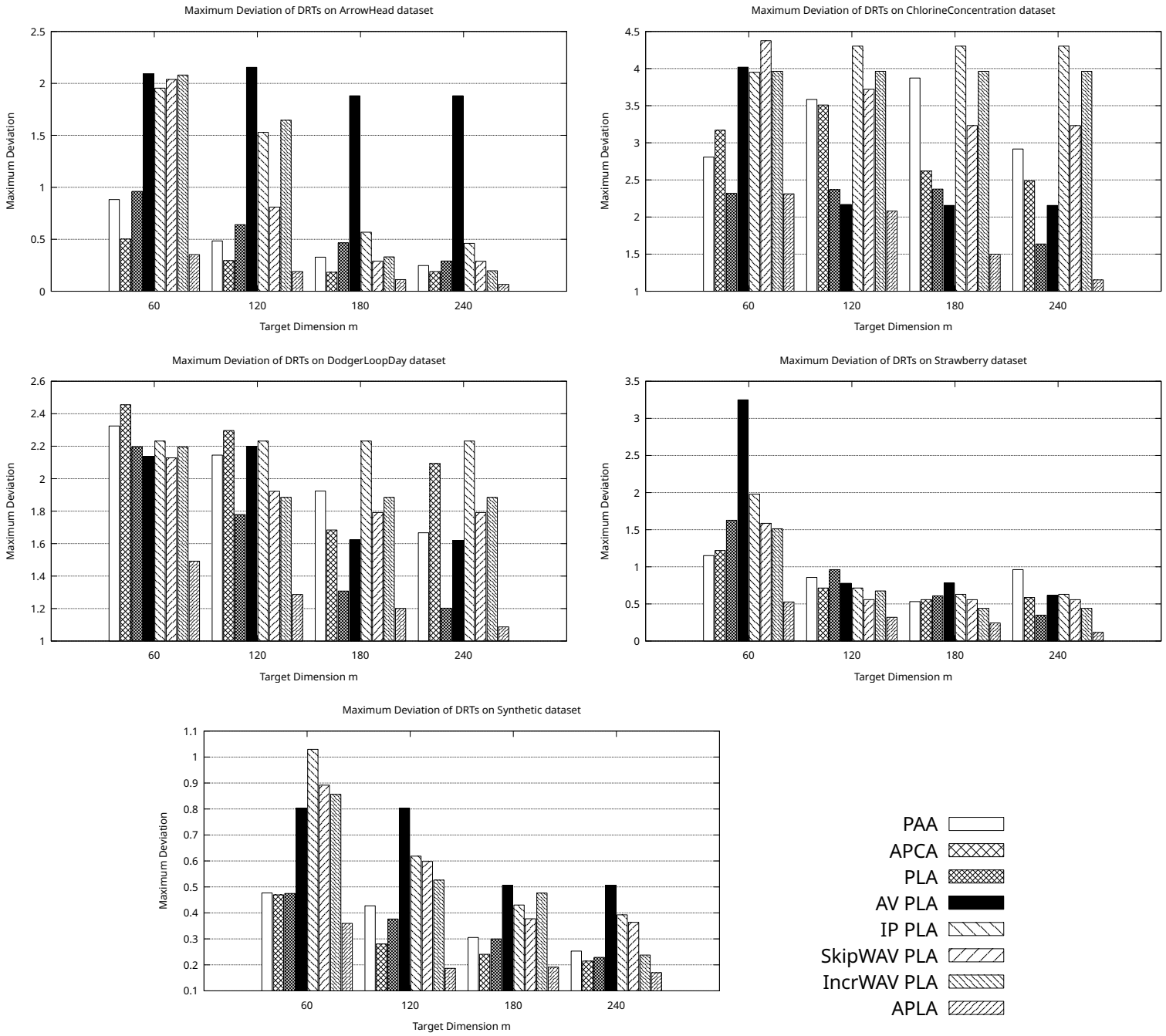


Figure 10.2: Maximum Deviation of algorithms on the 5 datasets.

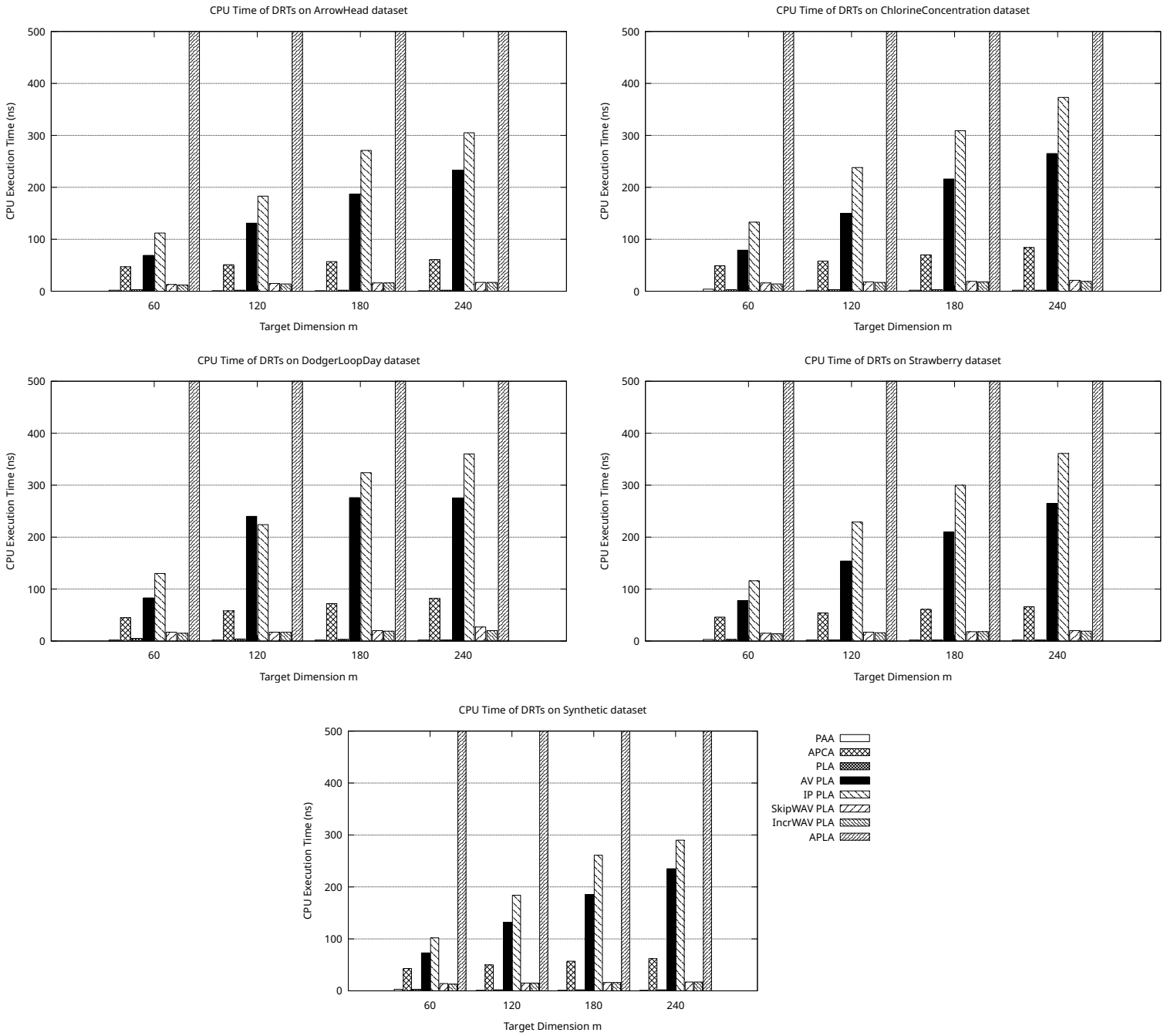


Figure 10.3: CPU Time to execute algorithms on the 5 datasets.

10.1.1 Observations

- **APLA is most accurate and slowest**

The modification of APLA for general PLA has lower Euclidean Distance and Maximum Deviation than any other Dimension Reduction Technique measured on all datasets. This is to be expected for the other PLA techniques as by construction, APLA is the optimal approximation, but not for other techniques.

APLA is significantly slower than the other techniques. It took APLA approximately 30 seconds to compute a partition into 80 segments whilst all other techniques took less than 0.5 microseconds (according to C++'s measuring library **Chrono**).

- **Double Window algorithms were inaccurate**

All Double Window techniques had larger Euclidean Error and Maximum Deviation than other methods, despite being more complicated and slower than simpler methods like PAA and PLA. The additional hypotheses of Weighted Average Value showed some improvement over Average Value but little improvement to Interval Projection.

- **Priority Queue Double Windows are faster**

SkipWAV PLA and *IncrWAV PLA* were implemented to use a priority queue to avoid recomputing results. Both were significantly quicker to execute than other double window algorithms (*AV PLA* and *IP PLA*).

10.2 Epsilon Precision Algorithms

Compression Ratio

The ε precision algorithms do not guarantee the same number of segments and cannot be analysed by changing the target dimension, instead the quality of an approximation was assessed by the number of segments the algorithm creates. The *Compression Ratio* is obtained by dividing the dimension of the approximation (the number of segments times 3) by the length of the original sequence. This is proportional to the number of segments and is a standard measurement that can be compared against the compression ratios of other DRTs.

Parameters

Evaluation of algorithms will be varied by the choice of ε.

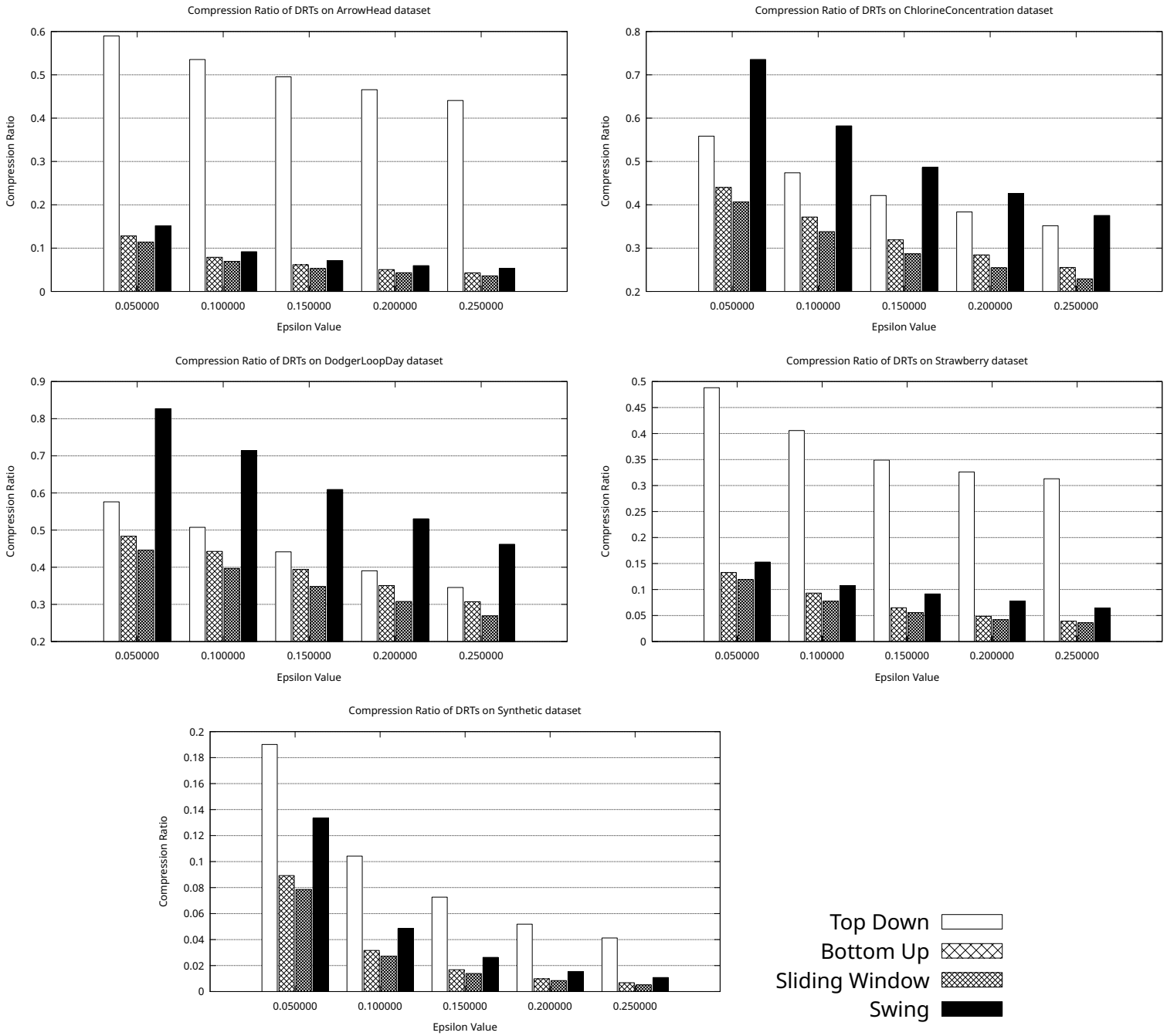


Figure 10.4: Compression Ratios of algorithms on the 5 datasets.

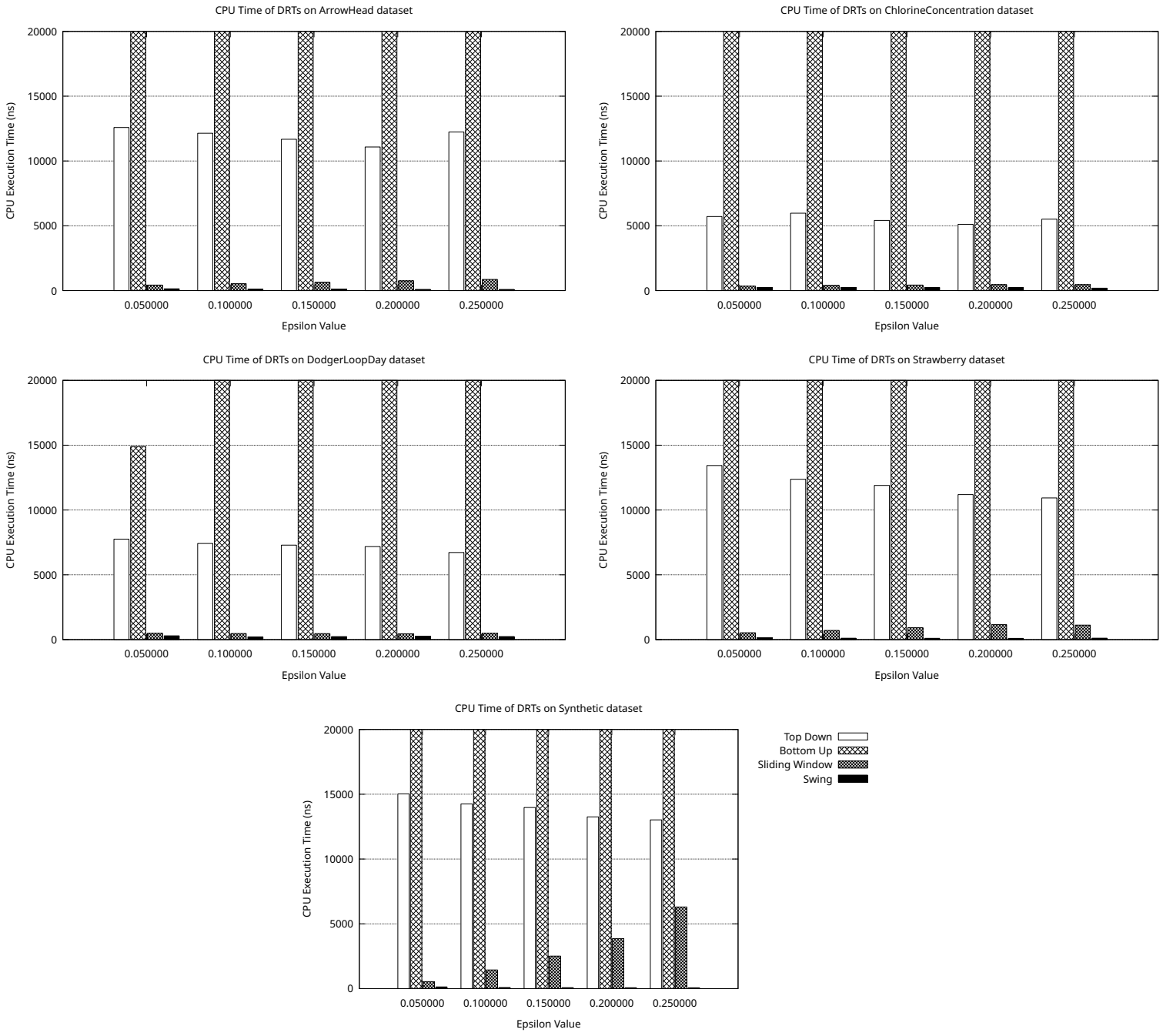


Figure 10.5: Time for CPU to execute algorithms on the 5 datasets.

Figures 10.4 and 10.5 show *Top Down*, *Bottom Up*, *Sliding Window* and *SWING* algorithms run on the 5 datasets. Dataset size for these plots was 20,000 as the algorithms being evaluated are more efficient.

10.2.1 Observations

- **Bottom Up is slower than other algorithms**

Bottom Up is slower than all other approaches, taking half a second to calculate the approximation. Bottom Up chooses the best segment to merge each iteration and requires deletions in the middle of the array. Data structures that perform deletions in constant time are inefficient in practice as there is a lack of cache locality. Using a less efficient data structure (a resizing array) that does maintain cache locality is also ineffective as it necessitates $O(n)$ cost deletions. By theoretical analysis using the optimisations, an execution time similar to Top Down could be achieved. Results incorporating Split and Merge into the approximations show Bottom Up performs faster alongside them (this can be seen in Figure 10.8).

Bottom Down has a high compression ratio (lower on graph) in comparison to other methods tested. This was confirmed in the review of Bottom Up by Keogh et al.[6].

- **Top Down has slow execution and a low Compression Ratio**

Top Down has the same problems as Bottom Up that asymptotic analysis won't detect. If it was necessary to maintain the order of the segments (as for Bottom Up) then Top Down's execution time would would be similar to Bottom Up's. The segments in Top Down are *split* not *merged*, so the order of segments in a data structure does not matter and can be appended and deleted from the end. Resizeable arrays have a $O(1)$ cost to inserting and deleting from the end allowing both cache locality and efficient asymptotic time.

Top Down has a poor Compression Ratio (denoted by a higher value) in comparison to the other techniques. It performs worse than other algorithms on smooth datasets, *ArrowHead* and *Strawberry*.

- **Sliding Window algorithm has a high Compression Ratio**

For every dataset and every Epsilon Value, Sliding Window offers the highest Compression Ratio. This appeared to be the case for the majority of datasets in the UCR Time Series Archive. Only for *ScreenType*, *FreezerRegularTrain* and *FreezerSmallTrain* datasets did Bottom Up outperform Sliding Window when comparing the first 2000 values of the TRAIN datasets with an epsilon of 0.15.

The low Compression Ratio of SWING indicates that fixing the endpoints of segments has a sizeable impact on the quality of an approximation.

10.2.2 Precision Methods with Exact

Figures 10.6, 10.7 and 10.8 were run on every dataset in the UCR Time Series Archive of size 10,000. Every DRT was included apart from APLA due to time constraints. The Double Sliding Window algorithms had window sizes of 5 and epsilon algorithms have $\varepsilon = 0.1$. Split and Merge were used to adapt precision precision algorithms to have a fixed number of segments. Table 10.2 explains all abbreviations of keys. Figures for all other time series in the UCR Archive are

available on GitHub in the project repository[22]. It should be noted that the plots of Figure 10.8 have logarithmic scaling to enable all algorithms to appear on the same plot.

PAA	Piecewise Aggregate Approximation
APCA	Adaptive Piecewise Constant Approximation
PLA	Piecewise Linear Approximation
AV PLA:	Average Value Double Window
IP PLA:	Interval Projection Double Window
SkipWAV PLA:	Weighted Average Value with Skip First Hypothesis
IncrWAV PLA:	Weighted Average Value with Increasing Value Hypothesis
RDP:	Ramer-Douglas-Peucker (Top Down)
B-U:	Bottom Up
SW:	Sliding Window
SWING:	SWING filter

Table 10.2: Key for Figures 10.6, 10.7 and 10.8.

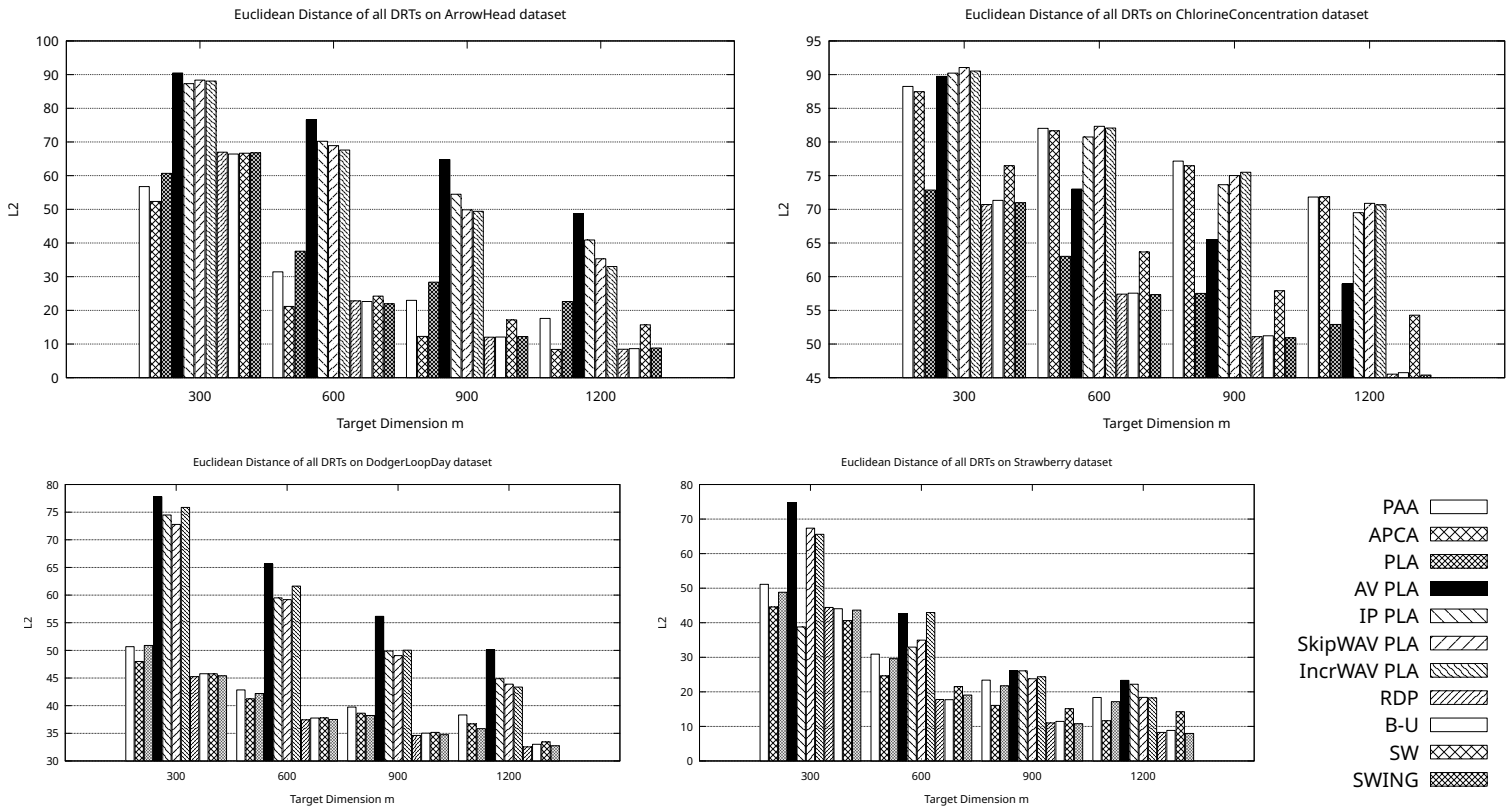


Figure 10.6: Euclidean Error of both algorithm types on the 4 UCR datasets.

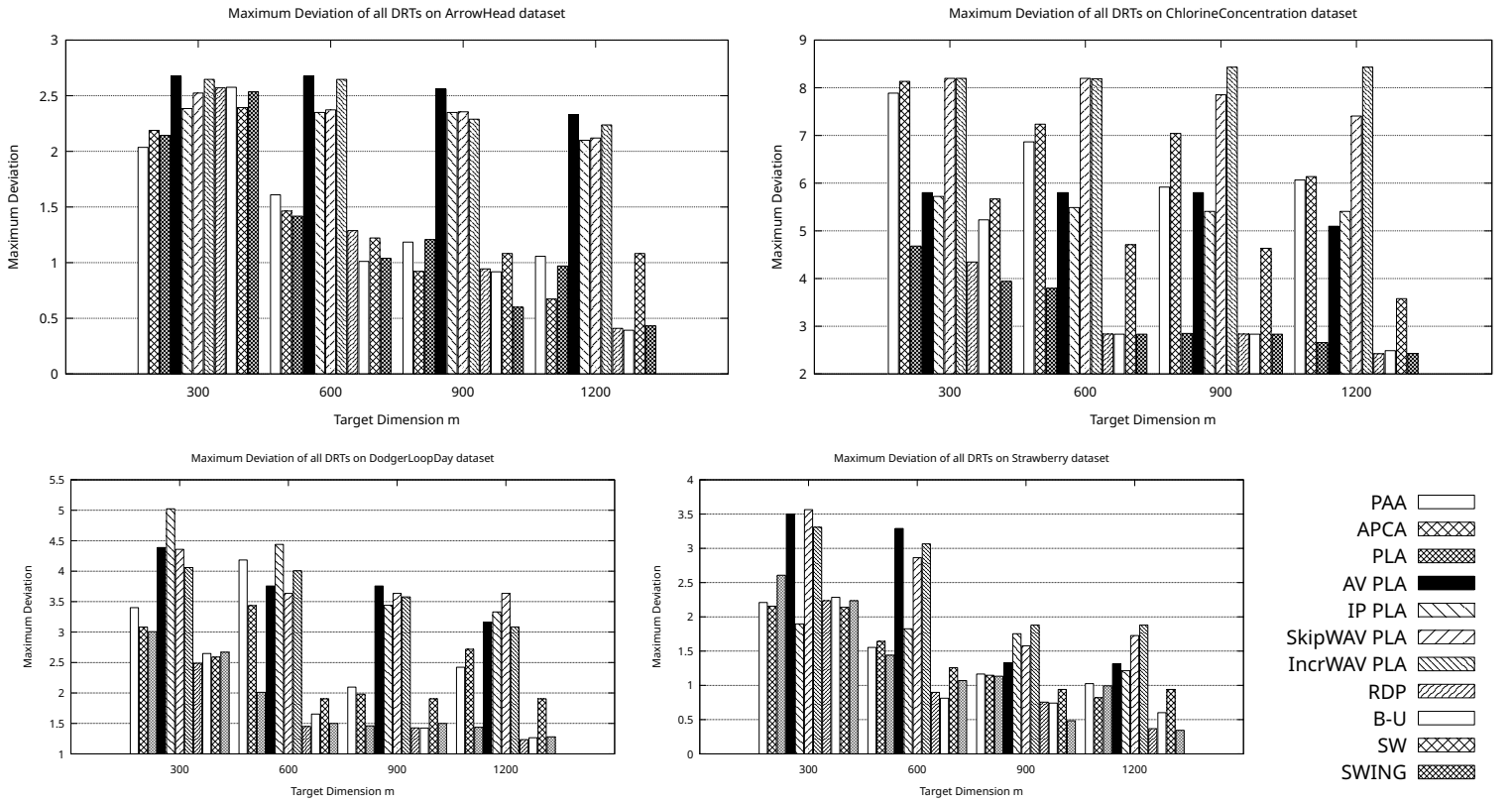


Figure 10.7: Maximum Deviation of both algorithm types on the 4 UCR datasets.

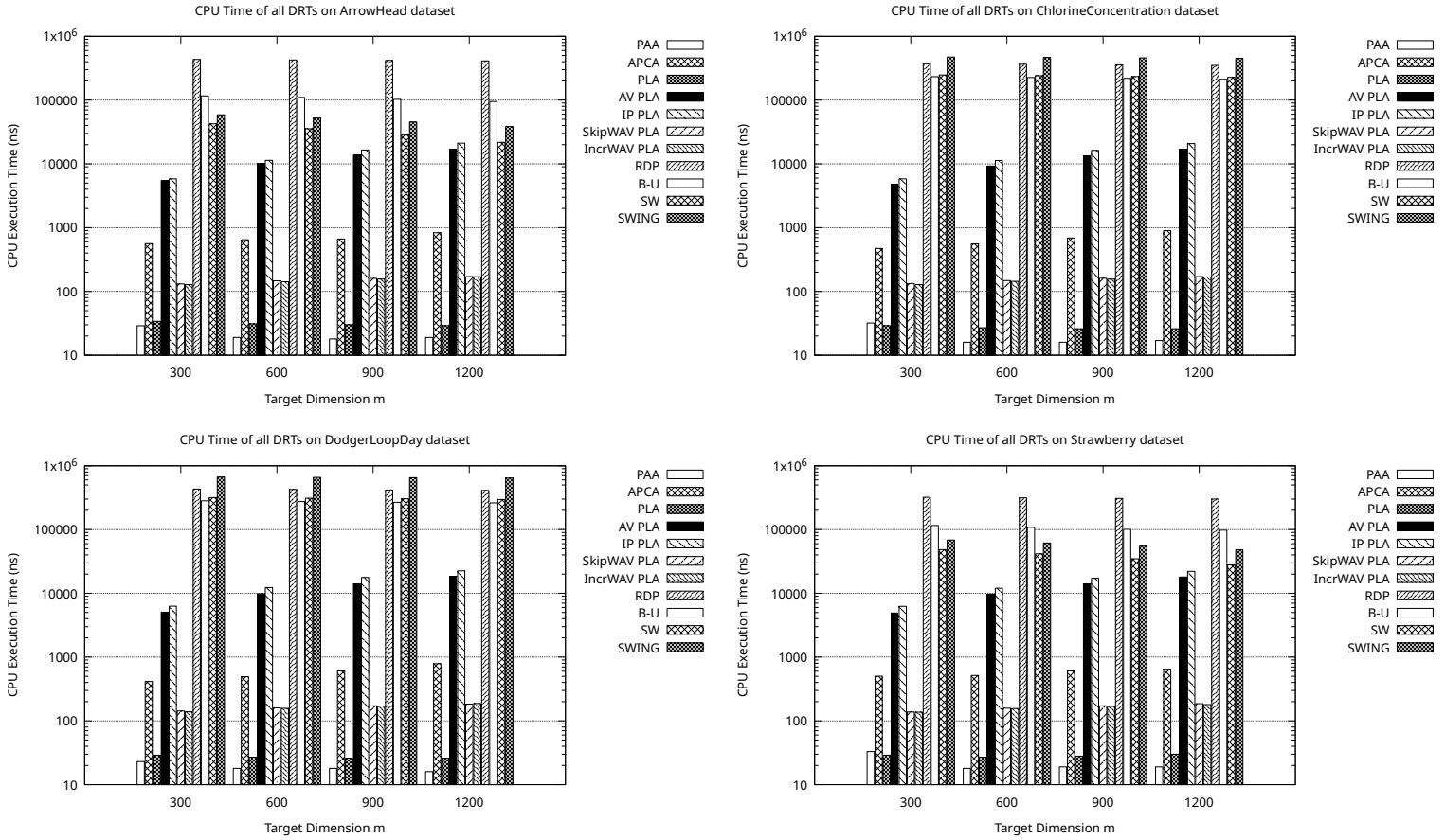


Figure 10.8: CPU Time of both algorithm types on the 4 UCR datasets.

10.3 Pruning Power

The pruning power is calculated by x divided by y. Where x is the number of *false dismissals* an indexing scheme makes before it guarantees a 1 Nearest Neighbour search of a query and y is the total number of subsequences. For a given query sequence Q and sequence S (and after creation of the index scheme upon S) x is the number of subsequences s of S for which the distance from s to Q must be calculated and y is the total number of subsequences of S. A greater pruning power is a smaller value, as less subsequences of S have been considered in the search.

The pruning power of Average Value Double Window (AV), Interval Projection Double Window (IP), Top Down (TD), Bottom Up (BU) and Sliding Window (SW) were found for the indexing scheme and shown in Figures 10.9 to 10.12.

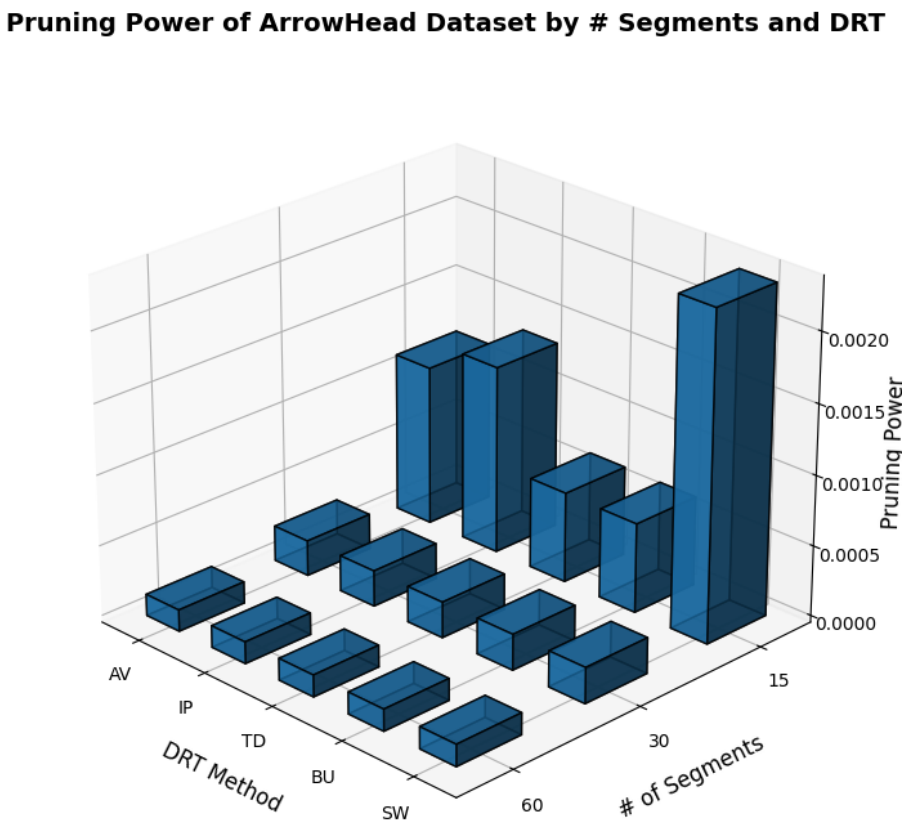


Figure 10.9: Pruning Power of Adaptive PLA methods on ArrowHead.

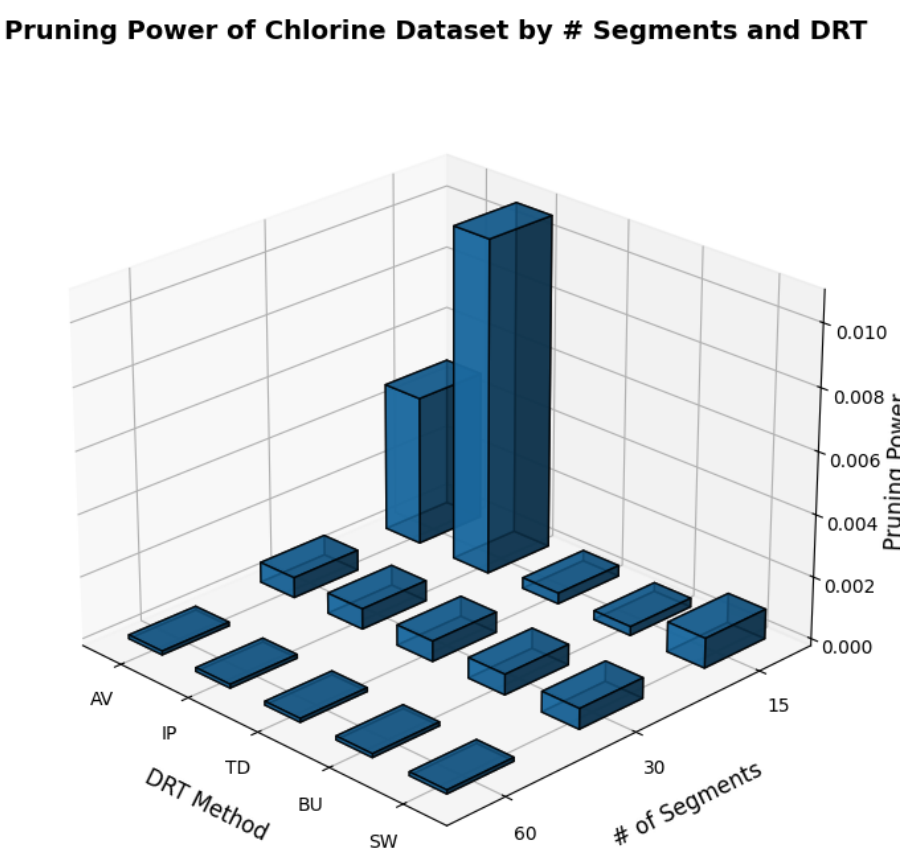


Figure 10.10: Pruning Power of Adaptive PLA methods on Chlorine.

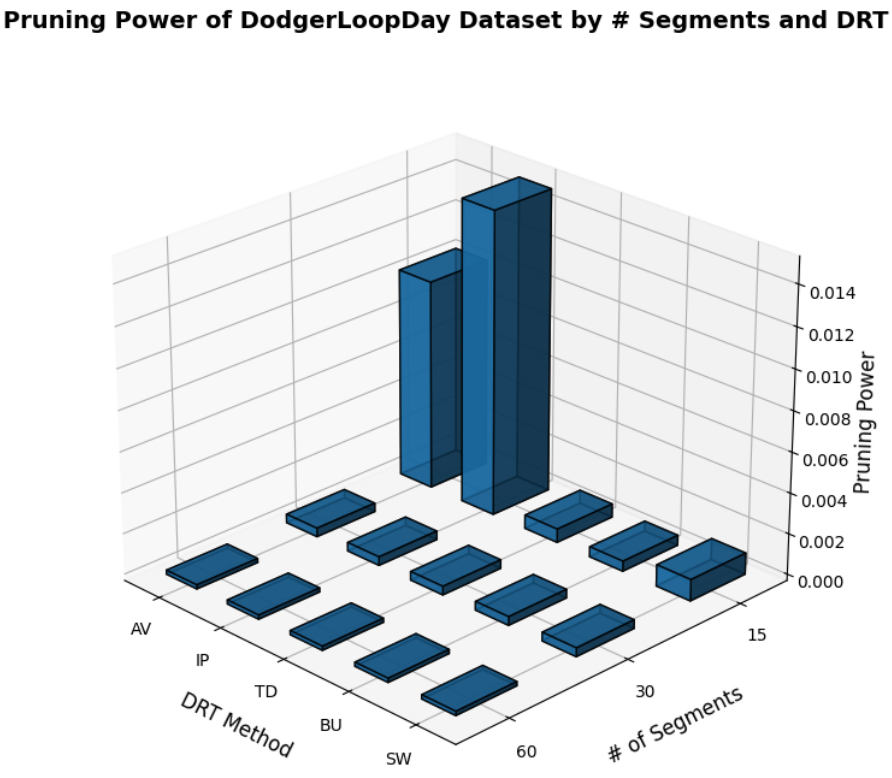


Figure 10.11: Pruning Power of Adaptive PLA methods on DodgerLoopDay.

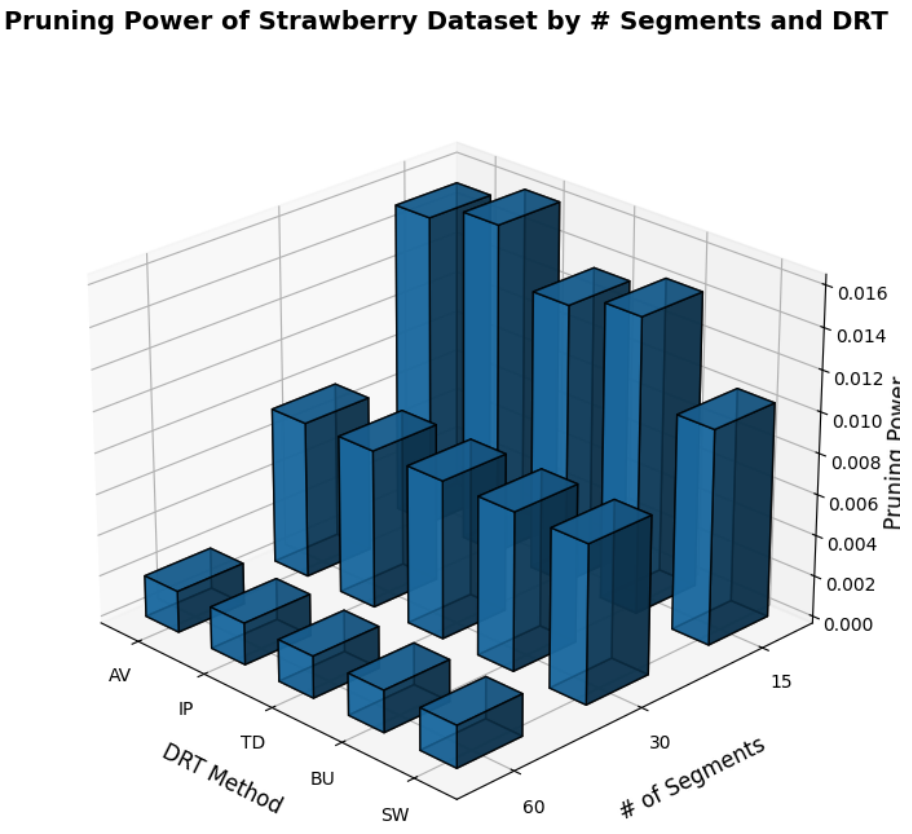


Figure 10.12: Pruning Power of Adaptive PLA methods on Strawberry.

Top Down, Bottom Up and Sliding Window are ε precision guarantee algorithms, so the Split and Merge algorithms have been used to guarantee that any subsequence is mapped to the same number of segments. The datasets are of size 20000 and subsequences of size 600 were indexed. Every subsequence was mapped to 15, 30 and 60 segments for Adaptive PLA respectively and query subsequences were generated by indexing 1000 evenly spaced subsequences of the larger sequence. Each query subsequence had every value perturbed by some Normal(0, 0.1) noise so queries were 'new' and the average of the pruning power for each query was recorded.

10.3.1 Observations

- **All pruning powers are excellent results**

A pruning power of 0.015 (the highest value recorded in Figures 10.9 to 10.12) indicates that only 300 entries were scanned for the 1 Nearest Neighbour search. The indexing scheme appears to be robust against the choice of Dimension Reduction meaning that different schemes appear to have minimal impact on performance. This is probably due to the loss of accuracy by merging Partition Covers exceeding any difference made by choices in DRT or numerical precision becoming an issue after thousands of merges.

- **Average Value and Interval Projection have lower pruning power**

All four figures show that Average Value and Interval Projection have pruning powers greater than 0.001 when dividing the subsequences into 15 segments. This is poorer than other algorithms and fits with the evaluations of Exact Segment algorithms, that showed they have high Euclidean Error and Maximum Deviations for approximations less than a tenth of their original size.

- **Sliding Window has lower pruning power than Top Down or Bottom Up**

Sliding Window has less pruning power than Top Down or Bottom Up in the majority of the datasets. On the ArrowHead dataset, Sliding Window has a significantly poorer pruning power than other methods, with a value twice as large as any other method.

10.4 Similarity Search and K-NN

Figure 10.13 shows the indexing scheme completing both Similarity Search and K-NN.

Similarity Search was performed on the first 2000 values of the *ECGFiveDays* TEST dataset with the *pink* line as the query subsequence. Subsequences that matched within a $\varepsilon = 7$ are shown in *green* and the underlying series is shown in *purple*. The dataset was split into subsets of size 50 and each was converted to an Adaptive PLA representation of 5 segments by a Bottom Up approach.

K-NN was performed on the first 1000 values of *Strawberry* TRAIN dataset from the UCR Time Series Archive. The *pink* line is the query subsequence (lines 150-230), the 15 closest subsequences are each highlighted in *green* and the underlying sequence in *purple*. To match all 4 of the 'dips' that we recognise, the 15 closest subsequences were required. Subsequences that are shifted from a very close match

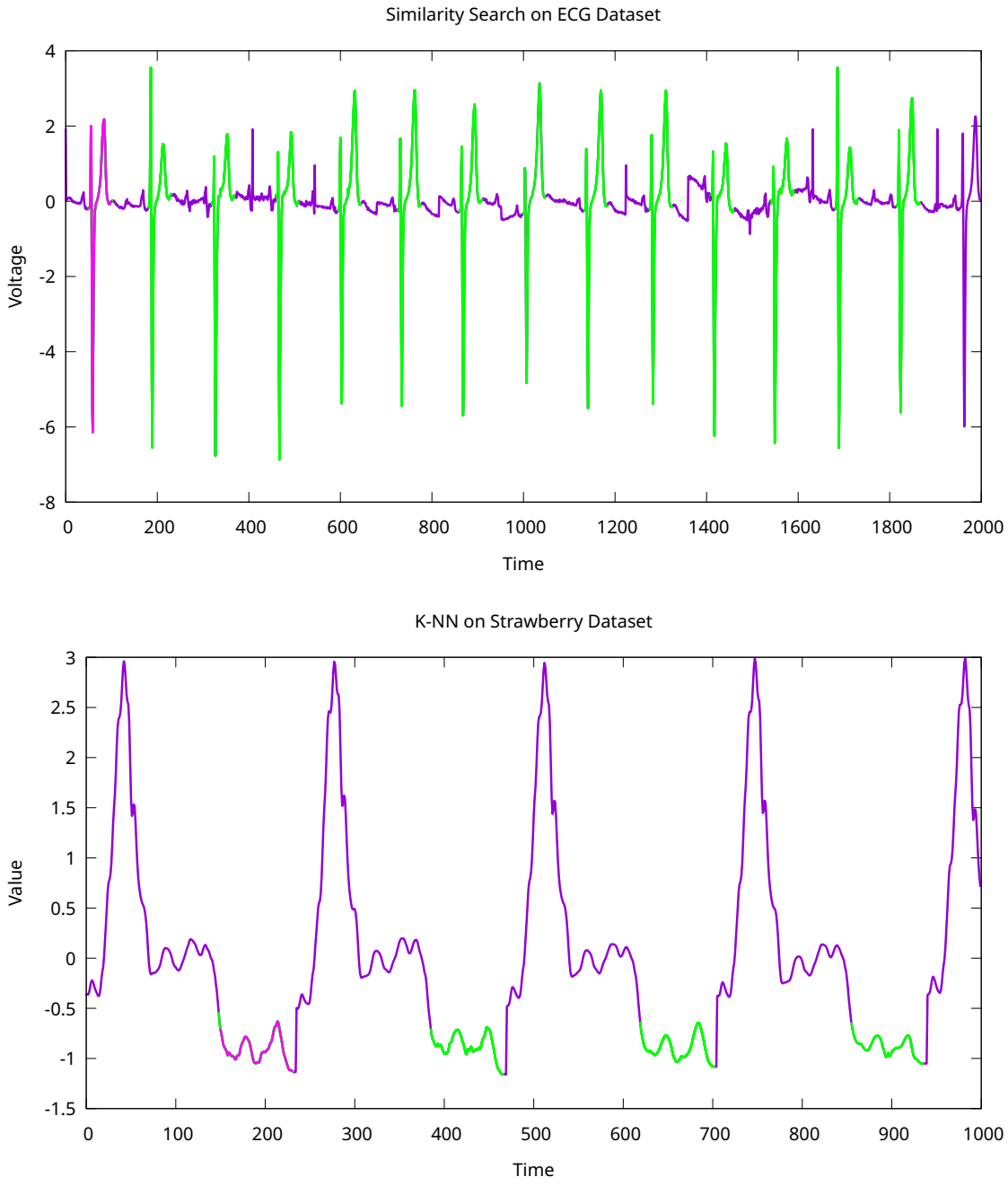


Figure 10.13: Execution of Similarity Search and K-NN on UCR Datasets.

are often closer than separate subsequences that a human would recognise. The sequence was split into subsequences of size 80 and expressed by 8 segments by the Top Down algorithm.

To program Similarity Search and K-NN a spatial indexing structure was required. For this project a R-Tree[23] was chosen, with a maximum of 40 leaves and a minimum of 10 leaves per node. Partition Covers were chosen to be merged by the minimal area the merged Partition Cover would occupy in the plane.

Chapter 11

Project Management

11.1 Achieving Objectives

In the beginning of the academic year, the project specification was created with 5 objectives:

- Research novel PLA techniques to solve Similarity Search problem on Time Series Data.
- Research methods to improve PLA via optimisation techniques.
- Provide description and theoretical analysis of methods (time complexity or if time error bound guarantees).
- Design implementations of these methods.
- Test implementations on real and synthetic datasets. Provide reproducible benchmarks, test scalability and compare methods.

By the end of term 1, I had researched Top Down and APLA methods, designed the Double Window technique with help from my supervisor, integrated the UCR Time Series Archive and synthetic data into the project and conjectured optimisation via the priority queue. I was ahead of schedule so I created extensions that would develop the project further.

- Research indexing methods for adaptive PLA.
- Research and implement the GEMINI framework for PLA.

Researching indexing methods for adaptive PLA took considerably longer than anticipated over the course of the second term as papers that contained indexing methods (APLA by Ljosa and Singh[7] and SAPLA by Xue et al.[5]) imposed additional constraints or provided too few details for implementation.

Over the course of this report, I have described 7 techniques to produce adaptive PLA representations of Time Series Data, all of which can be implemented with the indexing scheme to solve Similarity Search. Priority Queue optimisations were found for the Split and Merge algorithm and Double Window, reducing asymptotic time complexities.

For all the algorithms in this paper, detailed descriptions are provided, with pseudocode and analysis of the algorithms design and time complexity. Most of

the algorithms had error bound guarantees and proofs of these error bounds were provided.

All algorithms have been implemented in C++ and are unified in framework, allowing them to integrate into the larger suite of tools developed for this project. Implementations can be seen on the GitHub repository for this project[22].

All algorithms have been tested by the accuracy and efficiency of their approximations on both real and synthetic datasets. The source of the data and how it can be generated and z-normalised has been described already. All parameters of the algorithms are stated and run on the same hardware for consistent comparisons.

No suitable indexing scheme for this project was found, so one was created using techniques from Chakrabarti et al. [4]. Algorithms were tested within the indexing scheme to analyse their pruning power.

Finally a full implementation of GEMINI was created, enabling Similarity Search and K-Nearest Neighbours to be performed on Time Series Data.

11.2 Project Organisation

Figure 11.1 shows three timetables describing the plan for the project and their execution. *Project Specification Timetable* shows the timetable created for the project in the Project Specification in week 2 term 1. *Progress Report Timetable* shows the updated timetable released in the Progress Report with additional objectives included due to being ahead of schedule. *Actual Timetable* shows how much time was actually spent on each task over the course of both terms.

The timetables show some key differences. I over-estimated the time tasks in the first term would take, so a lot of smaller tasks were completed quickly in term 1 and the actual timetable displays a lot more bars. This encouraged the additional objectives declared in the Progress Report and necessitated an updated timetable, hence only the second term being displayed.

The *Progress Report Timetable* shows *Implement GEMINI* was only allotted 3 weeks to find and implement an indexing scheme. The Progress Report stated "in the worst case, any PLA technique can be applied to the GEMINI framework via additional tools set out in [5]" but this was incorrect as SAPLA offers a different structure to indexing. This mistake caused a large amount of time to be dedicated to researching Indexing Schemes and led to the development of Partition Covers. A task expected to take 3 weeks took 7 weeks to complete.

Despite these deviations, efforts were made to stick to the timetable as much as possible.

The generous time allocated for easier tasks in the Project Specification (quote "the timetable allows for periods of little progress by generous deadlines") meant some tasks finished a week earlier than scheduled, and allowed extra time to be allocated to harder tasks so they were still finished only a week after scheduled.

11.2.1 Methodology Evaluation

Research Methodology

Both **Algorithm Discovery** and **Optimisations Research** involved researching challenging topics that risked delaying the project. Bi-weekly meetings with my project supervisor alleviated the majority of the issues I encountered, but I realised

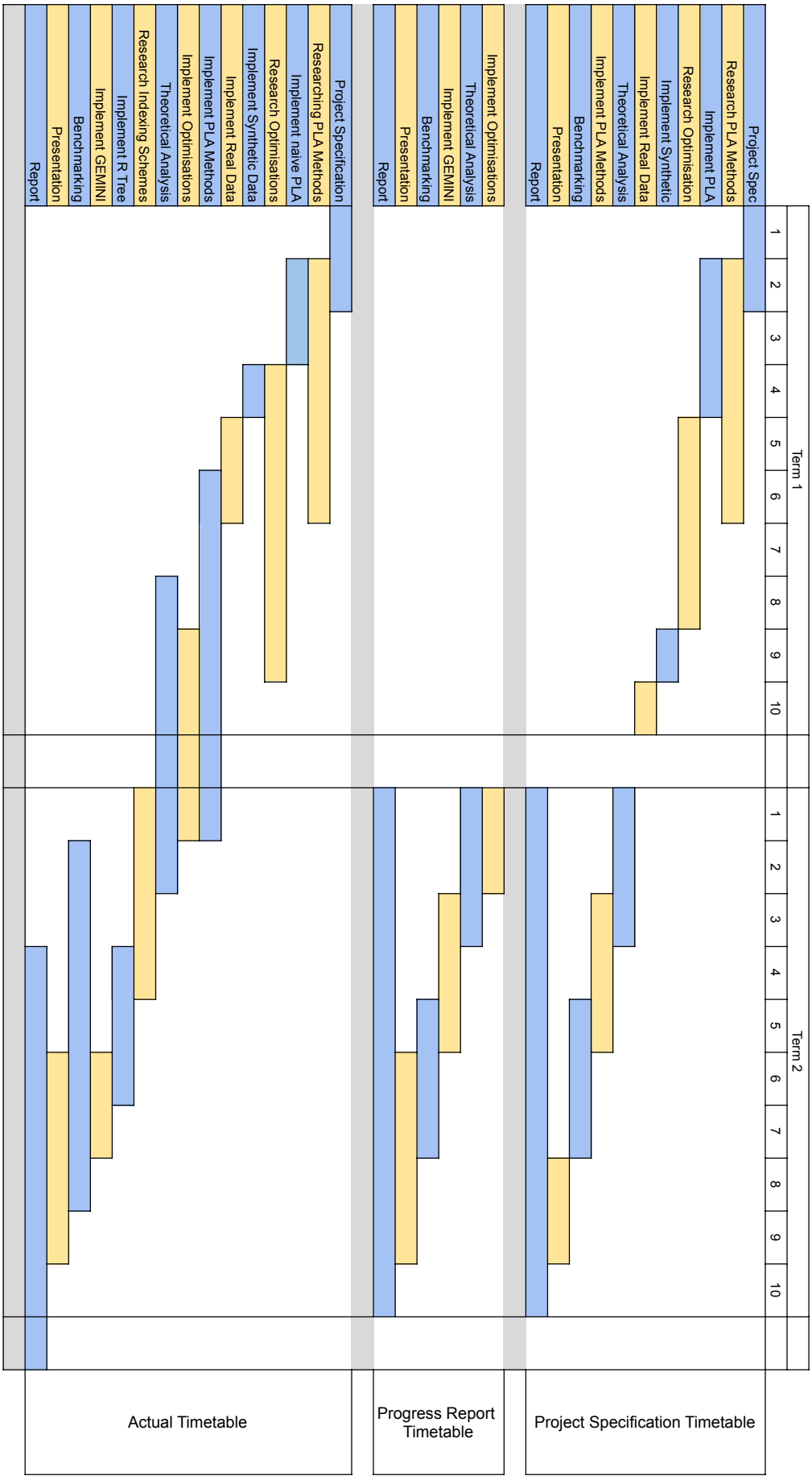


Figure 11.1: Three timetables of project.

that an additional framework was required to ensure that research time overwhelm the project. Two key risks were identified with research and steps to mitigate the risks were detailed.

- Difficulty Understanding
  - Many papers develop complex algorithms to leverage aspects of the time series that may be difficult to parse or understand to form better linear approximations.
  - Writing my own pseudocode enabled challenging issues to be identified at an early stage allowing them to be discussed with my supervisor before too much time was invested.
- Research Creep
  - Papers developing algorithms may reference multiple other papers which then need to be reviewed, making a simple research task much larger than expected.
  - For example, the details for APCA indexing scheme are split across APCA paper [4], GEMINI Framework [3] and Spatial Indexing Structures like R-Trees[23].
  - This became more of a problem with the additional topics in term 2, and mitigation had to be added in - formalising the intentions of research before it was undertaken and sticking to a strict 1 week maximum for research.

Practical Methodology

The project is primarily a research project evaluating algorithms but it necessitates a large project codebase to achieve this, so a robust practical methodology was required as well. Multiple risks affecting development were conjectured and new ones encountered over the project. Solutions to each are detailed below.

- **Machine malfunction:** If the project became inaccessible on the machine, regular pushes to *GitHub* meant that the project was backed up to the Cloud regularly and could be restarted from another machine if necessary.
- **Dataset Inaccessibility:** If datasets became inaccessible either online or on my machine then fortunately there was enough space on the host machine to install all datasets. Also many of the UCR Archive [12] datasets existed elsewhere online as well. Synthetic data was also produced for this project, so a lack of data would not be an issue.
- **External Library Issues:** If libraries chosen didn't fulfil the role intended for, or became unsuitable over the course of the project, the *Nix* repository enabled rapid installation of others and even test environments to ensure their suitability.
  - For example, the *matplotlibcpp* library was initially intended to render test results. The project switched to using the *gnuplot-iostream* library by changing a single line in *flake.nix*.

The algorithm development process created upon Agile values[13] enabled many algorithms to be quickly and correctly implemented for the project. Demonstrations of the algorithms with my supervisor encouraged adaptions and new algorithms; Interval Projection came as a suggestion of my supervisor.

11.2.2 Considerations of time limitations

Under-estimation of the time required to implement the GEMINI Framework and a bout of sickness in term 2 meant that despite my best efforts, time was a constraint upon the project. The project achieved more than I planned in my Project Specification (ie. there was no expectation of creating an indexing scheme or lower bounding distance) but other interesting avenues for research that could have been investigated and developed as part of the project could not be explored. These include those mentioned below.

- **SLIDE Filter**
  - Elmeleegy et al.[16] has a second filter, SLIDE, that does not require that segments must have matching endpoints, producing better approximations with a time complexity of $O(n^2)$.
  - I planned to implement SLIDE as an optimisation of Sliding Window, but I didn't have time to complete SLIDE analysis.
- **Optimised GEMINI**
  - GEMINI has been implemented using the indexing scheme, but using only a simple Spatial Indexing Structure and without optimisations.
  - GEMINI was completed whilst ill and with the presentation and report both needing to be completed. The R-Tree was chosen as a Spatial Indexing Structure for its simplicity with only methods for insertion and searching implemented (no deletion) to obtain a minimum viable solution. Ideally I would have implemented deletion methods as optimisations to R-Tree require altering the tree or chosen a more efficient Indexing Structure like R*-Trees.
  - Practical optimisations for the indexing scheme via improving the selection of which Partition Covers to merge did not have time to be completed meaning that currently it doesn't achieve effective pruning of the search space.

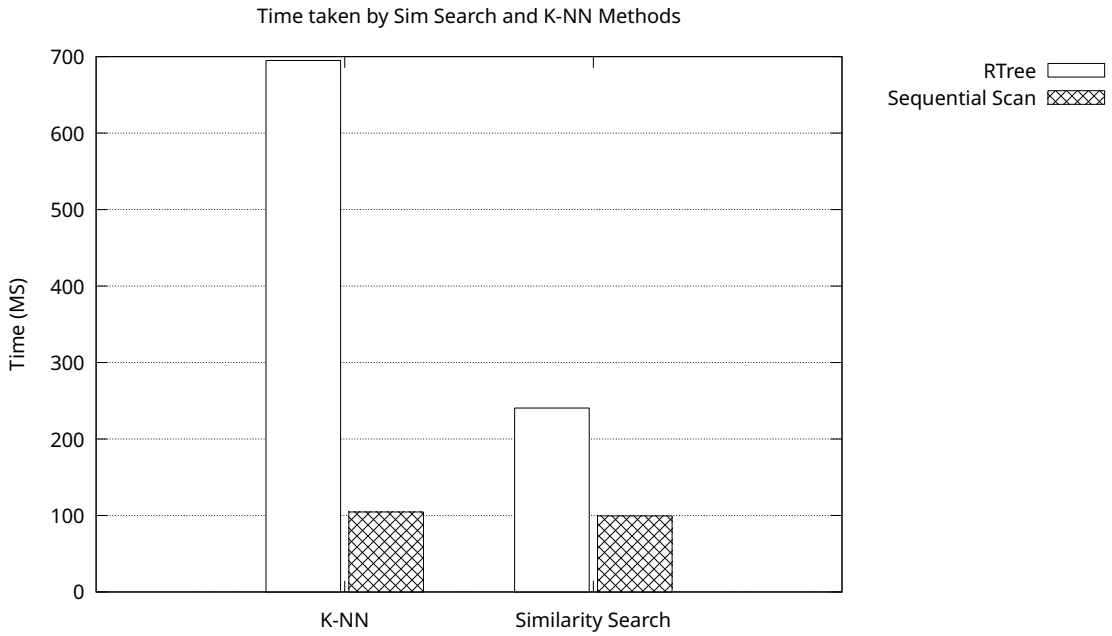


Figure 11.2: Plot of Unoptimised GEMINI and Sequential Scan

– Figure 11.2 show the impact of this upon the execution time. A synthetic Normal(0,1) random walk of size 500,000 was generated (seed for pseudo-random generator of 1) and subsequences of size 600 were compressed to 30 segments by Top Down. 500 subsequences of the walk were perturbed by the walk generated by N(0, 0.001) (seed 0) and results for a $\varepsilon = 0.1$ SimSearch and $k = 30$ K-NN search were found for both GEMINI and sequential.

– If the framework could be optimised close to the projected pruning powers, then this indexing scheme will outperform the Sequential Scan method.

11.3 Discussion of Ethical, Social and Legal considerations

Research

Research All research sources have been fully attributed. Any quotations taken from papers are taken in their entirety. All sources of inspirations from techniques from others are cited and these are referenced.

Data Real data used in this project all comes from the UCR Time Series Archive[12]. This is a collection of datasets assembled by the Time Series Data community to enable consistent and reproducible results of research. All data in this Archive has been granted free for research use and quote "none of the data sets in the UCR Archive motivates the need for privacy"[24].

Software and Libraries Table 11.1 displays the licenses of software and libraries used. In every case the software is allowed for the use of this project, and the project can be published online without special considerations under the MIT License.

Software name	Use Allowed	Source
pgbar	Yes	MIT License https://github.com/Konvt/pgbar/blob/main/LICENSE
Boost	Yes	Boost Software License https://www.boost.org/LICENSE_1_0.txt
GnuPlot	Yes	http://gnuplot.info/faq/index.html#x1-110001.6
GTest	Yes	https://github.com/google/googletest/blob/main/LICENSE
GnuPlot Iostream	Yes	MIT License https://github.com/dstahlke/gnuplot-iostream/blob/master/LICENSE
Valgrind	Yes	GNU General License https://valgrind.org/docs/manual/licenses.html
Doxygen	Yes	GNU General License https://doxygen.nl/license.html
CMake	Yes	BSD 3-clause License https://opensource.org/license/BSD-3-Clause

Table 11.1: Table of Licenses for Software and Libraries

Chapter 12

Conclusions

12.1 Project Conclusions

This project reviewed state of the art techniques for Time Series Data approximation via Piecewise Linear Approximation. This involved discussing algorithms, optimisations and conversion methods to fixed number of segments. A new indexing scheme was created that can be used to perform Similarity Search using Adaptive PLA and algorithms were compared. Bottom Up was the best performing across all distance measures assessed for compression algorithms with reasonable execution times.

A project repository was created with a suite of tools for data generation, parsing and cleaning. Functionalities for saving data were also created so random walks could be generated once and saved. The repository is available online and open source on GitHub at [22].

A large variety of algorithms for 'local' Dimension Reduction Techniques were implemented and optimisations found to improve performance. Split and Merge were implemented to convert precision guarantee algorithms to a fixed number of segments.

Algorithms were evaluated both as approximations alone and within the indexing scheme with Bottom Up performing the best. Results show that my algorithms to approximate using a fixed number of segments are considerably faster but less accurate than algorithms with precision guarantees.

A new indexing scheme was found for Adaptive PLA. The scheme has advantages over previous schemes [7][5][25], as once Adaptive PLA is calculated for subsequences, the original sequence can be moved to storage and isn't needed for Partition Cover construction. With approximations that have low Euclidean Error, this indexing scheme offers Pruning Power. Implementation of Similarity Search and K-Nearest Neighbour show the indexing scheme performs well on real world data, locating heart beats and features of spectroscopy graphs (Figure 10.13).

12.2 Future Work

Future work of this project would primarily address the current limitations of the project, completing SLIDE filters and optimising the GEMINI implementation for my indexing scheme.

Dynamic Time Warping

Dynamic Time Warping could be created as an additional mechanism for comparing Time Series and assessing similarity between sequences. The distance measure is more *elastic* and better approximates scenarios in real world data where sensors may report too late or not at all, incurring dissimilarities in data that would otherwise be the same.

A distance measure 'matches' elements of two sequences together and compares all matches. Euclidean Distance and Maximum Deviation match only elements with the same indices but Dynamic Time Warping allows elements of the first sequence to match with multiple elements of the second and with elements before or after its index. Dynamic Time Warping takes these matches and considers the minimum cost match for every element of the first sequence.

Other indexing schemes

This paper focussed on an indexing scheme that uses the Adaptive PLA algorithm but indexing schemes exist that use other compression methods like Discrete Fourier Transform or APCA. A review into other Dimension Reduction Techniques with indexing schemes ([26] [27] [3] [26] [4]) and other current Adaptive PLA approaches ([7] [5] [25]) would be a substantial project in its own right.

Big Data

Large datasets cannot be stored entirely within main memory and require handling in parts. Indexing is done via approximations of the data and indexes of subsequences within ε (some indexes may be false positives) are returned. These indexes are then used to pull the corresponding subsequences into main memory, meaning that the whole sequence never has to all be in main memory at once. A full implementation of storage and retrieval of large time series data is not present in the project repository but would offer an example of using the indexing scheme and Adaptive PLA algorithms for Similarity Search on Big Data.

12.3 Closing Remarks

This project was conceived as the analysis of a few algorithms upon a fundamental problem of Time Series Data with the assumption that the GEMINI Framework was a closed box that took a Dimension Reduction Technique and 'achieved' Similarity Search. This assumption was incorrect and the project developed into something larger and more impressive than its original conception.

What the final project offers is twofold; insight into a subsection of the TSD Analysis ecosystem and software offering a suite of tools to analyse approximation algorithms on Time Series Data and perform high dimensional indexing. This report contains the methods, requirements and details necessary to implement and analyse approximation algorithms for Adaptive PLA and the structure for analysis of any approximation algorithm.

The repository contains 4000 lines of data generation, approximation algorithms, spatial indexing techniques and analysis tools - all documented to enable anyone to experiment, replicate or extend with the work presented here.

Bibliography

[1] Statista. *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2023, with forecasts from 2024 to 2028*. https://www.statista.com/statistics/871513/worldwide-data-created/. 2025. (Visited on 04/01/2025).

[2] Thanawin Rakthanmanon et al. "Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping". In: *KDD : proceedings. International Conference on Knowledge Discovery & Data Mining* 2012 (2012), pp. 262–270. URL: https://api.semanticscholar.org/CorpusID:3018103.

[3] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. "Fast subsequence matching in time-series databases". In: *ACM SIGMOD Conference*. 1994. URL: https://api.semanticscholar.org/CorpusID:6595462.

[4] Kaushik Chakrabarti et al. "Locally adaptive dimensionality reduction for indexing large time series databases". In: *ACM SIGMOD Conference*. 2001. URL: https://api.semanticscholar.org/CorpusID:15192608.

[5] Ruidong Xue, Weiren Yu, and Hongxia Wang. "An Indexable Time Series Dimensionality Reduction Method for Maximum Deviation Reduction and Similarity Search". In: *International Conference on Extending Database Technology*. 2022. URL: https://api.semanticscholar.org/CorpusID:247848593.

[6] Eamonn Keogh et al. "An online algorithm for segmenting time series". In: *Proceedings 2001 IEEE International Conference on Data Mining*. 2001, pp. 289–296. DOI: 10.1109/ICDM.2001.989531.

[7] Vebjorn Ljosa and Ambuj K. Singh. "APLA: Indexing Arbitrary Probability Distributions". In: *2007 IEEE 23rd International Conference on Data Engineering* (2007), pp. 946–955. URL: https://api.semanticscholar.org/CorpusID:14903202.

[8] Yasushi Sakurai, Yasuko Matsubara, and Christos Faloutsos. "Mining and Forecasting of Big Time-series Data". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 919–922. ISBN: 9781450327589. DOI: 10.1145/2723372.2731081. URL: https://doi.org/10.1145/2723372.2731081.

[9] *Google Scholar*. https://scholar.google.co.uk/. (Visited on 03/18/2025).

[10] *Semantic Scholar*. https://www.semanticscholar.org/. (Visited on 03/18/2025).

[11] Eamonn Keogh and Shruti Kasetty. "On the Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration. Data Mining and Knowledge Discovery". In: *Data Mining and Knowledge Discovery* 7 (Oct. 2003), pp. 349–371. DOI: 10.1023/A:1024988512476.

[12] Hoang Anh Dau et al. *The UCR Time Series Classification Archive*. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/. Oct. 2018.

[13] Kent Beck et al. *Manifesto for Agile Software Development*. https://agilemanifesto.org/. Accessed: 2025-04-18.

[14] Eamonn Keogh et al. "Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases". In: *Knowledge and Information Systems* 3 (Jan. 2002). DOI: 10.1007/PL00011669.

[15] Qiuxia Chen et al. "Indexable PLA for Efficient Similarity Search". In: *Very Large Data Bases Conference*. 2007. URL: https://api.semanticscholar.org/CorpusID:18230.

[16] Hazem Elmeleegy et al. "Online piece-wise linear approximation of numerical streams with precision guarantees". In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), pp. 145–156. ISSN: 2150-8097. DOI: 10.14778/1687627.1687645. URL: https://doi.org/10.14778/1687627.1687645.

[17] Yahyia Benyahmed et al. "Adaptive sliding window algorithm for weather data segmentation". In: *Journal of theoretical and applied information technology* 80 (2015), pp. 322–333. URL: https://api.semanticscholar.org/CorpusID:61178005.

[18] Xiaoyan Liu, Zhenjiang Lin, and Huaiqing Wang. "Novel Online Methods for Time Series Segmentation". In: *IEEE Transactions on Knowledge and Data Engineering* 20.12 (2008), pp. 1616–1626. DOI: 10.1109/TKDE.2008.29.

[19] Urs Ramer. "An iterative procedure for the polygonal approximation of plane curves". In: *Computer Graphics and Image Processing* 1.3 (1972), pp. 244–256. ISSN: 0146-664X. DOI: https://doi.org/10.1016/S0146-664X(72)80017-0. URL: https://www.sciencedirect.com/science/article/pii/S0146664X72800170.

[20] Deng Min, Fan Zi-de, and Liu Hui-min. "Performance Evaluation of Line Simplification Algorithms Based on Hierarchical Information Content". In: 2013. URL: https://api.semanticscholar.org/CorpusID:64247458.

[21] Iosif Lazaridis and Sharad Mehrotra. "Capturing sensor-generated time series with quality guarantees". In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. 2003, pp. 429–440. DOI: 10.1109/ICDE.2003.1260811.

[22] *Github of the investigation*. https://github.com/jamesThompson2805/third-year-project. (Visited on 04/18/2025).

[23] Antonin Guttman. "R-trees: a dynamic index structure for spatial searching". In: *ACM SIGMOD Conference*. 1984. URL: https://api.semanticscholar.org/CorpusID:876601.

[24] Hoang Anh Dau et al. "The UCR time series archive". In: *IEEE/CAA Journal of Automatica Sinica* 6 (2018), pp. 1293–1305. URL: https://api.semanticscholar.org/CorpusID:53019704.

[25] Victor Ionescu, Rodica Potolea, and Mihaela Dinsoreanu. "Data driven structural similarity: A Distance measure for adaptive linear approximations of time series". In: *2015 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K)*. Vol. 01. 2015, pp. 67–74.

[26] Yuhan Cai and Raymond Ng. "Indexing spatio-temporal trajectories with Chebyshev polynomials". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: Association for Computing Machinery, 2004, pp. 599–610. ISBN: 1581138598. DOI: 10.1145/1007568.1007636. URL: https://doi.org/10.1145/1007568.1007636.

[27] Yuhan Cai and Raymond T. Ng. "Indexing spatio-temporal trajectories with Chebyshev polynomials". In: *ACM SIGMOD Conference*. 2004. URL: https://api.semanticscholar.org/CorpusID:12711593.