University of Alberta

Time Series Discords

by

David Alexis Mueller

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

©David Alexis Mueller
Fall 2013
Edmonton, Alberta

Abstract

Time series discords, as introduced in by Keogh et al. [5] is described as the subsequence in the time series which is maximally different from the rest of the subsequences. Discovery of time series discords has been applied to several diverse domains including space shuttle telemetry, industry, and medicine [5] to detect anomalies in the data which can identify equipment failure, unusual patterns of activity and health problems.

In this thesis we will examine the problem of finding time series discords, with detailed analysis of the problem and analysis of the effectiveness of prior work. Three different areas of discord discovery will be examined: Top Discord, Variable Length Discords, and Top-K Discords. In each of these areas, we strive to reduce the number or ease the selection of input parameters required by the end user. Emphasis is also placed on improved runtime and scalability of discord discovery methods.

Acknowledgements

I would like to than my wife and children for their support and understanding during my studies and research work, as well as my parents for their encouragement. It has been a long journey that I could not have completed alone.

Furthermore, I would like to thank my supervisors, Dr. Jörg Sander and Dr. Ian Parsons. Your knowledge and advice have been critical the completion of this work and your mentorship has helped me grow and develop both academically and personally. I am truly appreciative of your endless support.

Table of Contents

1 Introduction 1
  1.1 Problem definition 1
  1.2 Difficulties in Finding Time Series discords 1
  1.3 Thesis Outline 2
  1.4 Thesis Contribution 3

2 Concepts and Related Work 4
  2.1 Introduction 4
  2.2 Anomaly Detection and Time Series Discords 4
  2.3 Definitions 6
  2.4 Distance Measure Used 7
  2.5 Symbolic Aggregate approXimation 7
  2.6 Heuristic Discord Discovery 9
  2.7 Anomaly Detection in Time Series 14
  2.8 Time Series Motifs 15
  2.9 Datasets 16
  2.10 Implementation 17

3 Top Discord 19
  3.1 Introduction 19
  3.2 Preprocessing 19
  3.3 Range of Expected Performance 20
  3.4 Optimizations 26
  3.5 Performance Evaluation 47
  3.6 Conclusion 57

4 Variable Length Discords 58
  4.1 Introduction 58
  4.2 Dissimilarity Measure 58
  4.3 Discovery of Variable Length Discords 60
  4.4 Evaluation of Variable Length Discords 62
  4.5 Conclusion 64

5 Top-*K* Discords 65
  5.1 Introduction 65
  5.2 Methods 65
  5.3 Performance evaluation 72
  5.4 Conclusion 75

6 Conclusion 76
  6.1 Summary of Work 76
  6.2 Future Work 77

7 References 78

Appendix A: Additional Experimental Results from Chapter 3 80

List of Figures

Figure 1: Illustration of the top 2 time series discords in sensor data 1

Figure 2: Illustration of steps to discretize data using SAX 9

Figure 3: Pseudo code for brute force discord discovery 10

Figure 4: Illustration of data structures used in Hot SAX 12

Figure 5: Distance distributions for an ECG dataset 23

Figure 6: Distance distributions for a Random Walk dataset 23

Figure 7: Average number of distance calculations to find the discord using a random ordering heuristic 25

Figure 8: Number of inner loops required to early abandon 25

Figure 9: Runtime comparison for computing Euclidean vs. squared Euclidean distance 27

Figure 10: Discord search with early abandonment 28

Figure 11: Discord search with early abandoning and distance lookup 30

Figure 12: Discord search algorithm tracking best-so-far distance for each subsequence 31

Figure 13: Comparison between discord discovery methods implementing abandoning optimization and several strategies for distance lookups 32

Figure 14: Comparison of the average difference between nearest neighbour distances as a function of proximity to each other for a subsequence length of 128 34

Figure 15: Comparison of the average difference between nearest neighbour distances as a function of proximity to each other for a subsequence length of 512 34

Figure 16: Discord search with Random ordering and early abandoning 36

Figure 17: Percentage of duplicate distance calls for the Random Search heuristic 37

Figure 18: Number of distance calls required when resorting after 20, 40 & 60 outer loops. 38

Figure 19: Performance of SAX based methods for different word lengths and alphabet sizes in terms of average runtime performance 40

Figure 20: Effect of looking up indexes in the second stage of the inner loop 42

Figure 21: Pseudo code for my implementation of Hot SAX algorithm 43

Figure 22: Self tuning method for building sax structures of desired size 46

Figure 23: Runtime comparison of pre-processing steps by subsequence length 49

Figure 24: Illustration of the time spent in the double loop compared to the total time searching and interacting with the SAX data structures 47

Figure 25: Pre-processing, SAX Time and Search Time for small, medium and large number of SAX words 50
Figure 26: Runtime performance of Hot Sax with Word Hierarchy showing the contribution of preprocessing time, time to interact with the SAX data structures, and the time spent in the double loop for different ranges of distinct number of SAX words 50
Figure 27: Comparison of discord discovery methods in terms of average distance calls 53
Figure 28: Runtime performance comparison between different discord search methods 55
Figure 29: Comparison of methods showing maximum, average and minimum number of calls to the distance function 56
Figure 30: Comparison of methods showing maximum, average and minimum runtime for the discord discovery methods 57
Figure 31: Illustration of the dissimilarity measure 59
Figure 32: Pseudo code for computing the dissimilarity between two subsequences 60
Figure 33: Variable length discord algorithm 62
Figure 34: Illustration of the top 2 variable length discords in the shuttle dataset 63
Figure 35: Comparison of Variable Length Discord algorithm to a fixed length algorithm 64
Figure 36: Pseudo code for finding top-K discords by re-running the algorithm 66
Figure 37: Time spent to find each of the top 4 discord with the cumulative time and the distance to the discord 67
Figure 38: Illustration of the challenge in maintaining listing of K discord candidates 68
Figure 39: Comparison of Top-K discord methods in terms of distance calls 73
Figure 40: Comparison of Top-K discord methods in terms of runtime 73
Figure 41: Nearest neighbor distances of top discords returned by variable-k methods 74
Figure 42: Complete set of results for top discord methods by distance calls 80
Figure 43: Complete set of top discord methods by runtime results 81
Figure 44: Range of performance in terms of calls to the distance function for top discord methods 82
Figure 45: Range of performance in terms of CPU time for top discord methods 83

1 Introduction

1.1 Problem definition

The notion of time series discords was introduced by Keogh et al. [5] as the subsequence in a time series which is maximally different than the rest of the subsequences, as determined by a dissimilarity measure, such as Euclidean distance. Discovery of time series discords has been applied to several diverse domains including space shuttle telemetry, industry, and medicine [5] to detect anomalies in the data which can identify equipment failure, unusual patterns of activity and health problems.

An example discord is displayed in Figure 1, where the top two discords are highlighted and are the subsequences that differ most from the rest of the time series. This example of time series discords is from the shuttle dataset, where the discord corresponds to an irregular activation cycle of the valve. Discovery of this type of anomaly is important in fault detection.

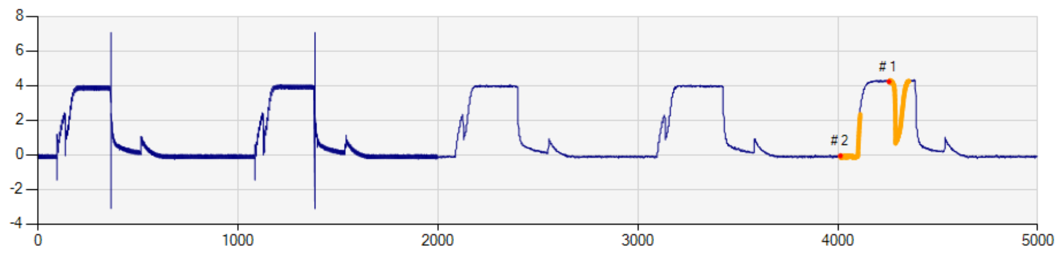


Figure 1: Illustration of the top 2 time series discords in sensor data from Marotta Valve activation cycle from the space shuttle.

1.2 Difficulties in Finding Time Series discords

The process to find time series discords begins by extracting the set of all possible subsequences of length *n* from the time series T. This can be achieved by using a simple sliding window technique, where the first subsequence, S_1, will contain the ordered set of points $t_1, t_2, \dots, t_n$ from the time series T. Subsequence S_2 will contain points $t_2, t_3, \dots, t_{n+1}$. This simple process will continue until the set is complete with a total of $|T| - n + 1$ subsequences.

By definition, discord discovery implies that a nearest neighbour search must be conducted for each subsequence, and the nearest neighbour search must compare each subsequence to all other subsequences for a runtime complexity of $O(|S|^2)$ where $|S|$ is the total number of distinct subsequences in T. A brute force approach is easily implemented using a double nested loop, where each subsequence is compared to every other subsequence in order to find its nearest neighbour, and then the subsequence with the largest nearest neighbour distance is returned. This brute force approach to discovering a time series discords quickly becomes impractical as the length of the time series increases, since the time to solve the problem grows quadratically.

Since the first paper introduced the method *Hot SAX* [5], others have worked on methods to improve the efficient discovery of time series discord [11][1][14]. While each of these authors presents improved algorithms to discover time series discords; three major questions arise from their work. First, each of these methods relies on tuning parameters, which will be shown to have a significant effect on runtime performance. Can we reduce or eliminate the need for tuning parameters? Second, these methods have, primarily, been evaluated using a naive method of comparison based on the number of calls to the distance function, as the dominating computational function, while ignoring the impact of the preprocessing steps. Is this an appropriate method for evaluating performance? Third, the work presented by Bu et al. [1] discusses finding the top-k discords, while providing little in the way of evaluation. What is the performance of this approach and are there better approaches?

1.3 Thesis Outline

This work examines three different problems related to discord discovery in time series data. First, it examines the problem of top discord discovery, then examines variable length discords and finally, extends the top discord problem to the top K discords.

The work is organized in the following way. Chapter 1 has introduced the problem of time series discords. Chapter 2 will review related work and introduce some key concepts and definitions.

In Chapter 3 will examine the top discord problem, where we will evaluate prior methods and propose new methods that are competitive without the need for tuning parameters. Chapter 4 will propose a new method for discovery of variable length discords. The discovery of top-k discords will be examined in chapter 5, where prior work will be evaluated and new methods will be proposed. Finally, chapter 6 will summarise the work presented and discuss future work.

1.4 Thesis Contribution

In this thesis, we will examine the effectiveness of prior works in finding the top discord, taking all preprocessing steps into account, and evaluating the methods in terms of calls to the distance function and runtime performance. We will present a new strategy for tracking nearest neighbour distances that will improve both the runtime performance and scalability of the methods. The impact of tuning parameter selection will be evaluated and we will present new methods that eliminate tuning parameters while maintaining competitive performance.

We will present a new approach that we have developed to find variable length discords. This new method will determine both the exact position and the length of the discord; where the length is within a user specified range.

The work on the top-k discords will examine and evaluate the performance of prior methods and present several new methods for finding the top-k discords. We will also examine two methods for finding all discords within the time series, without the need for the user to specify k (how many discords to discover).

2 Concepts and Related Work

2.1 Introduction

In this chapter, section 2.2 will explain discord discovery in terms of anomaly detection, then some basic concepts and definitions will be presented in sections 2.3 through 2.5; including discussion of distance measures and Symbolic Aggregate approXimation (SAX). Then a review of related work will be presented in sections 2.6 to 2.8. The related work falls into three categories: discord discovery, time series anomalies and motif discovery. These classes of work are interesting to contrast as they relate to the opposite problems of finding rare versus finding frequent patterns in time series data. Both these problems share many preprocessing steps, and can both take advantage of the same discretization techniques, such as SAX, to improve runtime performance. Finally, the datasets used for testing will be introduced in section 2.9, and some important implementation details will be discussed in section 2.10.

2.2 Anomaly Detection and Time Series Discords

Anomaly detection is a diverse and important problem area which, in general, refers to finding patterns in data that differs from expected behaviour. The survey work by Chandola et al. [2] describes three classes of anomalies: point anomalies, where individual instances can be considered as anomalous; collective anomalies, where a collection of related data instances are anomalous; and contextual anomalies, when the anomalous data is context specific. Anomalies in time series data are often contextual anomalies since individual data points may not be anomalous, however, the context, or sequence, of data points may be considered anomalous.

Many works have been proposed to discover anomalies in data across many diverse domains [2], such as fraud detection, system monitoring, intrusion detection, health care, and security systems. The choice of method used to detect anomalies is dependant of the type of data (binary, continuous, categorical, ect.) and on the type of anomaly, as listed above. Chandola et al. [2] group these methods into several categories:

- *Classification based techniques:* Labelled data is used to train a model that is then used to classify test instances as normal or anomalous.
- *Clustering based techniques:* Similar data instances are grouped into clusters using unsupervised or semi-supervised techniques, where normal data clusters together and anomalous data does not fall into any cluster, or anomalous data is far from any cluster centre.
- *Nearest neighbour techniques:* This class of techniques assumes that normal data falls into dense neighbourhoods, while anomalies are sparse and fall far away from their nearest neighbours.
- *Statistical techniques:* Anomalies in the data fall into low probability regions of the stochastic model which describes the data.
- *Information theoretic:* Assumes that anomalies introduce irregularities in the data that affect information theoretic measures, such as entropy or Kolomogorov Complexity.
- *Spectral Techniques:* Spectral techniques try to embed data into a lower dimensional subspace which differentiates between normal and abnormal behaviour.

Each of these methods have advantages and disadvantages, and are applicable to different types of problems and data sets. This work focuses on discord discovery in real-valued time series data. Discords are contextual anomalies that make comparisons based on the 'shape' of data, and the approach taken in this work is a nearest neighbour approach.

As described by Chandola et al. [2], the key advantages of nearest neighbour based techniques is that they are purely data-driven, unsupervised approaches and straight-forward to apply to different data types; however, these methods can miss anomalies when they have a close enough nearest neighbour and can be computationally complex.

2.3 Definitions

Before we delve into the details of discord discovery, some key definitions will be introduced. These definitions will apply to all future sections of the work.

Definition 1: A *Time series* T of length m is an ordered sequence of real values, where $T = t_1,\ t_2,\ \ldots,\ t_m$. The length of a sequence T is, in general, denoted by $|T|$.

Definition 2: A *Subsequence*, S_i, of time series T is an ordered sequence of T starting at position i sampling n continuous data points such that $S_i = t_i, t_{i+1}, \ldots, t_{i+n-1}$ where n is called the length of S_i.

Definition 3: For a time series T and a subsequence length n, S will represent the set of all possible subsequences of T. $|S|$ denotes the number of elements in S, i.e., $|S| = |T| - n + 1$.

Definition 4: Two subsequences S_i and S_j are called *non-overlapping* if and only if $|i - j| \geq n$.

The exclusion of overlapping subsequences is important to the problem of time series analysis as these subsequences will be very similar by nature of proximity, rather than by the nature of the underlying process. Consider, for example, subsequence S_1 and subsequence S_2. In this case $n - 1$ points in the subsequences will be the same, offset by a position of 1. For any $n \gg 1$ the shape of these two subsequences will be next to identical and inclusion of the distance between these overlapping subsequences will give non-intuitive results.

Definition 5: For the purpose of finding time series discords, *Distance* is a measure of dissimilarity between two subsequences where a distance function returns a real value such that $distance(S_i, S_j) = distance(S_j, S_i)$ and $distance(S_i, S_j) \geq 0$.

Definition 6: The *Nearest Neighbour* to a subsequence S_i is defined as a non-overlapping subsequence S_j that has the minimum distance to S_i among all non-overlapping subsequences in S.

Definition 7: A *Time Series Discord*, as introduced by Keogh et al. [5], is the subsequence S_i with the largest distance to its non-overlapping nearest neighbour.

Definition 8: The *top k*th discord is the subsequence with the *k*th largest distance to its non-overlapping nearest neighbour and is non-overlapping with the top *K-p* discords where $p = 1, \dots, k-1$.

2.4 Distance Measure Used

Current methods in the literature focus on Euclidian distance and this distance measure is used exclusively in this thesis. All methods discussed in this thesis, however, allow for the distance function to be replaced with a measure other than Euclidean distance, such as Dynamic Time Warping (DTW) [13].

Changing the distance measure will affect the performance of the methods based on the complexity of the distance function and may change the optimal approach taken to solve the problem. For example, if the Euclidian distance function was replaced with a function that is significantly more computationally intensive, then an approach with higher pre-processing costs that further reduces distance computations may improve overall performance. Likewise, for a lower complexity distance function, additional distance computations may be tolerated in order to reduce the complexity in other areas of the algorithm. Finding the correct balance is dependent on the distance function chosen and will require some experimentation.

The Euclidian distance between two subsequences $S_i$and S_j is defined as:

$$\boldsymbol{dist(S_i, S_j) \equiv \sqrt{\sum_{m=1}^{n} \left((S_i)_m - (S_j)_m\right)^2}} \qquad \textbf{Equation 1}$$

2.5 Symbolic Aggregate approXimation

Symbolic Aggregate approXimation (SAX), as presented by Lin et al. [7], is a dimensionality reducing symbolic approximation for time series data. An interesting property of the SAX approximation is that it provides lower bounding of distance measures. This means that the distance between two subsequences cannot be less than the distance between the SAX representations of those subsequences.

This symbolic approximation allows for discrete-value algorithms to be applied to real-valued time series data, and it can be effectively applied to solving time series problems [7][8]. The authors propose a generic framework for solving time series problems that includes: approximate data using SAX, find approximate solution, and then refine the approximate solution on original data.

The steps to produce a SAX approximation are illustrated in Figure 2. A subsequence is first normalized to a mean of 0 and standard deviation of 1. Following the normalization, a piecewise aggregate approximation (PAA) [4] of the subsequence is computed for a fixed window size. Computing the PAA involves splitting the subsequence into segments equal to the window size and then representing each of the windows by the average of the values within that window. In Figure 2, this is represented by the horizontal lines of fixed length. Each component of the PAA is then assigned a symbol (from a fixed alphabet) so that each symbol has an equally probable chance of occurring in the PAA based on a normal distribution. In Figure 2, this is illustrated by the dotted horizontal lines and shows the position of the symbol break-points in relation to the normal distribution, displayed along the left side of the chart, for an alphabet size of 3. The resulting sequence of symbols obtained by computing the SAX approximation is referred to as a SAX word.

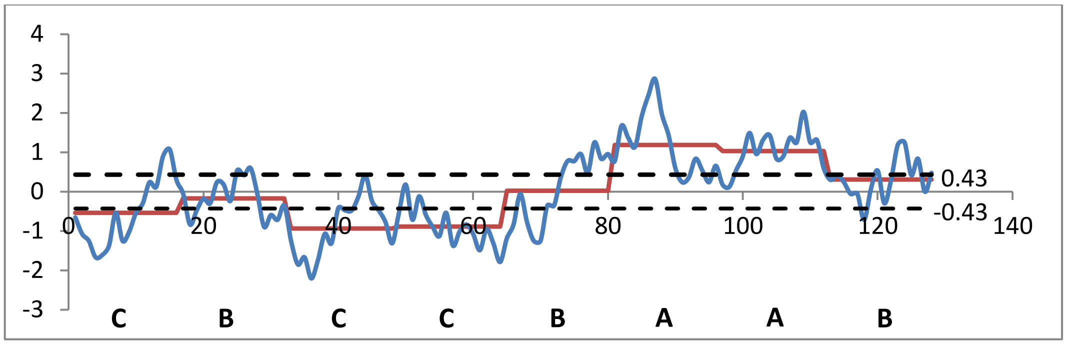


Figure 2: Illustration of steps to discretize data using SAX. First, the piecewise aggregate approximation is calculated as the average of each equally wide segment of the normalized sequence. Second, each segment is assigned a symbol based on the value of the approximation and breakpoints indicated by the dotted lines. The breakpoints used for symbol assignment are selected to give equally probable regions based on a normal distribution.

2.6 Heuristic Discord Discovery

The primary objective of this work is to examine time series discords, or anomalies in time series data. As introduced, this is important to many diverse domains such as health care and industry. For any process which produces real valued readings over time, it may be important to find patterns of activities which differ from the rest of the data. These anomalies could indicate process upset (in the case of industrial data), intrusion detection (in the case of web traffic monitoring) or patient illness (in the case of patient monitoring).

The simple brute force approach, as illustrated in Figure 3, is implemented with two nested loops which compares each subsequence to every other subsequence. This implementation is quadratic in runtime complexity, which quickly becomes impractical as the length of the time series grows.

```
// S = Set of all distinct subsequences of length n in time series T
// |S| = |T| -n + 1
1. Function [Distance, Location] = BruteForce(S)
2.      DiscordPosition = NaN
3.      DiscordDistance = 0
4.      FOR i = 1 to |S|                                    // outer loop
5.              NNDist = INFINITY
6.              FOR j = 1 to |S|                            // inner loop
7.                      IF |i-j| > n                        // non overlapping
8.                              d = Distance($S_i$, $S_j$)          // compute distance
9.                              IF d < NNDist
10.                                     NNDist = d
11.                             END
12.                     END
13.             END                                         // end inner loop
14.             IF NNDist > DiscordDistance
15.                     DiscordDistance = NNDist
16.                     DiscordPosition = i
17.             END
18.     END                                                 // end outer loop
19.     RETURN [DiscordDistance, DiscordPosition]
20. END
```

Figure 3: Pseudo code for brute force discord discovery. This simple algorithm is implemented using a double nested loop which compares each subsequence in the outer loop to every other subsequence in the inner loop, while tracking which subsequence has the largest distance to its nearest neighbour.

When comparing the subsequences, it is important to include only 'non-overlapping' subsequences, as discussed earlier. In the brute force method, this is accounted for by the simple if-statement on line 7, which ensures the difference between the subsequence indices is greater than the length of the subsequences.

Keogh et al. [5] introduces the problem of time series discords and proposes a solution to improve the performance of discord discovery over the brute force approach. The performance improvements are based on two simple observations.

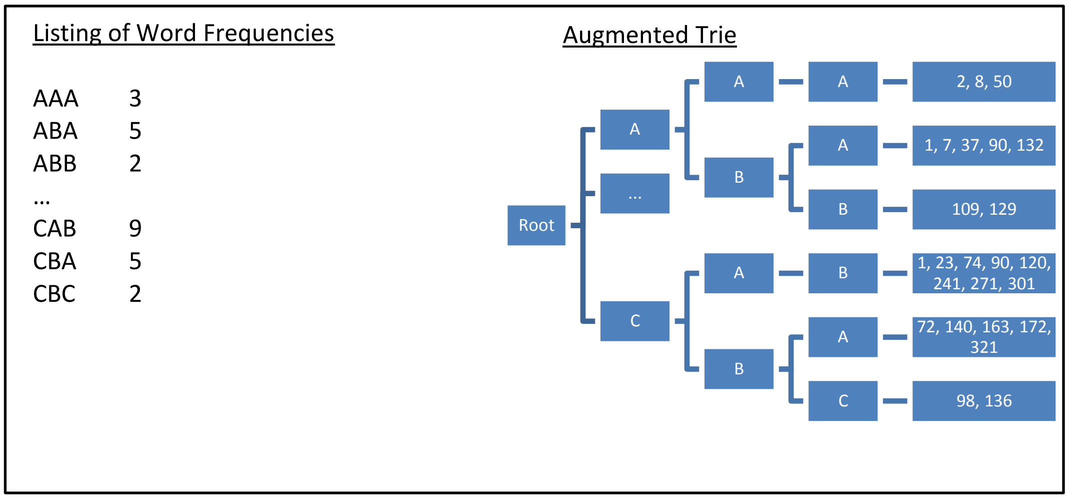
1. In the inner loop, we do not need to find the true nearest neighbour. We can break out of the inner loop as soon as we find a neighbour with a distance that is smaller than the current discord distance. Since the objective is to find the subsequence with the maximum distance to its nearest neighbour, any subsequence with a neighbour distance smaller than the current discord's nearest neighbour distance can be ruled out as the

discord. The current nearest neighbour distance is referred to as the Best-So-Far-Distance, and the optimization is referred to as Early Abandoning.

2. The performance of an early abandoning discord discovery algorithm is dependent on the order in which the subsequences are examined in the inner and outer loops. In the ideal situation, the true discord would be examined in the first outer loop and the nearest neighbour to the current subsequence in the outer loop would be considered first in the inner loop.

The ideal loop ordering, referred to in point 2 above, is impractical and would require knowledge of the true discord and nearest neighbour distances in order to achieve. Since an ideal ordering is not practical, an approximation is proposed based on the Symbolic Aggregate ApproXimation (SAX) [7] of the subsequences.

Once the SAX representation of each normalized subsequence is obtained, the approximations can be used to obtain a good loop ordering for the early abandoning discord discovery algorithm. This is achieved by the creation of two data structures. The first data structure is a listing of all distinct SAX words and the count of the words' frequency. The second data structure is an augmented trie (or prefix tree) of depth equal to the SAX word length, where path from root to leaf represents a SAX words and each leaf contains an array of subsequence indexes that map word represented by the path the trie. These data structures are illustrated in Figure 4.


Listing of Word Frequencies

AAA	3
ABA	5
ABB	2
…	
CAB	9
CBA	5
CBC	2

Figure 4: Illustration of data structures used in Hot SAX. The first data structure on the left is a listing of all distinct SAX words with a count of how many subsequences map to the word. The second data structure is an augmented trie with non-leaf nodes representing the sax letters of the sax word, and the leaf not containing a listing of all subsequences that map to the word.

For the Hot SAX method, the outer loop of the search proceeds in the order of SAX word frequency with infrequent words first, under the assumption that unusually shaped subsequence are represented by rare SAX words. For the inner loop, subsequences that map to the same SAX word are examined first, and then all remaining subsequences are examined randomly. The inner loop assumption is that similar subsequences will map to the same word and give small distances that can be abandoned on.

The generalization of this method is referred to as Heuristic Discord Discovery (HDD) where an ordering heuristic is used to guide the search in the double nested loop. For the Hot SAX method, SAX is the basis of the ordering heuristic. Keogh et al. [5] indicate a random ordering heuristic could be used, which considers subsequences in random, and note that it performs surprisingly well, but only provide a performance evaluation of their SAX based heuristic.

In the work by Pham et al. [11] it is argued that the use of fixed breakpoints based on a Gaussian distribution, when applied to the PAA, is not able to adapt to the unique nature of the data. They propose a k-means based approach to determine adaptive breakpoints, referred to as adaptive Symbolic Aggregate approximation (aSAX). They argue that this adaptive method is able to offer greater pruning power than the previous approach, since the adaptive breakpoints will prevent similar values (after the piecewise aggregate approximation) from being split and assigned different symbols. In their work, the loop ordering proceeds in much the same manner as Hot SAX; look at subsequences that map to rare words first in the outer loop, then look at words that map to the same word in the inner loop, followed by all other subsequences randomly.

Bu et al. [1] extend the previous Heuristic Discord Discovery (HDD) works by using a Haar Wavelet transform on the data prior to SAX. This proposed transformation is done in such a way as to dynamically determine the appropriate word size for SAX, effectively reducing one of the tuning parameters. The authors also suggest a method for determining the top-k discords, which entails finding the top discord and then re-running the algorithm to find the second discord and so on until the top-k discords are found. The work discusses the use of distances computed in early iterations to reduce the work required to find the 2nd to the kth discord. This method is claimed to be effective, but no performance results were provided. The authors recognize the difficulty in selecting the discord length and suggest a method to store properties of the Haar wavelets in order to efficiently try a range of subsequence lengths; however, no method is presented to determine how to select the final discord from the range of potential subsequence lengths.

In the work by Son et al. [14], the authors propose enhancements to HDD by using a symbol frequency table and some additional ordering heuristics that can be used to achieve better loop ordering. These loop ordering heuristics look at additional information, such as the likelihood of a particular symbol in a particular position, which is contained in the symbol frequency table. The paper presents numerous loop ordering heuristics, with differing performance depending on the dataset. As such, it introduces additional unintuitive inputs to the method. The authors compared their work to other works using both runtime and distance calls and show improvement. I have obtained this code and instructions to run from the author; it was compiled with optimizations and ran as best as possible. The code was tested on a range of input parameters. Performance was very poor for some selection of input parameters and comparable to our random search approach (which will be introduced in section 3.4.5) for the best set of parameters.

2.7 Anomaly Detection in Time Series

Several other methods have been proposed that address the problem of anomaly detection in time series datasets that are fundamentally different than that of Heuristic Discord Discovery methods presented so far.

Keogh et al. [6] propose a method for mining sequential data which is robust against length, sampling rates, missing values and dimensionality based on Kolmogorov complexity. The proposed method uses a Compression based Dissimilarity Measure (CDM) that is based on the compression of two strings and is defined as $CDM(x, y) = \frac{C(xy)}{C(x)+ C(Y)}$ where $C(xy)$ is the compressed size of x concatenated to y and $C(x), C(y)$ is the compressed size of x, y respectively. The CDM of two sequences will approach 0 as the similarity increases. To limit the effect of data representation, the authors present SAX as a solution to discretize the data beforehand. This dissimilarity measure is demonstrated in a divide-and-conquer algorithm and applied to clustering, anomaly detection and classification problems.

Previous works discussed require that the data reside in main memory. To address the problem of large (terabyte size) datasets, Yankov et al. [15] present a method that can find discords in datasets that exceed the space available in main memory. This method is able to find the discord in two linear scans of the dataset with a tiny buffer of main memory. The advantages of this method are only applicable to these large datasets. Our work focuses on datasets that fit into main memory.

Cheng el al. [3] developed a robust graph based algorithm that employees a kernel alignment method in order to find anomalies in multivariate time series. This method attempts to find time stamps where measurement values in one or more time series deviates significantly from the normal behaviour. Normal behaviour is described as the expected values of the time series based on historical changes as well as its relationship to other time series.

In the work by Preston et al. [12] the problem of finding sub-intervals, which are statistically significant from the underlying noise is addressed. This work is very different than those examined by HDD methods and in our work. In their work, the authors are interested in astronomical data where the noise characteristics are difficult to generalize into a model. An analytical and Monte Carlo approach are proposed to determine areas of interest based on the probability of their occurrence, where regions of low probability can be used to determine the starting point and size of statistically significant event in the data.

Luo et al. [9] recognize the challenge in parameter selection for discord discovery. In their work, they propose a parameter free discord search algorithm for quasi-periodic time series, which takes advantage of the repetition in the data. This method was shown to outperform Hot SAX in terms of calls to the distance function for periodic datasets. Testing was also conducted on non-periodic datasets and the authors only note it is 100s of times faster than brute force.

2.8 Time Series Motifs

The goal of motif discovery is to find sub-sequences that occur frequently in the time series, rather than rare sub-sequences as in discord discovery. Often this is described as the pair of subsequences that have the shortest distance to each other; however, range motifs are also described where sets of subsequences within a specified distance constitute a motif. Like discord discovery, many approaches are distance based and thus understanding this work can lend some insight into Discord discovery. For example, SAX discretization is a common pre-processing step in both tasks.

Yankov et al. [16] attempt to account for the inherent variability of scale in time series data as it applies to motif discovery. Their approach is to define a similarity measure which calculates the distance under several scaling factors. This approach is argued to be better than using distance measures such as Dynamic Time Warping (DTW) [13] when searching for short motifs (which are less prone to scaling issues). The short motifs discovered can then be ‘grown’ into longer patterns. Dynamic Time Warping is a distance measure which is able to ‘warp’ the subsequences as the distance is computed to get good alignment of the features in the data.

Mueen et al. [10] project the d-dimensional data into 1-dimensional space using an arbitrary reference point. Using the triangular inequality, they can lower bound the distance between any two points and prune large amounts of search space. The lower bounded distance can effectively and efficiently be calculated on the SAX approximation of the subsequences.

While these motif discovery methods do share some commonality with discord discovery, such as distance measures and preprocessing steps, the works are quite different. One optimization common and effective to motif discovery is the lower bounding of distances. This is unfortunately unavailable to discord discovery, as we are looking for maximum distances and there is no method of upper bounding the distance using approximations.

2.9 Datasets

All experimentation performed in this thesis was conducted on several diverse datasets that have been used in prior works on Discord Discovery. These datasets were obtained from the Bu et al. [1] along with the source code for their work.

The datasets include:

- ECG: Electrocardiogram dataset
- Shuttle: Sensor data from Marotta Valve activation cycle from the space shuttle.
- ERP
- Random Walk. Random walk dataset generated with transition values sampled from a normal distribution with mean of 0 and standard deviation of 1.
- Tickwise

The ECG and shuttle dataset are periodic, while the ERP and Random walk datasets do not exhibit periodic behaviour. It is important to differentiate and test on these two classes of data as it has a significant impact on our expectation of nearest neighbours. For example, with the periodic datasets, we expect each subsequence to have many near neighbours, since the pattern is approximately repeated throughout the data, leading to small distances. Any deviation from this pattern represents a discord. This is not the case for time series that are not periodic; however there are still subsections in the time series data that represent unusual behaviour as compared to the rest of the dataset.

All of the datasets used here, with the exception of shuttle dataset, are quite long and experiments were performed on time series of lengths 5,000, 10,000, 50,000 and 100,000 data points. The shuttle dataset is short and only allowed for experiments on time series of 5,000 data points. Prior works typically ran experiments on datasets shorter than 10,000 data points. One general constraint of this method is that the dataset fits in main memory. For datasets that do not fit in main memory, a significantly different approach would need to be used that limits access requests to secondary storage, like the one proposed in the work by Yankov et al. [15].

2.10 Implementation

Algorithms in this work were developed using C# in Microsoft Visual Studio 2010. When implementing prior works, care was taken to maintain the intent of each algorithm while ensuring that the resulting performance accurately reflects the effect of the optimizations presented. Deviations in implementation details are noted throughout the thesis. An example deviation is the use of a sorted dictionary, rather than a trie when implementing the Hot SAX code. Both of these data structures offer the same lookup performance of O(log(n)).

This approach to redevelopment of each method in a common environment is important so that runtime differences between algorithms are not due to differences in coding style, language or runtime environments.

As an aside to this, the actual code, written in C, was obtained for one of the prior algorithms. The code was compiled, as received, with optimizations and ran on several datasets. In addition to the code being difficult to read, understand, and test, the runtime performance was slow and reported inconsistent discord positions compared with other methods (including simple brute force). The slow performance could be due to the code being optimized for minimizing distance calculations, rather than minimizing runtime performance.

3 Top Discord

3.1 Introduction

The brute force approach to discord discovery is impractical on large datasets due to the quadratic runtime complexity, as already noted. Current works on improving discord discovery are based on the Heuristic Discord Discovery (HDD) and early abandoning as introduced by Keogh et al. [5]. This approach proposes the use of a search ordering heuristic to increase the rate of early abandoning in order to improve runtime performance.

The current works have not provided adequate performance evaluation in terms of CPU runtime and focus performance evaluation on the number of calls to the distance function. It is argued that computing the distance between subsequences accounts for the majority of the runtime [5], however this does not give a true evaluation of the methods. In order to provide a fair comparison between methods, we need to compare them in terms of actual runtime performance, to account for the overhead of the ordering heuristics. For example, one method may offer marginal improvement in calls to the distance function at significant preprocessing overhead in computing the ordering heuristics. Overall performance could then be much slower than without the proposed optimization.

In this chapter, we will examine the preprocessing steps in section 3.2. Then the baseline expectation of the complexity of discord discovery will be discussed in section 3.3. Previously published and our new optimizations will be described in section 3.4. Finally, performance evaluation of these optimizations will be presented in section 3.5.

3.2 Preprocessing

Given an input time series T of length $|T|$, a couple of preprocessing steps are required for all methods.

First is the extraction of the complete set of all distinct subsequences of length n, where subsequence S_i is given as

$$S_i = t_i, t_{i+1}, \ldots, t_{i+n-1}.$$

The second preprocessing step normalizes each S_i to a mean of 0 and standard deviation of 1. As will be seen this second preprocessing step takes considerable computation effort.

3.3 Range of Expected Performance

Mathematical examination of the problem of discord discovery will allow us to specify best case and worst case for performance in terms of calls to the distance function. Also, careful examination of the problem and several datasets will allow us to understand how difficult the problem is. For example, the degree in which the discord is dissimilar from 'normal' data influences performance. If a dataset is periodic, such as ECG data, there exists recurring patterns which are similar, thus finding a close neighbour should be easier, as any 'normal' repeat of the pattern should be a close neighbour. However, it also means that the range of true nearest neighbour distances may all be small compared to the range of all pairwise distances. This knowledge will allow us to understand how much room there is left to improve, as compared to the best case, for a given method (in terms of the number of distance calls required to find the discord). Any proposed method would need to balance the cost of the overhead with the number of distance calls to achieve better performance.

3.3.1.1 Worst Case Performance

In the brute force approach, all distances between pairs of non-overlapping subsequences need to be calculated. Due to the symmetry of the distance function, we only need to compute half of the total pair wise distances. This can be computed as:

$$\boldsymbol{Distance\ Calls} = \sum_{i=1}^{|S|-n} i = \frac{(|S|-n)(|S|-n+1)}{2} \qquad \textbf{Equation 2}$$

To illustrate the correctness of this formula, we can think of the simple brute force approach. In the first outer loop, we need to compute the distance to all $|S| - n$ non-overlapping subsequences. In the next outer loop, we again need to compute the distance to all non-overlapping subsequences, except for the reciprocal distance computed in the first outer loop. In the third outer loop, compute the distance to all non-overlapping subsequences, except for the two distances computed in the previous two outer loops. This pattern continues to obtain the summation given above.

As an example, if the time series contains 5,000 data points, and we are looking for a discord of length 128 we have:

$$|S| = |T| - n + 1 = 5000 - 128 + 1 = 4{,}873$$

$$|S| - n = 4{,}873 - 128 = 4{,}745$$

$$Distance\ Calls = \sum_{i=1}^{4{,}745} i = 11{,}259{,}885$$

It can be seen that for a small dataset, there are a large number of distance calculations that must be performed. For a larger dataset, such as 50,000 data points and subsequence length of 128, this number quickly grows to 1,237,307,385 distance calls.

3.3.1.2 Best Case Performance

In the ideal case, the discord discovery algorithm would look at the true discord in the first iteration of the outer loop. Then it would consider the true nearest neighbour to the subsequence in outer loop in the first inner loop. In this case, the first outer loop would compute the distance to all non-overlapping subsequences. Every subsequent outer loop would abandon on the first inner loop. The number of distance calls would then compute as:

$$\boldsymbol{Distance\ Calls = 2(|S| - n) - 1} \qquad \textbf{Equation 3}$$

This is only 9,489 distance calculations for the same example with 5,000 data points and 99,489 for the 50,000 data point example. This is an enormous difference and implies that there may be a lot of room to improve over the brute force approach.

3.3.1.3 Factors Affecting Performance

As discussed, early abandoning is a simple optimization that allows us to reduce the calls to the distance function through the observation that we do not need to know the true nearest neighbour of a subsequence in order to rule it out as the discord. It is sufficient to find a distance that is closer than the current best-so-far discord distance to rule out a subsequence as the discord. It can be easily shown that actual performance is dependent on the order in which the subsequences are examined. For example, if processing the subsequences in order, one would expect a large difference in performance if the true discord is located at the beginning of the time series as opposed to the end of the time series. This is because the true discord has the largest distance to its nearest neighbour and therefore will have greatest pruning power. This is the intuition behind the ordering heuristics as proposed by Keogh et al. [5], where we want to examine rare subsequences, and ideally the true discord, early in the search.

The dataset itself will also have a significant impact on performance and the difficulty of the problem. The first observation is that if the true discord is not too dissimilar from the other subsequences, early abandoning will not be as effective compared to the case when the discord is very different from the other subsequences. This is illustrated in Figure 5 and Figure 6. In these two figures, we see a histogram of all pair-wise distances compared to a histogram of all nearest neighbour distances for an ECG and a Random walk datasets.

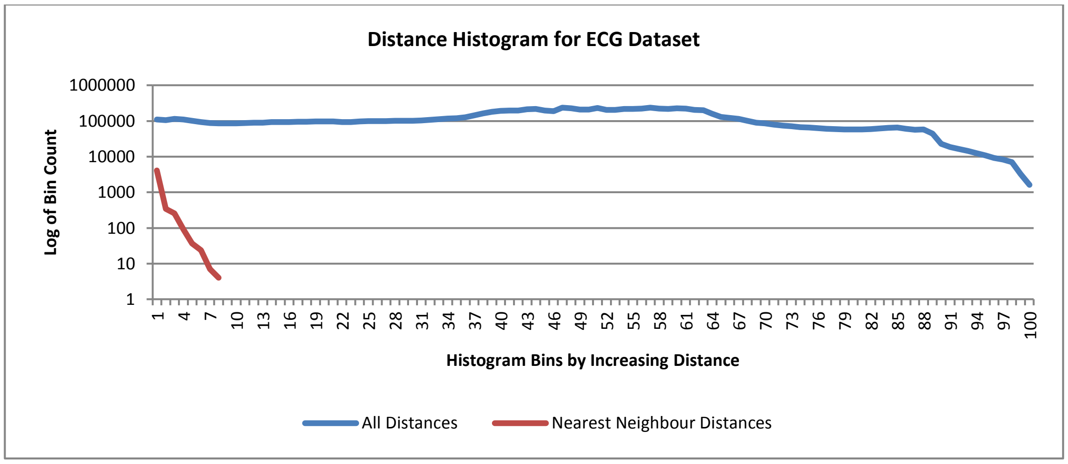


Figure 5: Distance distributions for an ECG dataset of 5,000 data points and subsequence length of 128. The chart shows the distribution of true nearest neighbour distances for each subsequence, in red, compared to the distribution of all pair wise distances in the dataset, in blue. Each bin number corresponds to a fixed-width-bin (of distances), with the smallest bin number starting at the smallest pairwise distance and the largest number bin ending at the maximum pairwise distance.

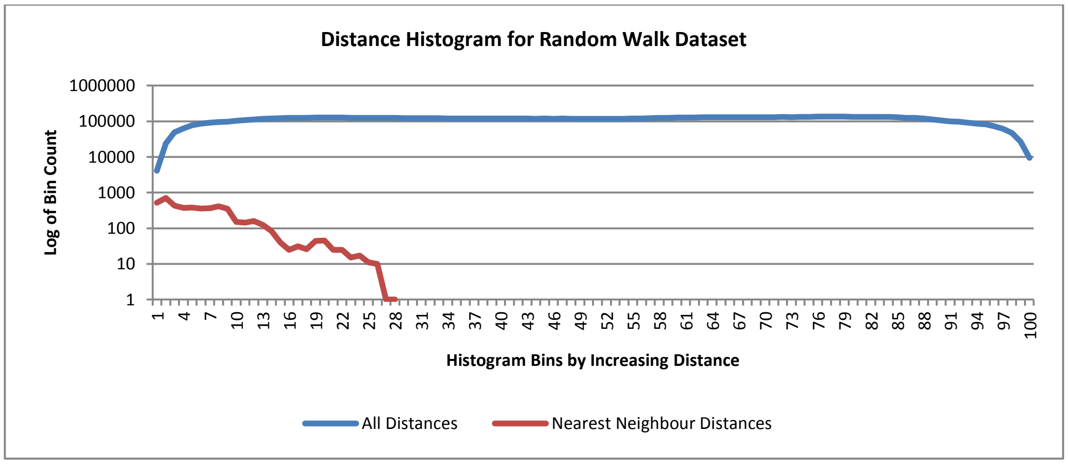


Figure 6: Distance distributions for a Random Walk dataset of 5,000 data points and subsequence length of 128. The chart shows the distribution of true nearest neighbour distances for each subsequence, in red, compared to the distribution of all pair wise distances in the dataset, in blue. Each bin number corresponds to a fixed-width-bin (of distances), with the smallest bin number starting at the smallest pairwise distance and the largest number bin ending at the maximum pairwise distance

Comparing the figures between the two datasets shows that the distribution of all pairwise distances follows the same general shape; consistent bin membership with a small decrease towards the tail ends. The nearest neighbour distance distributions also have the same general shape for the two datasets, with the majority of subsequences having small nearest neighbour distances and a decreasing trend in the number of subsequences as the nearest neighbour distances increase. This general shape holds for all of the datasets examined; however, the range in which the nearest neighbour distance distributions extend into the distribution of all pairwise distances varies. As for the two datasets shown above, the ECG dataset has a narrow range of nearest neighbour distances as compared to all distances; whereas the random walk dataset has a much wider range of nearest neighbour distances.

There are two consequences of these distributions. First, the distance between any two randomly selected subsequences is almost equally likely to fall into any of the bins of the histogram. Second, finding a good nearest neighbour distance to prune on in the random walk dataset should be easier than in the ECG dataset, since more nearest neighbour distances are larger than a larger percentage of pairwise distances.

Figure 7 demonstrates the effect different datasets have when using a linear ordering with early abandoning and random ordering heuristic with early abandoning and compares the results to the theoretical minimum and maximum number of distance calls for a time series of 5,000 data points and subsequence length of 128. The random search ordering heuristic will be explored in section 3.4.5.

In Figure 7 we can see that the early abandoning coupled with the random ordering heuristic provides a huge improvement over the theoretical maximum and that the dataset has a significant effect on performance. The random ordering also offers a significant improvement over the linear ordering; however, there is less difference between datasets for the linear ordering. Differences in the linear ordering are caused by factors such as position of discords in the dataset. This figure also gives us an idea of the range available for improvement by any other loop ordering heuristics of about 10 to 40 times improvement in distance calls, depending on the dataset.

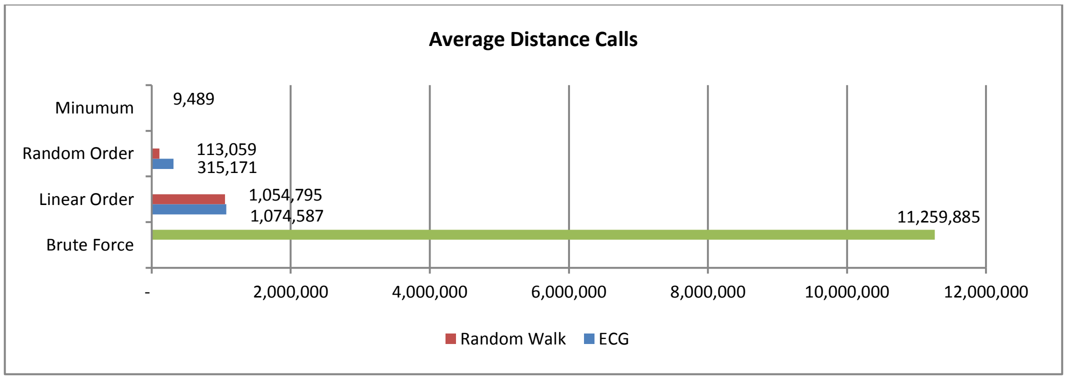


Figure 7: Average number of distance calculations to find the discord using a linear and random ordering heuristic for ECG and Random Walk datasets compared to minimum number of distance calls and the number of distance calls brute force for datasets of 5,000 data points and subsequence length of 128. Values for each of the datasets represent the average of ten runs of the experiment.

The number of actual distance computations performed is surprisingly small given the distributions that we have seen in Figure 5 and Figure 6. For the ECG dataset, finding the true discord using 427,505 distance calls means that on average, the outer loop abandons on the 87th inner loop (as computed by total distance calls divided by total number of outer loops). As for the Random Walk Dataset, the outer loop abandons on the 20th inner loop, on average, for this experiment.

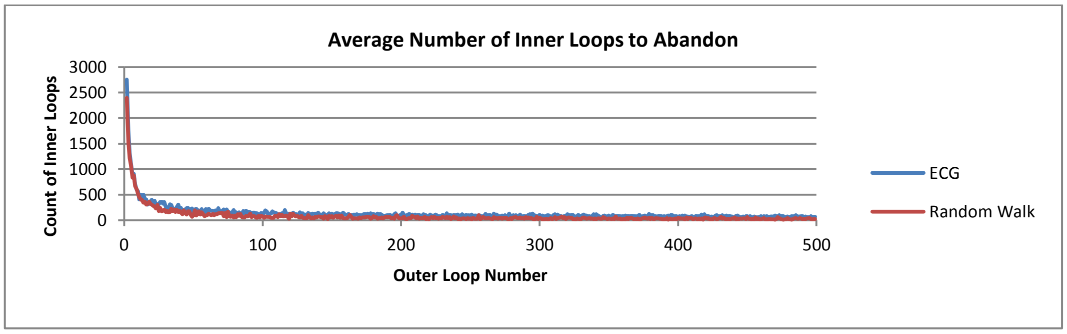


Figure 8: Number of inner loops required to abandon for outer loops numbered 2 to 500. Results are for the ECG and Random Walk datasets of 5,000 data points and subsequence length of 128. The results display the average of 100 runs for each dataset using the random search heuristic with early abandoning.

Figure 8 displays the average number of distance call required to abandon the inner loop as determined experimentally. In this figure, we can see that the early abandoning rate very quickly drops to below 500 inner loops. By the 200th outer loop (out of 4,872) , Random Walk dataset consistently abandons in less than 50 inner loops and the ECG dataset consistently abandons in less than 100 inner loops. With each successive loop, the trend is a further decreasing number of inner loops required before abandoning. Note that the experimental results displayed in Figure 8 are averaged over 100 repetitions due to the large variability between runs. Early in the search, it is common to see the occasional inner loop abandon in fewer than 5 inner loops.

3.4 Optimizations

In order to have a clear understanding of how different optimizations affect the runtime performance, we begin with the brute force approach and then build optimizations on top of this method. This allows easy comparisons and demonstrates how the changes affect the runtime performance of the method.

The following sub-sections will examine the following optimizations:

- 3.4.1: effect of removing the square root from distance function
- 3.4.2: early abandonment
- 3.4.3: distance lookup strategies
- 3.4.4: array storage of nearest neighbours
- 3.4.5: random search heuristic
- 3.4.6: effect of different SAX tuning parameters
- 3.4.7: Hot SAX
- 3.4.8: Hot SAX with work hierarchy
- 3.4.9: Sax Self tuning.

3.4.1 Remove Square Root from Distance Function

The strict definition of Euclidian distance, as given, is not needed to find the discord provided that the ranking of subsequences based on distance is preserved. A simple optimization is to omit the square root operation in the Euclidian distance function presented in Equation 1. The potential runtime impact of removing the square root was tested and the results are displayed in Figure 9. This optimization has a limited impact on runtime performance particularly as the subsequence length increases. For short subsequences of length 32, the difference is less than 10 % improvement in time to compute the pairwise distances. Once the subsequence length exceeds 128, the difference is less than 1% increase in runtime. All other experimentation conducted in this work used subsequence lengths greater than or equal to 128, so this would have limited impact on the results. For this work, the same distance function was used by all methods implemented (using squared distances).

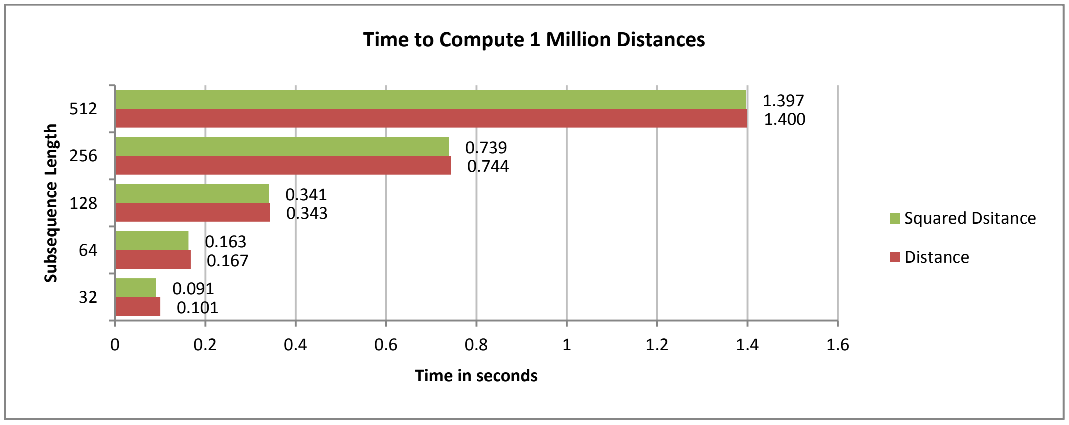


Figure 9: Runtime comparison for computing Euclidean distance versus squared Euclidean distance between two subsequences, i.e. with and without the final square root operation. This chart illustrates that the final square root operation has limited impact on the overall runtime of the distance calculation as the subsequence length increases.

3.4.2 Early abandonment

The key improvement over the brute force approach is the early abandonment optimization, which offers significant improvement in terms runtime and calls to the distance function. Like brute force, this method examines each subsequence in the outer loop, and compares it to all other subsequences in the inner loop. The outer and inner loops progress in order of position in the dataset.

The pseudo code for this algorithm is shown in Figure 10.

```
// S = Set of all distinct subsequences of length n in time series T
// |S| = |T| -n + 1
1. Function [Distance, Location] = EarlyAbandoning(S)
2.      DiscordPosition = NaN
3.      BestSoFarDistance = 0
4.      FOR i = 1 to |S|                                  // outer loop
5.              NNDist = INFINITY
6.              FOR j = 1 to |S|                          // inner loop
7.                      IF |i-j| > n                      // non overlapping
8.                              d = Distance(S_i, S_j)    // compute distance
9.                              IF d < NNDist
10.                                     NNDist = d
11.                             END
12.                             IF NNDist < BestSoFarDistance
13.                                     BREAK inner loop  // abandon loop
14.                             END
15.                     END
16.             END                                       // end inner loop
17.             IF NNDist > BestSoFarDistance
18.                     BestSoFarDistance = NNDist
19.                     DiscordPosition = i
20.             END
21.     END                                               // end outer loop
22.     RETURN [BestSoFarDistance, DiscordPosition]
23. END
```

Figure 10: Discord search with early abandonment. The lines which are bold indicate the difference from the Brute force approach shown in Figure 3.

Note that while the pseudo code shown in Figure 10 implements the simple early abandoning implementation, in lines 12 to 14, we are still not taking advantage of the symmetry of the distance function. By the time we reach the end of the inner loop, at line 16, we need to be sure that we have either abandon the inner loop or have considered every non-overlapping pair of subsequences. Storing pairwise distances that have been computed, will allow us to take advantage of symmetry of the distance function, as discussed in the next section.

3.4.3 Distance Lookups

Prior works have suggested using a $|S|\ x\ |S|$ matrix to store distances computed. This method can effectively be used to guarantee that we call the distance function only once for any pair of subsequences. Provided that we are proceeding in linear order and are not implementing early abandoning, we can limit calls to the distance function to when $i < j$, and then look up the distances when $j > i$. This will ensure that the subsequence pair i, j is considered before j, i when i iterates from 1 to $|S|$ in the outer loop and j iterates from 1 to $|S|$ in the inner loop.

In the case when we are not proceeding in linear order over the data or are implementing early abandoning, we can initialize a $|S|x|S|$ matrix to -1, and then check the appropriate value in our distance matrix prior to computing it. If the distance is greater than or equal to 0, we can use the stored distance, otherwise we must compute it. While this can save on distance calculations, there is a time and space cost associated with maintaining a matrix of pair-wise distances.

This optimization can be implemented using a simple square matrix, as shown in Figure 11; or with some additional logic, be implemented with a triangular matrix. To implement the triangular matrix approach, a space is allocated for an upper triangular matrix, and then all read/write accesses to the matrix are through the logic $distance = distances[\min(i, j), \max(i, j)]$ to ensure that only the upper half of the matrix is accessed. As will be shown in Figure 13, the additional logic of using a triangular matrix (due to the min/max functions) will have a small performance impact over using a square matrix, but will have half the space requirement. Applied to the early abandoning method, we see that this optimization reduces the number of distance calculations by 15% to 20%; however, it only improves the runtime performance as the subsequence length increases, and the computational cost of the distance function increases as a consequence.

```
// S = Set of all distinct subsequences of length n in time series T
// |S| = |T| -n + 1
Function [Distance, Location] = EarlyAbandoningLookupMatrix(S)
    DiscordPosition = NaN
    BestSoFarDistance = 0
    // create square matrix with cells initialized to -1
    Distances[,] = InitSquareMatrix(|S|, -1)
    FOR i = 1 to |S|                                    // outer loop
        NNDist = INFINITY
        FOR j = 1 to |S|                                // inner loop
            IF |i-j| > n                                // non overlapping
                IF(Distances(i,j) < 0)
                    d = Distance(Si, Sj)                // compute distance
                    Distances(j,i) = d
                ELSE
                    d = Distances(i,j)                  // retrieve distance
                END
                IF d < NNDist
                    NNDist = d
                END
                IF NNDist < BestSoFarDistance
                    BREAK inner loop                    // abandon loop
                END
            END
        END                                             // end inner loop
        IF NNDist > BestSoFarDistance
            BestSoFarDistance = NNDist
            DiscordPosition = i
        END
    END                                                 // end outer loop
    RETURN [BestSoFarDistance, DiscordPosition]
END
```

Figure 11: Discord search with early abandoning and distance lookup.

The primary concern with this strategy is not improvement or degradation in runtime performance, but with the additional space requirement. For a small dataset of 5,000 data points and a subsequence length of 128, a 4,873 by 4,873 matrix of distances is required. Assuming 8 byte doubles are used, this requires 181 megabytes of storage for the distances alone. This space requirement significantly limits the length of time series that this method can be applied to. Given our hardware setup, this approach limited our dataset to around 10,000 data points. Longer datasets would generate out-of-memory exceptions or would require additional logic to rely on slow secondary storage options.

Storage and lookup of pair wise distances in any method was abandoned in this work due to the limitations imposed by the $O(n^2)$ space requirements. Prior works [5][1][14], however, include this optimization and only evaluate using relatively small datasets. This may be due to the strategy used for performance evaluation in their work of only considering distance computations.

3.4.4 Array storage of Nearest Neighbour Distances

As an alternative to storing all pairwise distances, we have developed a different approach. Given that the objective is to find the nearest neighbour, we only need to concern ourselves with the current nearest neighbour distance for each subsequence and we can effectively couple it with early abandoning. This method is illustrated in Figure 12. First we must initialize an array of length |S| to *MaxValue* of a double. Then prior to each inner loop, we can see if a nearest neighbour for the outer loop subsequence has been found that is less than the current best-so-far distance. If so, we can rule out the current subsequence as being the discord without entering the inner loop. Inside the inner loop, we need to update the minimum distance found for both subsequences, i and j, in the current inner and outer loops.

```
// S = Set of all distinct subsequences of length n in time series T
// |S| = |T| -n + 1
Function [Distance, Location] = EarlyAbandoningLookupArray(S)
    DiscordPosition = NaN
    BestSoFarDistance = 0
    // create array matrix with entries initialized to max value
    NNDist[] = InitArray(|S|, MaxValue)
    FOR i = 1 to |S|                                // outer loop
        IF NNDist [i] < BestSoFarDistance
            NEXT i  // i cannot be discord skip inner loop
        FOR j = 1 to |S|                            // inner loop
            IF |i-j| > n                            // non overlapping
                d = Distance($S_i$, $S_j$)          // compute distance
                // Update nearest neighbour distances
                NNDist[i] = MIN(NNDist[i], d)
                NNDist[j] = MIN(NNDist[j], d)
                IF NNDist [i] < BestSoFarDistance
                    BREAK inner loop                // abandon loop
                END
            END
        END                                         // end inner loop
        IF NNDist [i] > BestSoFarDistance
            BestSoFarDistance = NNDist[i]
            DiscordPosition = i
        END
    END                                             // end outer loop
    RETURN [BestSoFarDistance, DiscordPosition]
END
```

Figure 12: Discord search algorithm tracking best-so-far distance for each subsequence. This algorithm implements early abandoning and allows for inner loops to be skipped entirely.

This strategy may lead to the situation where some distance pairs are computed twice. This is unavoidable without tracking exactly which distance pairs have been computed since we need to ensure that by the end of the inner loop we have abandoned or considered every subsequence. However, tracking the nearest neighbour distances for each subsequence can be effective for pruning and can be used to skip the inner loop entirely.

Figure 13 compares the three distance lookup strategies with a 'no-lookup' strategy in terms of calls to the distance function and runtime. We can see from the data that the array lookup strategy is typically best in terms of distance computations and is always best in terms of runtime performance. It is also interesting to note that while the matrix look-up strategies both reduce the number of distance computations over no-lookup strategy, there is minimal improvement on runtime performance and for short subsequence. It can in fact lead to an increase in runtime.

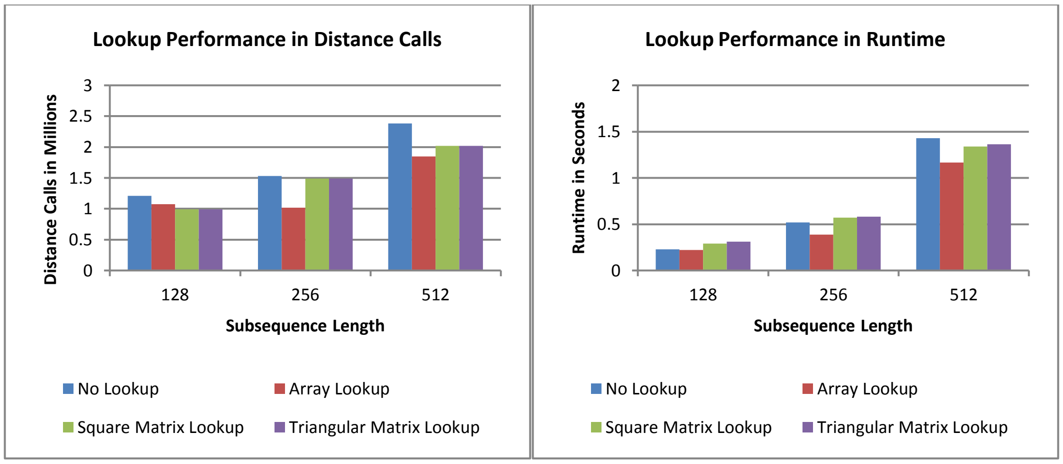


Figure 13: Comparison between discord discovery methods implementing early abandoning optimization and several strategies for distance lookups. The charts display the average of ten runs for the ECG dataset of 5,000 data points. All lookup strategies offer an improvement in the number of distance calculations over having no lookup strategy. Runtime performance and distances calculations required are typically the best for the array lookup strategy, due to the inner loop skipping and computational simplicity. While using a triangular matrix reduces storage requirement, it has a small increase in runtime performance over using a square matrix. These results are consistent among all 5 datasets examined.

It is also important to note that the loop skipping implemented in the array lookup strategy could be used with the matrix lookup strategies but would require either; repeatedly searching for max values along dimensions of the matrix adding to the runtime complexity, or combining the array and matrix strategies adding to the space complexity.

All further work presented in this thesis will implement the array lookup strategy due to the increased performance, simplicity and low storage requirement.

3.4.5 Random Search

Keogh et al. [5] noted that a random search heuristic could be used in the Heuristic Discord Discovery (HDD) framework and noted that the performance is surprisingly good, but no performance results or analysis of this approach have been provided. We will provide both here.

One of the problems with the early abandoning method is that distances between adjacent subsequences are not independent. This is due to overlap between the subsequences, which leads to similar distances, particularly where there is significant overlap. For example, *dist*(S_i,S_j) is expected to be very similar to *dist*(S_i,S_{j+1}) in all but the most extreme cases. Similarly, the nearest neighbour distance of subsequence S_i is expected to be very similar to the nearest neighbour distance of subsequence S_{i+1}. This is illustrated in Figure 14 and Figure 15 where we see the average difference between nearest neighbour distances offset by 1 to 150 indexes for two different subsequence lengths.

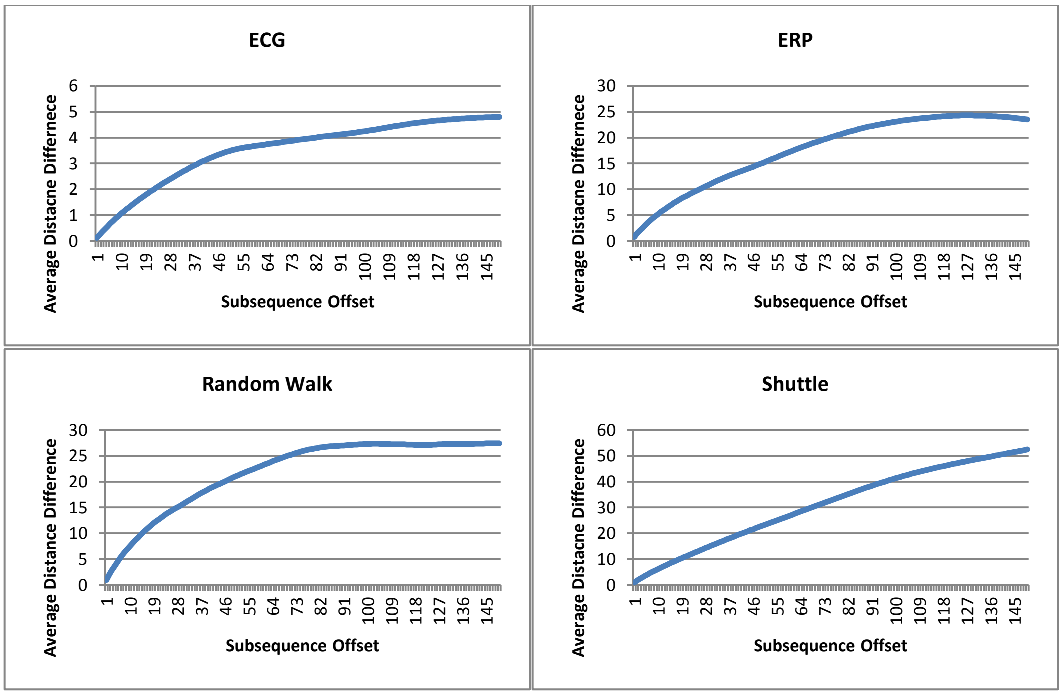


Figure 14: Comparison of the average difference between nearest neighbour distances as a function of proximity to each other for a subsequence length of 128. Here we see a clear trend of similar nearest neighbour distances for subsequences that are close to each other.

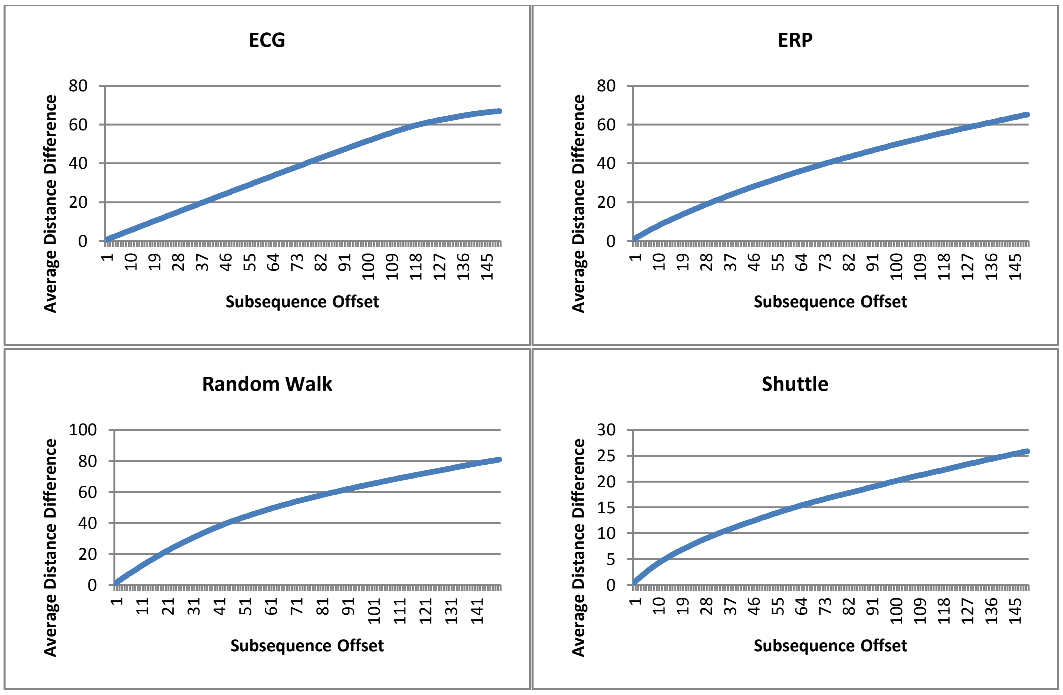


Figure 15: Comparison of the average difference of between nearest neighbour distances as a function of proximity to each other for a subsequence length of 512. Here we see a clear trend of similar nearest neighbour distances for subsequences that are close to each other.

For all of these four datasets and both subsequence lengths, we can see that close subsequences have similar nearest neighbour distances and the nearest neighbour distances increase as the distances between the positions of the subsequences increases.

The implication of this finding is that if the search cannot abandon on loop j in the inner loop, it is unlikely that it will be able to abandon on the next inner loop $j + 1$. This is due to the fact that the shape of the subsequences j and $j + 1$ will be very similar, due their significant overlap and thus $dist(i, j)$ will be very similar to $dist(i, j + 1)$.

Additionally, if the outer loop *i* does not have a large nearest neighbour distance, the next outer loop *i+1* will generally not have a large nearest neighbour distance, as we can conclude from Figure 14 and Figure 15. Therefore, if a subsequence i in the outer loop does not have a large nearest neighbour distance that would be good for pruning, then next subsequence $i + 1$ will not have a large nearest neighbour distance, and neither will be able to improve on our abandoning rate.

A random search ordering will address the issues highlighted above as compared to a linear progression with early abandonment. The random search method considers the subsequences in random order in both the outer and inner loops. This random ordering will increase the variability in each consecutive distance computed and helps increase our chances of finding a large best-so-far distance early in the outer loop and finding a near neighbour early in the inner loop, regardless of where they are located in the dataset. To save on overhead, the same random ordering can be effectively applied to all inner loops. The implementation of the random search method is presented in Figure 16.

```
// S = Set of all distinct subsequences of length n in time series T
// |S| = |T| -n + 1
Function [Distance, Location] = EarlyAbandoningRandomSearch(S)
    DiscordPosition = NaN
    BestSoFarDistance = 0
    // create randomly ordered array of all indexes
    OuterSearchOrder[] = GenerateRandomOrdering(|S|)
    InnerSearchOrder[] = GenerateRandomOrdering(|S|)
    // create array matrix with entries initialized to max value
    NNDist[] = InitArray(|S|, MaxValue)
    FOREACH i IN OuterSearchOrder                          // outer loop
        IF NNDist[i] < BestSoFarDistance
            NEXT i  // i cannot be discord skip inner loop
        FOREACH j IN InnserSearchOrder                     // inner loop
            IF |i-j| > n                                   // non overlapping
                d = Distance($S_i$, $S_j$)                 // compute distance
                // Store distance
                NNDist[i] = MIN(NNDist[i], d)
                NNDist[j] = MIN(NNDist[j], d)
                IF NNDist[i] < BestSoFarDistance
                    BREAK inner loop                       // abandon loop
                END
            END
        END                                                // end inner loop
        IF NNDist[i] > BestSoFarDistance
            BestSoFarDistance = NNDist[i]
            DiscordPosition = i
        END
    END                                                    // end outer loop
    RETURN [BestSoFarDistance, DiscordPosition]
END
```

Figure 16: Discord search with Random ordering and early abandoning. Highlighted lines indicate changes from the previous early abandoning method which searches in linear order.

As discussed in the previous section, using the array storage of nearest neighbour distances leads to some pairwise distances being computed twice. In practice, however, this does not occur too frequently, as seen in Figure 17, and is not a significant contributor to the overall runtime, as the cost of the extra distance computations is less than maintaining and looking up the distances.

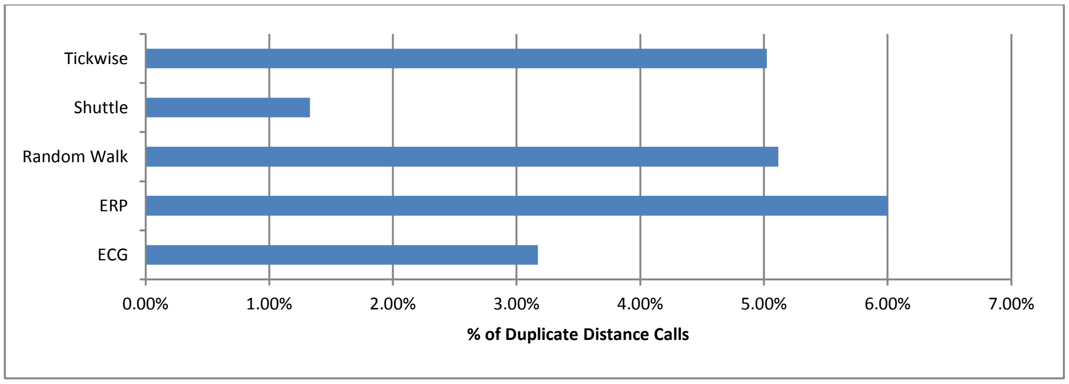


Figure 17: Percentage of duplicate distance calls for the Random Search heuristic, where duplicate distance calls are due to the symmetric nature of the distance function. The data represents the average value over ten runs with a time series length of 5,000 and subsequence length of 128.

3.4.5.1 Outer Loop Re-Ordering

We have developed and tested two variants inspired by the random search ordering heuristic. The first variant begins with the random search approach, maintaining the array of nearest neighbours for each subsequence. Then, after a specified number of outer loops, a new outer loop ordering is generated based on the current nearest neighbour distances that have been computed for each subsequence.

With each outer loop, the current nearest neighbour distances are updated for both the outer loop subsequence and the inner loop subsequence. With each successive outer loop the current nearest neighbour distances begin to approach the true nearest neighbour distances. So, the intuition behind this approach is that the current nearest neighbour distances calculated for each subsequence can be used to estimate true nearest neighbour distances for the subsequences. By resorting (in decreasing order of current nearest neighbour distances) the outer loop indexes that have not been considered, we will examine subsequences that appear to have large nearest neighbour distances earlier in the subsequent outer loops. The inner loop remains random in this method.

Experimentation was conducted on all datasets, file lengths of {5000, 10000, 50000, 100000} and subsequence lengths of {128, 256, 512} comparing performance in terms of distance calls and runtime performance with resorting after 20, 40 and 60 initial loops. Experimental results for 4 datasets of length 10,000 and subsequence length of 128 is shown in Figure 18. Here we see that none of the cases is best for all datasets. This is consistent across all of the experiments performed, where we see small performance differences with no case being significantly better. However, resorting after 40 loops appeared to give slightly better overall performance in many cases. So, for comparison to other methods, resorting after 40 initial loops will be used in the performance evaluation presented in section 3.5.

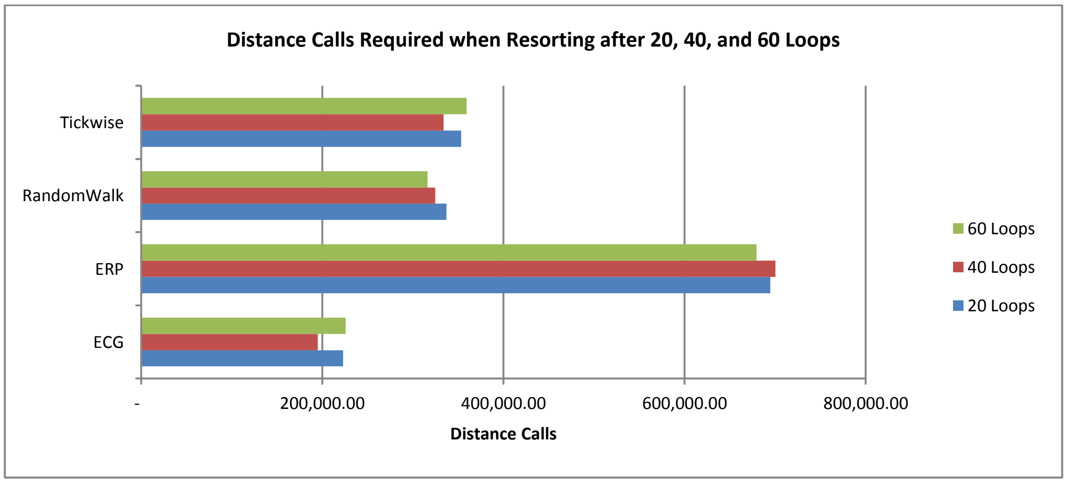


Figure 18: Number of distance calls required when resorting after 20, 40 and 60 outer loops. The figure displays the results for 4 datasets of 10,000 data points and subsequence length of 128.

3.4.5.2 Random Search Neighbour Pruning

The second variant we developed is based on the distance between adjacent subsequences. Prior to the double nested loop, the distance between all adjacent subsequences are calculated and stored. Then in the inner loop, if we abandon on subsequence S_i , that is

$$NearestNeighbour[S_i] < BestSoFar$$

then we can also abandon on S_{i+1}if

$$NearestNeighbour[S_i] + Dist[S_i] < BestSoFar$$

and abandon on S_{i+2} if

$$NearestNeighbour[S_i] + Dist[S_i] + Dist[S_{i+1}] < BestSoFar$$

and so on, as long as the summation is less than BestSoFar. In the above formulas, $Dist[S_i]$ corresponds to the pre-calculated distance between subsequences S_i and S_{i+1}.

Due to the triangular inequality (which states that the sum of any two sides of a triangle must be greater than the length of the remaining side) this approach will guarantee that the true nearest neighbour distance for the subsequence is less than the left hand side of the equation. In this situation, for subsequence S_i with nearest neighbour S_j we are saying that

$$dist(S_i, S_{j+1}) < dist(S_i, S_j) + dist(S_j, S_{j+1})$$

We can perform the same pruning for S_{i-1}, S_{i-2} and so on.

This approach will be shown to have a significant reduction in the number of full distance calculations that must be performed, as well as a significant reduction in runtime over other random search approach.

Evaluation of these two approaches will be presented in detail in Section *3.5*.

3.4.6 SAX Tuning Parameters

All Sax based methods rely on two parameters: SAX word length and alphabet size. These parameters are used in the discretization of each subsequence to its representative SAX approximation. The SAX word length specifies how many letters are used in the representation and the alphabet size specifies the number of letters to use in the discretization.

Keogh et al. [5] offers guidance on selecting these parameters and suggests that an alphabet size of 3 or 4 is best for virtually any task and they set this value to 3 in all their experiments. For word lengths, the recommendation is to use smaller word lengths for slowly changing datasets and longer word lengths for complex datasets. No further guidance on what is ‘smaller’ and ‘longer’ or ‘slowly changing’ and ‘complex’ was provided.

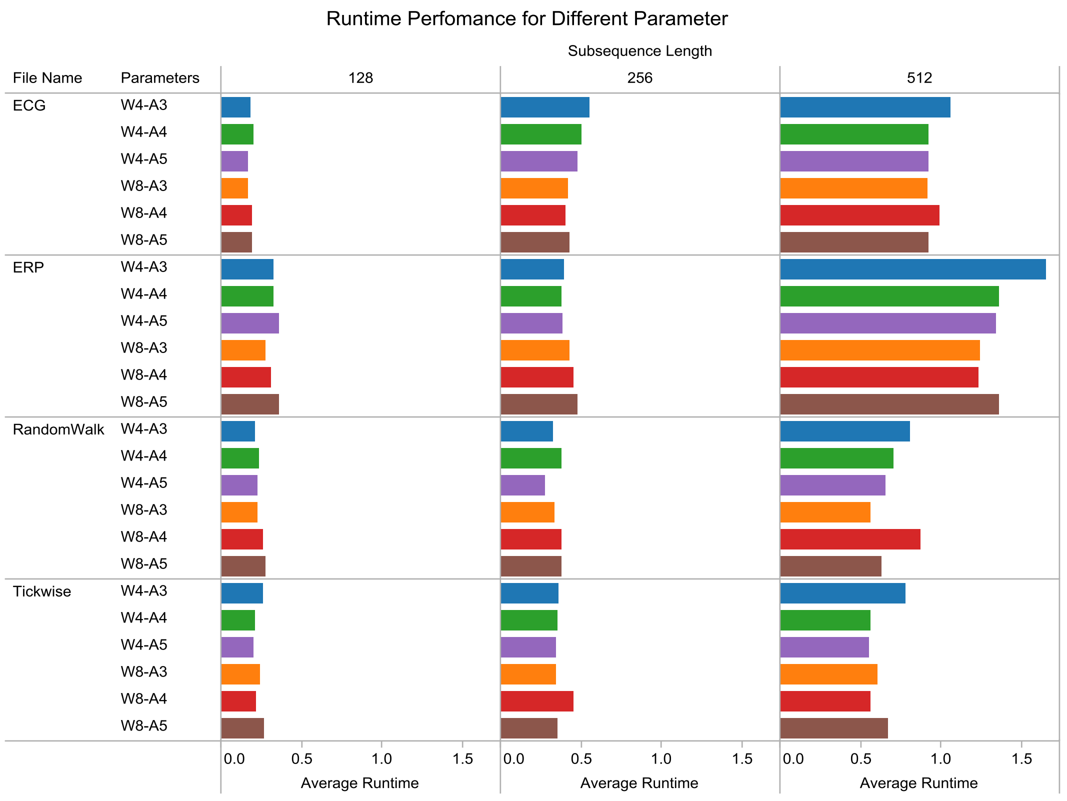


Figure 19: Performance of SAX based methods for different word lengths and alphabet sizes in terms of average runtime performance. The row W4-A3 corresponds to a word length of 4 and alphabet size of 3. Results are based on ten repeats of each experiment with subsequence lengths of 128, 256 and 512 on a dataset of 10,000 data points. Word lengths longer than 8 were excluded due to the extremely poor performance on the majority of datasets.

Extensive experimentation was performed in this work on a range of input parameters for both alphabet size and for word length on each of the datasets and each of the SAX variants that will be examined. Results are presented in Figure 19. This figure demonstrates the variability in the performance for different parameters on the same dataset and different subsequence lengths, as well in the variability in the performance for different parameters across different datasets. The figure also shows the clear increase in runtime for increasing subsequence length, which is caused by the increasing time to compute the distance between longer subsequences.

This resulted in two important findings. First, optimal performance occurred with a word length of 4 and alphabet of 3 to 6 or word length of 8 and alphabet size of 3 or 4. Second, the choice of input parameters has significant effect on runtime performance and demonstrates the danger of only reporting the performance of the top set of parameters.

3.4.7 Hot SAX

HOT Sax, as described earlier, presents an ordering heuristic to improve the performance of discord discovery by ordering the inner and outer loops to increase the rate of early abandonment of the inner loop. This work re-implemented the Hot SAX method based on the original paper by Keogh et al. [5] to allow accurate comparison in terms of both runtime performance and in terms of distance function calls. The implementation as presented in Figure 21 follows the principal of the original method, but differed in the following areas. First, data structures provided by the Microsoft .net libraries were used, rather than custom data structures. Objects with comparable runtime complexity to the data structures discussed in the paper were selected and used in our implementation. For example, a sorted dictionary was used instead of a trie for the distinct SAX words, both of which offer *log (n)* lookup performance. This dictionary stores all distinct SAX words as the key, and then stores a list to all subsequence indexes that map to the word as the value. In addition to the dictionary, an array of SAX words was created and sorted in order of the number of subsequences that map to the word.

The second change to the original method is the use of the array to store nearest neighbour distances, rather than a matrix to lookup distances computed. This was to allow experimentation to be performed on larger datasets, and has been shown to improve runtime performance as this structure allows for inner loop skipping, improving over the original SAX method.

Once the SAX data-structures are constructed, the double loop search proceeds by examining SAX words in the order of rarest to most frequent in the outer loop. And then the inner loop proceeds in two stages. In the first stage, all subsequences that map to the same word as being considered in the outer loop are examined first, and then in the second stage all remaining subsequences (i.e. do not map to the same word) are examined in random order. This implies that the second stage of the inner loop must search for each index in the list of indexes considered in the first stage of the inner loop. While this strategy will ensure that distances are only calculated once, my experimentation has shown that this search in the second stage of the inner loop has a large negative impact on the runtime performance, as shown in Figure 20, and it is faster to recalculate these distances. Herein is the third change to my implementation of the original Hot SAX method; my implementation examines *all* subsequences in random order in the second stage of the inner loop, rather than check if the subsequence was considered in the first stage of the inner loop.

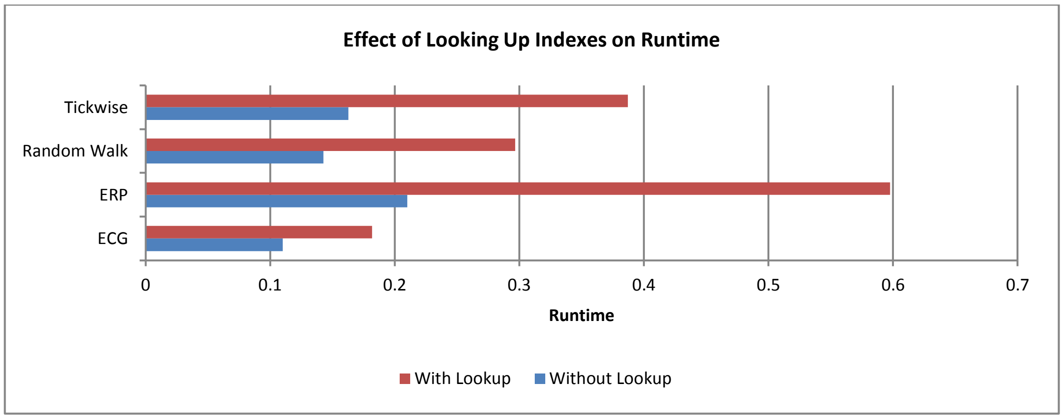


Figure 20: Effect of looking up indexes in the second stage of the inner loop. Data is for four dataset of length 10,000 data points, subsequence length of 128, word length of 4 and alphabet of 3. Results are consistent with other sets of tuning parameters.

The pseudo code for my implementation of HotSAX is presented in Figure 21.

```
// S = Set of all distinct subsequences of length n in time series T
// |S| = |T| -n + 1
Function [Distance, Location] = HotSAX(S, wordLen, alphabetSize)
    DiscordPosition = NaN, BestSoFarDistance = 0
    // sax structure is sorted dictionary with sax word key and
    //      array of indexes that map to word as value
    SaxStruct = BuildSaxStructure(S, wordLen, alphabetSize)
    // outer word order is array of sax words in
    //      increasing order of frequency
    OuterWordOrder = GenerateWordOrder(SaxStruct)
    InnerSearchOrder[] = GenerateRandomOrdering(|S|)
    // create array matrix with entries initialized to max value
    NNDist[] = InitArray(|S|, MaxValue)
    FOREACH word IN OuterWordOrder                   // outer loop
      FOREACH i IN SaxStruct[word]
            IF NNDist[i] < BestSoFarDistance
                  NEXT i  // i cannot be discord skip inner loop
            // check indexes that map to same word first
            FOREACH j IN SaxStruct[word]
                  IF |i-j| > n                      // non overlapping
                        d = Distance($S_i$, $S_j$)          // compute distance
                        // Store distance
                        …
                        IF NNDist[i] < BestSoFarDistance
                              BREAK inner loop      // abandon loop
                  END
            END
            // check all indexes in random order
            FOREACH j IN InnerSearchOrder[]
                  IF |i-j| > n                      // non overlapping
                        d = Distance($S_i$, $S_j$)          // compute distance
                        // Store distance
                        …
                        IF NNDist[i] < BestSoFarDistance
                              BREAK inner loop      // abandon loop
                  END
            END                                     // end inner loop
            IF NNDist[i] > BestSoFarDistance
                  BestSoFarDistance = NNDist[i]
                  DiscordPosition = i
            END
      END
    END                                             // end outer loop
    RETURN [BestSoFarDistance, DiscordPosition]
END
```

Figure 21: Pseudo code for my implementation of Hot SAX algorithm.

3.4.8 Hot SAX with Word Hierarchy

In the Hot SAX approach, subsequences that map to the same SAX word are considered first in the inner loop, then all remaining subsequences are searched in random order. When examining the rare SAX words early on in the search, we quickly resort to examining all subsequences in random order in the second stage of the inner loop, since only a few subsequences map to these rare words. As a better strategy for the inner loop, we would like to examine the subsequences that map to the same word as in the current outer loop first, and then consider the remaining subsequences in order of SAX word similarity. That is to consider the subsequences that map to the most similar SAX word in the next outer loop, assuming that similar subsequences map to similar SAX words.

Son et al. [14] implemented this approach using a lower bounding distance measure between SAX words based on the symbol breakpoints of a normal distribution. For example, the following table gives the distance between symbols for an alphabet of size 4:

	A	B	C	D
A	0	0	0.67	1.34
B	0	0	0	0.67
C	0.67	0	0	0
D	1.34	0.67	0	0

Then for SAX words AAA and AAC, the distance is calculated as $0 + 0 + 0.67 = 0.67$. While this method preserves the strict lower bounding distance between SAX words, it does not differentiate between similar SAX works such as AAA and AAB, which may be mapped to by nearly identical subsequences but are more likely to be mapped to by slightly different subsequences.

For our implementation, I defined a simple similarity measure between SAX words where the distance is the sum of the distance of each SAX letter in each position with symbol differences given by the following lookup table:

	A	B	C	D
A	0	1	2	3
B	1	0	1	2
C	2	1	0	1
D	3	2	1	0

Where $SymbolDist(A, A) = 0$ and $SymbolDist(A, B) = 1$. Then the distance between two SAX words can be defined as:

$$WordDistance(X,Y) = \sum_{i=1}^{w} SymbolDist(X_i, Y_i)$$

Using this dissimilarity measure, the distance between AAA and AAC would be 2 and the distance between AAA and AAB would be 1.

Compared to the prior work, this method will preserve the same general ordering, however provide slightly more granularity and definition between the similar sax words.

The algorithm then considers SAX words in increasing order of frequency in the outer loop, and then looks at subsequences that map to words in increasing order of distance from the current word.

3.4.9 SAX Self Tuning

One problem with the Hot SAX approach is the dependence on tuning parameters, as discussed. Elimination of the tuning parameters is advantageous as long as it gives consistent and comparable performance to other methods, as it simplifies application of discord discovery to end users.

The experimentation on input parameters performed in section *3.4.6* also demonstrated that the number of distinct SAX words is related to performance. There is a balance between separating unusual subsequences into rare SAX words and the increasing cost of searching the growing number of SAX words. If the data maps to too few SAX Words, the ‘rare’ Sax words will not be rare enough and may represent too many subsequences. On the other hand if there are too many SAX words, many of them will contain a few subsequences. This increases the time to build and search the SAX data structures and too many SAX words will be considered rare, which lessens the discriminative power of the heuristic.

We have developed a self-tuning approach that is based on a target for the minimum number of SAX words. This method will adjust the parameters for the alphabet size and word length until the first combination of parameters produces a discretization that exceeds the target number of words. Since the number of subsequences mapping to each unique SAX word increases with the number of subsequences, a simple heuristic was used to determine a target number of SAX words that is dependent on the number of subsequences. This was set to the square root of the number of subsequences. The rationale behind this selection is based on experimentation that will be shown later which compares the search time to the time to build and access the SAX data structures.

During the preprocessing, word lengths, *w*, are tested in order of {4, 8, 16, …}. Once the Piecewise Aggregate Approximation is computed for *w*, then each alphabet size, *a*, is tested in order of {3, 4, 5} creating the SAX data structures for *w* and *a*. Once the first combination of *w* and *a* that exceeds the target word number this preprocessing step ends and the double loop proceeds with the SAX data structures created. The pseudo code for this step is seen in Figure 22.

```
Function [SaxStruct] = BuildSaxStructure(S)
    w = 4
    targetWords = Round(SquareRoot(|S|))
    DO
        double[][] paa = ComputePAA(s, w)
        FOREACH i in {3,4,5}          // for each alphabet size
            SaxStruct = BuildSaxStruct(paa, i)
            IF(SaxStruct.Length > targetWords)
                RETURN SaxStruct
        END // FOREACH
        w = w*2
    WHILE(TRUE) // DO
END
```

Figure 22: Self tuning method for building sax structures of desired size.

The steps to build the SAX data structures proceed in order to minimize number of times the PAA data structure is built, as this is computationally intensive. Each of the possible alphabet sizes are tested in increasing order on one PAA representation, since the alphabet assignment is significantly faster than the PAA computation. As soon as the target number of words is achieved, we proceed with the current discretization; else we double the word length and retest each of the alphabet sizes.

Once the SAX data structure is created, this method continues in the same manner as in section 3.4.8.

3.5 Performance Evaluation

Three areas of performance will be considered here. First, we will look at the performance of the pre-processing steps, then the performance of input parameters, and finally the performance of the discord discovery methods in terms of both distance calculations and total runtime performance (including all preprocessing).

3.5.1 Pre-Processing Performance

As mentioned before, experimentation was performed in order to determine the cost of the pre-processing steps. During experimentation, it was determined that the three pre-processing steps that have the largest contribution to the overall runtime are: reading data from disk, extracting the subsequences and normalizing the data. In Figure 23 we can see the contribution of each of these steps for a dataset of 50,000 data points and three subsequence lengths. As expected, reading the data from disk is affected only by data length, however extracting the subsequences and normalizing the data are affected by both data length and subsequence length. Other preprocessing steps, such as array of nearest neighbour initialization, generating random loop orderings have negligible effect on the overall runtime. These preprocessing steps are incurred by all methods, regardless of loop ordering heuristics. Understanding the fixed preprocessing costs is important when evaluating the effectiveness in any discord search approach as it allows us to evaluate the margins available for improving the search component of the method.

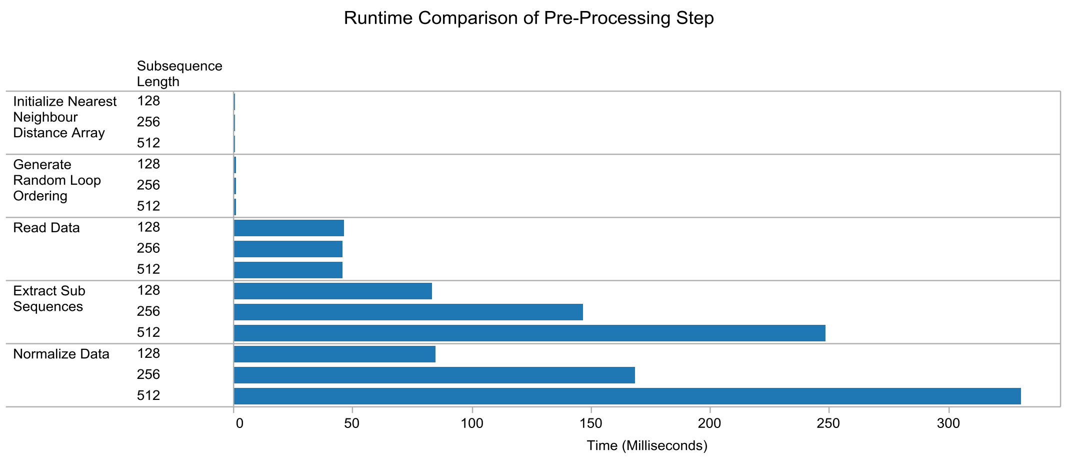


Figure 23: Runtime comparison of pre-processing steps by subsequence length. The data summarizes averaged values over multiple runs on each of the datasets of 50,000 data points. This is consistent between all data lengths.

In the next section, we will see the preprocessing costs compared to the overall cost of the discord search method.

3.5.2 Input Parameter Performance

Correct selection of input parameters is of critical importance for any method implementing the SAX approximations for generating loop orderings. As discussed, there are two input parameters used in the computation of the SAX approximations of the subsequences. These are the alphabet size and the word length, where the word length specifies how many regions are used in the computation of the piecewise aggregate approximation (PAA), and the alphabet size specifies how many regions are used in the discretization of each PAA segment.

The selection of good input parameters and its impact on runtime performance is critical to having good search performance. The objective is to have good separation in the data, where unusual subsequences are discretized into rare SAX words, without a large overhead in creation and interaction with unnecessarily large SAX data structures. This is demonstrated in Figure 24 where we see an illustration of the time to create and access the data structures (labeled as SAX Time) compared to the time spent in the double loop (labeled as search time) as a function of the number of distinct SAX words.

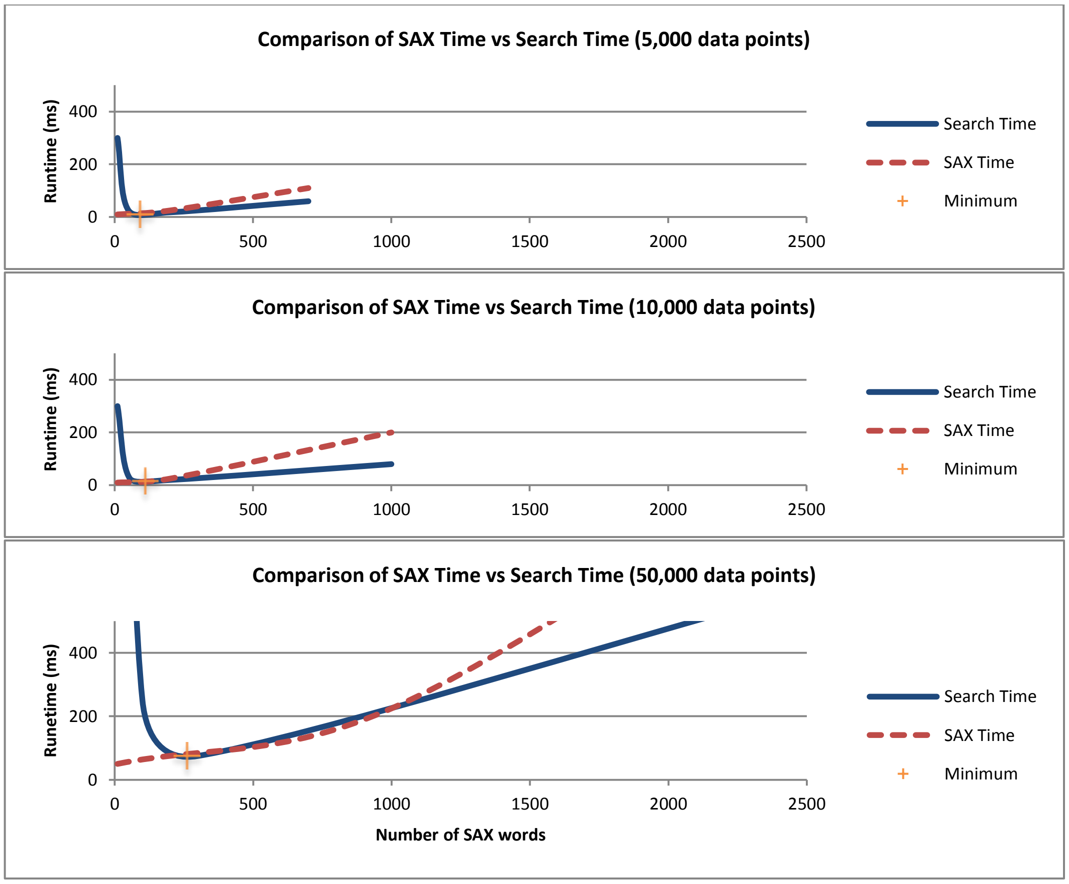


Figure 24: Illustration of the time spent in the double loop (labelled as search time) shown as a solid blue line, compared to the total time searching and interacting with the SAX data structures (labeled as SAX time) shown as a dashed red line as a function of the number of distinct SAX words. The results presented are for the Hot SAX with Word Hierarchy method averaged over ten runs for all datasets of 5,000 data points on the top, 10,000 data points in the middle and 50,000 data points in the bottom chart.

In Figure 24 we can see an increasing trend in the time to create and interact with the SAX data structures as the number of distinct SAX words increases, which is intuitive. We can also see very poor search performance with a small number of SAX words. This poor performance is due to the unusual or rare subsequences not being differentiated from the normal subsequences. We can also see an increasing trend in search time after a local minimum is reached. This increase in runtime is caused by both rare and not so rare subsequences being mapped to rare SAX words. The local minimum also is seen to shift depending on the data size. For the 3 data lengths, the minimums appear to be roughly around 100, 150, and 250 respectively.

Figure 25 shows the runtime contribution of pre-processing, SAX Time and Search time summarized over three ranges of distinct word counts: small (less than 70 SAX words), medium (70 to 650 SAX words) and large (over 650 SAX words). The same experimental results, from Figure 25, is also shown in Figure 26, but stacked to give clear comparison of the overall performance.

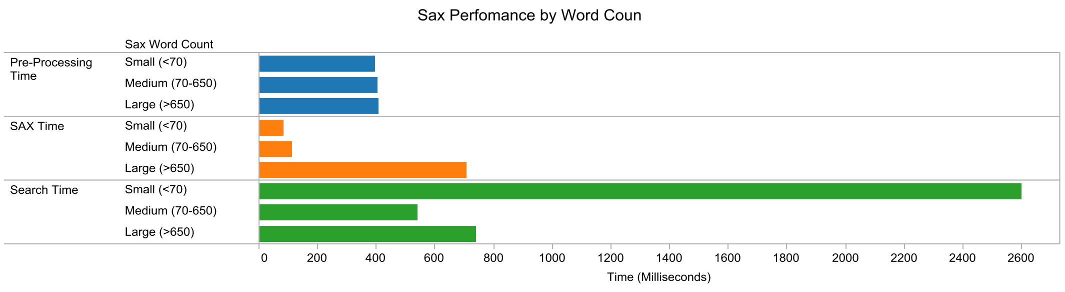


Figure 25: Pre-processing, SAX Time and Search Time for small, medium and large number of SAX words. The results presented are for the Hot SAX with Word Hierarchy method averaged over ten runs for all datasets and subsequence lengths and data size of 50,000 data points.

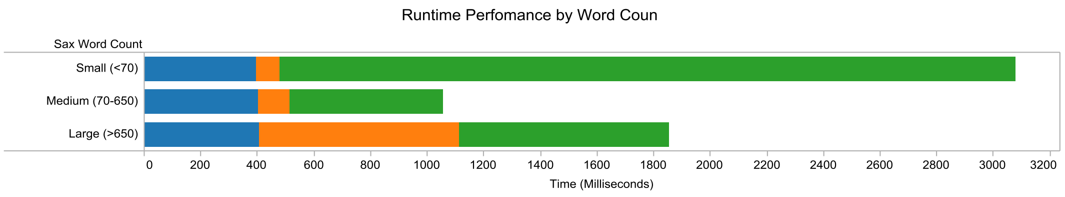


Figure 26: Runtime performance of Hot Sax with Word Hierarchy showing the contribution of preprocessing time in blue, time to interact with the SAX data structures in orange, and the time spent in the double loop in green for different ranges of distinct number of SAX words.

The three discoveries from this work are that: optimal performance occurs when there is a good balance between data separation and overhead costs, and that as the dataset grows we can tolerate more overhead to achieve a good separation of the data. This balance between data-separation and overhead costs lead to the intuition of the self-tuning method with the target number of sax words equal to the square root of the data length, which maps well to the local minimums seen in Figure 24. Finally, with an appropriate discretization (producing between 70 and 650 distinct words) the search time accounts for approximately 50% of the overall runtime. This does not leave a large margin to improve on.

3.5.3 Discord Discovery Performance

Each of the following algorithms were evaluated in terms of Runtime performance:

- *Random Search*

  Random search method implementing nearest neighbour tracking in an array and inner loop skipping.

- *Outer Loop Re-Ordering*

  Our new method that begins with random inner and outer loops, and then re-orders the outer loop after 40 outer loops. This method implements nearest neighbour tracking in an array and inner loop skipping.

- *Neighbour Pruning*

  Our second random search method that implements neighbour pruning based on pre-calculated distance between neighbours. This method implements nearest neighbour tracking in an array and inner loop skipping.

- *Hot SAX*

  Implementation of Hot Sax with SAX outer loop ordering and searching the same word first in the inner loop, then all subsequences randomly. This method implements nearest neighbour tracking in an array and inner loop skipping.

- *SAX Word Hierarchy*

  SAX outer loop ordering and inner loop ordering considers SAX words in increasing order of distance between the words. This method implements nearest neighbour tracking in an array and inner loop skipping.

- *SAX Self Tune*

  Our new self-tuning implementation of Hot SAX based on a target number of SAX

Words. This method implements nearest neighbour tracking in an array and inner loop skipping.

For performance evaluation each method was run on each dataset of lengths 5,000, 10,000, 50,000 and 100,000 and subsequence lengths of 128, 256 and 512. Each of the SAX based methods, except for the self-tuning method, were run with input parameters of

{[w 4, a 3], [w 4, a 4], [w 4, a 5], [w 8, a 3], [w 8, a 4], [w 8, a 5]}.

where w is the word length and a is the alphabet size. Word lengths of 16 and alphabet sizes of 6 were excluded from analysis due to the extremely poor performance on some datasets.

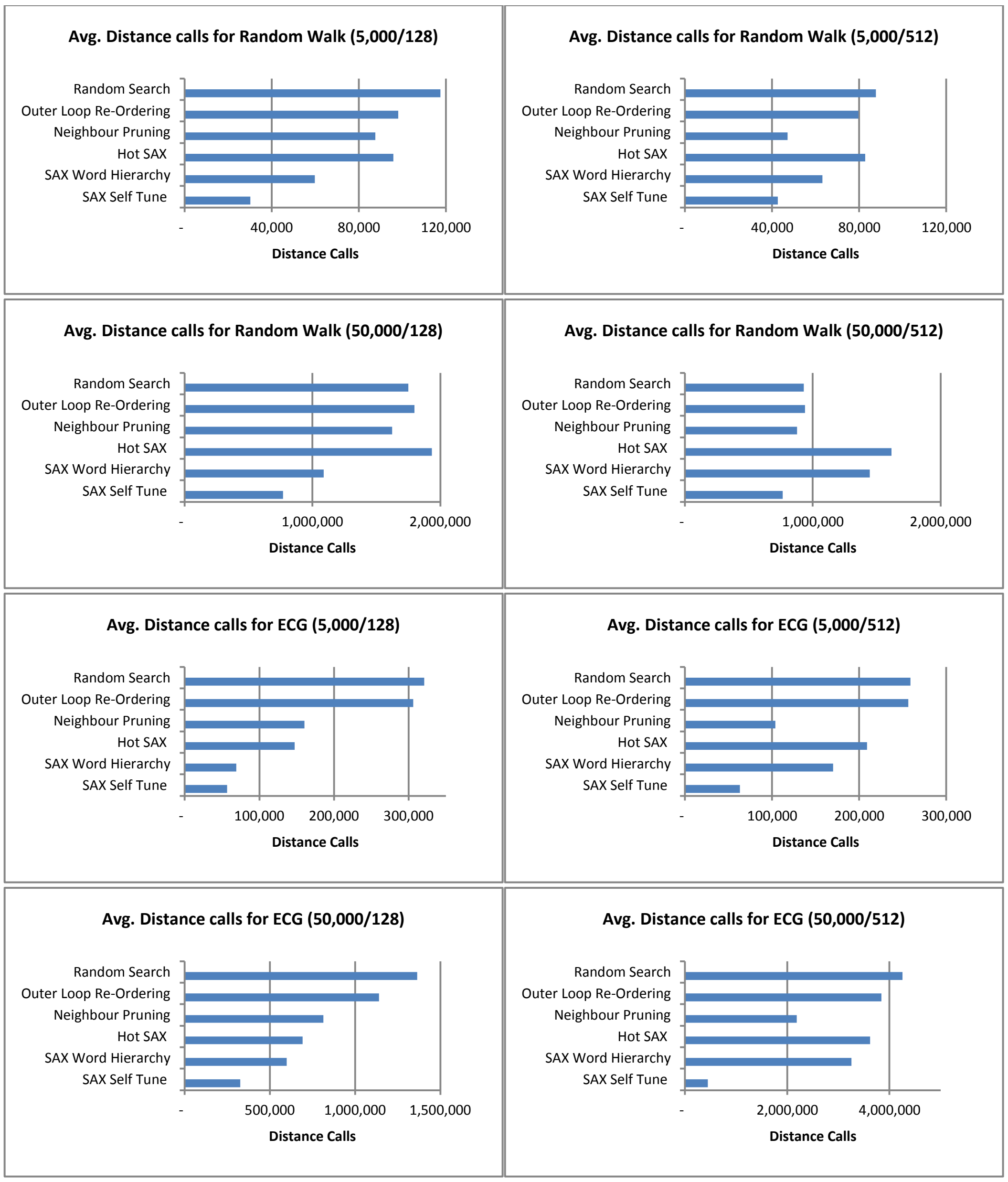


Figure 27: Comparison of discord discovery methods in terms of average distance calls on the Random Walk and ECG datasets of length 5,000 and 50,000 data points and subsequent lengths of 128 and 256. The average values over ten repeats of each experiment are shown. The Hot SAX and SAX Word Hierarchy methods are also averaged over the range of input parameters, as indicated above

Figure 27 compares the methods in terms of average calls to the distance function over ten runs for two of the datasets (more results are shown in Figure 42 of Appendix A). The Hot SAX and SAX Word Hierarchy methods are also averaged over the input parameters indicated above to represent 'expected' performance since a user would not know the best parameters and would have to choose a set of parameters 'randomly'.

For the non-SAX based methods, there is comparable performance between the simple random search and the outer loop re-ordering methods; however, the approach that uses neighbour pruning is able to significantly reduce the number of calls to the distance function and can outperform the SAX based methods in the literature for approximately 50% of the experiments performed. See Figure 42 in Appendix A.

In terms of calls to the distance function, we see that ordering the inner loop using the distance between SAX words improves performance. We can also see significant performance improvement in using the self-tuning approach over other all other approaches, when considering average performance over a range of reasonable input parameters used in the other SAX based methods.

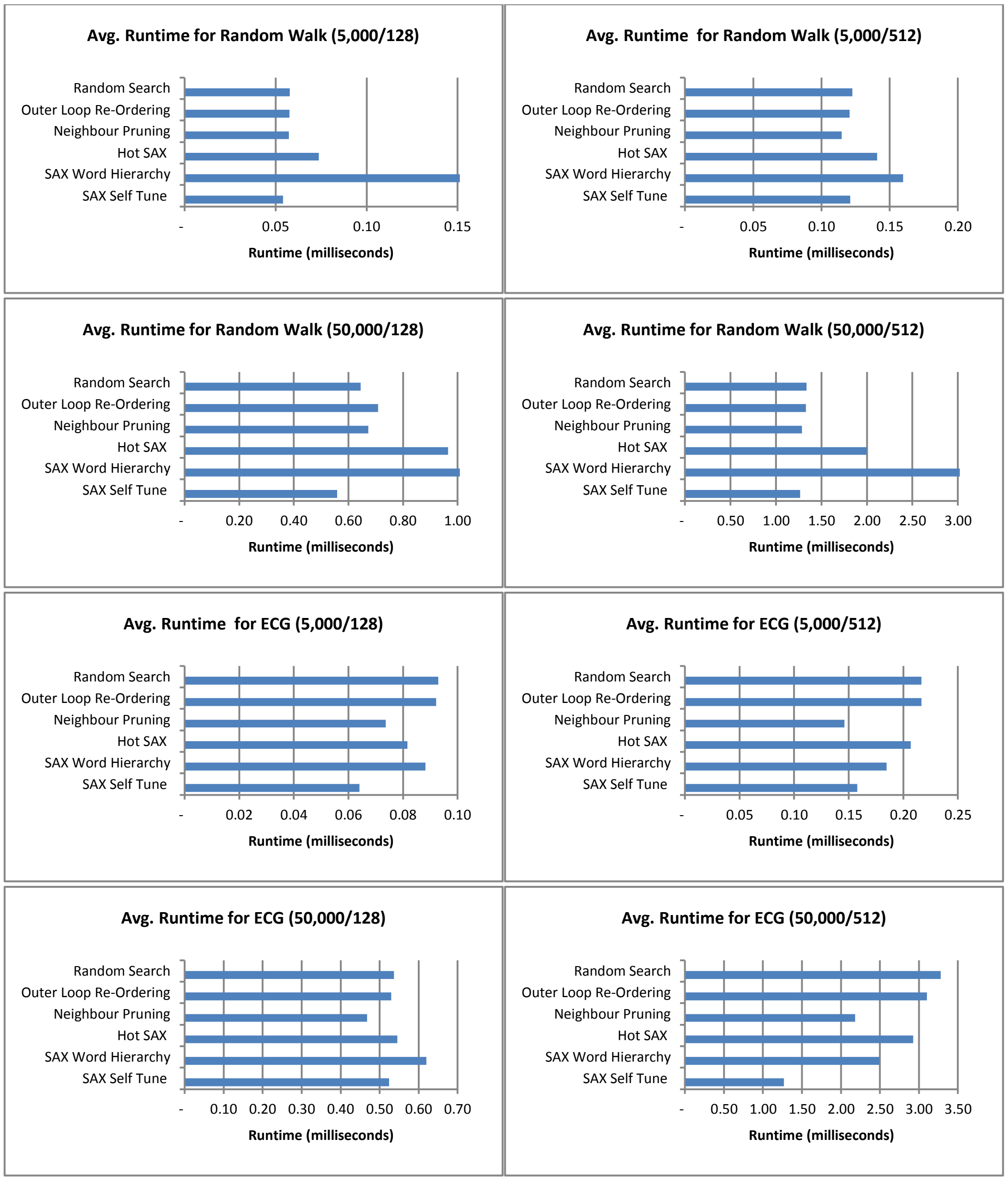
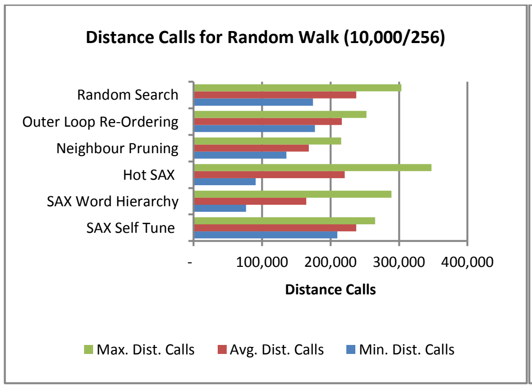


Figure 28: Runtime performance comparison between the different discord search methods on the Random Walk and ECG datasets of length 5,000 and 50,000 data points and subsequent lengths of 128 and 512. Data summarizes average runtime performance for 10 repeats on each of the datasets. The Hot SAX and Sax Word Hierarchy methods were also averaged for each set of input parameters.

Figure 28 summarizes performance by average runtime in milliseconds for two of the datasets (other datasets show similar results, see Figure 43 in Appendix A). Here we see that in terms of average runtime performance, the Sax Self Tune method outperforms the other methods in most cases. Interestingly, some of the Random methods, Random Neighbor Pruning in particular, perform very well in comparison to the Sax based methods. This is due to the limited overhead and consistent performance.

Other works have reported only on the best set of input parameters. Figure 29 and Figure 30 illustrates the danger in this approach where it is not uncommon to see twice as many distance calls for the worst set of parameters for the Hot SAX and Sax Word Hierarchy methods. The random methods and Sax Self Tune method perform much more consistently between best and worst performers. This is more pronounced when looking at runtime performance; particularly for the Sax Word Hierarchy method which can have over 5 times worse performance for the worst set of parameters when compared to the best (additional results can be found in Figure 44 and Figure 45 of Appendix A).



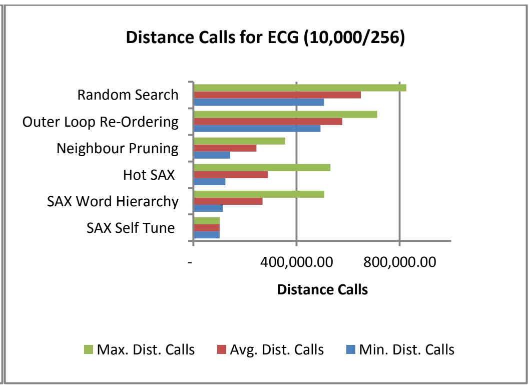


Figure 29: Comparison of methods showing maximum, average and minimum number of calls to the distance function for the Random Walk and ECG datasets of 10,000 data points and a subsequence length of 256.

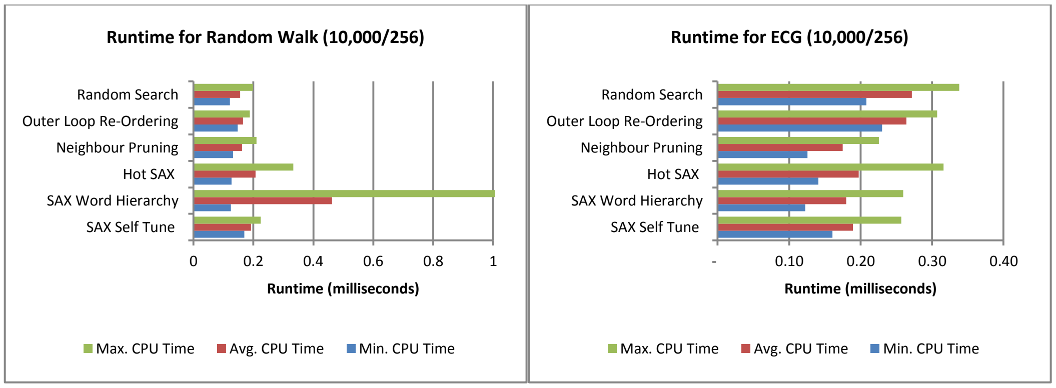


Figure 30: Comparison of methods showing maximum, average and minimum runtime for the discord discovery methods for the Random Walk and ECG datasets of 10,000 data points and a subsequence length of 256.

3.6 Conclusion

In this section, we have examined the problem of finding the top time series discord. This analysis included a discussion on the problem, expectations, effect of datasets and optimizations. We have provided analysis of these optimizations in terms of both runtime performance and number of calls to the distance function.

Several new approaches have been presented that have competitive performance, such as the random search method with neighbour pruning and the self-tuning SAX method. One key advantage of both of these methods is that they do not rely on tuning parameters, which have been shown to have a significant impact on runtime performance. Additionally, both of these methods have competitive performance across all datasets and subsequence lengths tested.

4 Variable Length Discords

4.1 Introduction

In this section we will present a framework for discovering variable length discords. The core of this method is similar to Heuristic Discord Discovery, with two key differences. First, our method will use a random loop ordering heuristic. Second, a new dissimilarity measure will be introduced which allows for the length of the discord to adjust to the discord itself, rather than specifying the length of the discord in advance. This will be accomplished by searching a range of possible discord lengths and reporting the discord as well as the length of the discord. This framework simplifies the task of finding discords as the only input parameters are the minimum and maximum length range of discords to discover, rather than specifying a specific length.

4.2 Dissimilarity Measure

The objective of a variable length search approach to discord discovery is to lessen the challenge of selecting the discord length in advance. Bu et al. [1] also identifies this challenge and suggest running their algorithm for multiple discord lengths multiple times. They recognize that much of the computational work for calculating Euclidian distance and their loop heuristics can be re-used between runs and suggest storing this information at the cost of additional memory. They do not, however, suggest a strategy for selecting the discord after the multiple runs, and there is no evidence in their evaluation that they tested this approach. Furthermore, they do not provide any empirical results to show that such an approach can actually improve runtime and that the gain in distance computations is not outweighed by the infrastructure overhead.

The method presented here will take a different approach and is based on a dissimilarity function which allows us to determine both the discord position and the discord length. For any two subsequences S_i and S_j we would like to measure their maximum dissimilarity within a given range of lengths. We could calculate the Euclidean distance for each length within the range; however since this distance function is monotonically increasing with the increasing length of the subsequences it will always favour the longest length within the range of the minimum and maximum lengths. In order to address this issue, we normalize each of these distances by the length at which they are computed to obtain the weighted distance.

We can formally define the dissimilarity between two subsequences S_i and S_j as:

$$\boldsymbol{Dissimilarity}(S_i, S_j) = \max_{n \in [minLen, maxLen]} \left(\frac{\sqrt{\sum_{m=1}^{n} \left((S_i)_m - (S_j)_m\right)^2}}{n} \right) \qquad \textbf{Equation 4}$$

The corresponding length of dissimilarity can be defined as:

$$\boldsymbol{DissimilarityLen}(S_i, S_j) = \operatorname{argmax}_{n \in [minLen, maxLen]} \left(\frac{\sqrt{\sum_{m=1}^{n} \left((S_i)_m - (S_j)_m\right)^2}}{n} \right) \qquad \textbf{Equation 5}$$

This is illustrated in Figure 31 were we can see two sequences which diverge in the first half, and then become more similar in the second half. The dissimilarity between them grows as long as they continue to diverge to a maximum dissimilarity at length 13, at which point the series values are approaching each other and the dissimilarity begins to decrease.

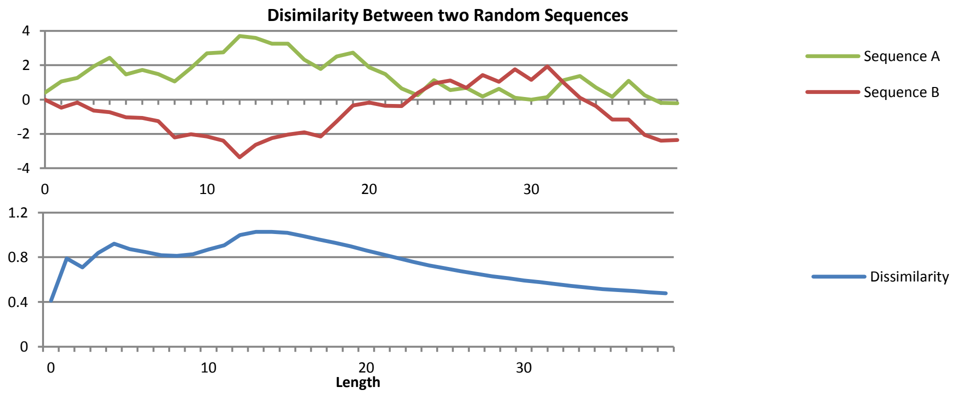


Figure 31: Illustration of the dissimilarity measure is seen on the on the bottom for the two random subsequences on the top plot. As long as the two sequences diverge, the dissimilarity grows. At Length 13, for these sequences, the dissimilarity is at a maximum and after this point the sequences come together.

In order to optimize runtime performance of this calculation, we can take advantage of the fact that as *n* increases from the minimum length to the maximum length all of the previous squared differences are valid. The pseudo code for this method can be seen in Figure 32. For a pair of subsequences, the sum of the squared distance is calculated for each pair of points starting at the beginning of the series. After the difference between each successive pair of points is added to the sum, the weighted distance is calculated and if it is greater than the current maximum distance the new maximum distance and length is recorded. The maximum dissimilarity and the length at which it occurs are reported at the end of the function.

```
Function [Dissimilarity, Length] = Dissimilarity(s1, s2, minLen, maxLen)
    double maxDistance = 0, distance = 0, sum = 0;
    int length = 0;
    FOREACH i = 1 to maxLength
        sum += (s1[i] - s2[i])
        distance = sum/(i+1)
        IF(i >= minLength and distance > maxDistance)
            maxDistance = distance
            length = i
        END
    END
    RETURN [maxDistance, length]
END
```

Figure 32: Pseudo code for computing the dissimilarity between two subsequences.

When incorporated into our discord search framework, this dissimilarity measure will allow determination of both the position and length of the discord, within the range specified.

4.3 Discovery of Variable Length Discords

With the dissimilarity measure presented above, we can now examine the framework for discovering variable length discords, as seen in Figure 33. This variable length method accepts the set of subsequences extracted using the sliding window technique for the maximum length of discord to discover. Then a list of best-so-far distances and corresponding lengths is created for each subsequence and initialized to infinity for distance and zero for the length. This list of best-so-far distances stores only the closest neighbour distance and length, and effectively supports loop abandonment as demonstrated in section 3. A discord candidate is initialized to position of NaN, length of 0 and distance of 0. Then in the outer loop each subsequence is considered in random order. In line 10, we can check the best-so-far distance for the current subsequence and compare it to the current discord candidate distance. If the best-so-far distance is less than the candidate distance, we can skip the entire inner loop. Lines 18 to 25 ensure that the current best-so-far distances are maintained for each subsequence.

In the inner loop, we then compare the current subsequence, from the outer loop, to all non-overlapping subsequences in random order. Note that we must check for overlap at the minimum length in line 13, then in line 15 we must adjust the maximum length for those subsequences that do not overlap at the minimum length but do overlap as the length increases to the maximum length. Following the distance calculations, we can record the best-so-far distances and length for both the inner and outer loops subsequence, if required. Then, if the best-so-far distance for the outer loop index is less than the current discord candidate distance we can abandon the inner loop as the subsequence cannot be the discord. Line 30 to 34 updates the current discord candidate if best-so-far distance for the outer index is larger than the current candidate distance.

```
// S = Set of all distinct subsequences of length n in time series T
// |S| = |T| -n + 1
Function [Distance, Location, Length]
= VariableLengthDiscord(S, minLength, maxLength)
    DiscordPosition = NaN
    BestSoFarDistance = 0
    DiscordLength = 0
    // array of initialized to max value for distance and 0 for length
    NNDist[] = InitArray(|S|, MaxValue, 0)
    OuterSearchOrder[] = GenerateRandomOrdering(|S|)
    InnerSearchOrder [] = GenerateRandomOrdering(|S|)
    FOREACH i IN OuterSearchOrder
        IF NNDist[i].Distance < BestSoFarDistance
            NEXT i  // i cannot be discord skip inner loop
        FOREACH j IN InnerSearchOrder         // inner loop
            IF |i-j| > minLength             // do not overlap
                // Adjust max length to ensure no overlap
                adjustedMaxLen = minimum(|i-j|, maxLength)
                [distance, length]
                = Distance(Si, Sj, minLength, adjustedMaxLen)
                // Store distances and lengths if needed
                IF distance < NNDist[i].Distance
                    NNDist[i].Distance = distance
                    NNDist[i].Length = length
                END
                IF distance < NNDist[j].Distance
                    NNDist[j].Distance = distance
                    NNDist[j].Length = length
                END
                IF NNDist[i].Distance < BestSoFarDistance
                    Break inner loop
                END
            END
        END
        IF NNDist[i].Distance > BestSoFarDistance
            BestSoFarDistance = NNDist[i].Distance
            DiscordPosition = i
            DiscordLength = NNDist[i].Length
        END
    RETURN [BestSoFarDistance, DiscordPosition, DiscordLength]
END
```

Figure 33: Variable length discord algorithm.

It is important to note that the variable length discord method presented here could make use of any of the previous ordering heuristics, instead of random search. The random approach, however, was chosen here due to its simplicity in design and consistent runtime performance, as already seen.

4.4 Evaluation of Variable Length Discords

We have already discussed the benefit of variable length discords in easing the selection of the true discord length upfront, allowing the user to select a range in which to search for the discord.

Now we will consider the effectiveness of this method and the efficiency of finding the variable length discords.

Figure 34, shows result of the variable length discord discovery method on the shuttle dataset. For this test, the range used for the discord discovery was 75 to 300, with the top discord of length 103 being discovered at position 2254, and the second discord of length 86 starting at position 2064.

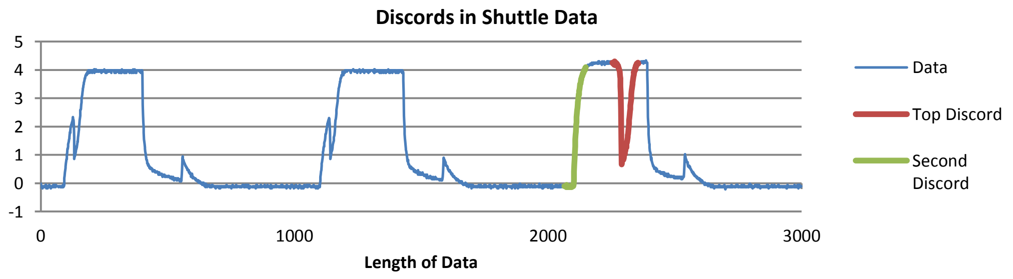


Figure 34: Illustration of the top 2 variable length discord discovered on shuttle dataset

The alternative to this approach, as speculated by Bu et al. [1], is to run the discord discovery method multiple times for the range of distances. Since this was not included in the evaluation of the SAX paper [1], we will evaluate the runtime performance of our method to running our fixed length discord for multiple subsequence lengths. This experiment was run 10 times on the random walk dataset with lengths of 2k, 4k, 8k, 16k and 32k points. For the variable length discord method, we used 75-300 as the range of discord to be found. For comparison, we used the random method tested above and selected discord lengths of 75, 150 and 300. These lengths were used as to match with the minimum length and maximum length in the variable length method, as well as an intermediate length.

As can be seen in Figure 35, the variable length discord method's performance is comparable to that of running the fixed length discord method a mere three times; however, the variable length method considers the complete range of distanced between the minimum and maximum distances.

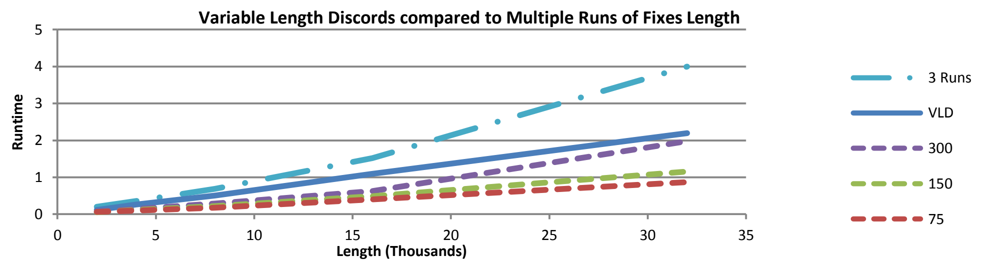


Figure 35: Comparison of Variable Length Discord algorithm to a fixed length algorithm. The variable length method was run on a range of 75 to 300 and the fixed length methods were run for subsequence lengths of 75, 150 and 300.

The variable length discord method was effective at finding the discords and length. However, it often appeared to favour short discords, which may have been valid for the datasets used in testing. Also, it was found that the discords found by the variable length method would overlap the discords found by the fixed length method used for comparable lengths. So, while this method appeared to be effective, its usefulness is questionable as the fixed length discord method seems to return the same regions of discords faster.

4.5 Conclusion

In this section we have examined a new approach to discovery of variable length discords. This method was seen to be effective and efficient compared to running a comparable fixed length discord method two to three times, with the variable length method considering all distances in the range. However, the utility of this method was found to be questionable, as the fixed length method and variable length method would both return overlapping discords. This limits the benefits provided by the variable length method, considering the increase in runtime, if one is only interested in the ‘approximate’ position of the discord, which typically would be subjected to further analysis that may find the correct alignment of the discord around the position found. This also highlights that discord discovery is not particularly sensitive to the length of discord.

5 Top-*K* Discords

5.1 Introduction

Within any process that generates time series data, the process is not limited to generating only one unusual subsequence. Therefore, it is reasonable that one may want to find the top-*K* discords. Ideally, we would like to find all unusual subsequences in the data, as outliers in the dataset. In this section, we will examine the problem of discovering the top *K* discords. We will also examine two methods where all unusual subsequences are returned, without specifying *K*.

5.2 Methods

One prior approach discussed in the literature will be examined alongside several competing methods that we have developed. While no adequate runtime evaluation has been presented in the literature, we will provide evaluation of all these methods.

5.2.1 Re-run Algorithm

The first approach we will discuss is simple and intuitive, and entails re-running the discord discovery algorithm from 1 to k times, as has been presented by Bu et al. [1]. There are two keys to this approach, first is to exclude any subsequence that overlap with a previously discovered discord; and second is to make use of the computational effort in preceding loops. While Bu et al. [1] discussed these ideas, they provided no further evaluation.

Several key aspects are important to make note of in the pseudo code in Figure 36. First, while this code implements a random search heuristics, any ordering heuristics could be incorporated into this framework. Second, line 2 initializes the array of nearest neighbour distances. These nearest neighbour distances are relevant for each subsequent loop, while determining the $i^{th} + 1$ discord and is thus initialized before the *while* loop. The use of distance computations from prior loops is effective at reducing the work in subsequent loops. Finally, on line 12, in addition to checking if the subsequences overlap, we must check if the subsequence overlaps with any discord already found. This is to ensure that each discord is a distinct anomaly in the data.

```
// S = Set of all distinct subsequences of length n in time series T
// |S| = |T| -n + 1
Function [DiscordsFound] = FindTopKDiscords(S, K)
    NNDist[] = InitArray(|S|, MaxValue)
    DiscordsFound[] = EmtpyArray()
    WHILE DiscordsFound.Count < K
        DiscordPosition = NaN;        BestSoFarDistance = 0
        OuterSearchOrder[] = GenerateRandomOrdering(|S|)
        InnerSearchOrder[] = GenerateRandomOrdering(|S|)
        FOREACH i IN OuterSearchOrder                    // outer loop
            IF NNDist[i] < BestSoFarDistance
                NEXT i  // i skip inner loop
            FOREACH j IN InnserSearchOrder            // inner loop
                IF |i-j| > n & !Overlaps(DiscordsFound,i,j)
                    // non overlapping
                    d = Distance($S_i$, $S_j$)    // compute distance
                    // Store distance
                    NNDist[i] = MIN(NNDist[i], d)
                    NNDist[j] = MIN(NNDist[j], d)
                    IF NNDist[i] < BestSoFarDistance
                        BREAK inner loop // abandon loop
                    END
                END
            END                                       // end inner loop
            IF NNDist[i] > BestSoFarDistance
                BestSoFarDistance = NNDist[i]
                DiscordPosition = i
            END
        END                                           // end outer loop
    DiscordsFound.Add(BestSoFarDistance, DiscordPosition)
    END // end while loop
    RETURN DiscordsFound
END
```

Figure 36: Pseudo code for finding top-K discords by re-running the algorithm.

One observation made during the analysis of the runtime performance is that the time to find the *i*th discord tends to decrease, or remain approximately constant, with finding each subsequent *i*th *+1* discord until the point where we are finding discords that are not so unusual. At this point we see an increase in runtime and distance calls, as it is harder to differentiate the next discord from the rest of the subsequences. This phenomenon is illustrated in Figure 37 and is a particular problem for large values of *K*.

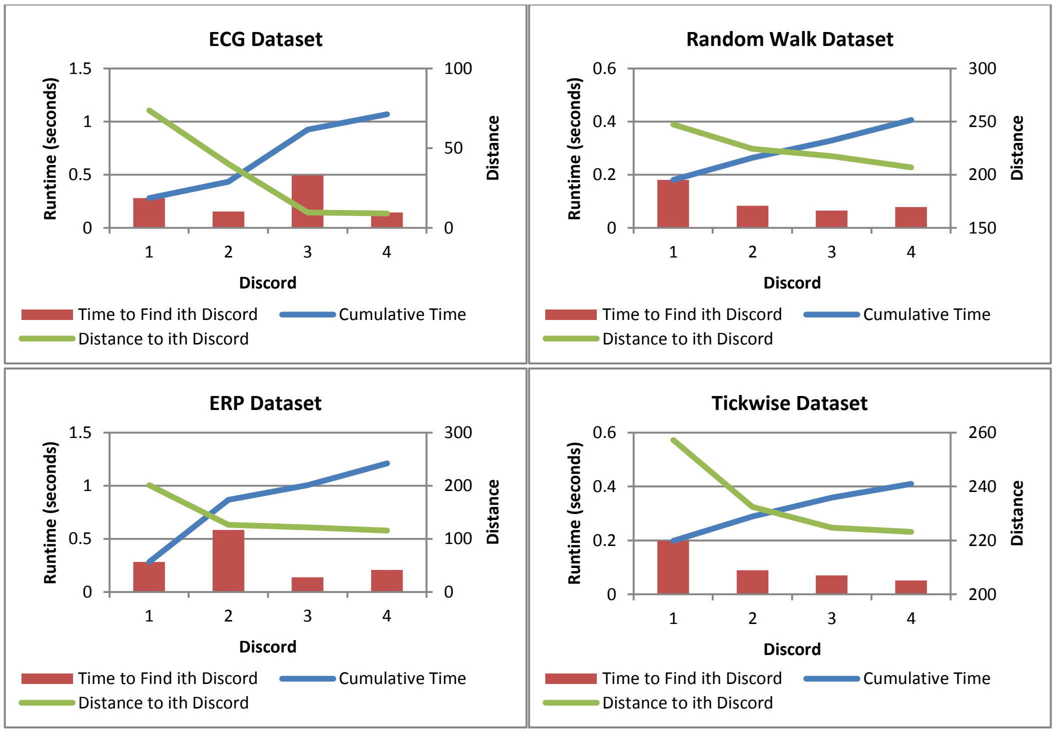


Figure 37: Time spent to find each of the top 4 discord with the cumulative time along with the distance to the discord. Each data point represents the average of 10 runs for a file length of 10,000 data points and subsequence length of 256.

In Figure 37 we see a large increase in time to find the 3rd and 2nd discords for the ECG and ERP datasets respectively, and do not see this trend for the Random Walk or Tickwise datasets. This may indicate that the discords found when there is a large increase in runtime are not so different from the rest of the data. Looking at the results from the ECG dataset, we see a large increase in runtime to find the 3rd discord and then little change in the discord distances for the 3rd and 4th discords. The same trend is present for the ERP dataset, with the 2nd discord seeing the spike in runtime. This seems to support the notion that these discords may not be so dissimilar. This trend is not seen for the top 4 discords in the random walk or tickwise datasets, where there continues to be decreasing runtimes and discord distances for each subsequent discord.

These observations are motivation for the self-terminating and statistical approach that we have developed and will present later in this section.

5.2.2 Prune on K^{th} Item

In this section, we will present an alternate method that we have developed. This method will keep a listing of the *K* best-so-far discords and then prune on the k^{th} item in the ordered list of discord candidates. This eliminates the need to re-run the double nested loop, but involves some additional complexity.

While this method sounds straight forward, the complexity lies in the management of the top *K* discord candidates. As will be demonstrated, it is not enough to simply maintain *K* candidates; we must maintain a list of all possible discord candidates, which will be significantly longer than *K.* This lengthened list of candidates is needed since we cannot guarantee that a subsequence previously considered to be a discord will not become a valid discord again. This condition is illustrated in Figure 38. Suppose we are looking for the top 2 discords and we first discover discord *A* with a distance of 10 to its nearest neighbour. Then we find discord *B*, which overlaps *A,* and has a greater nearest neighbour distance of 15. Under this condition *B* will replace *A* as a candidate. Later in the search, we may find discord *C*, which replaces *B* in the same way that B replaced A. At this point, candidate *A* should once again be a valid, non-overlapping discord candidate. Without maintaining the subsequence *A* as a candidate, we would not produce the correct result.

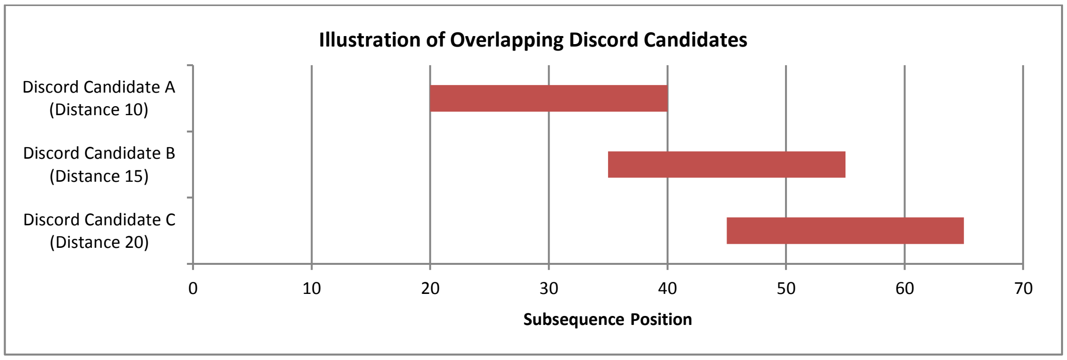


Figure 38: Illustration of the challenge in maintaining listing of K discord candidates.

In this method, we maintain a sorted list of all discord candidates ordered by nearest-neighbour-distance (largest first). Then to obtain the *k*th best-so-far discord distance needed for pruning, we must search the list of candidates for the *k*th non-overlapping candidate in this list. This listing of discord candidates can be pruned on the *K*th*+1* non-overlapping item.

This strategy produces a correct result and only iterates the double nested loop one time. However, maintenance of list of discord candidates requires considerable effort and has a negative impact on runtime performance. For the experiments performed for time series length of 10,000 data points, the length of the list of discord candidates ranged from 10 to over 600, with an average of around 160.

As compared to the previous method, where the double nested loop is ran *K* times, this method does not offer any improvement in the number of distance calls or runtime performance.

5.2.3 Re-Run With Sorting

Our second method is similar to the first approach discussed, of re-running the algorithm *K* times, with one key change. For the first loop, this method uses a random inner and outer loop ordering (same as the first method) and then a new outer loop ordering is generated after finding each successive discord, 2 to K.

During the first loop to find the top discord, many, but not all of the pairwise distances are computed and the nearest neighbour distance for each subsequence stored. This first outer loop only guarantees that all pairwise distances are computed for the top discord and that a distance is found for each of the other subsequences that is less than the distance for the top discord. Many of these distances, which have effectively been used to abandon on, are significantly smaller than the discord distance.

After finding the *i*th discord, a new outer loop ordering is generated by sorting the current nearest neighbour distances in descending order. The intuition behind this is that the nearest neighbour distance array is an approximation to the true nearest neighbour distance for each subsequence.

By generating a new outer loop ordering, we will examine subsequences that have a higher chance of having a large nearest neighbour distance early in the search to find the next discord, while leaving all subsequences with a close neighbour to the end.

5.2.4 Self-Termination

Ideally, we do not want to specify the *K* in any top-K discord search and would like the algorithm to return all outliers and nothing else. Having the user specify a value for *K* has one of two potential problems. First, if a value of *K* is too small valid discords will not be returned. Second, specifying a value of *K* that is too large will have a significant and negative effect on runtime performance.

The idea behind this method is to use the currently computed nearest-neighbour distances to determine when to stop searching for additional discords. The method's implementation is very similar to the first method presented, of re-running the double loop, with one key difference. Rather than stopping the discord search after the K^{th} discord is found, a stopping criteria is defined that determines when the algorithm has found all the discords.

Following each run, and having found the i^{th} discord, the mean and standard deviation of the nearest-neighbour distances is computed. As noted, these nearest neighbour distances are approaching the true nearest neighbour distances and can be used to represent the population of all nearest neighbour distances. Then we can stop our discord search when the nearest neighbour distance of the last discord found is less than $\mu + x\sigma$ where μ and σ are the mean and standard deviation of the nearest neighbour distances for each subsequence, and x is an input parameter that specifies how many standard deviations from the mean we would like the threshold for a discord to be.

Intuitively this makes sense. At first the estimated mean and estimated standard deviation are larger than the true mean and standard deviation, since it is based on the current nearest neighbour distances, so the method is conservative. With each iteration this estimation improves and the value approaches the true mean and standard deviation and we can feel more confident about quitting the search once the stopping criteria is meet.

This method replaces the input parameter *k* with the new threshold *x*. This threshold may be easier to select than *k*, but still needs to be specified and can impact the number of discords returned.

Experimentation did give good results and was successful at returning a range of discords; however, it is important to note the following. First, when the threshold *x* was too high, greater than 1.5 or 2, no discords were returned for some datasets, however the top discord would still be returned. Second, with regards to performance, the final loop was often slow to compute since the discord being found was not too dissimilar from the rest of the subsequences. Third, on some datasets, a large number of discords were returned. And finally, the number of discords returned in each successive run did vary. This is due to the variability in the stopping criteria; since it is based on it is based on a randomly chosen subset of distance computations.

5.2.5 Statistical Approach

The final idea tested for finding the top *k* discords was a statistical approach. For this method, a subset of the subsequences was sampled at a fixed interval and the true nearest neighbour was found for each subsequence. The mean and standard deviation were computed for the sample set's nearest neighbour distances and used as an estimation of the true mean and standard deviation for the entire set of nearest neighbour distances. This information was used to set an abandoning distance.

The subsequences chosen for the sampling were chosen on a fixed interval of $3/4n$, where n is the subsequence length. This fixed length sampling was chosen to get a nearest neighbour distance calculated that overlaps with each of the subsequences.

After the sampling and estimation, the method proceeded with the double nested loop, but instead of abandoning on the best-so-far distance (as in all previous methods), this method abandons on the value of $\mu + x\, \sigma$, where μ and σ are the mean and standard deviation of the nearest neighbour distances calculated from the sample set, and x is an input parameter. Then all subsequence that are not abandoned in the inner loop at the threshold $\mu + x\, \sigma$ are added to the list of discords.

Some of the notes from the self-terminating method apply to this method as well. First, when the threshold of x is too large no discords are returned. Second, the number of discords returned can vary between repeats of the same run. Also, when many discords are returned, the runtime is slow.

5.3 Performance evaluation

The performance of these 5 methods is presented in terms of both distance calls and runtime. Figure 39 shows the results in terms of distance calls. In the left column of this chart, we see the number of discords returned. For the first three fixed-K methods (re-run, prune on K, and re-run with sort) K was specified at 4, and remaining two variable-k methods would return a variable number of discords. For the fixed-K methods, we can see that the re-sorting method is consistently best in terms of distance calls. The variable-k methods performance degrades significantly as the number of discords returned increased. The Statistical approach would consistently return a large number of discords and would require a significantly larger number of distance calls to complete.

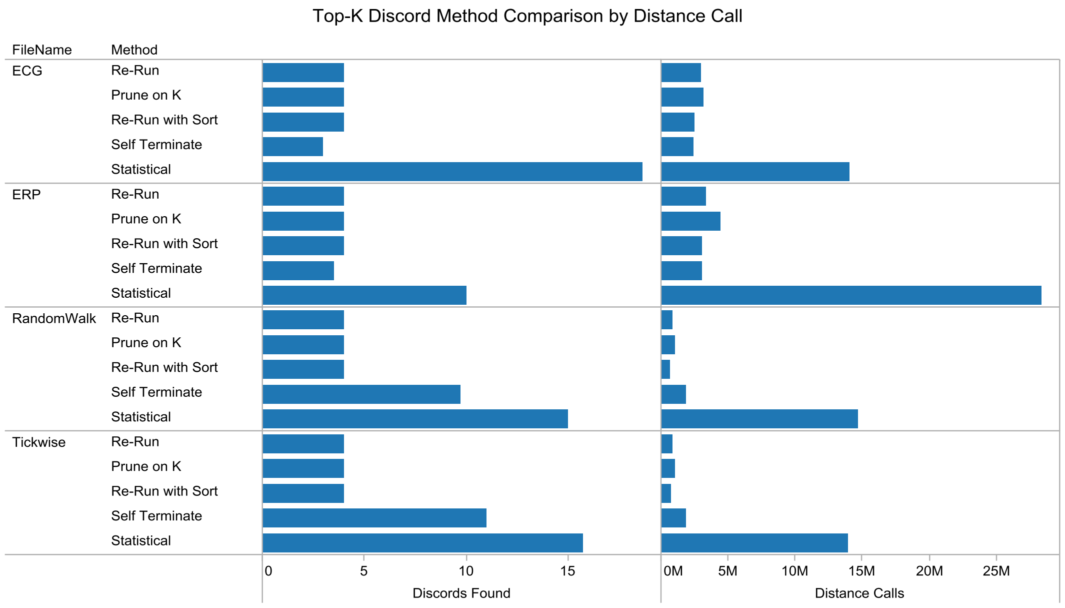


Figure 39: Comparison of Top-K discord methods in terms of distance calls. These figures are the average of ten runs for a subsequence length of 256 and file length of 10,000 data points.

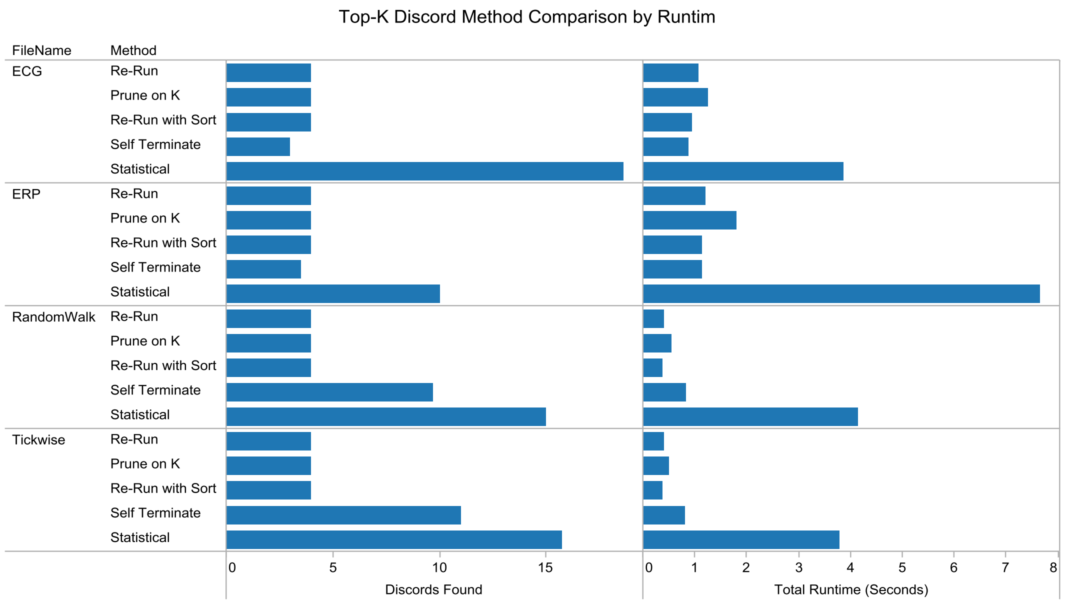


Figure 40: Comparison of Top-K discord methods in terms of runtime. These figures are the average of ten runs for a subsequence length of 256 and file length of 10,000 data points.

Figure 40 displays the performance results in terms of runtime. Here we see the same trend, the re-run with sort method is best for the fixed-K methods. For the variable-k methods, we see that the self-terminating approach performs much better, but consistently returns fewer discords then the statistical approach.

Comparing the re-run with sort to the self-terminating method, we see that the self-terminating method's performance is comparable, and only significantly slower as the number of discords returned increases significantly over the fixed K.

Given the large number of discords returned by the variable-k methods, it is important to see the distances that these discords are found at. This will help in understanding the quality of the discords returned.

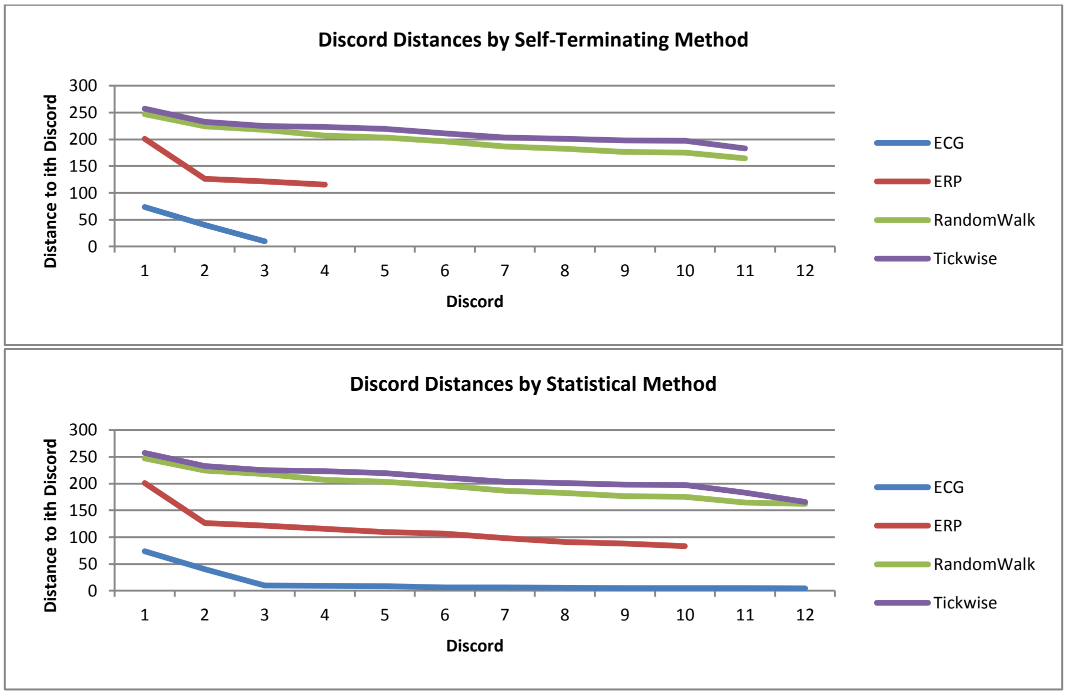


Figure 41: Nearest neighbour distances of top discords returned by variable-k methods.

Figure 41 displays the distances that the top discords are returned at for both of the variable-k methods. The self-terminating method in the top graph appears to terminate at a more appropriate K for the ERP and ECG datasets, when the improvement in discord distances shows little change, while the statistical method continues to return discords for these two datasets. The Random Walk and Tickwise datasets have a large number of discords returned in both cases.

The self-terminating approach appears to be more effective at returning an appropriate number of discords as representative of the outliers in the dataset. This method is comparable to the performance of the re-run with sort method for a comparable number of discords returned and is a good choice where the user does not want to specify a specific value for K.

5.4 Conclusion

In this section, we have seen five different approaches to finding the top-k discords; one of which has been discussed before in the literature and 4 novel ideas we have developed. The performance of these methods was compared and the re-run with sort method is best in terms of overall performance, however the self-terminating method is competitive and will return all discords in the dataset.

6 Conclusion

6.1 Summary of Work

In this work we have examined the problem of finding time series discords. This work focused on three different areas of discord discovery: Top Discord, Variable Length Discords, and Top-K Discords. In all of these works, we have strived to reduce the number or ease the selection of input parameters required by the end user.

First, the top-discord work examined optimizations as already presented in the literature and some new methods that we have developed. It also included an analysis of the problem, expectation, and the effect of different datasets. In this area our self-tuning approach proved to be competitive in terms of both calls to the distance function and runtime, without the need to choose tuning parameters. Also, the specific optimization to maintain an array of nearest neighbour distances, rather than a matrix of all pairwise distances, greatly extended the scalability of the method to larger datasets.

In the second section, we have developed a new approach for finding variable length discords, rather than discords of fixed, use specified length. This method simultaneously searches a range of discord lengths, where the user specifies the range (the thought being that selecting the range is easier that specifying a specific discord length). This method has been shown to be effective and the increase in runtime comparable to running the fixed length discord method 2-3 times of varying lengths. However, the discords returned in the variable length approach would consistently overlap with locations found by the fixed length methods, which limits the benefits provided by this method, where approximate discord position is sufficient.

Finally, we have examined five different approaches to finding the top-k discords, one of which has been discussed in the literature. One novel idea we have developed is a self-terminating approach which returns all unusual subsequences in the dataset as determined by the stopping criteria. This reduces the need for the user to try and guess how many anomalies are in the dataset.

6.2 Future Work

One weakness of the work presented here is the original definition of time series discord. This weakness in the definition can be explained using the ECG data as an example. If a patient has a particular cardiac problem characterised by an anomalous heartbeat, this irregular heartbeat will show up as a discord in the data, provided that only one anomaly is recorded. If the ECG dataset contains two of the same anomalous patters, they will be nearest neighbours to each other and will have small nearest neighbour distances. This could lead to the condition where the true discords in the data are not returned. One idea to consider would be to alter the definition of discord to allow several close matches, and define a discord as the subsequence with the largest distance to its k^{th} nearest neighbour.

7 References

[1] Bu, Y.; Leung, T.; Fu, A.; Keogh, E.; Pei, J.; Meshkin, S. WAT: Finding Top-K Discords in Time Series Database. Proceedings of the 7th SIAM International Conference. 2007.

[2] Chandola, V; Banerjee, A.; Kumar, V. Anomaly Detection: A Survey. ACM Comput. Surv. 2009

[3] Cheng, H.; Tan, P.; Potter, C.; Klooster, S. Detection and Characterization of Anomalies in Multivariate Time Series. SDM. 2009.

[4] Keogh, E.; Chakrabarti, K.; Pazzani, M.; Mehrotra, S. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. Journal of Knowledge and Information System. 2000.

[5] Keogh, E.; Lin, J.; Fu, A. Hot sax: Efficiently Finding the Most Unusual Time Series Subsequence. Proceedings of ICDM. 2005.

[6] Keogh, E.; Lonardi, S.; Ratanamahatana, C. A.; Wei, L.; Handley, J. Compression-based data mining of sequential data. Data Mining and Knowledge Discovery. 2007.

[7] Lin, J.; Keogh, E.; Lonardi, S. A symbolic representation of time series, with implications for streaming algorithms. Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery. ACM. 2003.

[8] Lin, J.; Keogh, E.; Wei, L.; Lonardi, S. Experiencing SAX: A Novel Symbolic Representation of Time Series. Journal of Data Mining Knowledge Discovery. 2007.

[9] Luo, W.; Gallagher, M. Faster and parameter-free discord search in quasi-periodic time series. Advances in Knowledge Discovery and Data Mining. 2011.

[10] Mueen, A.; Keogh, E.; Zhu, Q.; Cash, S.; Westover, M. Exact Discovery of Time Series Motifs. SDM. 2009.

[11] Pham, N.; Le, Q.; Dang, T. HOT aSAX: A Novel Adaptive Symbolic Representation for Time Series Discords Discovery.

[12] Preston, D.; Protopapas, P.; Brodley, C. Event Discovery in Time Series. arXiv preprint arXiv. 2009.

[13] Salvador, S; Philip C. Toward accurate dynamic time warping in linear time and space. Intelligent Data Analysis 11. 2007

[14] Son, M.; Anh, D. Some Novel Heuristics for Finding the Most Unusual Time Series Subsequences. Advances in Intelligent Information and Database Systems. 2010.

[15] Yankov, D, Keogh, E and Rebbapragada, U. Disk Aware Discord Discovery: Finding Unusual Time Series in Terabyte Sized Data Sets. Knowledge and Information Systems. 2008.

[16] Yankov, D.; Keogh, E.; Medina, J.; Chiu, B.; Zordan, V. Detecting time series motifs under uniform scaling. Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. 2007.

Appendix A: Additional Experimental Results from Chapter 3

Figure 42 contains the complete set of results, in terms of calls to the distance function, for the top discord work presented in section 3.5.3. The data represents the average of ten runs for each dataset and file length (as seen in the rows), and each subsequence length (in the columns). Each method is displayed in a different colour.

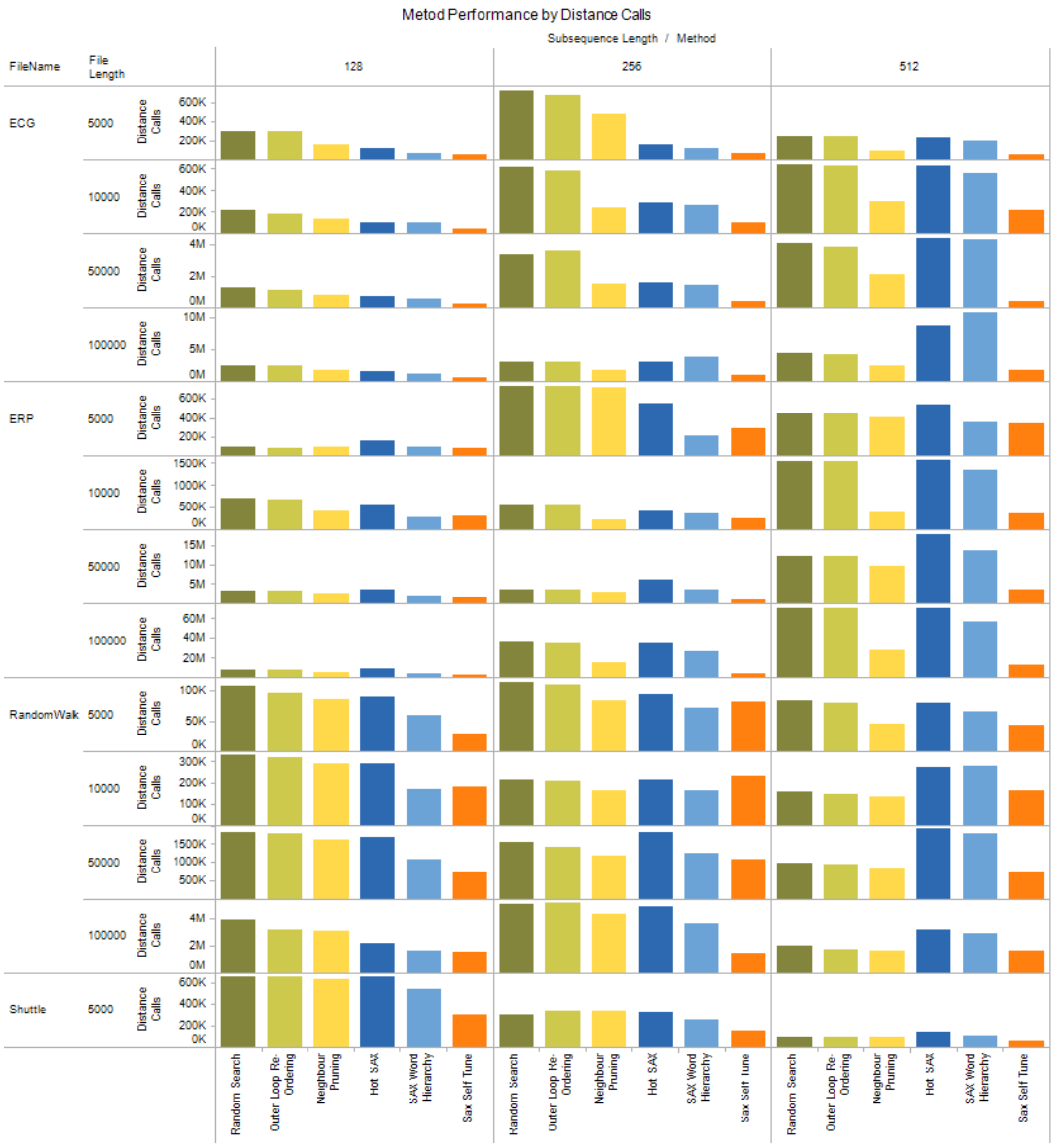


Figure 42: Complete set of results by distance calls for 4 datasets, different data lengths, and subsequence lengths for each of the 6 methods compared.

Figure 43 contains the complete set of results, in terms of runtime performance, for the top discord work presented in section 3.5.3. The data represents the average of ten runs for each dataset and file length (as seen in the rows), and each subsequence length (in the columns). Each method is displayed in a different colour. As discussed in the text, our self-tuning SAX approach is the top performer in the majority of cases.

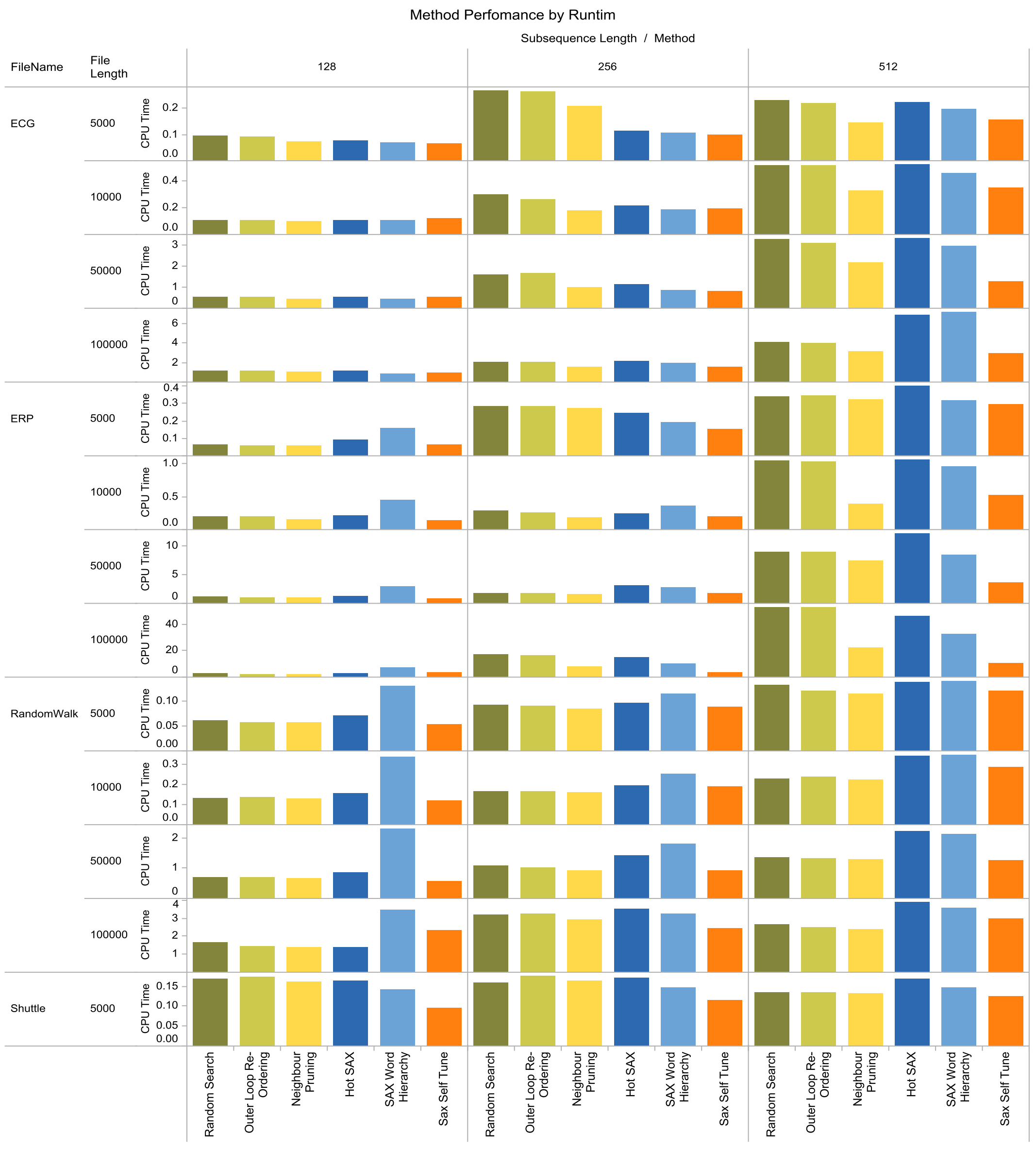


Figure 43: Complete set of runtime results for 4 datasets, different data lengths, and subsequence lengths for each of the 6 methods compared.

Figure 44 and Figure 45 display the range of performance for each of the methods in terms of calls to the distance function (first figure) and CPU time (second figure) for four datasets and each of the methods. These results are an extension of the results presented in section 3.5.3. As discussed in the text, we can see a large range in performance for the Hot SAX and Sax Word Hierarchy methods caused by the different tuning parameters. The random search methods and the SAX Self Tuning method that we have developed have more consistent performance. These figures illustrate the danger of reporting only the best set of tuning parameters.

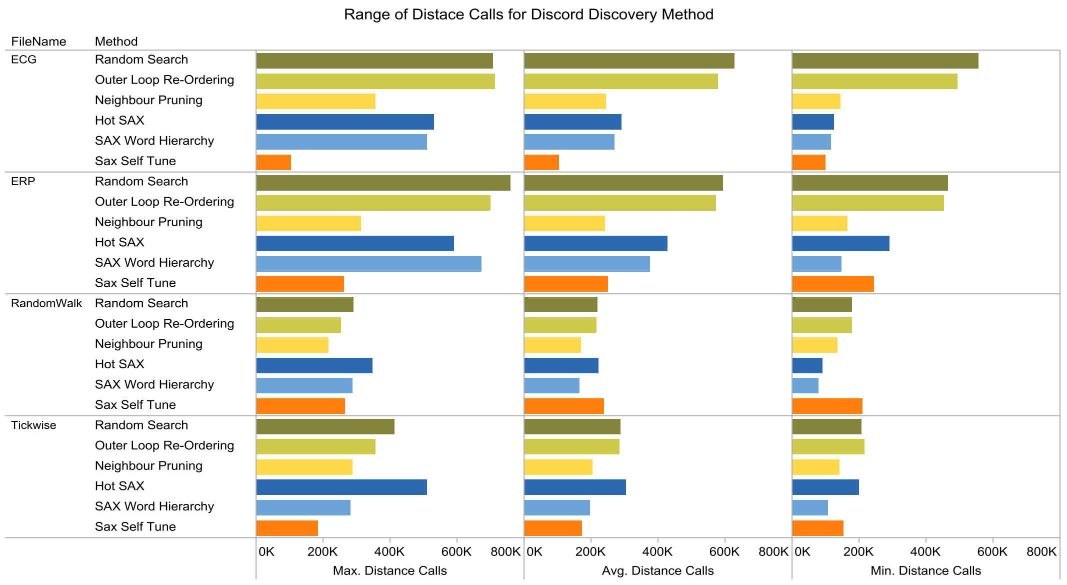


Figure 44: Range of performance in terms of calls to the distance function for each method and across four datasets. Results are for a data length of 10,000 data points and subsequence length of 256. The average of ten runs is displayed.

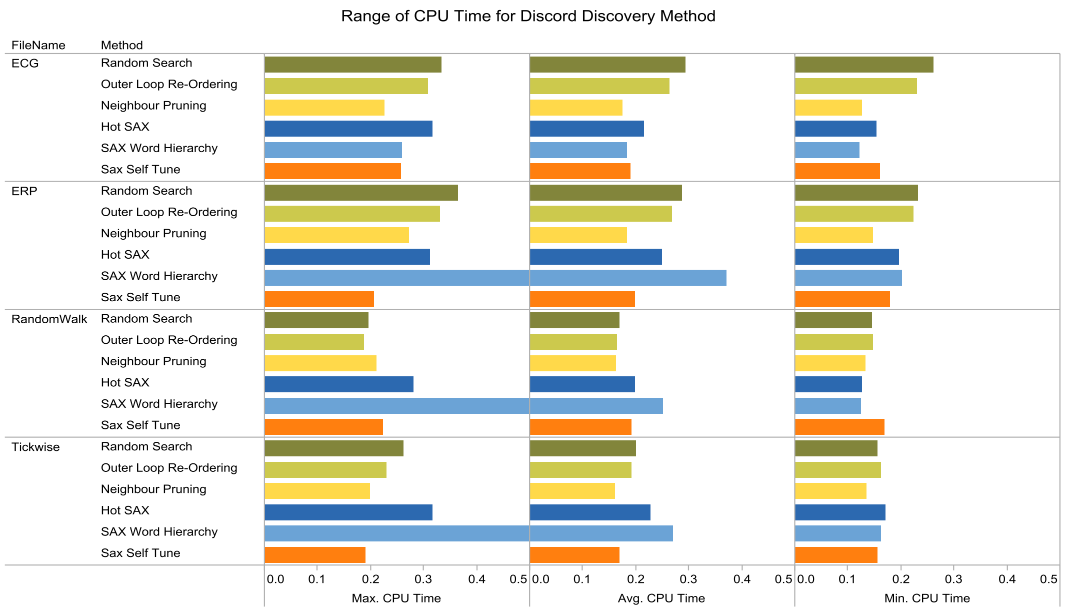


Figure 45: Range of performance in terms of CPU time for each method and across four datasets. Results are for a data length of 10,000 data points and subsequence length of 256. The average of ten runs is displayed.