Parallel Locality Sensitive Hashing for Network Discovery from Time Series

by

Svetlana Sodol

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

January 2022

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Post-doctoral Studies for acceptance, the thesis entitled:

Parallel Locality Sensitive Hashing for Network Discovery from Time Series

submitted by Svetlana Sodol in partial fulfillment of the requirements for the degree of Master of Science

in Computer Science

Examining Committee:

Dr. Alan Wagner, Computer Science, UBC
Supervisor

Dr. Aastha Mehta, Computer Science, UBC
Supervisory Committee Member

Abstract

Similarity operations on time series are a vital area in data mining research. Science and systems applications require a scalable solution that is fast enough to work with streaming of real data and provide reliable results in the presence of noise, high dimensionality and time dependency. Locality sensitive hashing (LSH) has shown advancement in limiting the number of comparisons required for similarity operations, thus reducing the overall time, by pre-processing the data into buckets as candidates for approximate nearest neighbours. This thesis proposes a scalable system based on locality sensitive hashing by implementing it in parallel with independent hash functions. The parallel system we present is implemented in a message-passing framework for four locality sensitive hashing methods – minhash, approximate binary correlation (ABC), symbolic aggregate approximation (SAX), and "sketch, shingle and hash" (SSH), each with its own pre-processing, hash creation, and similarity measure. A preliminary investigation implements minhash with a bag-of-words representation of text data to validate our proposed framework. The experimental part of the thesis focuses on comparing the other three LSH methods (ABC, SAX, SSH) on a real flight data set processed in a streaming fashion, flexible to the size of the time series used. The output of our parallel system is a similarity network discovered from the data, that we use to detect an anomaly present in the data set. The LSH methods are evaluated with respect to the time of execution, amount of communication, computation complexity, tuning of parameters and the required number of similarity operations. Our results indicate the feasibility of the implemented methods and proposed framework for this type of application and this sort of real-life time series. Our thesis concludes with discussion of the impact of the similarity measures on the network discovery results, as well as proposing further investigations into other parts of the parameter space.

Lay Summary

Finding similar items in data mining requires a fast solution to work on real-life data of large size and complexity. A method titled “locality sensitive hashing” (LSH) has shown promising results in achieving high speed by producing groups of items that are expected to be similar so that the main task can focus on comparing only inside the groups. In this thesis we propose to decrease the time of operation further by having multiple subtasks run at the same time. We show implementations of four LSH methods – minhash, ABC, SAX and SSH, each with its own way of defining similarity. Our experiments test the methods on a real-life airflight data set. We compare the results between the methods on time of execution and ability to show anomalies. Our results show that our proposed framework and implemented methods are suitable for this line of applications and data.

Preface

This thesis presents original unpublished work of the author, S. Sodol, done under supervision and guidance of A. Wagner

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor, Dr Alan Wagner, for guiding me through the journey that is graduate school. Every meeting I have learnt new perspectives and was charged with positivity even in front of hard work still to be done. A very special thank you to UBC and the Systopia Lab, for allowing the opportunity for my studies and research. I am very grateful to my second reader, Dr Aastha Mehta, for thorough and timely feedback.

I would like to thank the scientific community for persevering in face of the challenge that is research and for having contributed to our current level of knowledge and technology (and especially to all those continuing their work through the current pandemic). I especially want to extend my gratitude to the authors of the data sets that my work in this thesis used, Matthews [2012] and Dua and Graff [2017], for making their data available and accessible to others. Thank you for supporting reproducibility and scientific collaboration through fields, people, and years.

I am forever indebted to my parents and teachers, who throughout my life, have kindled my curiosity. Thank you to my grandparents and in-laws for their endless support and patience. My heartfelt gratitude to my husband who has been my greatest cheerleader through the difficult times. And lastly, a big thank you to my friends, Olya, Nick and Ana, for always having my back, even from halfway around the globe. I could not have done this without you!

Dedication

To the scientist of tomorrow

Chapter 1

Introduction

One of the marvelous abilities of the human brain is embodying the vast amount of learning throughout a person's life. What facilitates this learning is the concept of *similarity*. Similarity is present on many levels in the human brain. On the cell level, learning happens by neurons building connections to other neurons that activate in a similar pattern. The representations of similar objects and concepts are formed in the brain in close-by physical locations [Connolly et al., 2012]. We also see similarity in higher-order learning - which occurs by imitating, comparing and predicting the implicit sensory patterns and explicit examples of behaviour [Hebb, 1949]. Learning would not be possible without the notion of similarity.

Humans have become such experts at learning from similarity that modern technological systems still struggle to match human level of performance, especially in messy and time-dependent real-life situations. This time dependency is of crucial importance to the process of learning as the desired outcome is the ability to note the change from the past, form predictions for the future, and refine the knowledge over time and practice. This way the actual process of learning can only exist in the context of time, deeply ingrained and integrated with this ordered one-way dimension. Time dependency, unfortunately, introduces a lot of complexity into the data and makes analyzing *time series* (the data gathered over time) one of the top 10 most challenging problems of data mining [Längkvist et al., 2014]).

Time series are one of the most common datatypes that are present in virtually every social and science domain [Palpanas and Beckmann, 2016]. One can find time series collected in finance, environmental studies, biology, astronomy, engineering, neuroscience, and music. Time series are used in stock prediction, video, speech and music recognition, motion capture, analyzing odours, and physiological research [Längkvist et al., 2014]. The goals of the analysis of time series could be finding similar or abnormal patterns, timing the length of certain events, finding times of data variability or identifying the periodicity of patterns. For sets of time series, the goals become search amongst the data set, correlation calculations, network discovery, clustering, anomaly detection, and causal modelling [Palpanas and Beckmann, 2016]. Similarity search in particular is an expensive operation that if performed fast enough can help speed up the other analysis operations, as it is often a key step for them [Palpanas and Beckmann, 2016].

1.1 Challenges of Time Series

The vast size of the time series data sets — the data sets are often in the order of hundreds to billions of sequences [Chávez et al., 2001] — introduces a challenge for similarity search. The main bottleneck of execution time becomes the organization of the data to efficiently query for the requested item [Chávez et al., 2001]. Finding closest matches in a data set of large size can become problematic in terms of time complexity with its $O(n^2)$ time, as most approaches need to compare all the pairs of data items. The actual distance or similarity calculation is at least $O(d)$ in number of the dimensions of the data. This causes current applications in many areas to be unusable for real time systems and creates a need for systems that provide at least approximate results but faster. Preferably these systems will also be scalable and flexible to work on full sequences and subsequences [Palpanas and Beckmann, 2016]. One of the proposed strategies for decreasing the time could be focusing on reducing the number of comparisons overall [Yu et al., 2019].

Although it may seem insignificant to further increase the speed and resource usage in certain modern applications like movie recommender systems that are non-essential and already perform relatively fast for their use, there exist systems that work in real-time for industry quality control during production [Kammerer et al., 2019] or flight control. These can benefit from faster responses to arising problems, which can in turn result in great financial and physical cost reduction and even, in certain situations, prevent loss of human life. Time series are vital to monitoring complex systems and in particular predicting failure, for example, through maximum lifetime analysis and remaining useful life prediction for parts and subsystems of a complex system (e.g. airplane, nuclear plant, factory floor). The benefit of such analysis is to use the knowledge to exchange parts before they cause problems and become too costly to rectify the consequences. It is thus vital in many contexts to have robust and scalable solutions that come up with similarity results in real time from real noisy time-dependent data.

Other challenges when working on time series are the real-life nature and time-dependency of the data [Längkvist et al., 2014]. Real data is often high-dimensional, and solutions that perform dimensionality reduction must be wary of data loss. Coming from real life systems also means that it is challenging to have a model; moreover, it is often unclear whether the sensors used are correct and whether these sensors are sufficient to see the desired properties, as sensors can provide only a limited and indirect view of the state of the system. The time dependency increases the complexity as the patterns of interest might appear at different points of time, and thus it is unclear how much data should be stored and whether the analysis is to be performed over the whole time series or parts of the time series; and if over parts – what length should the parts be.

Another aspect of the challenge is the noisy nature of real time series. Robust applications need a way to reliably distinguish between fluctuations in data that are due to noise or are due to real patterns. Most theoretical approaches and computer science solutions focus on data that is already cleaned and are thus not entirely representative of results that could be obtained on real data [Palpanas and Beckmann, 2016] in real-time. Usually the way to deal with noise is to use dimensionality-reduction techniques – for example

using prior domain knowledge to choose a representation. This could focus on the whole sequence or segmenting out the main temporal patterns like motifs (repeating patterns), shapelets (segments characterizing source) and discords (unique anomalies) [Längkvist et al., 2014, Palpanas and Beckmann, 2016]. Working with these representations could also open up avenues for solutions that can provide fast scalable similarity search for other types of data that could be turned into time series like categorical data or text [Lee and Lee, 2012].

It has also been shown in the literature that there is no "one size fits all" similarity measure for all time series data sets and so it is likely that every application will require an area-specific investigation to discover which measures are the best for it [Paparrizos et al., 2020]. There is thus a gap in the available applications for fast, scalable similarity search solutions on streaming large and noisy sets of real-life time series, with investigation into domain-specific and general similarity measures.

1.2 Locality Sensitive Hashing: Potential Solution

One of the approaches to similarity search that has potential to deal with this gap for fast, scalable similarity search is *locality sensitive hashing* (LSH) – a solution to the "curse of dimensionality" for approximate results [Chávez et al., 2001]. There exist many versions of LSH, with diverse applications, multiple distance metrics, and theoretical guarantees, but they all improve similarity tasks by using hashing for quick look up for pre-computed candidate similar items. This allows to trade off the accuracy for faster performance [Jafari et al., 2021]. Unlike regular hashing, overlaps in the hashing bin are maximized, albeit only for items that are deemed to be similar, which is a direct consequence of the hashing technique used.

Multiple similarity definitions have been used with LSH that make it suitable to a plethora of domains, based on Hamming, Minkowski, angular, or Jaccard distances [Jafari et al., 2021]. A survey on LSH by Jafari et al. [2021] found more than a thousand application papers that utilize LSH, including papers in audio, image and video processing, security and privacy, blockchains, data mining, text and document analysis, biology, geology, graph theory [Zhang et al., 2015], machine learning, healthcare, duplicate detection for plagiarism, networks, software testing, ontology matching, social media and community applications, and robotics [Tanaka and Kondo, 2008].

Small index size, fast index maintenance, fast query performance, and theoretical guarantees on query accuracy make LSH an important technique for solving the approximate nearest neighbours problem [Jafari et al., 2021]. LSH is easily scalable across data, as the hash functions do not require change during runtime when adding new data items [Jafari et al., 2021]. LSH methods are also excellent candidates for parallelization, as each data item can be processed independently given the hash functions, and then in the table each bin is independent – and inside of the bins the pairwise distance metrics can also be calculated in parallel as needed. The "parallelizable nature" of LSH is widely recognized by researchers and one of the possible benefits is the opportunity to perform similarity search even faster [Alcantara et al., 2009].

Modern developments in hardware allow us to harvest even more resources out of available computational systems. One such development is the availability of multiple cores in a single machine and connecting multiple machines over networks – leading to applications relying on parallel computation for improvement of execution time. Parallelizing solutions that already attempt to increase the speed of similarity search by minimizing the number of similarity calculations like LSH, is a way to possibly achieve an empirical speed up "for free".

Previous works have introduced some parallel implementations for LSH. An approach was introduced by Sundaram et al. [2013] and has shown an 8 times improvement in query speed and 4 times improvement for table construction, over a non-parallel LSH implementation, however it has used 780 hash tables to reach a satisfiable accuracy result, leaving significant room for improvement.

1.3 Thesis Goals

LSH has been widely and successfully used for similarity tasks, utilizing its randomness and dimensionality reduction through hashing indexes, including real-time systems [Aydar and Ayvaz, 2018, 2019, Aytekin and Aytekin, 2019, Durmaz and Bilge, 2019, Zamora et al., 2016]. *The area of interest of our work is using parallel LSH on streaming time series data.* Focusing on streaming applications allows us to extend any benefit found to non-streaming applications as well. In instances where we have the data available upfront we can always use it as if it was streamed. Even applications that are not time sensitive may benefit from being solved faster. On the other hand, extending non-streaming solutions to applications that require fast decision making on partially available data such as real-time quality control for systems is non-trivial and thus we focus on streaming applications directly as it is more interesting and has potential for wider benefit.

Previous advances of using LSH on time series data include work by Rong et al. [2018] where LSH and data-driven techniques allowed the earthquake prediction system to efficiently use time series for a period of 10 years, a vast improvement over the 3 months windows that were used before. We see great advantages in using LSH for time series data in the physiological domain. LSH has improved the time for diagnosing heart attacks, labelling fMRI brain scans and identifying markers of ADHD [Hu et al., 2016, Kim et al., 2016, Liu et al., 2019]. A more novel application of LSH on time series is described by Safavi et al. [2019] in their work on brain network discovery. They were able, by use of LSH, to form a network of brain regions showcasing similar activation patterns, faster than other computational approaches. Network discovery by LSH is non-specific to the data used and can be extended to other domains with time series – like stock market prediction.

A related problem to the network discovery is the correlation set discovery problem – where one attempts to find all the data points that are correlated or similar enough in some measure. The detailed definitions and connections between these problems are provided later on in Section 2.3. Parallel LSH can

also be used successfully to solve this problem. We find support in literature that the correlation set problem solution times are benefitted by parallel implementations [Amagata and Hara, 2019] and we can apply correlation-based LSH methods successfully to time series data sets [Yu et al., 2019]. Since the solutions for these problems can easily be reformulated for the specific application, we will focus on the network discovery aspect in this thesis.

The network discovery problem is essentially a visualization and interpretation method of the discovered "similarity connections" between data items by LSH – we can build a graph where each edge is a similarity measure value between two data items that were found similar by our methods. To have more interpretable and realistic results we will draw from data sets with time series that are often compared by correlation-based methods to find connections. *Our experiments will use airflight sensor data that was collected over real flights*. Airline data is a perfect candidate for streaming applications in online quality assurance systems and the *data set has known anomalies to evaluate our similarity methods*. There is a need for robust and fast solutions to working with time series in this area to enable the optimal management of maintenance processes, as projected by Dr. Hohy Hong in a recent presentation at a Workshop on Time Series [Palpanas and Beckmann, 2016]. *Finally, we focus on network discovery* as it gives us the flexibility to include many parts of the similarity results – from groups of nearest neighbours and correlation sets, to the actual similarity between items.

One last example of interest in network discovery research is brain network discovery – the "connectome" project, where each physical region of the brain has its activity recorded in a time series fashion and two areas that have similar activation patterns over time are deemed to be physically or functionally connected. A recent notable LSH technique for this type of network discovery, is the approximate binary correlation LSH (ABC-LSH) [Safavi et al., 2017, 2019]. The authors design the hash functions to focus on whole pre-processed sequences from the time series, with the distance metric increasing in geometrical progression with each consecutive match between the sequences. The pre-processing that ABC uses is simplistic – as it is a binary encoding of each time series point as being above or below the series mean. Another method we investigate has a larger symbolic alphabet to encode the signal as if every time series can be represented as a normalized Gaussian distribution – symbolic aggregate approximation (SAX) [Lin et al., 2003]. The last method that we explore is the "sketch, shingle and hash" (SSH) that utilizes random projections applied to the subsequences of the time series to detect patterns and uses set similarities for comparisons [Luo and Shrivastava, 2016]. Moreover, there is the opportunity to explore how the hash function families are structured – using AND and OR families like in ABC-LSH [Safavi et al., 2017, 2019]. A bag-of-words procedure could also be used for binarizing non-time-dependent data like text for usage in this framework as if it was a time series, making these methods suitable for a wider range of data domains [Leskovec et al., 2014] and this is a part of our implementation of minhash, that we use to test our proposed framework.

Our thesis implements the four LSH approaches mentioned above in a parallel message passing framework. We first design a system for minhash LSH technique on text data, and then refine the design to implement ABC, SSH, SAX for streaming airflight data. The experiments will tune parameters, discuss metric properties, compare the approaches on execution and communication complexities, similarity, and anomaly detection results. We believe all three approaches will indicate further directions for research and show the multifaceted nature of solving the problems with similarity search and related problems with streaming time series. We expect all three approaches to reach a reasonable solution and results but at the same time highlight and discuss the trade-offs that some of the approaches require. We hope to use our experimental data to draw conclusions about communication and computation complexity, execution time, anomaly detection, and the number of distance calculations – which is a measure of the sparsity of the discovered network, as we discuss the comparisons over methods and over parameter ranges. This thesis will thus present

- a parallel message-passing system
- with four implementations of LSH
- for providing similarity network discovery and anomaly detection results
- on real aircraft flight sensor data set in a streaming fashion
- with a discussion of parameter tuning and performance.

Chapter 2 will outline the literature related to the background and context of our research with details on the theory of LSH, the tasks our research attempts to solve and the four LSH methods we have chosen to investigate. Chapter 3 will present the implementation details for our experiments and we will discuss the evaluations and results in Chapter 4, with the concluding remarks in Chapter 5.

Chapter 2

Background

In this chapter we delve into the theory of important concepts for this thesis. We start in Section 2.1 with defining time series and discussing the concept of similarity. Section 2.2 follows with explanation of the locality sensitive hashing technique and the specifics of the approaches that we implement in Chapter 3. The last Section 2.3 will provide context for the possible applications of LSH on time series in general and for our work in particular by providing references to previous works of interest and the definitions of some problems that our work is tackling.

2.1 Similarity in Time Series

This thesis focuses on a fast way to perform similarity operations on a datatype known as time series. Formally, a time series can be defined as follows [Esling and Agon, 2012, 2013]:

Definition 2.1.1 (Time Series) *A univariate time series T is an ordered sequence of n real-valued variables $T = (t_1, \ldots t_n), t_i \in R$.*

In many situations, the entire time series is not available or is impossible to store completely. Real world sensors create a continuous stream of data points and can have unknown or unbound ranges of recorded time. Let's consider an example of time series recorded from airline data. Currently, commercial flights can take up to over 18 hours of flight time (the longest flight is Singapore to New York with 18 hours 40 minutes of flight time). Several hundred sensors on board an airplane can be streaming data at a sampling rate of up to 8 data points per second. That is over $500,000$ data points for a single sensor. Because of the magnitude of the resulting data it will be necessary to divide the series into appropriately sized *subsequences* to apply the algorithms.

Subsequences can be defined as a part taken out of the time series, but with values still kept in the original order of monotonically increasing time. It can be over a specified length of time, over a certain event with a start and an end point, or a certain number of data points or time steps. The important notion is that the data points are kept in sequence with time and usually represent equal time intervals, since data is collected at a certain sampling rate for a specific series. The concept of a "sliding window" can be used to create the set of all subsequences. The length of the window is specified – this is the size of the subsequence – and how much to move this window to create the next one. This can create subsequences that either have a certain overlap, are next to each other or are separated by some time with no overlap at all. Alternatively,

we can specify some number of starting points for the windows we might need.

Domain-specific considerations can also dictate the best approach to create subsequences. For example, in airline data the flight is divided into phases – taxi, take off, ascend, cruise, descend. Behaviour of the plane and important variables will change depending on the phase, influencing the model and properties of the data and the type of information one might want to extract. Another consideration, especially for comparing multiple time series from different sources, is the difference in sampling rates. Specific preprocessing like smoothing, re-sampling procedures, interpolating and extrapolating may be needed to obtain sets of time series based on the same time scale.

Storing this type of data in a regular relational database with time as an extra dimension is not ideal as this causes difficulty in querying a certain time-period and creating a subsequence. Machine learning approaches like neural networks that work on multi-dimensional data are typically not sensitive enough to the time-dependency of this data – even though you can treat a time series like a vector ordered by the time parameter, the methods are not designed to treat the ordering as important and monotonically increasing. Some artificial neural networks that can tackle time-dependent data have a very limited memory [Längkvist et al., 2014].

One solution is to mine the time series for certain features and perform operations on the results. The most common features for time series are the three primitive temporal patterns – motifs (repeating patterns), shapelets (representative subsequences for source or type of series) and discords (most unique subsequences in a series) [Huang et al., 2015, Palpanas and Beckmann, 2016, Yeh et al., 2018]. However, existing methods for finding these features are still not fast enough to be used as part of a solution for online streaming data. This is due to the complexity of this problem, which is also exacerbated when there are hundreds, or potentially thousands, of time series of unbound lengths [Yeh et al., 2018].

When dealing with comparisons another challenge appears – which time series do we want to consider similar? Humans are able to infer similarity based on their learnt experience from the world and are able to abstract from many mathematical details that are not important for the overall shape of the time series, the task with which algorithms struggle [Esling and Agon, 2012, 2013]. An operational definition of similarity is thus required when working on algorithms that use this concept. Esling and Agon outline five aspects to a good similarity measure [Esling and Agon, 2012, 2013]:

- Providing recognition of perceptually similar objects, even if not mathematically identical.
- Being consistent with human intuition.
- Emphasizing the most salient features on both *local* and *global* scales.
- Allowing to identify or distinguish arbitrary objects.

- Abstracting from distortions and being invariant to a set of transformations like amplitude shifts, amplification, noise, and time dilation.

A common way to define similarity is to define it in conjunction with its "inverse" – *distance*. Let's look at an example with binary strings of the same length $x = 00100$ and $y = 11101$. A very simple *similarity* measure would be to count the number of matching bits – i.e. 2. The value of the length of the string corresponds to the maximum possible number of matching bits and therefore also to the maximum similarity possible for two strings of this length. Two out of possible five is consistent with our intuition of these strings not being very similar to each other. Then the *distance* between these strings is the number of mismatching bits – i.e. 3. It is the same to say that the distance is the difference between the length of the string 5 and the number of matching bits – the actual similarity of these strings (2). This is in fact the definition of Hamming distance. In this way, the concepts of similarity and distance are connected by being opposite to each other.

A special subclass of distance measures, that has the nice properties that allow, for instance, the usage of a hash table, is the class of distance *metrics*. A metric is what imposes a structure on the set of data and the ability to order items in this set – this is the basis for the notion of distance metric and metric space. *In the similarity-distance relation, this means that the more similar two items are, the closer together they are in the space defined by the distance metric, and vice versa for two dissimilar items*. Thus, the usage of a metric becomes important for the operationalization of the similarity definition. Some of the properties of a distance metric are also key in the type of fast neighbour retrieval we explore in this thesis, with the specific benefits for our research described further in Section 2.2. Mathematically, a distance metric is defined as follows [Rajaraman and Ullman, 2011]:

Definition 2.1.2 (Distance Metric) *A distance metric $d(x,y)$ is a distance measure that satisfies the following four axioms:*

1. *Identity:* $d(x,y) = 0$ *iff* $x = y$
2. *Non-negativity:* $d(x,y) \geq 0$
3. *Symmetry:* $d(x,y) = d(y,x)$
4. *Triangle inequality:* $d(x,y) \leq d(x,z) + d(z,y)$

The most familiar distance metric is the Euclidean distance (the L_2 Norm), which defines the Euclidean metric space. Hamming distance described above is also a distance metric, used in information retrieval and text mining. Another such popular metric is the Jaccard distance defined along with the Jaccard similarity for sets:

Definition 2.1.3 (Jaccard Set Similarity) *For two sets X and Y, both of size D:*

$$J(x,y) = \frac{\sum_{i=1}^{D} \min(x_i, y_i)}{\sum_{i=1}^{D} \max(x_i, y_i)} = \frac{|X \cap Y|}{|X \cup Y|}$$

The Jaccard distance is defined as the complement of the Jaccard set similarity $d(x,y) = 1 - J(x,y)$. For our two strings of length 5, $x = 00100$ and $y = 11101$, the Jaccard set similarity is $1/4$. We treat the ones in the binary string as indication of an item present in the set and zero as absent – and the comparison is done bit-wise. Since Jaccard similarity is a fraction, its maximum is 1, and thus the Jaccard distance for these two strings is $3/4$. We can see that this corresponds to our human notion of these two strings being more different than similar.

Jaccard distance is a metric. The first three properties are trivial to show, with the triangle inequality requiring a more involved proof.

1. **Identity**

   (a) Assume $d(x,y) = 0$. Since the distance is defined as $1 - J(x,y)$, $J(x,y) = 1$. This can only happen when $|X \cap Y| = |X \cup Y|$, which can only happen when $X = Y$.

   (b) Assume $X = Y$. Then, $|X \cap Y| = |X \cup Y|$ and $J(x,y) = 1$. Since distance is defined as $1 - J(x,y) = 1 - 1 = 0$.

2. **Non-negativity**

   Since J(x,y) is a ratio between the sizes of the intersection and union of two sets, $0 \leq J(x,y) \leq 1$ and thus the distance $0 \leq d(x,y) \leq 1$.

3. **Symmetry**

   The symmetry property follows from the symmetry of the set intersection and union operations.

4. **Triangle Inequality**

   Multiple proofs exist for the triangle inequality of the Jaccard distance [Gilbert, 1972, Kosub, 2019, Levandowsky and Winter, 1971], we will include the most straight forward set-based proof by Gilbert [1972].

   Suppose we have three sets S_1, S_2, and S_3, for which $\cup S_i = U$ and $\cap S_i = V$, and T_i is as shown in Figure 2.1 [Gilbert, 1972]. As given by Gilbert the triangle inequality follows from the inequality for $d(S_i, S_j)$ in 2.1.

$$\frac{|T_1| + |T_2| + |T_3|}{|U|} = 1 - \frac{|V|}{|U|} \geq d(S_i, S_j) \geq \frac{|T_i| + |T_j|}{|U|} \tag{2.1}$$

   The first part holds, as from the Venn diagram we can formulate the following:

$$\frac{|T_1| + |T_2| + |T_3|}{|U|} = \frac{|U| - |V|}{|U|} = \frac{|U|}{|U|} - \frac{|V|}{|U|} = 1 - \frac{|V|}{|U|}$$

   Substituting the definition of Jaccard distance and simplifying according to the Venn diagram:

$$d(S_i, S_j) = 1 - \frac{|S_i \cap S_j|}{|S_i \cup S_j|} = \frac{|S_i \cup S_j| - |S_i \cap S_j|}{|S_i \cup S_j|} = \frac{|T_i| + |T_j|}{|S_i \cup S_j|}$$

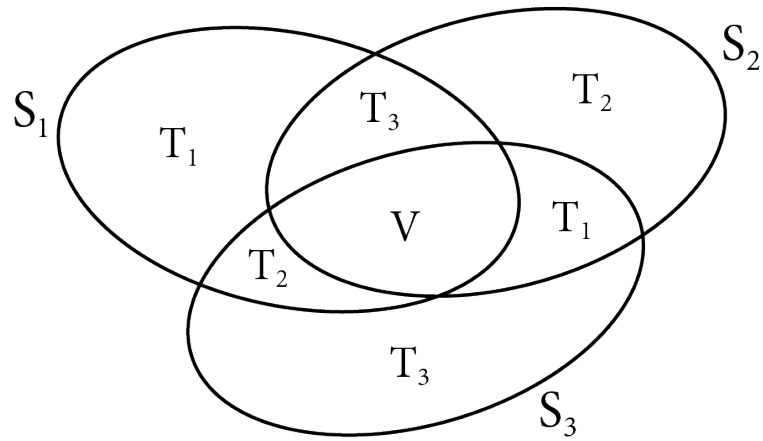

Figure 2.1: Venn Diagram for Jaccard Set Distance Triangle Inequality Proof [Gilbert, 1972]

The right-hand side of the inequality in 2.1

$$d(S_i, S_j) = \frac{|T_i| + |T_j|}{|S_i \cup S_j|} \geq \frac{|T_i| + |T_j|}{|U|} \tag{2.2}$$

follows because $|U|$ is larger than $|S_i \cup S_j|$.

The left-hand size of the inequality in 2.1

$$\frac{|T_1| + |T_2| + |T_3|}{|U|} \geq \frac{|T_i| + |T_j|}{|S_i \cup S_j|} = d(S_i, S_j) \tag{2.3}$$

is a result of carefully considering the regions from the Venn diagram. The numerators of the two fractions differ by $|T_k|$ and the denominators by $|T_k \cap S_k| < |T_k|$, and since the top of the fraction decreases more than the bottom:

$$\frac{|T_1| + |T_2| + |T_3|}{|U|} - \frac{|T_i| + |T_j|}{|S_i \cup S_j|} \geq 0$$

The inequality in 2.1 is sufficient to prove the triangle inequality property $d(S_i, S_j) \leq d(S_i, S_k) + d(S_k, S_j)$. Consider, without loss of generality, the distances $d(S_1, S_2)$, $d(S_2, S_3)$, and $d(S_1, S_3)$.

$$\begin{aligned}
d(S_1, S_3) + d(S_2, S_3) &\geq \frac{|T_1| + |T_3|}{|U|} + \frac{|T_2| + |T_3|}{|U|} \quad \text{by applying 2.2 twice} \\
&\geq \frac{|T_1| + |T_2| + |T_3|}{|U|} + \frac{|T_3|}{|U|} \\
&\geq \frac{|T_1| + |T_2| + |T_3|}{|U|} \\
&\geq d(S_1, S_2) \quad \text{by 2.3}
\end{aligned}$$

We have shown that Jaccard distance is a metric, satisfying the four properties. ■

Various distance and similarity measures exist for time series. A meta-analysis of multiple studies and distance measures shows variance in results and performance using the same techniques, processing procedures and similarity measures on time series from different domains and for different purposes [Paparrizos et al., 2020]. This highlights the opportunity for further exploration in this area. Specifically, one promising area for research in time series is a well-known similarity search technique, namely locality sensitive hashing.

2.2 Locality Sensitive Hashing

Locality sensitive hashing (LSH) was first introduced by Indyk and Motwani [1998] as a method for achieving fast nearest neighbour search. The main idea is to organize the data in a way so that similar items are placed closer together. This produces a form of clustering of the data items by similarity. The key idea behind LSH is that the processing is done using hash functions, such that the probability of collisions (being placed in the same bucket of the hash table) of things close in the metric space is higher than for things that are far apart. This is unlike regular use of hashing functions, where collisions are generally avoided as much as possible. These buckets can be treated as clusters of data and can be used to narrow down the search for the nearest neighbours to only one bucket per hash table, eliminating the $O(n^2)$ distance computations. LSH can be implemented in a Map-Reduce like framework, with the Map step applying the ith hash function to the jth data item, and the Reduction step forming the buckets according to the mapping results. The basic idea of LSH as described by Indyk and Motwani [1998]:

> "The idea of this approach is to hash the points in a way that the probability of collision is much higher for points which are close (with the distance r) to each other than for those which are far apart (with distance at least cr, for some constant c). Given such hash functions, one can retrieve near neighbors by hashing the query point and retrieving elements stored in buckets containing that point."

This is done by generating a number of hash functions for a given similarity measure. A dissimilarity measure like a distance metric can also be used. These hash functions are used to place the data points during the processing into buckets with the same hash values. The query data item is also hashed using the same functions. LSH is guaranteed to have placed the query's ε-nearest neighbour (Definition 2.3.2), if it exists, to the same bucket with a constant probability [Indyk and Motwani, 1998]. Only those data points in the bucket with the query are then considered for distance comparison.

Definition 2.2.1 (Locality Sensitive Hashing [Christiani, 2017]) *Let H be a distribution over functions $h : X -> R$ (X is the space of original data, R is the space of the hash values). We say that H is (d_1, d_2, p_1, p_2)-sensitive family of functions if for $x, y \in X$ and h from H we have that:*

$$\text{If } d(x,y) \leq d_1 \text{ then } Pr[h(x) = h(y)] \geq p_1$$

$$\text{If } d(x,y) \geq d_2 \text{ then } Pr[h(x) = h(y)] \leq p_2$$

This implies that when two points are within some small distance d_1 of each other, there is a probability of at least p_1 for the two points to be hashed to the same bucket. The second formula ensures that when two points are a large distance d_2 apart there is only a small probability, p_2, of these points being hashed to the same bucket. This means that in order for this locality-sensitive family to be useful, it should satisfy the following inequalities [Indyk and Motwani [1998]]:

- $p_1 > p_2$ and $d_1 < d_2$ when d is a distance measure
- and $p_1 > p_2$ and $d_1 > d_2$ when d is a similarity measure

Similarity S is defined as the opposite measure to the distance d – the more similar the points, the smaller the distance, the larger the similarity. And vice versa – the less similar the points, the greater the distance between them, and the lesser the similarity. Given LSH's reliance on distance, this highlights the importance and benefits of using a distance metric. Albeit the fact that there are variants of LSH using a non-metric measure, however, usually LSH is metric-based [Mu and Yan, 2010].

The non-trivial part of the distance metric definition relevant to hashing is the triangle inequality. For example, correlation, which is not a metric but is frequently used for comparing time series, does not allow us to form a hash table with the desired properties. Contrary to a common misconception[1], Pearson's correlation is not transitive [Elisa et al., 2009], which is required for the triangle inequality to hold. Moore [2006] shows an example of the non-transitive nature of correlation occurring with film ratings, where length of the movie is positively correlated with both rating and year of release, however the rating and year are negatively correlated. There are also many examples of false correlations where two events appear correlated but are in fact completely un-related, spurious or coincidental correlations.

For a more general example, let's say we have two highly-correlated items A and B, that we would want to place in the same bucket. We also find item C that is highly correlated with B but not with A. The high correlation would incline us to place the item in the same bucket with B, however, the lack of correlation with A breaks the notion of locality by introducing an item in the same bucket with A that is not a candidate for its nearest neighbour. Placements like this would lead to an uncontrollable false positive rate, unlike when using LSH. On the other hand, placing B in one bucket with A, and then again into another bucket with C is also not possible – B has only one hash value and should appear only once in the table. This thus does not allow us to use correlation directly with LSH, however, it is possible to come up with distance metrics that are correlation-based, like ABC, to share the other beneficial properties like being close to the human definition of similarity.

However, if the LSH is based on a metric, we can be sure that all of the pairs of items that we can find in the same bucket will be close enough by our similarity measure, subject to a small probability that is controllable. Thus, the usage of a distance metric allows us to form hash tables with non-overlapping buckets with the desired properties. *We can treat an LSH hash table as a disjoint partition of the data space into neighbourhoods – with all the candidate nearest neighbours (as defined by the distance metric) in the same part (bucket) of it*. This is only possible if the table is formed with a distance measure that is transitive, which can be ensured by the triangle inequality.

[1] that correlation is causation

The control over the collision probability comes from the formation of the hash functions. For a specified number $k = |H|$, we can create a hash function $g(p) = (h_1(p), \ldots h_f(p))$, where each $h_i \in H$. This means that we can choose $f \leq k$ hash functions from the defined family to use to form f buckets for our processing. When constructing an LSH family we can control the probabilities of false negatives and false positives, which positively correlate with the values of p_1 and p_2 respectively [Safavi et al., 2017]. To control these, a new "amplified" family can be defined and used [Rajaraman and Ullman, 2011, Safavi et al., 2019]:

Definition 2.2.2 (AND and OR Group Constructions) *Given a (d_1, d_2, p_1, p_2)-sensitive hash function H:*

The AND construction creates a new hash function $g(p) = {h_i}^r$ as a logical AND of r members of H, chosen uniformly at random without replacement from [1,k]. And thus $g(x) = g(y)$ iff $h_i(x) = h_i(y)$ for all $i \in [1, r]$.

The OR construction creates a new hash function of similar structure but using the logical OR, or equivalently, $g(x) = g(y)$ iff $h_i(x) = h_i(y)$ for any $i \in [1, r]$.

The AND construction then provides a single hash table, so each data point has a single hash signature as a concatenation of the r hash values. The OR construction can be used to build a specified number of hash tables b, with each data point hashed independently b times. Adjustment of these r and b parameters controls the rates of false positives and false negatives [Yu et al., 2019]. The higher the r, the lower the probability of false positives; the higher the b, the lower the probability of false negatives.

In general, LSH allows to achieve sublinear computation time in search tasks [Indyk and Motwani, 1998]. The bottleneck of the computation then becomes the hash table creation [Christiani, 2017]. This is a challenge when working with time series data sets of large sizes, as each one of the items needs to be hashed. For example, as mentioned in Chapter 1, airline data can have hundreds of sensors streaming data resulting in terabytes of data to be mined even for a single flight. LSH also has a drawback of requiring multiple hash tables to produce good accuracy [Wu et al., 2020]. This "number of items times number of hash tables" computation is a great challenge time-wise in similarity tasks. This can be solved by utilizing the "parallelizable" nature of LSH [Alcantara et al., 2009]. The inherent parallelism in LSH is one of its advantages and a major reason for investigating techniques that lend themselves to this approach.

2.2.1 Parallel LSH

A parallel implementation of the LSH techniques is made possible by the independent nature of the hashing procedures. Each data item is hashed independently of every other data item and each specific hash function works independently from any other. These nice features inherent to LSH can be used to distribute the workload of producing the hash values as well as distributing the storage of the buckets across processing cores or machines.

For instance, when working with airline data, the time series for each sensor can be hashed independently in parallel. Each sensor's data can also be parallelized over the necessary hash functions and each

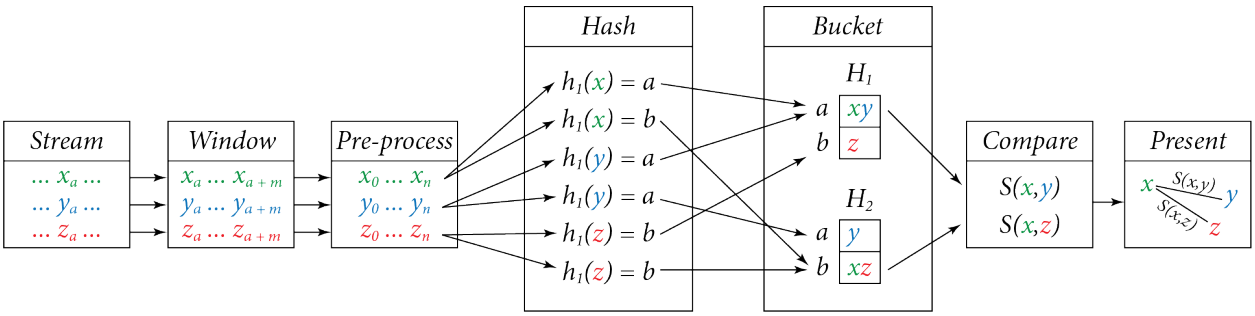

Figure 2.2: Parallel LSH Workflow

formed hash table and bucket can be stored separately. Therefore, the sequential steps in the workflow are the similarity calculations for pairs in the same bucket. The formation of the AND constructions and reporting of results in a network representation are also sequential but do not depend on having the previous steps fully completed for all data items; as items are hashed and new pairs are found, the results can be updated.

Previous parallel LSH implementations exist and show improvements over non-parallel solutions. Tripathi et al. [2019] use LSH for image super resolution with parallelization in Hadoop with search over each table in parallel. Pilz et al. [2020] calculate Hamming distance between LSH-generated signatures on DNA sequences in parallel on an FPGA. The paper by Teffer et al. [2019] approaches the problem of clustering non-structured data such as streaming text in a distributed environment. The solution is proposed that uses an adaptive and hierarchical LSH-like hashing, in a Map-Reduce-like framework.

In this thesis we describe a parallel framework for performing similarity tasks with LSH. We limit our exploration to four different LSH methods, which allow to reduce dimensionality enough for clearing out noise without explicit feature engineering but not as much as to lose the time-dependency and the details of fluctuations in the data.

Our general workflow to implementing LSH in a streaming parallel system is shown in Figure 2.2. The hash table creation is parallelized over items and hash functions, which are then pooled to form the hash tables. As an item has its hash value calculated for a table, it is added to the corresponding bucket and all previously-added items form the candidate neighbour pairs with it. These intra-bucket pairs of items then have the similarity calculated between them. In the last step, everything is combined in the resulting network.

One of the simplest hashing procedures with a popular distance metric (Jaccard) is minhash. It has benefits of being simple to implement and powerful enough to produce robust and theoretically-supported results. This makes a great candidate for validation of a parallel LSH framework, which we present in Section 3.1.

Since our main focus is streaming time series, for our main experiments we want to use techniques suitable for subsequences. One such novel technique is the ABC-LSH [Safavi et al., 2017, 2019]. Its sim-

Stage	**Minhash**	**ABC**	**SAX**	**SSH**
Pre-processing	Bag-of-words	Binarize	Normalize, PAA, Discretize	Sketch and Shingle
Hashing	Minhash	Subsequences	Subsequences	Weighted Minwise
Similarity	Jaccard	ABC	MINDIST	Jaccard
Analysis	*Pilot project*	Time, Computations, Network Discovery, Anomaly		

Table 2.1: Summary of Implemented Techniques

ilarity measure is correlation-based, but is also a valid distance metric. Its novelty allows us to provide new comparisons of this technique to more established ones. An older technique for comparing sequences that we chose is SAX [Lin et al., 2003]. The hashing procedure is similar to that of the ABC, however it has a variable number of symbols for the encoding, while ABC focuses on binarized time series. The last technique in our exploration is SSH [Luo and Shrivastava, 2016]. It has a fundamentally different way of pre-processing the original time series by extracting similar patterns at different time points, which opens up room for interesting results compared to the more simpler methods. It is also reliant on the weighted minhash procedure, which is a step up in complexity of its implementation from the minhash, that we use for the original pilot implementation. These three techniques allow us to investigate a few different approaches to comparing time series. Since all of these focus on subsequences, it makes them suitable to use on streaming data. The simplicity and unsupervised nature of pre-processing and hash table creation for these techniques, along with the general LSH advantages that they take on, are the key ingredients to a fast scalable parallel solution that can work for real-life streaming time series data.

In the following sections we provide details on the four LSH approaches under our exploration – minhash, ABC, SAX and SSH. Each method can be described in three main stages as summarized in Table 2.1: 1) the data pre-processing necessary, 2) hash values and hash signature formation and 3) distance measure (or similarity) calculation. For all of the approaches, we build a network of related time series. Here, by "related" we mean similar in the sense of our distance metric as defined by the specific LSH of the approach. The network has edges connecting the time series that are binned together in at least one hash table. This is flexible as described in the usage of OR and AND families construction (Definition 2.2.2) – the network can be specified to only include edges with a certain threshold of number of clashes between two items. The weights of the edges are calculated by the similarity measure defined by the approach. The last step is the analysis of the resulting network.

2.2.2 Minhash

Our exploration of LSH began with minhash. Minhash is an LSH approach for binarized sets of data that utilizes the Jaccard similarity measure and the corresponding distance metric. A weighted version of minhash is used in Section 2.2.5 in one of the approaches that we investigate in the main experiments of the thesis. We will begin with introducing the regular minhash procedure using Jaccard similarity in the context of applying it to text data.

Each *data item* for minhash is pre-processed into a sequence of the form $t = t_1 t_2 ... t_n$ that encodes the text excerpt, letter sequences or words, into a binarized *set*, where each $t_i = 1$ when excerpt i is present in the text and $t_i = 0$, if it is not. We assume the excerpts in this case are words. The collection of all appearing words in the full data set are collected in a "dictionary" and their order is kept the same across items; this is the bag-of-words procedure for text data. Each *hash function* for minhash is a permutation (without replacement) of the words' indexes – with the permutation's length as a parameter: the permutation might be of a certain length or might be over all possible words.

A way to formulate the minhash procedure is to use a characteristic matrix where each column is a sample of text (in form of binarized set over dictionary) and each row is a feature (in our case a word) [Leskovec et al., 2014]. For example if we want to encode three text excerpts: S_1: "green flying pigeon", S_2: "blue flying parrot" and S_3: "blue swimming whale" on dictionary of words "blue", "green", "flying" and "swimming", in this order, we could form the following characteristic matrix (Table 2.2):

Feature Index	***Feature***	S_1	S_2	S_3
0	*"blue"*	0	1	1
1	*"green"*	1	0	0
2	*"flying"*	1	1	0
3	*"swimming"*	0	0	1

Table 2.2: Original Characteristic Minhash Matrix

The *hash function* is then an ordering or a permutation of the rows, for example 3102: "swimming", "green", "blue", "flying". The *hash value* for a column of a characteristic matrix is then the minimum row index that has a 1 in that column – hence the "min" (min-wise independent permutations) part of the technique's name. The resulting matrix with permuted rows is shown below in Table 2.3, with the 1's at the minimum index in bold. The hash values for this hash function are thus S_1: 1, S_2 : 2, S_3: 0.

Hash Value	***Feature Index***	S_1	S_2	S_3
0	3	0	0	**1**
1	1	**1**	0	0
2	0	0	**1**	1
3	2	1	1	0

Table 2.3: Permuted Characteristic Minhash Matrix

The *hash signature* for a set, corresponding to a document, is the concatenation of its hash values for all hash functions. For sets that share at least one hash value (this is an OR construction), the distance is calculated. Minhash can be used to approximate the Jaccard similarity of two sets as the fraction of matching hash values over the total number of hash functions (proof is provided in sections 3.3.3 and 3.3.5 in Leskovec et al. [2014]).

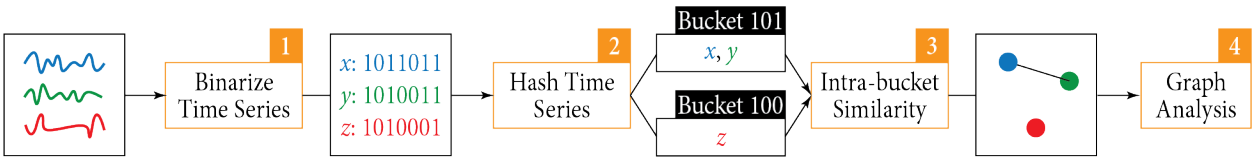

Figure 2.3: ABC-LSH Workflow [Safavi et al., 2019]

2.2.3 ABC

Works by Safavi et al. [2017, 2019] use LSH for network discovery with time series, specifically in fMRI brain activations. Previous approaches involved making a fully connected weighted graph between the nodes, where each node is a specific time series, generated from a small brain region. This is too slow for very large data sets. The authors design a new LSH-based approach since there are very few other papers reporting similar work. The proposed ABC-LSH uses correlation as a base for the similarity measure and this approach shows vast improvement in speed of the network discovery. An example of the full workflow for ABC is presented in Figure 2.3 [Safavi et al., 2017, 2019].

The data set of the time series is first normalized and then encoded as a binary code. Each time series is turned into a sequence of 1s and 0s – the 1s represent values that are above the mean of the time series and the 0s for values below the average. These codes are simple but are still representative of the fluctuations of the time series. For example, a time series that has real values "21 121 33 133 156 56" with the mean of 100, are represented in binary as "0 1 0 1 1 0".

The actual hash codes used for assigning time series to buckets are formed on the parameters that can be controlled. The parameters are the number of OR groups to consider and the length of subsequences. For each OR group a starting index for the subsequence is chosen. Time series whose binary codes match on the selected subsequences are bucketed together – this is intuitively similar to how the time series were fluctuating in the same way at the specified point of time and for the specified length of time.

The Approximate Binary Correlation (the ABC metric) is calculated for each pair of time series that was hashed to the same bucket. This is calculated from the sequence of matching bits between the two encoded time series, with each sequential matching bit receiving a heavier weight – the longer the matching subsequence, the more similar the time series are deemed to be and the more correlated their activations must be. The similarity calculation by ABC is described by the following formula:

$$s(x,y) = \sum_{i=1}^{p} \sum_{b=0}^{k_i} (1+\alpha)^b$$

Where x and y are the binarized time series of length n (or their corresponding subsequences of same length), and they have p matching subsequences of length k_i each – where the codes match exactly on each consecutive bit. α is a parameter that controls the emphasis on consecutiveness of the matches in the binary codes – the smaller it is, the closer this metric is to the inverse of Hamming distance, in which case the con-

secutiveness does not matter at all. The larger α is, the more weight we place on the bits that are matched consecutively.

The actual distance measure for the ABC similarity is the difference between maximum similarity for two time series of a given length and the actual similarity of the two time series – such that identical time series have a distance of 0. The maximum similarity $S(x,y)$ for two sequences of length n is calculated as follows:

$$S(x,y) = \sum_{i=0}^{n-1} (1+\alpha)^i$$

As shown by Safavi et al. [2017] the ABC similarity measure is a valid metric. We present the basic details of the proof here, with a brief sketch of the triangle inequality proof strategy.

1. **Identity**

   (a) Assume $d(x,y) = 0$. Since $d(x,y) = S(x,y) - s(x,y)$, then $S(x,y) = s(x,y)$;
   $(1+\alpha)^n - 1 = \sum_{i=1}^{p} (1+\alpha)^{k_i} - p$,
   which holds when $n = k_i$ and $p = 1$. Since k_i is the length of the matching run, if there are $p = 1$ of length n, then this implies that the two series are the same – there is only one matching run of length n.

   (b) Assume two series are the same. Then they have 1 matching run of length n: $p = 1$, $k_i = n$. Substituting these into the S(x,y) and s(x,y), we have $d(x,y) = S(x,y) - s(x,y) = 0$.

2. **Non-negativity**

   If the series match exactly, then $d(x,y) = 0$, as shown in 1. Otherwise the series match on the sum of the lengths of all matching runs – $\sum_{i=1}^{p} k_i$ elements, or do not match at all.

   (a) If the series do not match, then $d(x,y) = S(x,y) = \frac{(1+\alpha)^n - 1}{\alpha}$. Since all terms are ≥ 0, it follows that $d(x,y) \geq 0$.

   (b) If the series match on some elements, the maximum number of elements they can match on is $\sum_{i=1}^{p} k_i = n - p + 1$. The maximum matches provides the largest similarity, which is always smaller than the maximum similarity for the same n (as they don't match totally on all symbols), therefore $d(x,y)$ is always ≥ 0.

3. **Symmetry**

   The matching is done in order of elements which is the same across different series and the order of the series does not matter – so the measure is symmetric.

4. **Triangle Inequality**

   The proof is an induction on n – the length of the binary sequences compared.

   Base Case

   For the base case, all combinations of three binary symbols x, y, z (length of 1) are considered and it is trivial to show that the triangle inequality holds for any of them.

   Induction Step

   For the induction step, the setup is completed by comparing two sequences. Consider two sequences of length n with an already calculated similarity by ABC. How does appending a symbol at the end of the sequences change their similarity value? There are three cases to consider: 1) if the appended symbols create a new agreeing run between the sequences, 2) if they add onto an existing run and 3) if they are not the same. For all three cases, the distance change from being sequences of length n to being extended to length $n+1$ are derived.

   Enumerating all of the combinations of the three cases for three time series results in total of 27 combinations. However, of these, only 10 are possible. Some of the combinations are not feasible, as there are only two symbols that can be appended. Knowing whether the symbols match for two sequences determines the case with the third sequence. Furthermore, due to symmetry, 7 cases remain to be considered. By analyzing these 7 cases the authors show that for each one the triangle inequality holds.

   Therefore by induction it is proven that the ABC-LSH distance satisfies the triangle inequality and is a metric. For full details see the Appendix section in Safavi et al. [2019].

ABC [Safavi et al., 2019] is applied to data sets of fMRI brain scans and compared to Pearson correlation methods that are routinely used to identify networks from brain data. The networks from the ABC method are formed by representing the time series as nodes. Edges are added between each pair of time series that share a bucket in any of the OR subsequences or on all of them (the AND group), with the edge weighted by the ABC calculated similarity metric.

2.2.4 SAX

In their work, Lin et al. [2003] outline a more complex approach to encoding time series – symbolic aggregate approximation (SAX). This is a popular approach for utilizing discretized methods for real-valued data sets such as time series. This representation is shown to be bounded from below by the similarity metrics on the original real-valued time series, and using SAX allows researchers to benefit from using a symbolic approach without unduly introducing errors. The original paper experiments with the representation for usage in tasks of clustering, classification, querying by content and anomaly detection.

The SAX method requires the incoming data to be normalized before processing. The number of symbols to be used is pre-determined as a parameter. The approach and similarity calculations rely on the normalization of the time series and the assumption that a normal curve is an appropriate description of the distribution of the values of the time series. After normalization, the pre-processing procedure includes two stages – the piecewise aggregate approximation (PAA) that produces "words" and then the "discretization" step where each "word" is replaced by a symbol.

Normalization

The SAX method assumes normalized (standardized) input data with a mean of 0 and a standard deviation of 1. One way to do this is to have a known (or modelled) *mean* and a standard deviation *sd* for the real data distribution and calculate each new element as follows:

$$v_n = \frac{v - mean}{sd}$$

Piecewise Aggregate Approximation

Assuming that the incoming data is already normalized, the first step of SAX is to reduce dimensionality. The entire string is separated into non-overlapping "words" of a pre-defined length. Each "word" is then represented by the mean value of the data that was included in it.

Discretization

Since the data was originally normalized with a mean value of 0 and a standard deviation of 1, we can use the Gaussian curve to discretize the data by representing it as symbols. Once we have the data in a "words" string, we look up the breakpoints β_i in a statistical table like Table 2.4. The breakpoints divide the area under the Gaussian curve into equal-size intervals. For 3 intervals the breakpoints are -0.43 and 0.43. Thus, the values that are below -0.43 are encoded as the first symbol a, values between -0.43 and 0.43 as the second symbol b, and the values above the last breakpoint 0.43 as the third symbol c.

For example, a time series with real values of "-1 -1 0 0 1 1" (normalized with mean of 0) for encoding with 3 symbols and 3 "words" of length 2, will look like "-1 0 1" in its PAA representation. In the next step, this string will be symbolized as "a b c" according to the corresponding breakpoints.

	3	4	5	6	7	8	9	10
β_1	-0.43	-0.67	-0.84	-0.97	-1.07	-1.15	-1.22	-1.28
β_2	0.43	0	-0.25	-0.43	-0.57	-0.67	-0.76	-0.84
β_3		0.67	0.25	0	-0.18	-0.32	-0.43	-0.52
β_4			0.84	0.43	0.18	0	-0.14	-0.25
β_5				0.97	0.57	0.32	0.14	0
β_6					1.07	0.67	0.43	0.25
β_7						1.15	0.76	0.52
β_8							1.22	0.84
β_9								1.28

Table 2.4: SAX Breakpoint Values for 3 to 10 Symbols. Breakpoints β_i produce equal-sized areas under the Gaussian curve. The number of intervals is equal to the number of symbols, so that all values that fall into the same interval can be represented by a single symbol.

	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

Table 2.5: SAX Distances Table for 4 Symbols

Distance Calculation

The authors define a distance measure for the SAX representation based on the normal curve as well. For calculating distances between subsequences, we can pre-compute and store a look up table of distances between each pair of symbols, based on the normal curve, called $dist(r,c)$, shown in Table 2.5 where the r and c are the two symbols, represented by their index in the table. For given symbol indexes r, c, $cell(r,c)$ of the lookup table is calculated as follows:

$$cell_{r,c} = 0 \text{ , if } |r-c| \leq 1$$
$$cell_{r,c} = \beta_{(max(r,c)-1)} - \beta_{(min(r,c))} \text{ , otherwise.}$$

The minimum distance between the two SAX-encoded time series (or their corresponding "words" subsequences) is calculated by:

$$MINDIST(Q,C) = \sqrt{\frac{n}{w}} \times \sqrt{\sum_{i-1}^{w} (dist(q_i, c_i))}$$

Where Q and C are the two time series, n is the length of the original time series and w is the number of "words" generated by the PAA procedure. This distance can then be used as the edge weight. Albeit being an encoding method that was not designed with LSH in mind, this can be used in a very similar way to any other distance measure for LSH formulation and network discovery.

The measure presented is a lower bound of the Euclidean distance, which is a metric. However, the *MINDIST* is not a proper metric itself. This is due to the fact that two nearby symbols will have a distance of 0, as in the definition of the $dist(q,c)$ table above. This serves as a counterexample to the identity metric property being $dist(x,y) = 0$ iff $x == y$. For example, if $x = a$, $y = b$ then $x! = y$, but $dist(x,y) = 0$.

Due to two different symbols having a distance of zero, the triangle inequality does not hold in all cases. For example – $x = a$, $y = c$, and $z = b$. $dist(x,y) \geq 0$, but $dist(x,z) = dist(y,z) = 0$. This is because the *dist* function always takes the difference between the upper boundary of the "smaller" symbol and the lower boundary of the "larger" symbol. This might lead to interesting results for LSH in terms of how the choice of the number of symbols affects the performance. We explore this in Section 4.3.2

It is, however, possible for LSH to work with SAX, without it being a metric. Given the long length of the time series in question, the zeros over the full length will become less and less likely to affect the distance. This does require careful selection of the parameter for number of symbols, as using 2 or 3 symbols is too little to have any distinguishing power due to the sharing of lower and upper boundaries between nearby symbols.

2.2.5 SSH

The last LSH method we explore for streaming time series is Sketch, Shingle and Hash (SSH), presented by Luo and Shrivastava [2016]. This approach resembles the approach of LSH with minhash from Section 2.2.2 – the similarity is calculated based on Jaccard set similarity (Leskovec et al. [2014]), presented in Section 2.1. Below we describe the details of the 3 steps in the procedure – Sketch, Shingle and Hash.

Sketch

The real-valued data is encoded symbolically to form the sets by random projections. A number of vectors of a certain length are chosen – these could be random or indicate a certain pattern, like a filter for image processing. These vectors are multiplied by the subsequences of the time series to calculate dot products. The signs of these projections are then recorded as the binary representation. This stage is titled "Sketch" – the time series has been encoded as a binary string.

For example, if the chosen subsequence length is 2, a possible random filter for "sketch" is "-0.1 0.1" and we then apply this filter by sliding it across our time series of length n (for example 4) with the selected step size (let's say this parameter was set to 2). Thus, for a time series of "2 1 2 4", the *sketch* result is a string of length 2 with the following encoding:

$$B_x =< sgn(-0.1 \times 2 + 0.1 \times 1), sgn(-0.1 \times 2 + 0.1 \times 4) >=< sgn(-0.1), sgn(0.2) >=< -1, +1 >$$

where *sgn* is the sign function.

Shingle

In the "Shingle" stage the now binarized time series is turned into a weighted set of all the n-grams of a certain length that occur in its binary string. For example, if the binarized time series is 01010111, then all of the n-grams of length 4 and overlap of 2 are 0101, 0101, 0111. These can be described more compactly as a weighted set: 0101: 2, 0111: 1. These sets are placed into hash tables according to their calculated hash value by a weighted minwise hashing procedure during the last "Hash" step. Jaccard set similarity can then be used on the hashed sets to calculate the similarities and label edges in the graph for nodes that appear in the same bucket for the hash value.

Hash

The weighted minwise hashing procedure [Shrivastava, 2016] is similar to minhash described in Section 2.2.2. In minhash we sample elements in the order provided by the hash function until the first non-zero value – indicating that this element exists in the set. For the weighted version, however, we also need to consider the weights of the elements, as two sets that both have the word "dog", might differ in the number of times the word occurs. The weighted minwise hashing procedure is then implemented by the following steps:

1. For each element we define the weight of the element p for its number of appearances in the set. This is the fraction of number of the element appearances in the set over the maximum possible number of elements in the set, and so $0 \leq p \leq 1$. For SSH the elements are unique shingles. Given the constant length of shingle, overlap of the shingles when created and the total length of string for each set, this number is easily calculated and is the same across all sets. For an element that does not exist in the set $p = 0$. If the element appears 3 times in the set and any set can at most contain 5 shingles, then $p = 3/5 = 0.6$.

2. For each element, we define a "green region" as the interval $[0, p]$.

3. Hash functions specify an ordering of the elements. Each element in a hash function is represented as its index + a random number in the region $[0, 1]$. If the random sample falls in the "green region" of an element, then this is the element we stop at.

4. The hash value is returned. The hash value can be the index of the element that gave us the first value in the "green region" or alternatively the number of steps we completed. Unlike minhash, even with full length permutations, it is possible for sets to not have a valid hash value for some of the functions, as the random samples need to not only match a shingle occurring but also fall in its "green region" – i.e. match its weight.

Before moving on to describing the details of these four LSH implementations, we introduce the possible applications for the techniques in the next section.

2.3 Applications

LSH has been used in many fields for various tasks. For example, in Chen et al. [2019b], the authors use multi-probe LSH for service recommender systems to counter the sparseness of the data space and preserve user privacy, with higher robustness and acceptable accuracy compared to other methods. Beck et al. [2019] propose to improve the mean shift clustering (a step in gradient ascent method) from $O(n^2)$ to $O(n)$ by calculating the nearest neighbours by LSH. Chen et al. [2019a] use LSH to outperform deep learning on a CPU over the accepted best GPU approach for neural networks – by choosing sparse sets of neurons to activate by LSH.

Work by Tripathi et al. [2019] uses LSH for image super resolution, specifically manifold learning. The implementation of LSH is done in Hadoop to parallelize across the hash tables – the search over each table is done in parallel for each image patch. This cloud-based solution shows improvements over other accepted approaches for this task. Turrado García et al. [2019] use LSH to detect misspelled people's names or close duplicates in text. This is applicable for example in databases of customers. After finding the candidate pairs by LSH, a filter based on the full Damerau-Levenshtein distance (also known as the edit distance [Damerau, 1964, Levenshtein, 1966]) is used to distinguish misspellings from names from the same family group. The found candidate pairs are presented visually as a graph with similar names forming cliques.

LSH has also been commonly used for systems applications. For example, Nissim et al. [2019] tackle the problem of detecting malware on virtual/cloud machines through examining the volatile memory dumps. The proposed methods uses minhash, as it is efficient at comparing binaries statically and does not require feature engineering. This is done by designing a similarity classifier that works on the minhash-produced signatures, which can be quickly updated for new examples. The use of LSH in this context made the classification of malware more than 3 times faster. Another work by Yan et al. [2019] uses an LSH forest for balancing of memory, efficiency, accuracy and user privacy, along with collaborative filtering, for Social Internet of Things [2] service recommendation. Results on the WS-DREAM data set show decreased time and increased accuracy, while protecting user privacy.

LSH has a drawback of requiring multiple hash tables to produce good accuracy and the schemes are often single-machine based, which is not good for growing data sizes and distributed applications. Wu et al. [2020] investigate load balancing for peer-to-peer networks that are used for LSH-based similarity search, and build upon the authors' previous work. The results propose a static distributed similarity indexing scheme that better maintains load balancing. This achieves better Gini coefficient (measure of load imbalance) and relative recall in comparison to other methods.

For scientific applications, Hu et al. [2016] show how LSH can be used on the brain time series to identify ADHD subjects. Liu et al. [2019] use LSH to label brain scans. Kim and O'Reilly [2015] show an

[2] social networks combined with devices interconnected by the internet

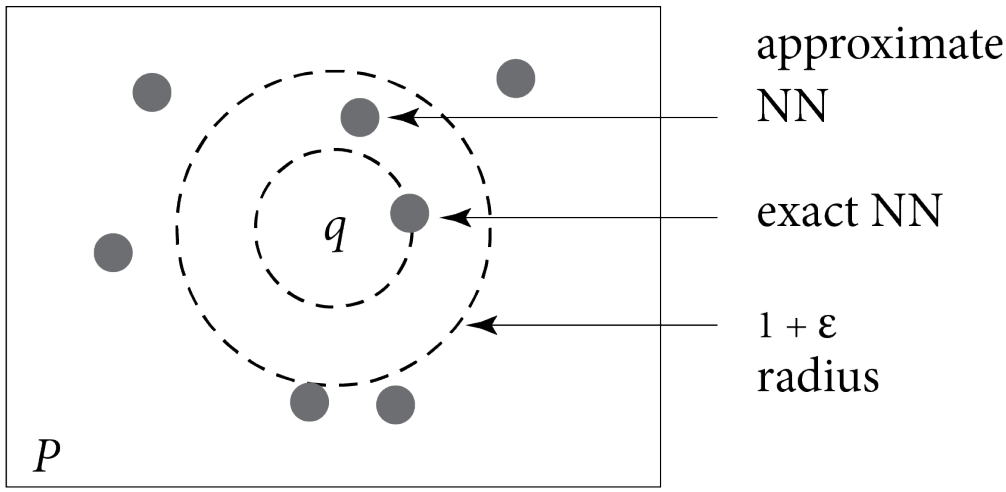

Figure 2.4: Approximate Nearest Neighbours

LSH approach with the L_1 distance used for retrieval of similar physiological waveform time series faster than naïve k nearest neighbours approaches without sacrificing accuracy. Such a solution is valuable for forecasting conditions of sepsis and hypotension based on blood pressure time series. Rong et al. [2018] use LSH to find similar time series of seismic activity to detect earthquakes with a two order-of-magnitude speedup and increase in scale of data used.

Originally, LSH was presented as part of an algorithm that showed an improvement in computational time over brute-force and other earlier approaches to solving the nearest neighbour (NN) search problem [Indyk and Motwani, 1998]. This problem was first formulated by Minsky and Papert in the 1960's [1969] and is as follows:

Definition 2.3.1 (Nearest Neighbour Search Problem) *Given a set of n points $P = p_1, p_2, \ldots, p_n$ in a space X^d, process P to efficiently answer queries for finding the nearest point in P to the query point $q \in X$.*

The problem is well-solved for low-dimensional spaces X. The more interesting cases we want to focus on are the cases of high-dimensional spaces X under some norm L_p, which are subject to the "curse of dimensionality". The computational complexity of working with items of multiple dimensions will often increase exponentially to the linearly increasing number of dimensions, along with increasing empty space between data points, making all points appear to be far from all others. Previous approaches all suffer from this "curse" along with the brute-force approach where each point $p \in P$ is compared to the query point q to determine the nearest neighbour. This limitation is also present for the more "relaxed" version of the NN search – the approximate NN search (ANN), formulated as follows:

Definition 2.3.2 (Approximate Nearest Neighbour Search Problem. Illustrated in Figure 2.4.) *Given a set of n points $P = p_1, p_2, \ldots, p_n$ in a space X^d, process P to efficiently answer queries for finding the ε-nearest point $p \in P$ to the query point $q \in X$, such that for all $p' \in P$, $d(p,q) \leq (1+\varepsilon)d(p',q)$.*

LSH is then introduced as the key ingredient for an algorithm that can solve the ANN with time linear in n and d for the processing stage and sublinear time in the query stage (given $\varepsilon > 1$) [Indyk and Motwani, 1998]. The time complexity for finding the ε-nearest neighbour for a query is shown to be O(n^{ρ}), where:

$$\rho = -\frac{\ln p_1}{\ln p_1/p_2} < 1$$

for an (d_1, d_2, p_1, p_2)-LSH family [Indyk and Motwani, 1998], as defined by Definition 2.2.1.

Two other related problem definitions for time series are correlation set and network discovery.

Definition 2.3.3 (Correlation set discovery problem [Amagata and Hara, 2019]) *Let $p(x,y)$ be the Pearson correlation between two time-series x and y. Given a result size k, a user-specified threshold θ, and a set of time-series data U, our problem is to compute a set $S \subset U$ such that $|S| = k$, $\forall x, y \in S, p(x,y) \geq \theta$.*

Definition 2.3.4 (Network discovery problem [Safavi et al., 2019]) *(Multiple time series to weighted graph) Given n time series $X = x_1, ..., x_n$, construct a sparse similarity graph where each node corresponds to a time series x_i and each edge is weighted according to the association of the nodes (x_i, x_j) it connects.*

Network discovery by LSH is applicable in many fields, such as neuroscience, genomics, climate science and economics, as LSH is easily generalizable. The problems of nearest neighbours, correlation set discovery and network discovery are interrelated and can all be tackled by LSH. The buckets formed from input data are essentially the correlation sets which consist of the candidates for the nearest neighbours. By calculating the distances between the points in the same bucket and combining this information between buckets, we can build a map of the LSH-processed space – the network of the data with preserved "locality" of points and therefore the topology of the input space. In this thesis, we will focus on the network discovery aspect, keeping in mind that the results that LSH provides can be re-interpreted for solutions of the other problems as described.

Using LSH for network discovery will provide us with a sparse locality-sensitive network. Following the definition and the theoretical bound on LSH query time of O(n^{ρ}), we can expect the time complexity of our approach to be on the order of: $O(n \times n^{\rho}) = O(n^{1+\rho}) < O(n^2)$, since $\rho < 1$. For example, for values that we use in Section 4.2.2 for setting the number of tables to use, the chosen probabilities were $p_1 = 0.9$ and $p_2 = 0.1$, which then produce the complexity constant $\rho = 0.048$. This allows us to avoid the $O(n^2)$ distance comparisons that would be required otherwise for building a correlation or some other type of network for n items.

There is previous work using LSH for network discovery. Davidson et al. [2013] tackle network discovery in the brain from fMRI data. The authors make many insightful observations on the nature of related tasks of finding nodes of the network, establishing edges and evaluating discovered clusters. Independent Component Analysis (ICA), clustering and belief networks are some of the methods named that are used for network discovery previously. However, using these in the context of brain network discovery leads to

simplifying data arbitrarily due to not having a way to easily encode domain expertise in this field. The authors note that existing approaches are not designed for the network discovery task specifically, but rather modify data in some way and then post-process to build the network. The authors also highlight the lack of existing solutions to infer networks from the brain that work with fMRI data and are both flexible and scalable. Safavi et al. [2017, 2019] propose the ABC-LSH as a candidate solution, as their correlation-based similarity measure shows vast improvement in speed of the network discovery. In our thesis, we build on their work by implementing ABC-LSH in parallel and comparing it to other LSH techniques.

An efficient LSH-based network discovery system can also assist in finding anomalies in the discovered networks. Anomaly detection is a vital field for complex systems requiring real-time supervision, such as equipment maintenance for operations where any downtime is very costly [Kammerer et al., 2019]. With LSH, anomalies can be detected by comparing the resulting buckets after performing LSH on a set of time series coming from the same system – certain changes to the correlation sets found can be indicative of changes to the underlying process.

One area in which anomaly detection is particularly important is aircraft operations. The airspace is one of the most complex man-made dynamical systems to exist, due to the vast amount of moving objects, all of which are made up of complex equipment [Matthews et al., 2013]. It has experienced large growth over time and requires constant improvement and research of techniques to keep the system at the established safety level [Matthews et al., 2014]. Previous research has discovered anomalies using trajectory, weather, terrain, surveillance, radar and flight recorder data. The latter can consist of both continuous and discrete data from the avionics and propulsion systems, control surfaces, landing gear and cockpit switch positions [Das et al., 2010]. Anomalies detected in both simulated and real airline data could include marking flights that have experienced emergency landings or even discovering risk events that were not known before when mining old data [Bhaduri et al., 2010, Das et al., 2010, Matthews et al., 2014].

Modern planes collect up to a terabyte of data per flight, with planes like Boeing 787 and Airbus A350 collecting 10,000 times more data than planes from two or three decades ago [AVM, 2017]. The obtained data is large enough to form baselines for trend and anomaly detection, and includes key measurements for operations, maintenance, training, and equipment optimization. Airlines struggle to keep up with the volume of data, however, and need technology solutions to extract value from the data [Negroni, 2021].

This amount of data cannot be decreased to a large extent, as a certain amount of replication can be necessary for detecting errors in other sensor measurements. Systems that double-check measurements between sensors can compare measurements to predicted calculated values and distinguish between a real emergency sensor reading and a sensor failing [Varela, 2019]. Failing sensors happen and are responsible, for instance, for the Boeing 373 Indonesia and Ethiopia Max8 crashes, as well as the Air France Flight 447, which killed 228 passengers and crew. The crash of Tuninter Flight 1153, resulting in 16 deaths, was caused by a faulty fuel quantity indicator sensor. Since fuel has a mass, its quantity affects the performance of the

aircraft during flight and it could have been possible to see that the weight shown by the faulty sensor did not correspond to the way the physical system was behaving as indicated by other sensors.

An example of a system that records data during a flight is the Engine Vibration Monitoring system by VibroMeter used in Airbus and Boeing planes for monitoring engine health [SA, 2019]. The engine is the most expensive part of the plane to replace – costing up to 20 million dollars – and is vital for the functioning and safety of the aircraft. Modern systems are capable of monitoring engine sensors and sending data over satellite during the flight. Having access to monitoring data before the black boxes can be retrieved has proven beneficial in crash investigations, according to Goglia [2014]. For example, for the Air France flight, the data transmitted allowed Airbus engineers to make a guess that the pilot tubes (sensors used in speed indication) froze and were giving the wrong measurements, which then lead to the crash. The changes to the pilot tubes were able to be implemented by Airbus even before the black boxes were retrieved and the cause was confirmed. The ability to monitor real-time flight data could also be crucial, for example, for the Malaysia Air Flight 370, which was found to be flying hours after a transponder stopped transmitting.

Although the data is important for usage in root cause analysis, the wish is to be able to use monitoring for predictions of problems before they occur, and to be able to use for improved planning in part logistics, shop visits and fleet management [AVM, 2017]. Using data to predict what components need checking at arrival can be useful for preventing inefficient fuel usage from improperly working parts as well as costly delays and labour costs. This would be especially vital to companies like Malaysia's AirAsia, whose business model relies on planes spending no more than 25 minutes at an airport gate [Negroni, 2021].

As discussed in the previous few paragraphs, the analytical analysis of airline data is a challenging area with a tremendous potential benefit to both saving lives and money. In the remainder of this section, we discuss some of the approaches to solving the problems associated with the huge amount of data and the time constraint for analysis, in the context of anomaly detection, fault analysis, and lifetime prediction for parts in aviation.

An example of a data-driven technology for real-time fault detection in engines is presented by Sarkar et al. [2011]. This system attempts to cover the gap of previous model-based solutions, like Kalman filtering, which suffer from becoming unreliable with increasing model complexity. On the other hand, existing data-driven neural network approaches either tend to focus on only a few sensors, considering others as a black box to prevent computational overload, or are sensitive to breaking down under noise and drift. Another issue with data-driven approaches is the reliance on a high volume of training data, which is hard to acquire for engine failures, as flights are very reliable [Sarkar et al., 2011]. The approach to solve this issue is to use snapshots of the sensor readings or time windows for monitoring. The authors use symbolic dynamic filtering for feature extraction to be optimized and pose the problem as a multi-class classification problem. The approach is to hide the noise and amplify the system fault signatures, while minimizing the number of false alarms and remaining real-time and scalable, which is critical for supervised fault detection.

One anomaly detection model is a full least squares regression approach by Chu et al. [2010]. The authors built a data-driven model from a database of flights from a fleet of the same plane model, limiting to cruise flight phases. Each new flight can then be added to the database and tested for having any deviations from the existing model, which would be indicative of anomalies. The regression approach is usually used in manufacturing and process industries, and is novel to the aircraft data with the planes operating under a range of possible flight conditions. A similar approach has been used before by Volvo using self-organizing maps in their bus operation [Chu et al., 2010], but many details of this and other approaches are proprietary. This model was tested on data from the NASA flight simulator [Kaneshige et al., 2000] and has shown sufficiently good results with the least squares, as compared to existing more complex approaches with clustering and neural nets.

A data-driven deep learning approach by Wang et al. [2021] is presented for estimation of remaining useful life for aircraft engines. It is trained using a feed of clipped sensor readings with provided life scores. The proposed approach uses a temporal convolutional network for the time component of the data and a graph convolutional network for the "space" of the sensor readings. The graph is built based on Pearson similarity between the streams of different sensors. The combination of these two types of networks is a novel application for this domain.

Our proposed work is also building a network between the sensor-created time series. The resulting network is made to capture the similarity and, to some extent, the correlations between the sensor measurements. However, by using LSH instead of correlation, we can improve the time of creation of this network by producing a sparser one. Our work in this thesis could be used in the contexts of anomaly detection, diagnosis of fault causes and prediction for maintenance, as all of these uses could be done from the network discovered by our approach. However, we will focus our attention on the anomaly discovery aspect and on the creation of the network for a specific streamed flight, not on the model creation for comparison. *The benefit of our approach is that LSH requires no training data and is fast and scalable, on which we improve further by utilizing a parallel framework*. Our LSH approach uses windows and subsequences instead of snapshots. This makes the implementation flexible to the size of incoming data and we can choose to focus on any part of the flight, for example a certain phase. Another benefit is that our implementation is evaluated on a real flight recorder data set, not on simulated data.

In this thesis we will use flight recorder data from duel-jet engine Boeing 727 aircraft. We describe the data set in more detail in Section 4.1. This is a realistic sample data for this area of application for such fast and scalable similarity-based systems. Using domain knowledge, we can expect certain parameters of the plane to be highly correlated and produce similar time series. For example, in planes with multiple engines, the sensors for engine speed can be expected to be similar. We would thus expect our LSH methods to place these parameters in the same bucket, with their calculated similarity scores being high. This also allows us to compare results between flights and possibly detect anomalous flights where the groupings of such parameters are no longer present. This could happen if an engine is breaking down. The anomaly could

be seen on the discovered network by our LSH approach, as it no longer contains the edges connecting the troubled and working engines.

Overall, our thesis will explore parallel LSH with applications of network discovery and anomaly detection from streaming real-life airline sensor time series.

Chapter 3

Implementation

In this chapter we describe our LSH implementation. First, we provide details on the preliminary parallel LSH project with minhash, and then go on to describe the implementation details of ABC, SAX and SSH methods in our parallel message passing framework.

The implementation is done in C with Fine-Grain MPI (FG-MPI) [Kamal and Wagner, 2014] [3]. FG-MPI is an extension of the Message Passing Interface (MPI) that introduces further concurrent MPI processes inside an OS-process.

The general network discovery workflow is shown in Figure 3.1.

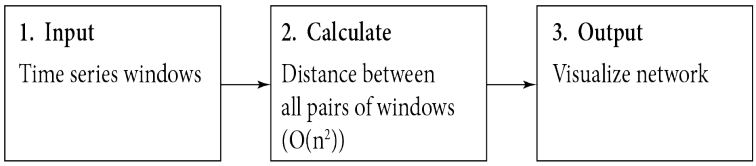

Figure 3.1: Network Discovery General Workflow

Our implementation makes two main changes to this general approach. First, we implement a parallel pipeline framework, where each time series window is processed independently and therefore the network is built incrementally in real-time. And second, by using LSH we decrease the number of distance calculations necessary, and therefore the complexity of the resulting network by increasing the sparseness.

The first implementation is minhash (Section 3.1), which is a Jaccard distance technique for binarized sets. Since the Jaccard distance calculation is on sets, the calculation does not emphasize the consecutiveness of the data and therefore not as well suited for time series. However, Jaccard-based minhash does provide a good initial test of LSH in our parallel framework.

Our main focus is on three LSH approaches that are capable of capturing time dependency – ABC, SAX and SSH. All implemented with the same architecture, they possess some common and some specific parameters affecting performance. The three parts that matter the most in each of the methods' implementations are their pre-processing procedures, hash value calculations and similarity measures, turning the workflow into what is shown in Figure 3.2.

[3] see https://www.cs.ubc.ca/ humaira/fgmpi.html

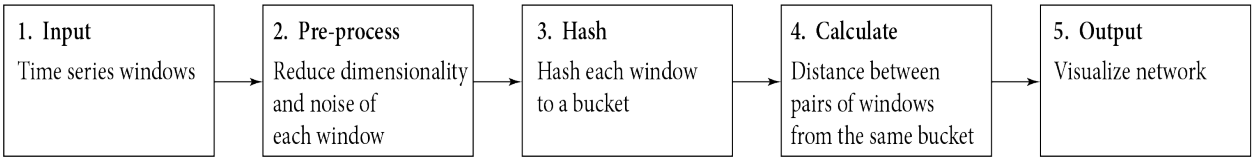

Figure 3.2: LSH Network Discovery Workflow

The details on the influence on the time and communication complexity of the approaches, and the specifics of the three implementations are presented in their respective Sections 3.2.1 to 3.2.3, which follow after the description of the common architecture of the proposed parallel pipeline framework in Section 3.2.

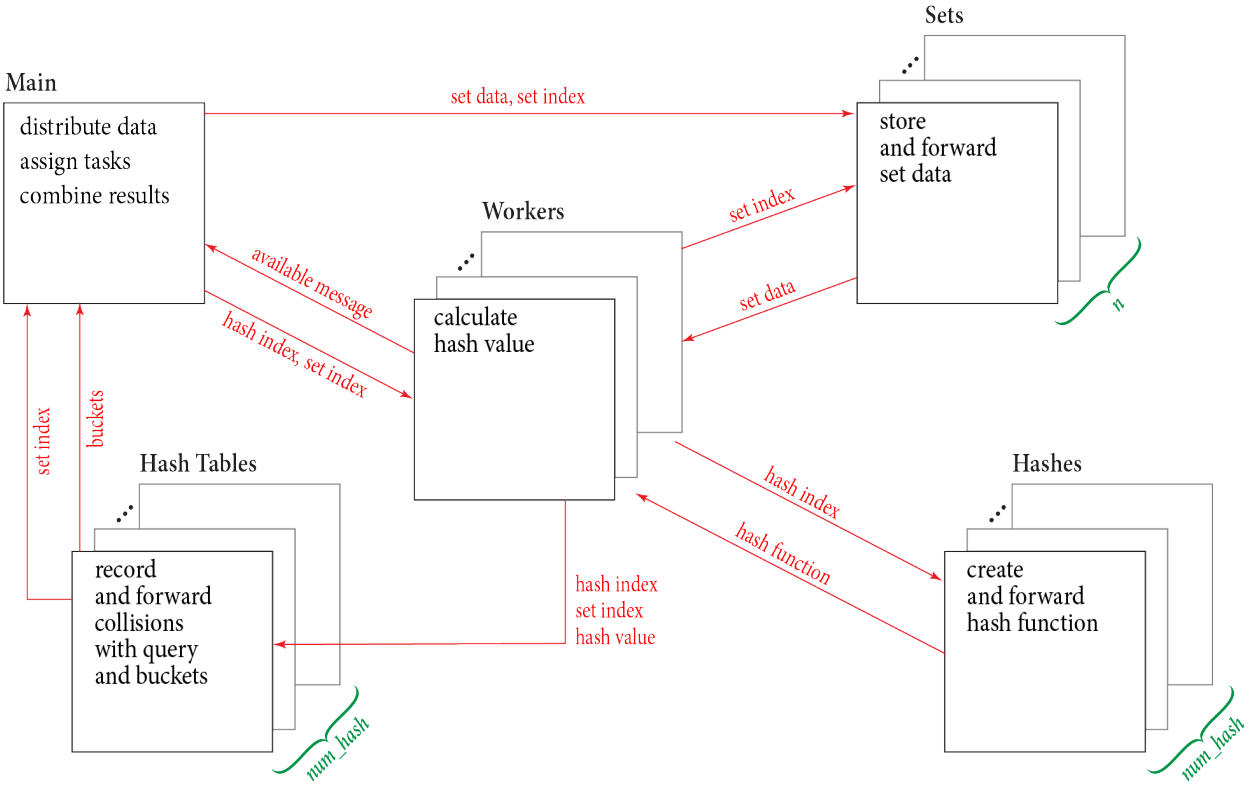

Figure 3.3: Minhash Implementation. This diagram shows the basics of the minhash LSH implementation in a parallel message passing framework. Some of the parameters are shown in green with the data flow in red.

3.1 Minhash Pilot Project

A preliminary pilot project on LSH in a parallel message passing framework was implemented, focusing on *minhash*, one definition and description of which is presented by Leskovec et al. [2014]. The details of the approach were described in Section 2.2.2. The 20 newsgroups [Dua and Graff, 2017] data set was used for validation and testing of the communication framework in the context of usage with LSH in our

implementation. The last set that is read in was treated as the query set and the algorithm focused on finding nearest neighbours of that set and approximating its Jaccard similarity with the best candidate found. The architecture consisted of a number of nodes: one reader, one manager, and a collection of workers, hashes, sets and hash table nodes, with the main data flow and tasks of processes shown in Figure 3.3. The bucket formation could be distributed further to allow for more flexibility and parallelization. The full code implemented in C with FG-MPI is publicly available [Sodol, 2020].

Computing LSH or the notion of fingerprinting a high dimensional vector is highly parallel. In minhash the use of permutations to construct the fingerprint for a vector can all be done in parallel. Although in the following we describe a process using FG-MPI and a light-weight process model, it a process that could be implemented in a many-core specialized hardware device such as FPGA or ASIC device.

The overall workflow and main tasks of all processes are presented in Figure 3.3. Further details are described in the following steps:

Step 1. The Manager process distributes the data of the streamed sets (see Listing 3.1).

Listing 3.1: Minhash Main Node

```
for each hash:
  for each set:
    receive(worker_available_message)
    send(set_index) //to requesting worker
    send(hash_fn_index) //to requesting worker
for each worker:
  send(stop_cmd)
find_candidates()
find_best_match()
find_jaccard_similarity(best_match, query)
```

Step 2. Each Worker process (Listing 3.2) implements a random walk through the rows of one column, after receiving the data from a Set node (Listing 3.3), as dictated by a hash function from a Hash node (Listing 3.4). This is equivalent to following a single column of the characteristic matrix that was permuted by the hash function in order. Returning to the example in Section 2.2.2, suppose the hash function is 3102, then the permuted characteristic matrix for S_1 is like the one in Table 3.1.

Hash Value	*Feature Index*	S_1
0	3	0
1	**1**	**1**
2	0	0
3	2	1

Table 3.1: Permuted Characteristic Minhash Matrix for One Set

The worker that was assigned H_1 and S_1 then follows the column from top to bottom until the first 1 and records the corresponding hash value. The actual hash value recorded could be the feature index value (what our implementation does) or the number of steps into the hash function that it took to find the 1 – this is equivalent to the "hash value" index in Table 3.1. Both of these ways can work, as long as they are consistently followed by all Workers. This is easily scalable as more rows, columns or hash functions can be added. All the hash values for all sets are computed in parallel.

Listing 3.2: Minhash Worker Node

```
while(1):
  send(worker_available_message) //to main
  receive(set_index)
  if set_index == stop_cmd:
    return
  receive(hash_index)
  send(worker_available_message) //to set_index
  receive(data)
  send(worker_available_massage) //to hash_index
  receive(hash_fn)
  for i in range(0, size_hash):
    if data[hash_fn[i]] == 1:
      send(set_index) //to hashtable node hash_index
      send(i) //hash_value to hashtable node hash_index
      break
    else if i == size_hash-1:
      //to hashtable node hash_index
      send(set_index)
      //marker of no value found to hashtable node hash_index
      send(num_words+1)
      break
```

Listing 3.3: Minhash Set Node

```
receive(data)
for each hash:
  receive(worker_available_message)
  send(data) //to requesting worker
```

Listing 3.4: Minhash Hash Node

```
create(hash_function)
for each set:
  receive(worker_available_message)
  send(hash_function) //to requesting worker
```

Step 3. The hash values for all columns for a single hash function are gathered in a single Hash Table node for easier candidate pairs recognition, with functionality described in Listing 3.5.

Listing 3.5: Minhash Hash Table Node

```
values[n]
for each set i:
  receive(message)
  if message == stop_cmd:
    break
  else:
    receive(hash_value)
    values[message] = hash_value
send(values) //to main for graph building
for each set i:
  if values[i] == values[-1]:
    //to main for query neighbour search
    send(i)
send(work_complete_command) //to main
```

Step 4. The Main node keeps track of all candidate pairs, which can be used to form the complete discovered network for the data set. The pseudocode is shown in Listing 3.6.

Listing 3.6: Minhash find candidates

```
  candidates{} //set of 3-tuples for pairs and their counts
  pairs[] //list of tuples for pairs
  counts[]
  t = num_hash/2 //threshold for candidate pairs
  for each hash i:
    for each set j:
      receive(hash_value)
      bucket = retrieve_bucket(hash_index, hash_value)
        for each item in bucket:
          if pairs.contains((j, item_index)):
            counts[index(j, item_index)]++
          else:
            pairs.append((j, item_index))
            count.append(0)
  //add all edges with weight above threshold to form the candidate graph
  for i in range(0, pairs):
    if counts[i] >= t:
      candidates.add({set1[i], set2[i], counts[i]})
```

Step 5. The Hash Table nodes report on any sets that share a bucket with the query set and forward this information to the Main node. The Main node returns the set with the most shared buckets with the query as its best match and compares its approximate similarity calculated by minhash and true Jaccard similarity. Details of this computation are presented in Listing 3.7

Listing 3.7: Minhash find best match

```
clashes[num_sets-1]
cnt = 0
//each hash table sends the indexes of sets that matched with the query
while (cnt < num_hash):
  //receive message from a hashtable node
  receive(set_index)
  if set_index > num_words:
    //marker that the signature node is done sending clashes for query set
        for this hash function
    cnt++
  else:
    //record the clash
    clashes[set_index]++
best_clashes = max(clashes)
best_match = argmax(clashes)
```

The minhash architecture implemented originally for use with text data as described above was reworked for our further investigation into LSH methods that work with time series. The next Section 3.2 provides details on the final implementation that was used for experiments on time series data.

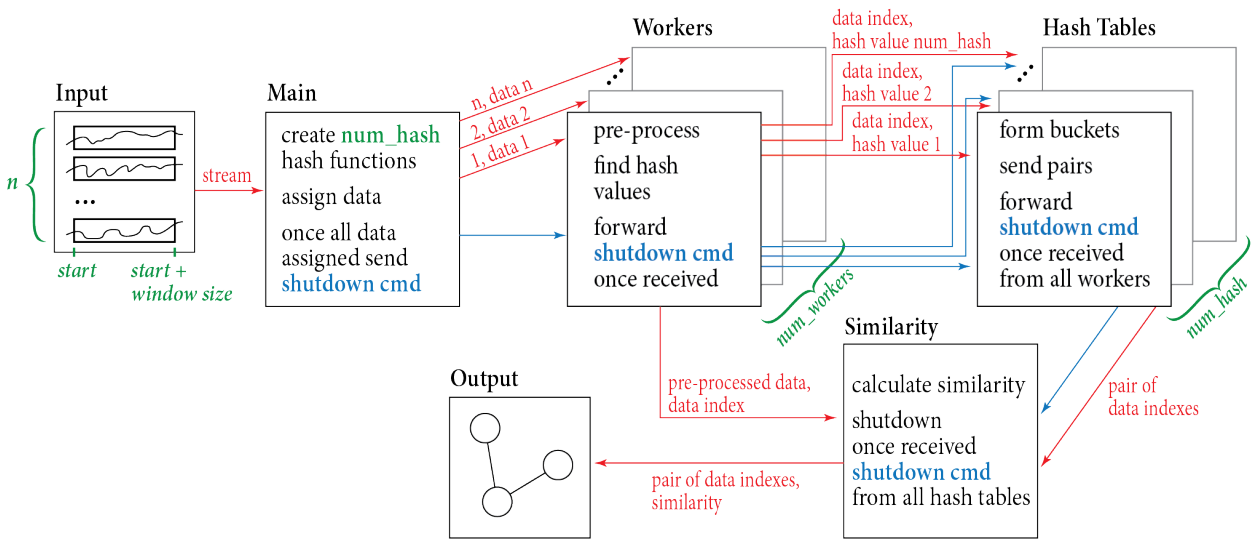

Figure 3.4: Process Architecture. This diagram shows each parallel process as its own box, with parameter names in green, data flow in red and the shut down procedure in blue.

3.2 Architecture

For our main experiments, a parallel implementation of three LSH approaches was completed with MPI (Message Passing Interface) [Sodol, 2021]. All three approaches use the same general architecture, with a similar structure of processing nodes as was done for the minhash project described in Section 3.1. The processing structure assumes the data items are streamed into the system. The architecture is represented in Figure 3.4 and shows the communication between nodes in red, the shut down procedure in blue and the common parameters in green.

One node is designated as the *Main* managing node, one node is the *Similarity* node and multiple *Worker* and *Hash Table* nodes are working in parallel. The incoming data stream is pipelined through the nodes in the order of Main – Worker – Hash Tables – Similarity node – this flow is shown in red in Figure 3.4. Each new stream is added to the hash tables as it is processed, and forwarded to the Similarity node with its pairs as necessary. This means that the output in the form of the network is formed dynamically with new nodes and edges being added as new data is arriving. Once the Main node has received and assigned all of the required streams, it sends a *shut down cmd* that is passed through the pipeline as outlined in blue on Figure 3.4 – the Workers forward this command to the Hash Tables, which forward it in turn to the Similarity node.

As in minhash in Section 3.1, the Main manager node assigns an available Worker a data item and its hash functions, as presented in Listing 3.8. The hash functions and all auxiliary data that needs to be initialized are created in the Main node. These are forwarded as needed, dependent on the assignment and current LSH method, along with the pre-specified current window of interest from the streaming data.

Listing 3.8: LSH Main Node

```
create preprocess_helpers, hash_functions

for each worker:
  send(preprocess_helpers, hash_functions)

if SAX:
  create distances
  send(distances) //to similarity

for each series in datafile:
  receive(data)
  receive(worker_available_message)
  send(index, data) //to worker

for each worker:
  send(work_complete_message)

while(1):
  probe(message)
  switch(message.tag):
    case WORKER_AVAILABLE:
      break
    case WORK_COMPLETE:
      return
```

The Workers complete the pre-processing and hash value calculations in parallel. The types of messages and respective operations are shown in Listing 3.9. Workers can receive new assignments after the current pairing results are sent to the Hash Table node in the form of data index and hash value. The pre-processed version of the data is stored in the Similarity node by its index for further use in distance calculations.

Listing 3.9: LSH Worker Node

```
while(1):

  probe(message)
  switch(message.tag):

    case HASH:
      receive(preprocess_helpers, hash_functions)
      break

    case DATA:
      receive(index, data)

      preprocess(data)
```

```
14        send(index, data) //to similarity
15
16        for each hash_function:
17
18          hash_value = lsh(data)
19          send(index, hash_value) //to hashtable
20
21        send(worker_available_message) //to main
22        break
23
24      case WORK_COMPLETE:
25        for each hashtable:
26          send(work_complete_message)
27        return
```

The Hash Table node forms the buckets and updates them as necessary, as shown in Listing 3.10. When an item is added to an existing bucket, all pairs of items formed between the new item and the ones previously in the bucket are considered candidates and, in the form of their index, the candidates are forwarded to the Similarity node. This node in turn computes the distance between the two items and saves the results in matrix form for output.

Listing 3.10: LSH Hash Table Node

```
1 work_complete_count = 0
2
3 initialize_hashtable()
4
5 while(1):
6   probe(message)
7   switch(message.tag):
8
9     case DATA:
10      receive(index, hash_value)
11
12      bucket = hashtable_lookup(hash_value)
13
14      for each item in bucket:
15        send(index, item.index) //to similarity
16
17      hashtable_insert(hash_value, data)
18      break
19
20    case WORK_COMPLETE:
21      work_complete_count++
22      if work_complete_count >= worker_count:
23        send(work_complete_message) //to similarity
24      return
```

The output of the Similarity node is a distance matrix giving the similarity values between pairs of time series items, formed and updated by messages and cases in Listing 3.11. For display purposes a threshold value t can be included to display the result as a graph with edges between two nodes i and j whenever the distance $d(i, j)$ is less than t.

Listing 3.11: LSH Similarity Node

```
storage_table[length_data][n]
processed_pairs[n][n]
similarity_matrix[n][n]
work_complete_count = 0

if SAX:
  distances[num_symbols][num_symbols]

while(1):
  probe(message)
  switch(message.tag):

    case SIMILARITY_HELPERS:
      receive(distances)

    case DATA:
      receive(index, data)
      storage_table[index] = data
      break

    case PAIR:
      receive(index1, index2)
      if proccessed_pairs[index1][index2] !=0:
        processed_pairs[index1][index2] += 1
        processed_pairs[index2][index1] += 1
        break
      else:
        processed_pairs[index1][index2] = 1
        processed_pairs[index2][index1] = 1

        similarity = sim_calc(storage_table[index1*length_data], storage_table[
            index2*length_data])

        similarity_matrix[index1][index2] = similarity
        similarity_matrix[index2][index1] = similarity
      break

    case WORK_COMPLETE:

      work_complete_count++

```

```
41        if work_complete_count >= num_hash:
42          write_to_file(similarity_matrix)
43          send(work_complete_message) //to main
44        return
```

The OR and AND constructions are the two ways in which LSH reduces the number of computations. Under the OR construction we only consider pairs of nodes that have ended up in the same bucket for at least one hash table. This means that the similarity is calculated for any and all pairs of indexes received from the Hash table nodes. Alternatively, we can set a parameter that controls how many hash tables are to be included in the AND construction. The AND construction in our implementation is performed at the last stage of the pipeline in the Similarity node. By determining a threshold of how many hash codes should be the same to be included in the output, we are essentially producing the union of all items that would be included in any possible AND construction of the given size with the given number of tables. This approach to the AND construction is similar to the Collision Frequency LSH that was shown to be superior with time and performance for L_1 norm by Kim [2017]. The details on setting the number of tables for the AND and OR constructions and how they affect the false positive and false negative rates are described in Section 4.2.2. Augmented LSH families allow one to harvest as much information as possible to narrow down the candidate sets to further decrease the number of distance calculations necessary for nearest neighbour search.

In addition to the common architecture and overall workflow, all three methods share the parameters of the window from the streaming data they are analyzing (specified by *start* time and length *window size*) and the total number of streams n, shown in green in Figure 3.4 in the Input section. These parameters share the same values for the three approaches to compare the results for the same windows. Other common parameters include the number of hash tables *num hash* (OR construction) – this parameter controls the number of processes for the Hash Table nodes, as shown in green in Figure 3.4 Hash Tables section. However, these particular values can be different depending on the specific LSH, as these will be tuned to produce the best results for the given approach. Remaining nodes are designated as the *num workers* Worker nodes.

For each method we also describe the time and communication complexity. One of the advantages of LSH is the ability to use a hash table, and this is the same for all implementations. The time complexity for placing each item into the hash table and querying a list of neighbours is constant as with any hash table implementation, and is done by the hash value calculated previously by the Worker.

As for minhash, the implementation of a process is simple, which is a great benefit of the message passing framework. Once a message is received, the behaviour of the node is determined based on the contents of the message. The computations each process performs are small, but in their collection and interaction, they form a system with an emergent complex behaviour. The following sections will provide details of the three LSH method implementations, focusing on the differences and the specifics of each method.

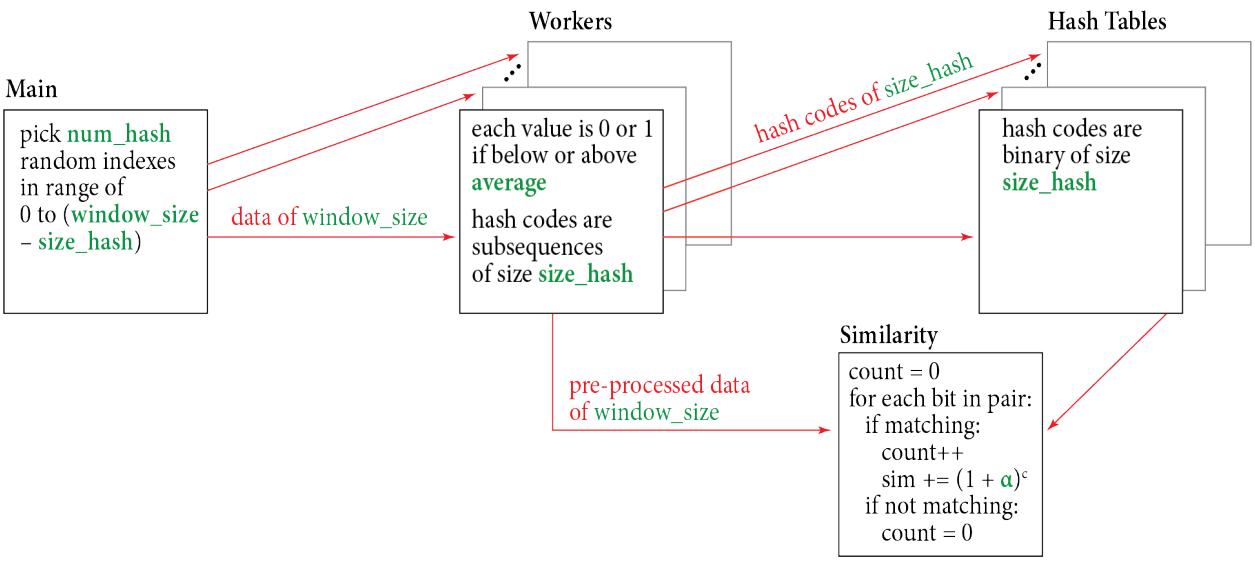

Figure 3.5: ABC Implementation. This diagram shows the extra details of the ABC implementation, beyond what is included in the common architecture from Figure 3.4. Parameters are named in green with the data flow in red.

3.2.1 ABC

Implementation Details and Parameters

The specifics for the ABC implementation are presented in Figure 3.5. This figure focuses on the differences and details of the ABC procedure as additions to the main architecture, with the parameters named in green and communication described in red. The parts not shown follow the overall architecture previously given in Figure 3.4.

The most important parameters for ABC are the number of hash tables *num hash*, and the length of resulting hash values *size hash*. Their usage is described in the Main section in Figure 3.5. For ABC, the hash functions are in the form of start indexes for the hash codes, and the hash codes are subsequences from the pre-processed window string. These hash value start indexes are taken randomly over their possible range. The possible indexes need to be followed by at least *hash size* bits, so the range is from 0 until the length of window *window size* minus the size of hash *size hash*. In this implementation we sample indexes uniformly until we have the correct number of non-repeating indexes and return them in increasing order.

The ABC pre-processing is straightforward and does not result in change of the length of the string, so the communication between the Main node and Workers is of the same size as between the Workers and the Similarity node, as noted on the red arrows in Figure 3.5. The window of the stream is binarized according to being above or below the mean, as described in Section 2.2.3. This is also shown in the Worker section of Figure 3.5. The *average* value for the specific stream can be calculated from the window available, inferred from domain knowledge or calculated over previously encountered data, as in this implementation.

The binary hash codes of *size hash* are passed from the Workers to the Hash Tables sections of Figure 3.5, with the pairs of indexes of the formed pairs from the same bucket passed in turn to the Similarity node. The last parameter for ABC is the α similarity parameter that controls the degree of consecutiveness when comparing two strings. Its usage can be seen on the Similarity section of Figure 3.5.

Time and Communication Complexity

For ABC, pre-processing and distance calculations are single pass procedures. The non-linear and non-constant component of the time complexity for ABC then results from the hash function calculation. Its time complexity is $num\,hash \times range + range^2$, where $range = window\,size – hash\,size$. The latter part comes from our implementation relying on quick sort, which is possible to avoid if the window size used is large and the indexes for subsequences are not needed to be produced in increasing order.

The non-simple terms for communication complexity for ABC include the *num workers* messages of *num hash* size for sending the hash functions, n messages of *window size* of sending pre-processed data, and $n \times num\,hash$ messages of *hash size* of sending the hash values to construct the hash tables.

3.2.2 SAX

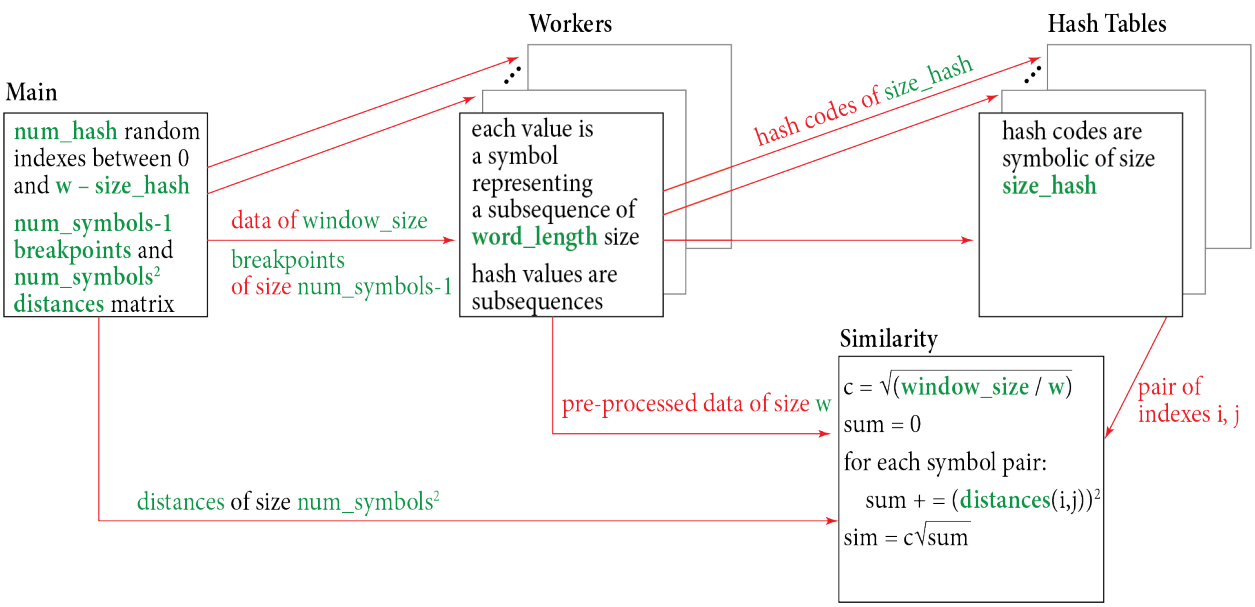

Figure 3.6: SAX implementation. This diagram shows the extra details of the SAX implementation, beyond what is included in the common architecture from Figure 3.4. Parameters are named in green with the data flow in red.

Implementation Details and Parameters

Specifics of the SAX approach are presented in Figure 3.6. The pre-processing for SAX is more involved than ABC and results in a shorter string (of size *w*) than original streamed data of *window size*. However, the parameters for hash table creation *num hash* and *size hash* behave similarly, with the exact same procedure for hash function generation through random indexes, as shown in the Main section of Figure 3.6.

The pre-processing procedure for SAX depends on parameters *w*, the number of resulting words from the original string, and *num symbols* – the number of symbols used to symbolize the "words" in the string. The usages of these parameters can be seen in green in Figure 3.6. The formation of "words" is done dependent on the *word length* parameter, with the *w* computed at start up from *word length* and the *window size*. Following the red arrows in Figure 3.6, the symbolized string is sent to the Similarity node and the symbolic hash code subsequences to the Hash Table nodes.

In addition to the hash functions, other pre-computed constants include the array of *breakpoints* of *num symbols*-1 size and the matrix of *distances* between single symbols of size *(num symbols)*^{2}. The breakpoints come from a statistical table – these values indicate cut-off values that separate the area under the normal curve into equal *num symbols* parts. This table is hard-coded in our implementation and the *breakpoints* array is simply constructed based on the value of *num symbols* on start up as a single column of Table 3.2, for each execution of the program. The *distances* matrix is computed once as the *num symbols* value is known and the *breakpoints* vector is selected. An example for 4 symbols is presented in Table 3.3 and the details of the formation and usage of these values are further described in Section 2.2.4.

	3	4	5	6	7	8	9	10
β_1	-0.43	-0.67	-0.84	-0.97	-1.07	-1.15	-1.22	-1.28
β_2	0.43	0	-0.25	-0.43	-0.57	-0.67	-0.76	-0.84
β_3		0.67	0.25	0	-0.18	-0.32	-0.43	-0.52
β_4			0.84	0.43	0.18	0	-0.14	-0.25
β_5				0.97	0.57	0.32	0.14	0
β_6					1.07	0.67	0.43	0.25
β_7						1.15	0.76	0.52
β_8							1.22	0.84
β_9								1.28

Table 3.2: SAX Breakpoint Values for 3 to 10 Symbols. Breakpoints β_i produce equal-sized areas under the Gaussian curve. The number of intervals is equal to the number of symbols, so that all values that fall into the same interval can be represented by a single symbol.

Prior to pre-processing, the streamed window is normalized with set parameters of *average* and *sd* – mean and standard deviation. As for ABC, we pre-compute these parameters from previously available data. SAX uses the same method for forming hash tables as ABC, and our implementation for other nodes is the same, except for parameter usage as described in this section above.

	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

Table 3.3: SAX Distances Table for 4 Symbols

Time and Communication Complexity

On top of similar complexities to ABC for hash creation, SAX also has the creation of *breakpoints* and *distances* matrix. With respect to time complexity, the *distances* take $(num\ symbols)^2$ to complete and the breakpoints are costnant time access from a hard-coded matrix. After single-pass normalization, the pre-processing for SAX takes $(w \times word\,length + window\,size + w + w \times num\,symbols) \times n$ time. Distance calculation is a single pass of w for SAX.

The *breakpoints* and *distances* matrix are sent through messages after creation, so for communication there are *num workers* messages of size $num\,symbols - 1 + (num\,symbols)^2$. The sending of pre-processed data is $n \times w$.

3.2.3 SSH

The specifics of the SSH approach in our framework are presented in Figure 3.7. Main parameters are shown in green and the communication data with its size in red. Unique parameters for SSH include the length for the sketching vector, *vector size*, along with the number of time steps to move this vector by, *vector overlap*. The signs of the dot products of the raw data with this sketching vector are recorded as a binary string of final size *size sketched*. This is the sketch step that is completed in the Workers section of Figure 3.7, producing the sketched string.

For shingling, one needs to set the length of the shingle *shingle size* and the overlap between shingles. The "shingle" step finds all of the subsequences of *shingle size* in the sketched string. The number of occurrences of each shingle appearing in the string are stored in a vector of size *size shingled*, which is calculated as the maximum possible number of shingles of the size *shingle size* formed from binary symbols. Since all of the shingles are binary we can treat them as binary numbers that can point us to the index of the location of the weight of that shingle in the original sketched string. This is the shingled string that is used for the hashing procedure to produce the integer hash codes for the Hash Tables section of Figure 3.7, and that is the string that represents the pre-processed data sent to the Similarity node.

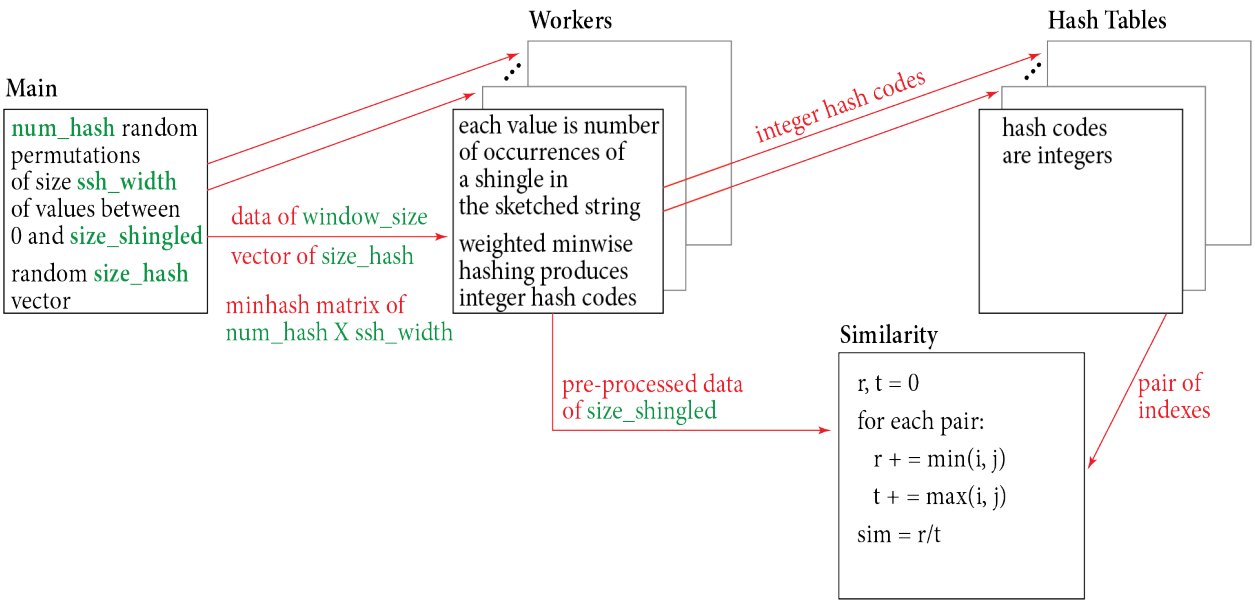

Figure 3.7: SSH Implementation. This diagram shows the extra details of the SSH implementation, beyond what is included in the common architecture from Figure 3.4. Parameters are named in green with the data flow in red.

The difference for the SSH hash table creation is that the hash index is always an integer, so we use a separate hashing node code from ABC and SAX that have a subsequence as the hash value index. For the hashing, we use a minhash procedure, so the necessary parameters become the number of hash functions *num hash* and their lengths *ssh width*, which becomes the width of the minhash matrix, where each row is generated as a random permutation. The weighted minwise hashing procedure is implemented as described in Section 2.2.5.

Time and Communication Complexity

The difference in communication as compared to the other two methods is the *size shingled* factor for the size of the pre-processed data to be passed and stored. Hash function creation and communication is $numhash \times sshwidth \times n$ time but the hash values are integers.

For pre-processing with the Sketch and Shingle, the time complexity is $(sizesketched \times sizehash \times sizehash + nshingles \times numsymbols \times numsymbols + sizeshingled) \times n$. SSH requires the formation of the minhash matrix, which can be done as a single pass of its size $sshwidth \times numhash$, and the sketch step requires a vector generated of size $numhash \times vectorsize$. These are then communicated once to every worker.

3.3 Similarity and Distance

ABC method [Safavi et al., 2017, 2019] presents the following connection between its similarity and distance measures. The similarity s of two time series x and y is defined as:

$$s(x,y) = \sum_{i=1}^{p} \sum_{b=0}^{k_i} (1+\alpha)^b$$

where p is the number of matching subsequences, k_i is the ith matching subsequence length and α is the consecutiveness parameter. The maximum similarity S for two time series of the same length n is:

$$S(x,y) = \sum_{i=0}^{n-1} (1+\alpha)^i$$

Two time series with the maximum similarity have $s = S$ and also a distance $d = 0$, thus:

$$d = S - s$$

Another way to express the similarity measure is to form a ratio of the form $\frac{s}{S}$, thus scaling the value to be in the range between 0 and 1. This will also match nicely for comparison to the SSH Jaccard measures [Luo and Shrivastava, 2016] that are a ratio between 0 and 1, with two identical elements having a similarity (maximum possible for that length) of 1 and therefore distance of 0:

$$J(x,y) = \frac{\sum_{i=1}^{D} \min(x_i, y_i)}{\sum_{i=1}^{D} \max(x_i, y_i)} = \frac{|X \cap Y|}{|X \cup Y|}$$

$$s(x,y) = 1 - J(x,y)$$

for two sets X and Y both of size D. SAX on the other hand only provides a distance measure with the following definition [Lin et al., 2003]:

$$d = MINDIST(X,Y) = \sqrt{\frac{n}{w}} \times \sqrt{\sum_{i-1}^{w} (dist(x_i, y_i))}$$

where n is the total length of the time series, w is the word length and $dist(x,y)$ are symbol-specific distance looked up from a distances table like Table 3.3 for 4 symbols.

The same procedure is used for SAX, as for ABC, to derive its related maximum similarity S and actual similarity s. The minimum distance is 0, this is the case when the two items match exactly and have the maximum similarity, also S = s in this case. In the opposite case, when the distance is maximum, the actual similarity is 0, which means that d = S. Thus for 4 symbols:

$$S = \sqrt{\frac{n}{w}} \times \sqrt{1.34w}$$

The value of 1.34 corresponds to the distance of the most distinct symbols a and d in Table 3.3. So then:

$$s = \sqrt{\frac{n}{w}} \times \sqrt{1.34w} - d$$

Our implementation will use the scaled similarity measures s/S, derived by the conversion of $s = S - d$.

3.4 Evaluation Considerations

Having described the implementation details, we move on to reporting the evaluation results in Chapter 4. The properties that we are interested in are the time of execution, the number of distance computations performed and the quality of the similarity connections discovered. This meant that our implementation included tracking these values and having an easy way to run our system with a range of parameter values. Specifying the parameter values that cannot be calculated from other parameters was done from outside the main implementation with the FG-MPI wrapper, which also included setting the number of parallel processes and cores to use.

Chapter 4

Evaluation

In this chapter we evaluate the three LSH techniques we implemented with regards to the airline flight data. Before describing the results of the experiments and comparing the techniques, we first describe in Section 4.1 the real flight data used in the experiments. Then in Section 4.2 we describe the valid parameter ranges and follow with the results of anomaly detection in Section 4.3. We conclude the chapter by discussing the network discovery results and drawing conclusions on the three methods under our investigation in Section 4.4.

4.1 Data

The data of interest are univariate time series that come from actual sensor readings aboard an operating plane. Each time series is a sequence of measurements taken at a constant sampling rate, representing the state of the measured quantity. The data is released by NASA as the DASHlink project [Matthews, 2012]. We use a portion of this data, in particular the *Flight Data for Tail 668* shared by Matthews [2012]. The full data set has 186 parameters over hundreds of flights over the course of 3 years of a single type of a regional commercial jet. The data for each flight has detailed measurements on aircraft dynamics, system performance and other engineering parameters, with all information that could lead to tracing of a particular manufacturer or airline not included.

We have further limited the data set to 9 flights with data for each containing time series for 28 sensors on board the plane for the entire duration of the flight (sampling at 1 through 8 times per second). The sensors are, for instance, exhaust gas temperatures, oil temperatures, pitch angle, air temperature, weight on wheels and phase of flight. For the full list of the parameters with their units and sampling rates see Table 4.1. The phases of the flight are recorded as a categorical variable with values of 0: unknown, 1: pre-flight, 2: taxi, 3: takeoff, 4: climb, 5: cruise, 6: approach, and 7: rollout. Of special interest are the variables in groups of four, that correspond to sensors in the four engines. We will particularly focus on turbine fan speeds of the four engines. These are named as n_{11}-n_{14} in Table 4.1 and these sensor names will be used in the remainder of the thesis. We expect there to be very high correlation in the fan speeds and the data set contains one flight with an anomaly in engine 3 (n_{13}) which decouples it from the correlation set. This will be used for the evaluation of the anomaly detection results in Section 4.3.

Variable Name	Description	Units	Rate
aoa_1	angle of attack 1	degrees	4
aoa_2	angle of attack 2	degrees	4
egt_1	exhaust gas temperature 1	degrees	4
egt_2	exhaust gas temperature 2	degrees	4
egt_3	exhaust gas temperature 3	degrees	4
egt_4	exhaust gas temperature 4	degrees	4
$elev_1$	elevator position left	degrees	1
$elev_2$	elevator position right	degrees	1
n_{11}	fan speed 1 lsp	%rpm	4
n_{12}	fan speed 2 lsp	%rpm	4
n_{13}	fan speed 3 lsp	%rpm	4
n_{14}	fan speed 4 lsp	%rpm	4
oit_1	oil temperature 1	degrees	1
oit_2	oil temperature 2	degrees	1
oit_3	oil temperature 3	degrees	1
oit_4	oil temperature 4	degrees	1
ph	flight phase from acms	categorical	1
pt	total pressure lsp	mb	2
$ptch$	pitch angle lsp	degrees	8
$rudd$	rudder position	degrees	2
sat	static air temperature	degrees	1
tas	true airspeed lsp	knots	4
tat	total air temperature	degrees	1
vib_1	engine vibration 1	in/sec	4
vib_2	engine vibration 2	in/sec	4
vib_3	engine vibration 3	in/sec	4
vib_4	engine vibration 4	in/sec	4
wow	weight on wheels	in/sec	1

Table 4.1: Flight Sensors

Since the raw data is in terms of timed events and each sensor has its own sampling rate, the data was first normalized to be of the same frequency – some values had to extrapolated and some were interpolated. This resulted in time series with equal time steps.

4.2 Tuning Parameters

In this section we describe the valid ranges of the parameters that are available for investigation for the three LSH methods. The summary is included in Table 4.2.

Method	Parameter Name	Range of Interest
all	*window size*	flight phase length
all	*num hash*	5 - 15
ABC	α	0 - 1
ABC	*size hash*	$\sqrt{window\,size}$
SAX	*word length*	≥ 3
SAX	w	$window\,size/word\,length$
SAX	*size hash*	$\sqrt{w}$
SAX	*num symbols*	3 - 10
SSH	*vector size*	2 - 10
SSH	*vector overlap*	half of sketch length
SSH	*size sketched*	$(window\,size - vector\,size)/vector\,overlap + 1$
SSH	*shingle length*	2 - 10
SSH	*shingle overlap*	half of shingle length
SSH	*size shingled*	$2^{shingle\,length}$
SSH	*ssh width*	1/2 - number of shingles

Table 4.2: Range of Parameter Values Used in Experiments

4.2.1 Common parameters

The *window size* is the most flexible parameter common to all methods. When looking at airline data we have many options to setting the size of the window to consider. For an online streaming application, we want to keep the size small enough for fast processing and to keeping up to date with the progress of the flight, but at the same time with enough data to draw conclusions. When having access to all of the flight's data for processing after the flight, we can harvest as much data as possible about a certain time period. Each phase, for example, dictates different behaviour of the plane and each flight has its own duration and ordering of these phases, so to keep results comparable between flights, the window can be set to be a phase of each flight.

The second common parameter for ABC and SAX is the length of the pre-processed subsequence that is used as the hash code – *size hash*. Authors of the ABC method recommend using the square root of the total length of the full sequence to be hashed as the length of the subsequence [Safavi et al., 2017, 2019]. Following this recommendation, we thus use the square root of the *window size* for ABC and the square root of w for SAX.

The common parameter that is used to control the false negative rate is the number of tables *num hash* to be used as a threshold for the AND group construction. We calculate this value according to the ABC method (Section 4.2.2) and re-use the same value for SAX. SSH required a larger value to produce useful results, as discussed in Section 4.3.3.

4.2.2 ABC

The *alpha* parameter for ABC controls the emphasis we want to place on the importance of bits matching in a sequential order, when calculating the distance between two sequences. The valid range is between 0 and 1, with 0 turning the ABC similarity measure into the inverse of Hamming distance, where the consecutiveness does not matter. The authors of the method suggest to use a value that is much smaller than 1, with the range around $0.01 - 0.0001$ being better suited. This parameter also controls the maximum possible similarity value in the absolute sense. This will be further explored in Section 4.3.

Optimizing the False Positive Rate

The presentation of the ABC-LSH method includes a discussion about controlling the accuracy of results through the false positive and false negative rates [Safavi et al., 2019]. According to the calculations in the paper, if we focus on the AND construction with 5 tables, this gives us 90% chance of clashing of items closer together than 2% of maximum similarity. Since we are working with the approximate nearest neighbours, we optimize the false positive rate, as we want to avoid calculating the distance to items that are not the closest neighbours. At the same time, we do not want to miss the nearest neighbour if it exists and we might not be able to predict how close it might be. This number of tables gives at most 10% probability of items being placed in the same bucket when they were more than 37% of maximum distance apart. These parameters are also specific to the length of the full sequence we want to compare (*window size*) and the similarity parameter *alpha* that we set.

The following information is used to calculate these values:

- An LSH family is (d_1, d_2, p_1, p_2)-sensitive (as defined in Definition 2.2.1)
- False negative rate is $(1 - p_1)$
- False positive rate is p_2
- LSH family under ABC metric is [Safavi et al., 2019]:

$$(d_1, d_2, p_1 = 1 - \alpha \frac{d_1}{(1+\alpha)^n - 1}, p_2 = 1 - \alpha \frac{d_2}{(1+\alpha)^n - 1})\text{- sensitive}$$

- AND construction with r tables turns this family to be (d_1, d_2, p_1^r, p_2^r)-sensitive [Safavi et al., 2019]
- OR construction with b tables turns this family to be $(d_1, d_2, 1-(1-p_1)^b, 1-(1-p_2)^b)$-sensitive [Safavi et al., 2019], however, we do not use the OR construction in the thesis for the accuracy tuning.

4.2.3 SAX

While technically, the *word length* and *num symbols* could be any integer larger than 1, these should act by reducing dimensionality in a meaningful way without loosing detail. For focusing on our flight data set with the *window size* being about 400 for a main phase of the flight, and the hash size for ABC being around 20, we want our *word length* to keep the patterns of that size. To keep the results comparable between methods, we set the *word length* to 3 for all trials.

The authors of the SAX approach provide recommendations on the *num symbols* that are appropriate to use for most data to be 3 to 10 symbols [Lin et al., 2003]. The effect of the number of symbols we explore in the experiments in Section 4.3.

4.2.4 SSH

To keep in line with the conclusions drawn on the parameter settings for the previous two methods, we keep the sizes of the sketching vector and *shingle length* relatively small, with the overlaps being approximately half of their lengths. The most effective length of the permutations is to have the full permutation, increasing the chance for all sets having a hash value. Depending on the data, it might be possible to have permutations of half or even quarter of total number of possible elements, with either much larger number of tables or for data that is less sparse, so that there are enough formed candidate pairs to draw conclusions.

	Regular Flight				**Anomaly Flight**			
	Variable 1	*Variable 2*	*Similarity*	*Buckets*	*Variable 1*	*Variable 2*	*Similarity*	*Buckets*
ABC	n11	n12	1	10	n11	n12	1	10
	n11	n13	1	10	n11	n14	1	10
	n11	n14	1	10				
SAX	n11	n12	1	10	n11	n12	1	10
	n11	n13	1	10	n11	n14	1	10
	n11	n14	1	10				
SSH	n11	n12	1	8	n11	n12	1	7
	n11	n13	1	8	n11	n14	1	7
	n11	n14	1	8				

Table 4.3: Sample Results for Anomaly Detection in Table Form. This table represents all of the discovered edges between sensor time series between n_{11} and other fan speed sensors. The anomaly flight is missing all of the edges with n_{13}.

4.3 Anomaly Detection

Eight flights of similar length and phase structure were selected out of the data set along with the flight with the engine anomaly. The anomaly was present in phase 4, with the fan speed of engine 3 n_{13} not correlating with fan speeds of the other engines – n_{11}, n_{12}, and n_{14}. The correlations and all LSH measures are symmetric. All three LSH methods were able to detect the anomaly, with the other eight flights showing all four engines highly similar and thus connected, and the anomaly flight missing the connections for one of the engines. For LSH this implies that for the anomaly flight the variable n_{13} was not hashed to the same buckets as the other n_{1x} variables.

One way to identify this anomaly is by looking at the resulting discovered network and comparing a regular flight to the flight with the known anomaly. Visually, the discovered networks with all 28 nodes are too difficult to visualize, so for the task of detecting this specific anomaly with n_{13} we will only include the part of the network with the fan speeds. Table 4.3 shows this part of the network for all three methods for a regular flight and the flight with the anomaly. Table 4.3 represents the network as a list of edges, where each edge is represented by a row of the table, with two variable names the edge connects and two options for its weight – either the scaled similarity or the number of shared buckets. Due to the transitivity and symmetry of being placed in the same bucket by LSH, it is only necessary to look at all of the edges for one of the engines. It can be inferred that all the variables having an edge with n_{11} and sharing all or most of its buckets, will also be connected by an edge with each other. The edges can have as the weight the scaled similarity value (see Section 3.3) or the number of buckets that the pair shares as given in the Table 4.3. Another way, used in further sections, is to represent this same information graphically – with each variable as a node, and with each similarity value as the edge weight. As in Table 4.3 and all other representations of successful anomaly detection, one can note how n_{13} is missing from the list of edges and does not have any edges in the graph for the anomaly flight.

All trials were based on phase 4 (climb) of the flight as the window and were run with 30 processes (10 main processes, with 3 sub-processes each). All experiments were performed on a Linux machine (Pop!OS 20.04 LTS 64-bit) with an Intel (i7) 5.1 GHz 8 core CPU (16 threads) and 16GB memory. All trials over parameter ranges were plotted by time of execution (in milliseconds) and resulting number of distance calculations. The total possible number of distance calculations for our data set is $n^2 = 784$. However, since the similarity measures are symmetric, the total number of calculations that produces a fully connected graph between all the nodes is $\frac{n \times (n-1)}{2} = 378$.

4.3.1 ABC

As noted in Section 4.2, *size hash* was set as the square root of the *window size*. The *num hash* necessary to obtain the required accuracy was 5 (in the AND group). The total number of hash tables used was 10 to account for the OR group formation, where matching on 5 was enough to be included in the output graph and similarity calculations. The correlated engines were placed in the same bucket for all of the 10 tables, however, we ran exploratory trials over values of the α parameter, which we present below in Table 4.4.

0	0.0001	0.001	0.01	0.1	0.25	0.5	0.75	1

Table 4.4: α Values Investigated

Results from varying α

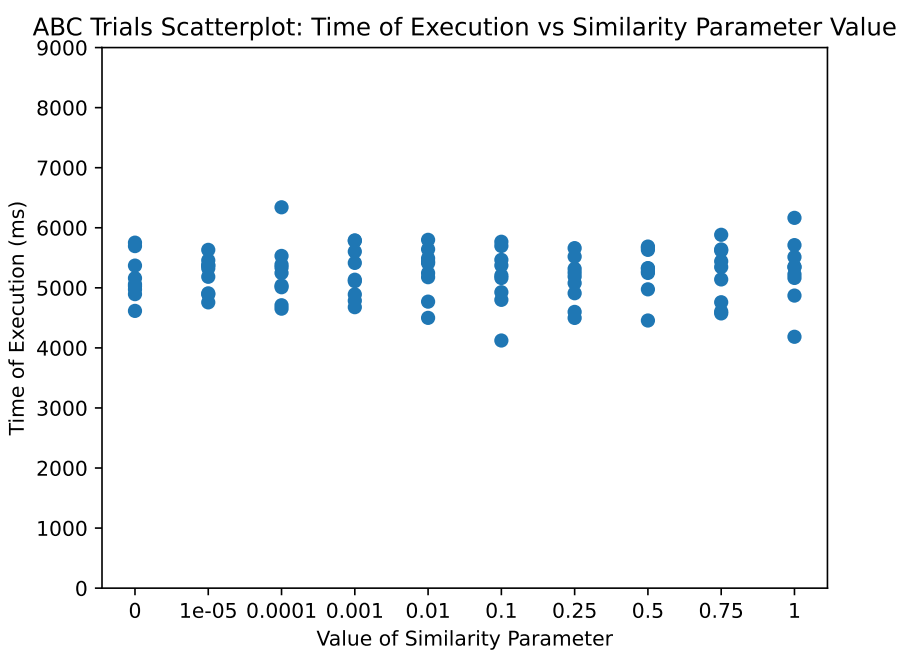

Figure 4.1: ABC Trials over Alpha Values vs Time of Execution

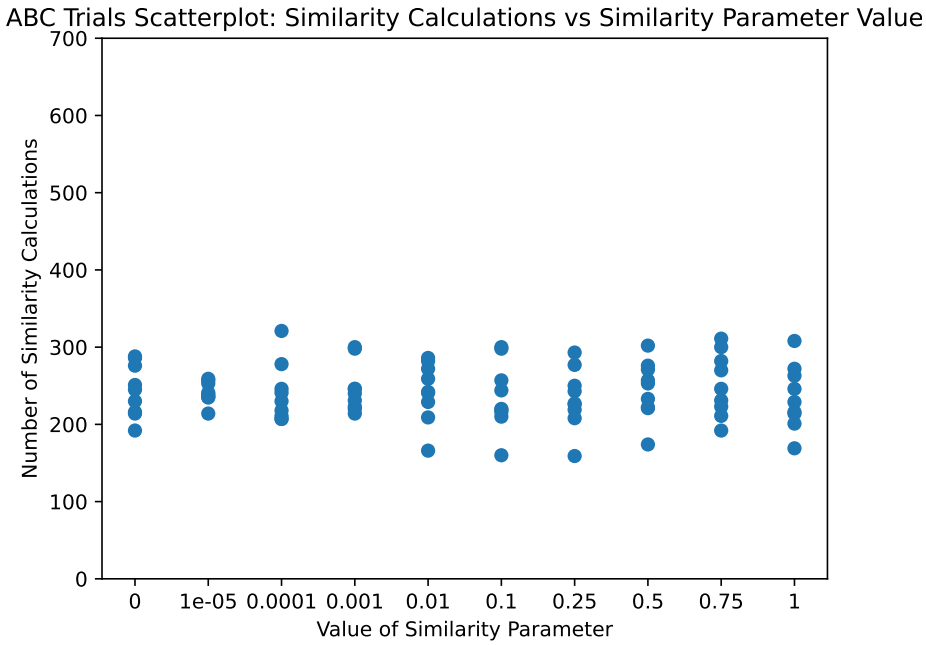

Figure 4.2: ABC Trials over Alpha Values vs Number of Similarity Calculations

All of the trials were able to clearly point out the anomaly. We have varied the values of α over the following values in the range $[0,1]$: 0.0 0.00001 0.0001 0.001 0.01 0.1 0.25 0.5 0.75 1. These are shown on the x-axis of the scatter plots in Figures 4.1 and 4.2. The 90 trials in total over the 9 flights are presented in the scatter plots. Figure 4.1 is graphing the time of execution in milliseconds (ms) for all trials in the y-axis and Figure 4.2 is using the total number of similarity calculations performed for each trial. There were no trends in time of execution or number of distance calculations with respect to the value of α.

A significant trend correlated with the values of α is the increase of absolute similarity values. For example, for two binarized time series with 217 out of 217 matching bits, the absolute similarity value was 217 at $\alpha = 0$ (behaves as an inverse of Hamming distance), went up to 242 at $\alpha = 0.001$ and then to 3.25^{38} for $\alpha = 0.5$. These also correspond to the maximum similarity values for that flight, as the phase was 217 time steps long and the two time series in question were highly correlated. Thus the scaled versions of these similarity values did not change over the value of α and remained constant at 1, as shown in Figure 4.3 in green.

We chose other two pairs of time series that showed smaller similarities in terms of inverse Hamming distance for this window of that same length – graphed in blue and red in Figure 4.3. The value for $\alpha = 0.001$ is missing from the graph, as the blue pair did not match in at least 5 tables to be included in the similarity calculations in this trial. This can happen because of the random nature of the hash functions, especially for two time series that are poor candidates for nearest neighbours.

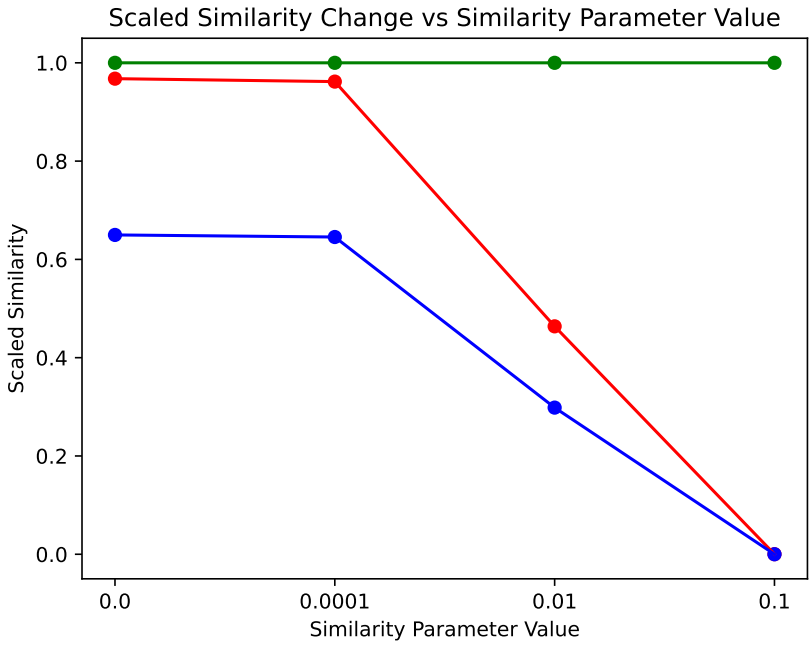

Figure 4.3: ABC Scaled Similarity Change Trend with Varying α Parameter

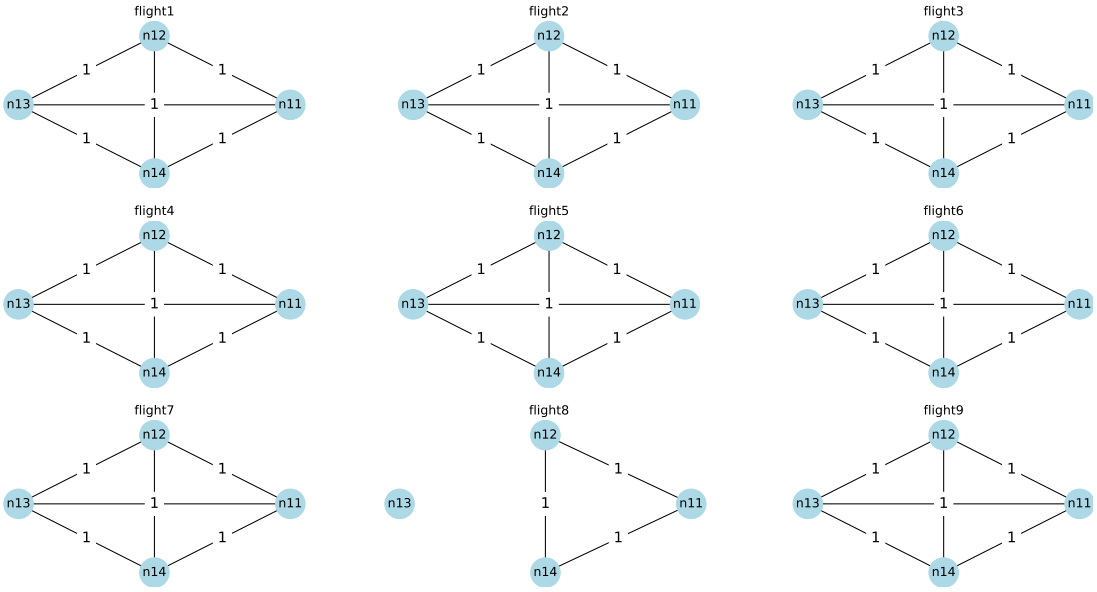

Figure 4.4: ABC Best Trial for Anomaly Detection Network Results With Scaled Similarity Values

The number of matching symbols for these were 210 and 141, respectively. Although the absolute values of the similarity explode as α increases, the maximum similarity for that length at that value grows even faster. As shown in Figure 4.3, over increasing α, the difference between the scaled similarity values for the red and blue pairs decreases, both ending up extremely close to 0 at $\alpha = 0.1$.

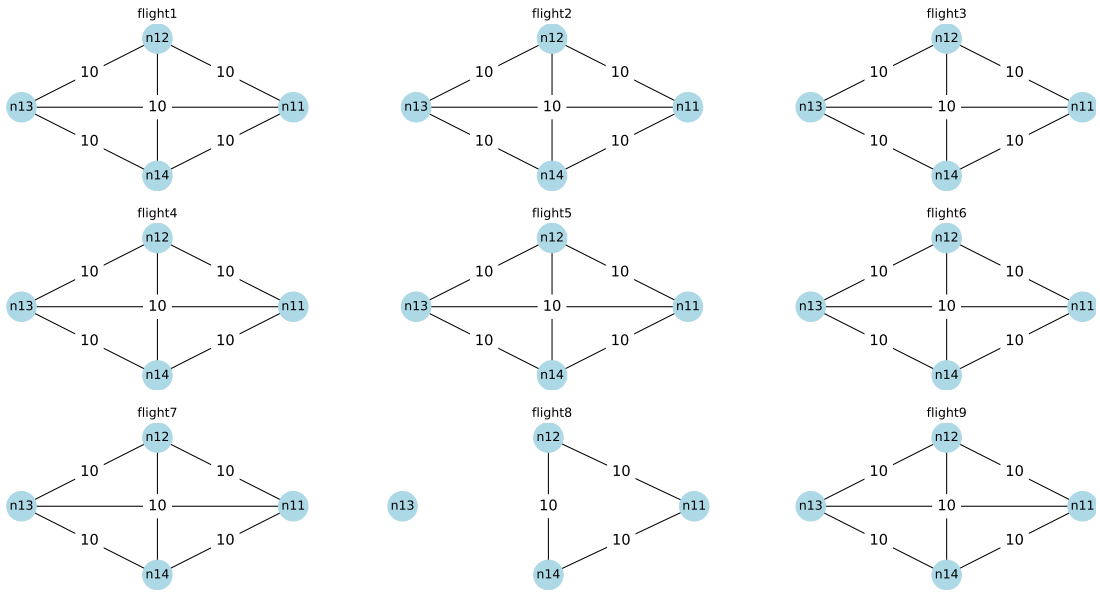

Figure 4.5: ABC Best Trial for Anomaly Detection Network Results with Table Values

For the final results, we choose $\alpha = 0.001$. The resulting networks for the four engine fan speeds for the 9 flights are shown in Figure 4.4, with the scaled similarity values as the edge weights, and in Figure 4.5 with the number of tables where the variables shared buckets. As mentioned previously, the anomaly is easy to see in the graph in the second column of the third row where n_{13} is not connected to the other nodes. All remaining edges are at maximum similarity and are sharing buckets in all of the 10 tables that were created.

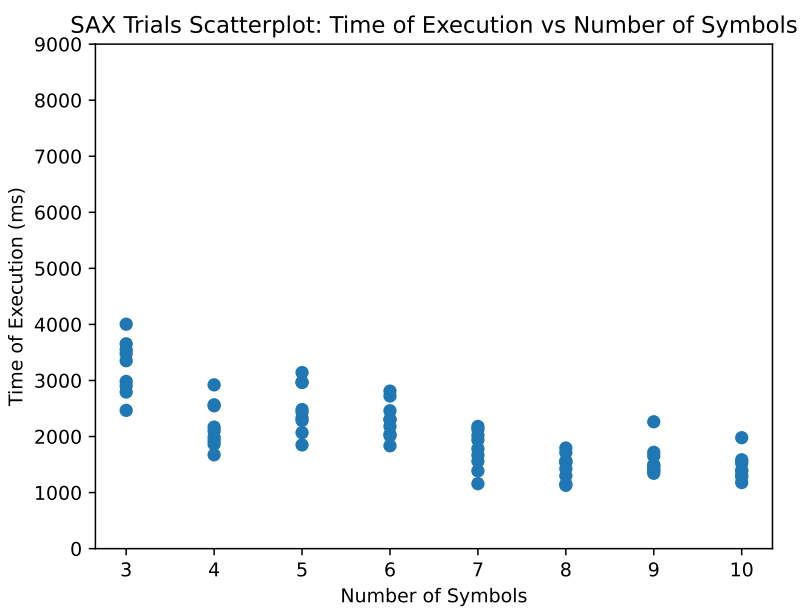

Figure 4.6: SAX Trials over Number of Symbols vs Time of Execution

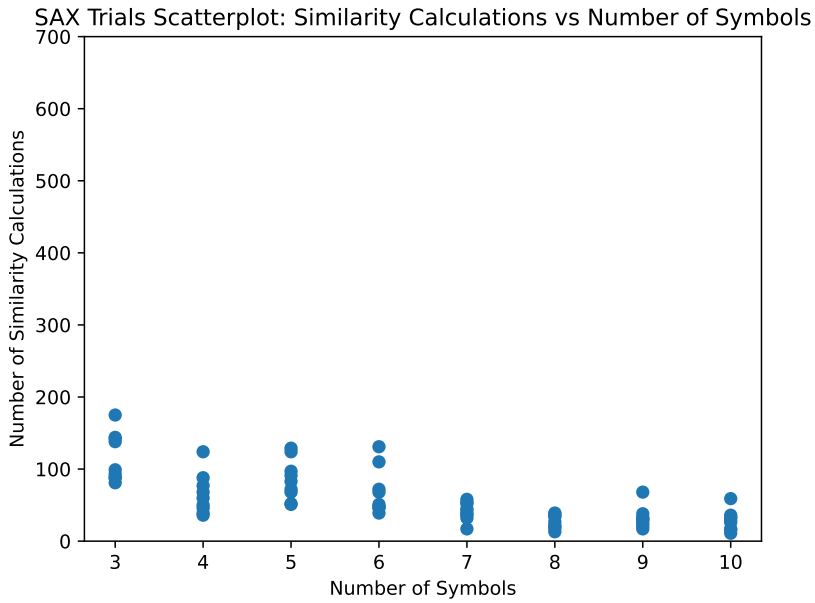

Figure 4.7: SAX Trials over Number of Symbols vs Number of Similarity Calculations

4.3.2 SAX

The length of hash *size hash* was set as the square root of the length of the full series after pre-processing (the number of words w). As in ABC, we used 10 tables and a threshold of 5 to be included in the output. Word length was set to 3, to not cause too much dimensionality reduction compared to the series length used by ABC, and values between 3 and 10 were investigated for the number of symbols parameter.

Number of symbols

Although, SAX with 2 symbols would be the most comparable to ABC, this would cause all of the distances to be zero for any two time series, due to the specifics of the distance measure, as discussed in Section 2.2.4. Larger values between 3 and 10 were thus investigated. The more symbols used, the less was the execution time, which may be due to the smaller number of distance calculations. The results are presented in Figures 4.6 and 4.7 respectively, as scatter plots of all 72 trials.

For our specific *word length* and *window size*, trials with 5, 6 and 7 symbols showed the clearest detection of the anomaly. For other values, smaller or larger, the anomaly flight was still missing the connections to n_{13}, however, some of the non-anomalous flights were missing some of the edges too. Figure 4.9 shows the 9 resulting networks for the engine fan speed cluster, with number of symbols set optimally to 6 with table values as the network edge weights. Figure 4.8 shows the same network structures with the scaled similarities as the weights. Interestingly enough, all the present edges are at maximum similarity and are sharing buckets for all 10 tables that were created.

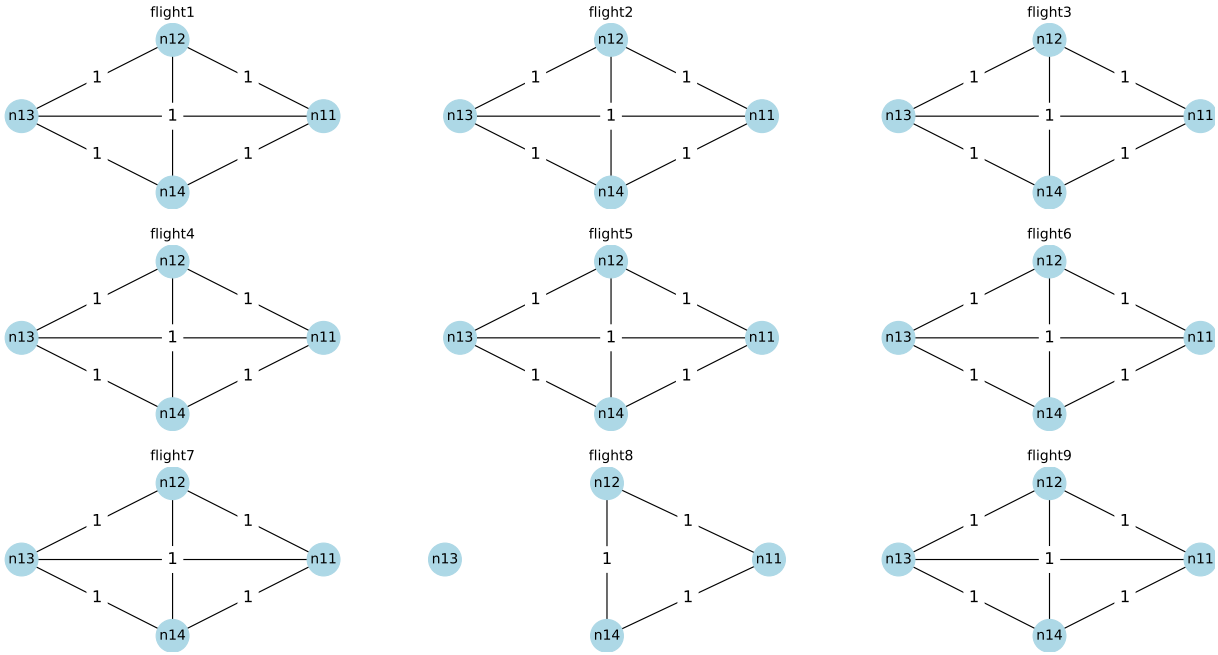

Figure 4.8: SAX Best Trial for Anomaly Detection Network Results with Scaled Similarity Values

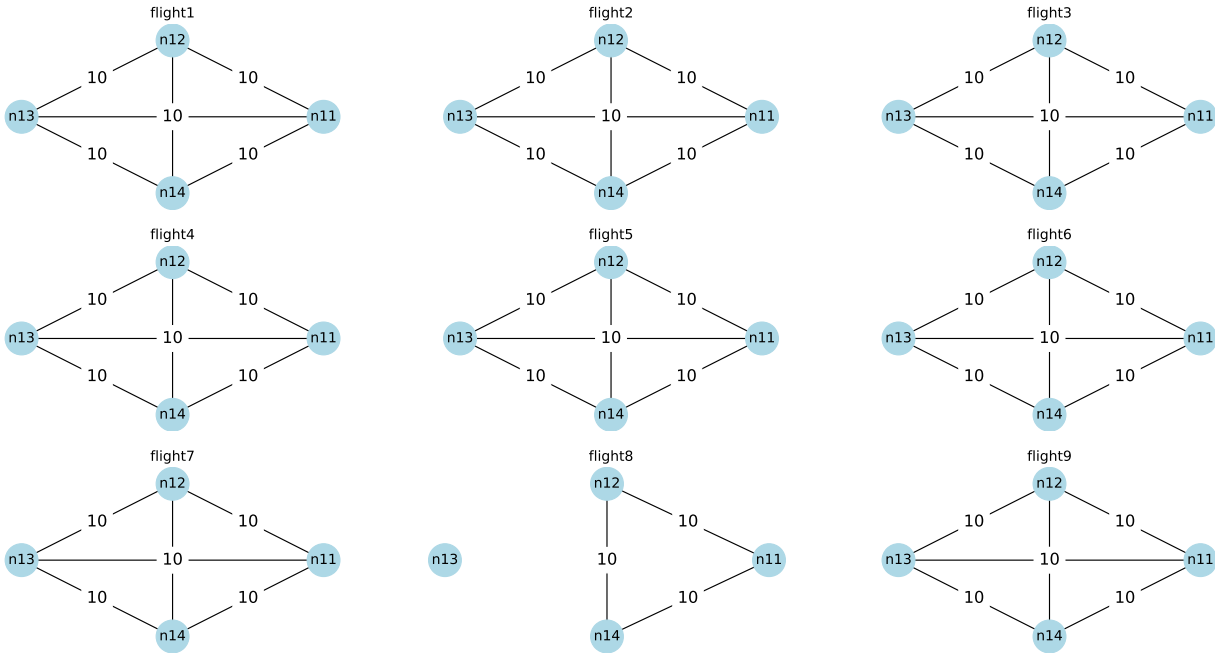

Figure 4.9: SAX Best Trial for Anomaly Detection Network Results with Table Values

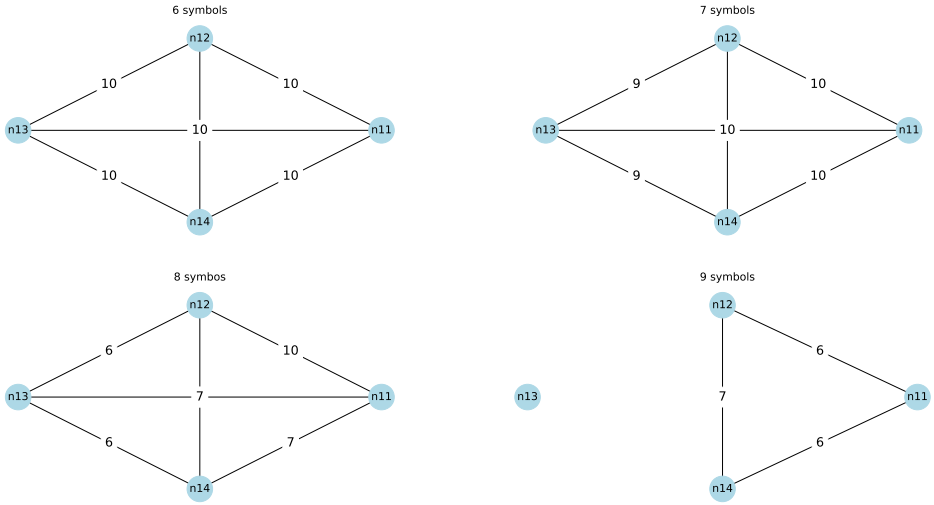

Figure 4.10: SAX Effect of Number of Symbols on Network Discovery

With the increasing number of symbols from 8 on, the number of shared buckets between even highly-correlated variables decreases, even though the distance remains the same (0). This is because the hash comparison is exact on symbol but the distance measurement for symbols differing by 1 is counted as 0. The more symbols there are, the more nuanced differences are detected in the symbolic representation which do not translate into an increase in the distance value. An example of this effect is shown in Figure 4.10, with 4 networks for the same flight, with varying number of symbols used from 6 to 9. One can see how the network weights decrease from all being 10 for the parameter value of 6, and to 6s and 7s for the value of 8, and then finally for the value of 9 the edges drop from the graph completely as they no longer meet the threshold of at least 5 collisions between variables.

4.3.3 SSH

We used 15 tables (5 for AND group) and the total number of possible shingles for the permutation length, as discussed in Section 4.2. Using fewer tables and shorter permutations did not provide enough edges for the engines to be correlated, due to the random nature of minhash and the sketching vector – too many of the time series did not receive many hash values. For data that is of a different size and sparsity this might not be an issue. Overlaps were set to half of the *vector size* or *shingle length*. Trials were run over values for shingle length and sketch vector size for ranges of 2 to 10, with each investigation thus running over 81 trials in total.

Figure 4.11: SSH Trials over Shingle Lengths vs Time of Execution

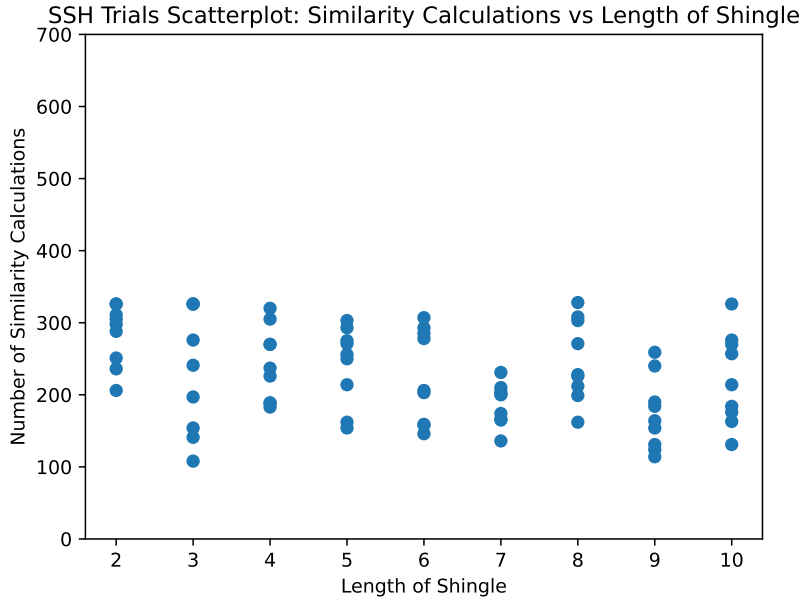

Figure 4.12: SSH Trials over Shingle Lengths vs Number of Similarity Calculations

Size of shingles

We consider shingles of short lengths and overlaps to make the dimensionality reduction small enough to be comparable with ABC and SAX. Increasing the *shingle length* resulted in decreasing time of execution (Figure 4.11) and number of calculations (Figure 4.12), with no notable effects on anomaly detection.

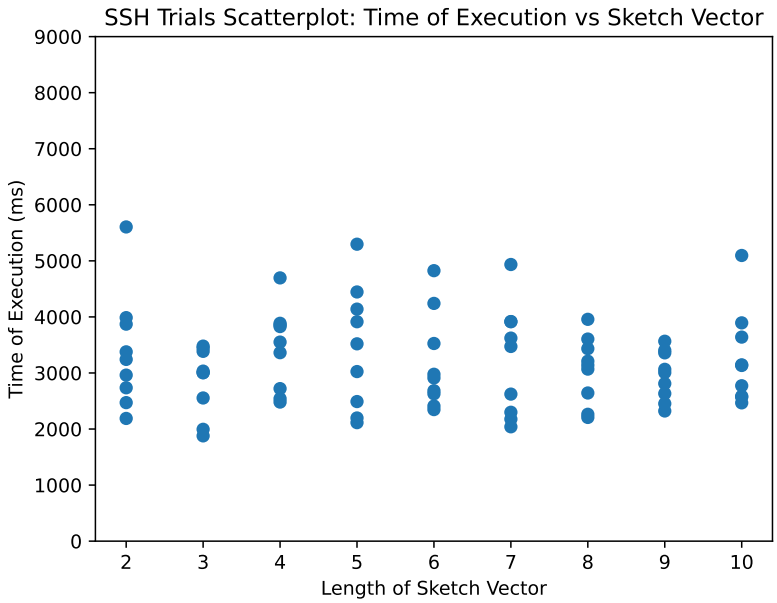

Figure 4.13: SSH Trials over Sketch Lengths vs Time of Execution

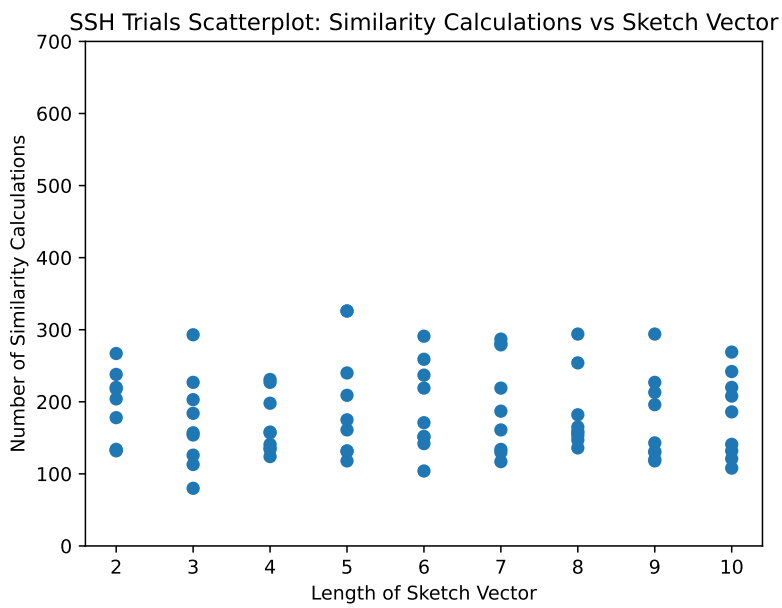

Figure 4.14: SSH Trials over Sketch Lengths vs Number of Similarity Calculations

Sketch Vector Length

Trials over the length of sketching vector from 2 to 10 elements showed no effects on anomaly detection, time of execution or number of calculations, as shown in Figures 4.13 and 4.14.

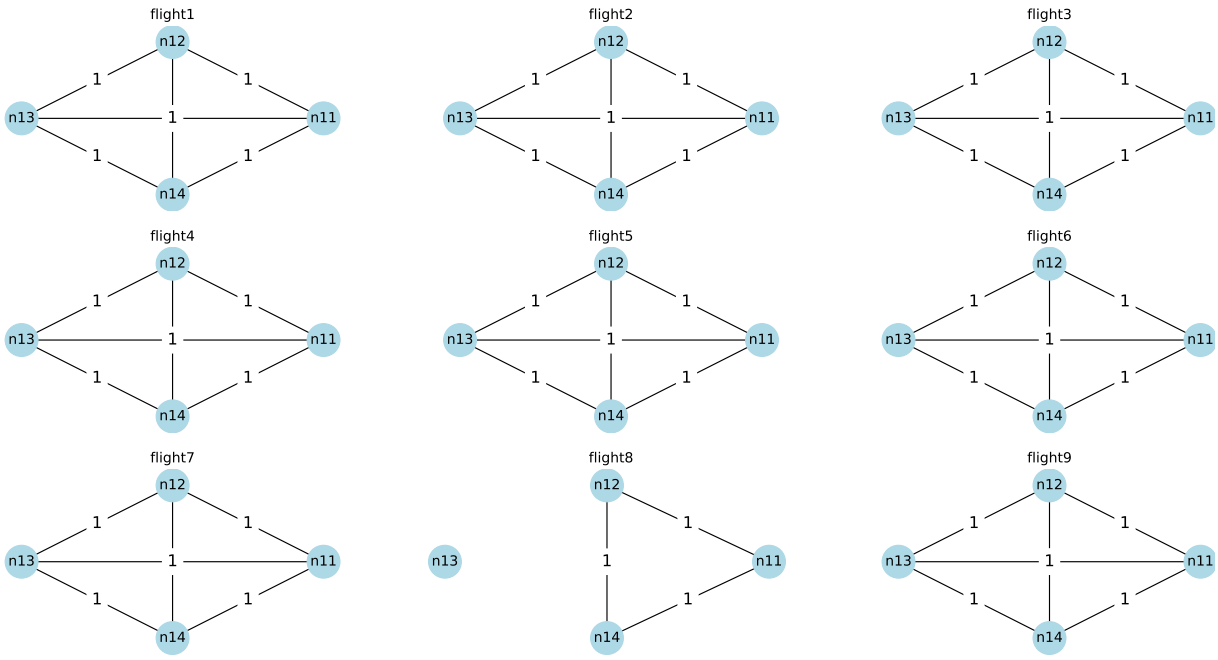

Figure 4.15: SSH Best Trial for Anomaly Detection Network Results with Scaled Similarity Values

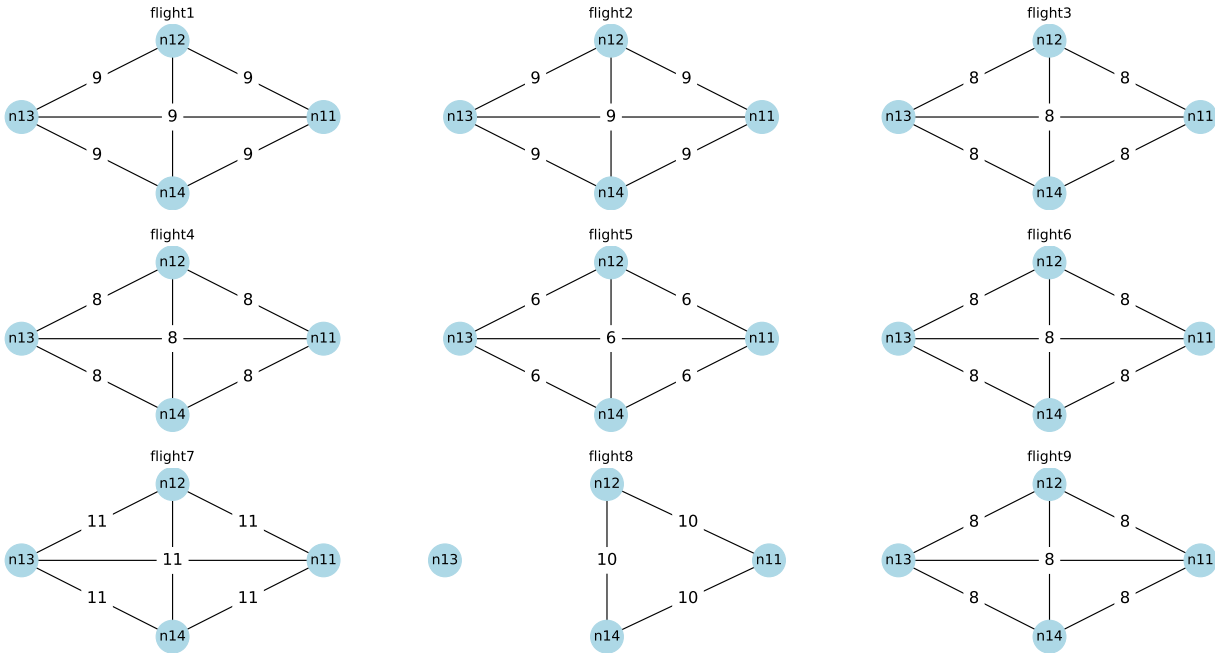

Figure 4.16: SSH Best Trial for Anomaly Detection Network Results with Table Values

For the optimal parameters we set both the sketch *vector size* and *shingle length* to 10, with the overlaps of 5. Results are shown in Figures 4.15 and 4.16 with scaled similarity and table values for edges respectively.

		Time of Execution (ms)			Number of Similarity Calculations		
		minimum	*average*	*maximum*	*minimum*	*average*	*maximum*
ABC	*Trials*	4123	5221	6341	159	241	321
	Best	4664	**5159**	5677	183	**242**	289
SAX	*Trials*	1130	2062	4004	11	58	175
	Best	1778	**2117**	2493	38	**51**	90
SSH	*Trials A*	2036	4284	7557	108	228	328
	Trials B	1738	3095	4946	78	186	325
	Best	2856	**3527**	4626	112	**196**	298

Table 4.5: Trial Descriptive Statistics for Time of Execution and Number of Similarity Calculations. The *Trials* rows represent all of the trials ran across the investigated parameter ranges over all flights. For SSH, *Trials A* refer to trials with the changing *shingle length* and *Trials B* to trials over *vector size*. An optimal set of parameters for each method was chosen and the program was ran again over all flights, with the results presented in the *Best* rows.

4.4 Discussion

In this section we draw comparisons between the three explored methods and discuss the results.

4.4.1 Parameters

Out of the three methods, ABC was the easiest and most straight-forward to set parameters for the data we used. The authors of the method provided recommendations that lead to successful and clear anomaly discovery as expected from the resulting networks. The SAX method, since it is somewhat more complex with more parameters to set, was a little more involved to make it produce a clear picture. Due to the non-metric nature of the method, the number of symbols used for SAX had an interesting effect on the properties of the resulting network. SSH required increasing the lengths of the permutations and the number of tables to have enough candidate pairs, but once that was set, changing the other parameters did not lead to any further significant improvements. Overall, the methods were mostly intuitive to use with the mathematical properties provided in the original papers.

4.4.2 Efficiency and Sparseness

Table 4.5 presents the descriptive statistics over all trials for all values of parameters used in the experiments, in the form of minimum, average and maximum for time of execution and the total number of similarity calculations. Table 4.5 has separate sections for each of the methods, and the two main trends we were considering. The *Trials* rows refer to the values obtained of running the studies over all 9 flights and over all investigated parameter values. For each method, an optimal parameter set is chosen and its time and number of calculations are recorded again from running over 9 trials – once for each flight. This is recorded in Table 4.5 in the *best* rows.

The average time of execution and number of similarity calculations for the best trials with optimal parameters can be seen in bold in the Table 4.5. ABC has shown to be the slowest method of the three, likely because it also had the largest number of resulting edges in the network – since every similarity calculation becomes an edge. SAX can be seen in Table 4.5 as having the smallest numbers for both time and number of calculations. All of the methods. however, showed some improvement over the 378 similarity calculations that could be required, with SSH's best average being just over half and SAX's best average being much lower.

4.4.3 Anomaly Detection

In terms of anomaly detection, ABC worked the best with the most simple and straight-forward parameter setting required. SAX showed great results for the specific numbers of symbols in the middle of the possible range. However, this was expected to be the case, due to the nature of the symbolization, where using too few symbols to represent the time series results in everything in the set being seen as very similar. Conversely, when using many more symbols, the small noise between even very similar items is enough to cause the distortion in the hash table collisions. Keeping these considerations in mind, both ABC and SAX would be good candidates for this kind of anomaly detection task and for time series similarity tasks in general.

SSH was able to also detect the anomaly, but the nature of the data required careful review of the parameters. For this method to be useful for anomaly detection, from the results of our trials, it would need to be carefully calibrated on either known anomalies or extensive domain knowledge. It could also be the case that this method requires a completely other approach to parameter setting and theoretical analysis than what we have used, where we based the tuning on the ABC process.

SSH however, offers more possibilities and intricacies to explore, as ABC and SAX are relatively simplistic, albeit effective methods. SSH has the potential to produce very sparse networks, targeting more specific patterns through the usage of specific properties in the sketching vector, which we did not explore in this thesis. Also of note, is that for SSH, unlike ABC and SAX, two time series having the maximum possible similarity did not indicate that they would share all of their hash table buckets, due to the random nature of the weighted minwise hashing procedure used (as described in Section 2.2.5).

Running the optimal set of parameters for anomaly detection on the flight part before the phase with the anomaly did not yield any results for any of the methods. This could be due to the methods not being fine-tuned enough or to the nature of the phases that came before the phase 4 (climb) that we looked at in anomaly detection – the operation of the engines in the previous phases might have not shown any issues. This could be an interesting area to further explore.

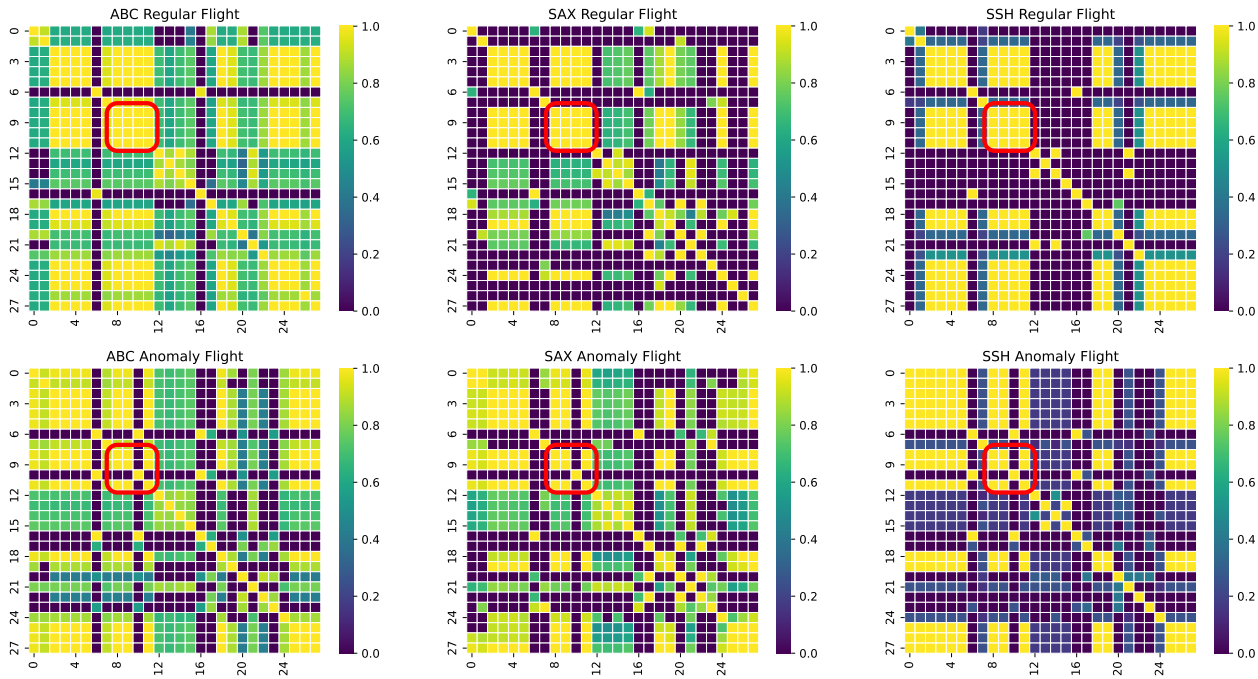

Figure 4.17: Scaled Similarity Values Heat Maps for all Flight Sensors. The region of interest with the anomaly is outlined in red. The anomaly is the loss of power to engine 3, resulting in lower similarity values with fan engine speed variable n_{13}, represented by row 10.

4.4.4 Network Discovery

Another way to visualize the discovered network is to represent it as a similarity matrix. This is useful for visually presenting the network as a heat map or comparing to a correlation matrix. This representation also allows one to easily visually evaluate the network in a snapshot and compare to other networks, as long as they were ordered consistently and shared the same colour coding. The limitation is that visually one can only notice the clusters of variables that happen to be placed next to each other in the matrix. This is the case for the variables that we looked at in the anomaly detection task.

Here we provide full heat maps of the similarity network matrix for two flights in Figure 4.17. The matrices are constructed such that the scaled similarity value for time series x and y is stored in the matrix at point $[x,y]$ (and symmetry is added for visual effect at $[y,x]$). This is then colour coded according to the value in each cell. The top row of Figure 4.17 shows the same regular flight matrix for each of the methods in order – ABC, SAX and SSH. Out of the many yellow clusters we will focus on the cluster that is between variables at rows 8-11, forming a bright yellow square at all three heat maps. This area is outlined in red on all the heat maps in Figure 4.17. The location of this highly correlated group where we expect to see the anomaly is the same for all heat maps. At the bottom row of Figure 4.17, are the matrices for the flight with the anomaly. Engine 3 fan speed n_{13} is the variable index 10 which clearly is now breaking up the cluster as compared to the regular flight patterns heat maps in the top row, and this can be seen in the matrices produced by all three methods.

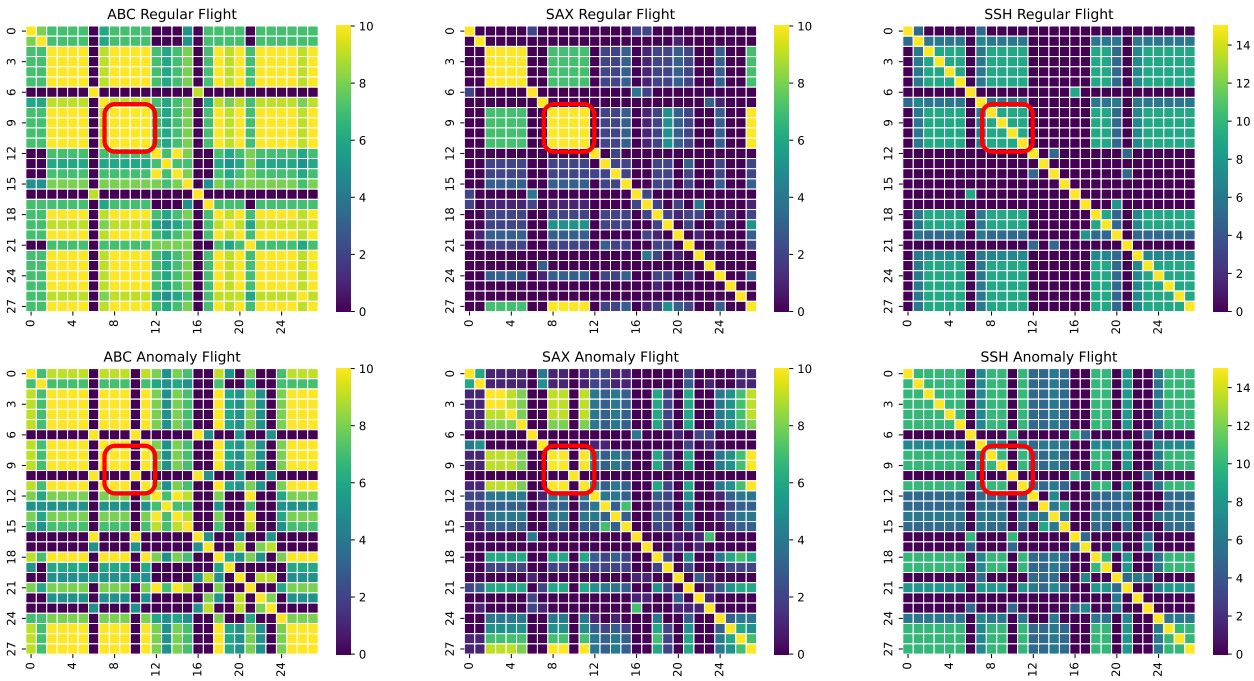

Figure 4.18: Number of Shared Buckets Heat Maps for all Flight Sensors. The region of interest with the anomaly is outlined in red. The anomaly is the loss of power to engine 3, resulting in lower number of shared hash buckets with fan engine speed variable n_{13}, represented by row 10.

An even more drastic change is the dark line at variable 10 across all of the variables, making it one of only a few other uniform lines across all variables, one of which is the identity diagonal. This diagonal was added at the value of 1 for scaled similarity for better visual effect as it obviously was not included in any of the calculated candidate pairs similarity.

In place of the scaled similarity values one could place the number of shared hash buckets for each pair of variables x, y in the cell $[x, y]$ and graph it a similar way. This is done for the same two flights as above with the similarity values and is included in Figure 4.18, with the region of the heat map with the anomaly outlined in red on all the heat maps.

The trend of the number of tables being correlated with the similarity value can be confirmed by looking at the patterns in Figure 4.17 and seeing almost identical patterns in Figure 4.18 for both the regular and the anomaly flight. This is most prevalent for SSH as we know that it's minhash procedure is able to approximate the Jaccard similarity through the number of tables where the two items share a bucket. This is essentially the reason why LSH can be boosted in accuracy with the AND formations as used in this thesis, with a collision frequency-like approach or having a threshold of the number of shared buckets for being included as candidate pairs in the output.

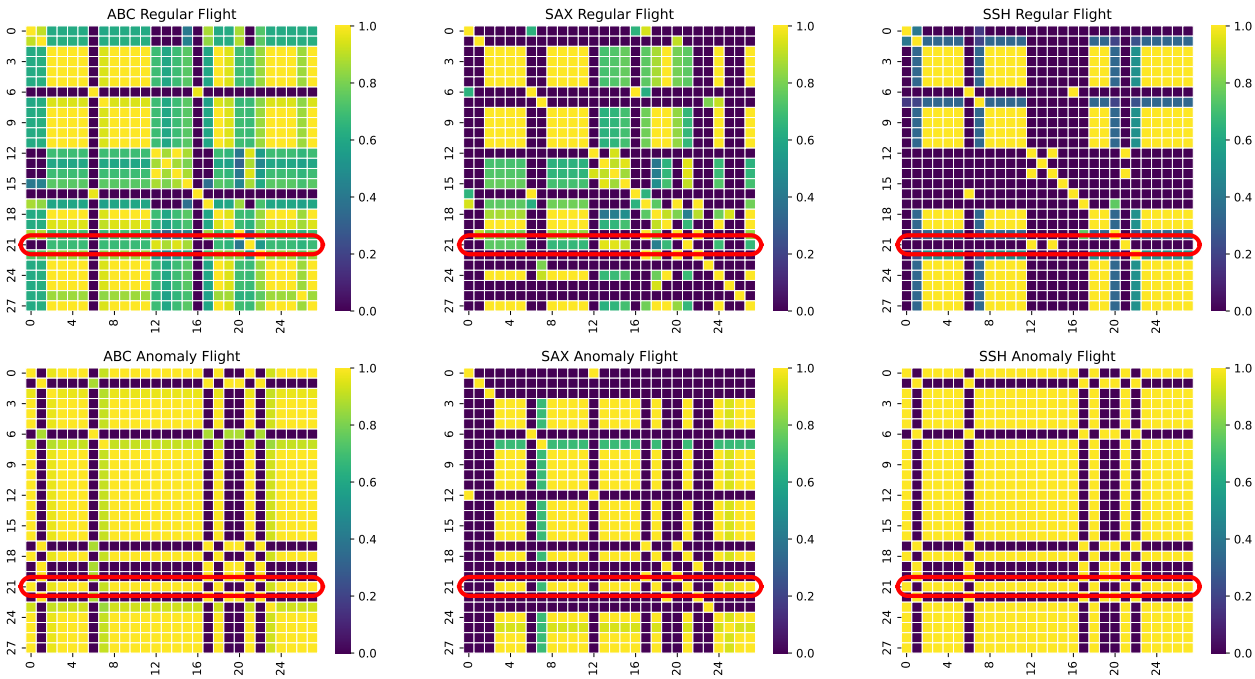

Figure 4.19: Scaled Similarity Values Heat Maps for all Flight Sensors. The region of interest with the anomaly of variable *tas*, row 21, is outlined in red.

Another type of anomaly was found in the data set by investigating the correlations of the variables over whole flights by each pair of sensors. The investigation was done by using Mutual Information Correlation [Reshef et al., 2011], and we tested if our LSH implementations can find the similar patterns. The anomaly discussed prior, mostly involved a decoupling of an engine group – the engine 3 fan speed n_{13} having a lower correlation and therefore similarity to the other ones in the anomaly flight. This anomaly was of a different sort, as the anomalous behaviour included mostly increase in correlations, specifically between the variable *tas* – true airspeed – and all the other ones, as compared to the rest of the flights.

All three methods were able to indicate this other anomaly when running with the optimal set of parameters. This can be seen as an increase in similarity values for the row 21 in the heat maps produced by both scaled similarity values in Figure 4.19 and number of shared tables in Figure 4.20. This row is indicated in red and is in the same location for all the heat maps. An overall increase in similarity across the variables present can be noted too.

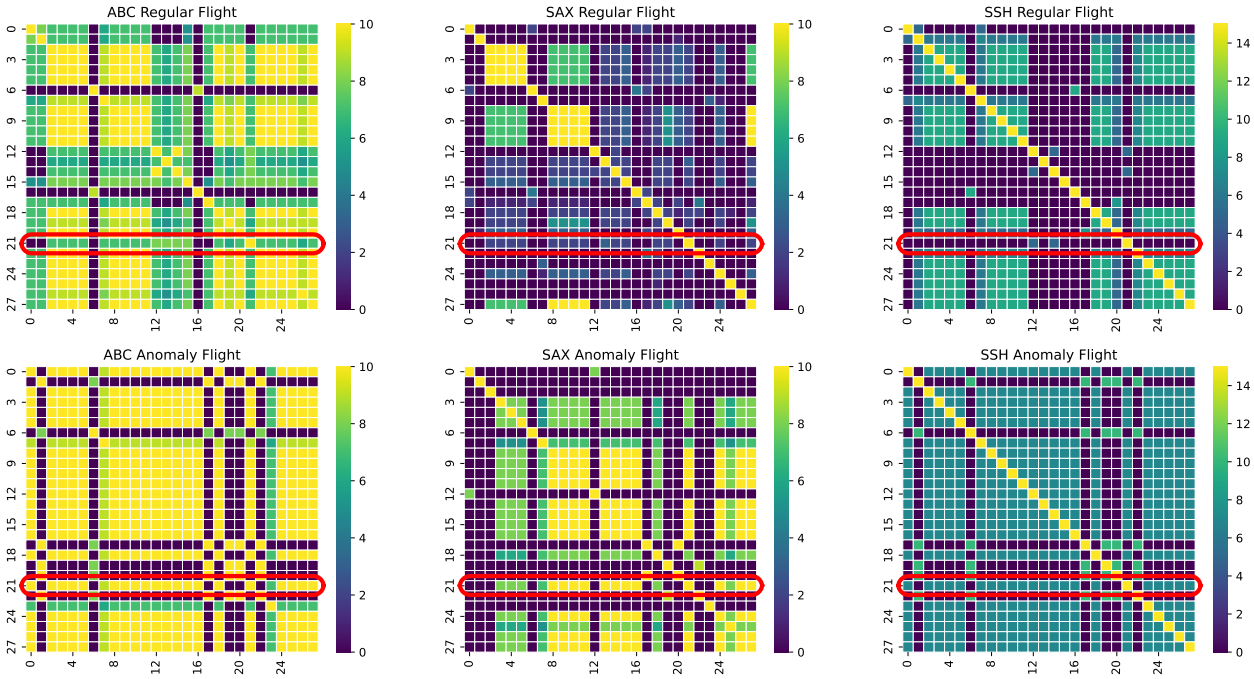

Figure 4.20: Number of Shared Buckets Heat Maps for all Flight Sensors. The region of interest with the anomaly of variable *tas*, row 21, is outlined in red.

4.4.5 Summary of Results

To summarize, ABC was the easiest to use but took the longest time to execute the network discovery with the most number of candidate pairs. SAX showed the best execution time and number of collisions performance once the number of symbols for its pre-processing was set to a value in the middle of the possible range. SSH required the largest number of hash tables and full lengths permutations to have enough data to be usable, but has the potential to better target detection of specific anomalies and produce very sparse networks. Overall, all three methods worked reasonably well in the anomaly detection task by identifying the known anomaly in the resulting similarity network.

Chapter 5

Conclusion

5.1 Contributions

The contribution of this thesis is a novel combination of the approaches we have used – systems, theory and application. From the systems perspective, we have designed and implemented a fast parallel system. For the time series analysis, we have relied on locality sensitive hashing methods that have known theoretical guarantees, a long history of usage, and a scalability advantage. For the application – we have successfully applied our implementation of LSH on real airline data for network discovery and anomaly detection.

We started with implementing minhash in parallel using a message-passing framework with FG-MPI. We then studied LSH approaches suitable for working with time series and formulated a workflow for network discovery with LSH, with the refined process structure of the parallel framework we used with minash. Having multiple worker processes splitting the computation of the hash values and communicating with multiple hash table nodes was key in our approach. The methods of ABC, SAX and SSH all had their own pre-processing of the time series and similarity measures, but the outputs of all trials were similarity matrices that could be later presented as edge lists, similarity networks or heat maps.

The ABC-LSH piqued our interest as it was proposed for usage for network discovery from brain data. The approach had detailed explanations of its mathematical background and clear recommendations on parameter setting, so the tuning of this method was easy and straight-forward to use successfully. We tried multiple α similarity parameter values to see if it had any impact on execution time or the number of resulting similarity calculations. It did not; however, there was an effect on the absolute and relative similarity values. The conclusion we made was to keep α at a 0.001 value, much closer to 0 then to its maximum of 1, although all values provided clear distinction between networks for anomaly detection. ABC also discovered the densest networks and required the most computation time.

For the SAX method, we investigated the number of symbols used for pre-processing the time series. An optimal number was set at 6 symbols, as expected from the general recommendations from the authors of the method. Increasing the number of symbols was resulting in fewer similarity calculations and thus lead to faster time of execution as compared to the other two methods. However, choosing an even larger number of symbols was not practical, as our further investigation showed that the distance measure not being a true metric caused the highly-correlated pairs of time series to not share buckets in tables as frequently, even though their distances remained as small as in the other cases.

The SSH method required the usage of the greatest number of hash tables to produce enough candidate pairs to be usable on our data. On top of this, and setting the permutations lengths for weighted minhash procedure to be of full lengths, any changes to the lengths of the sketch vector and the shingle did not impact the performance.

In the end, our expectations of being able to use our parallel implementations of ABC, SAX and SSH for anomaly detection through network discovery were successful, along with the opportunity for us to draw comparisons between methods and note interesting patterns about certain parameter values impacting results. Our main results are showcasing a parallel system for LSH-based network discovery from time series from a systems perspective, increasing our knowledge to form better impressions and comparisons on the new ABC-LSH approach, and proposing a working system with the potential to address the need for fast scalable systems in the application area of airline data, as well as for similarity tasks and anomaly detection.

5.2 Future Directions

We would have wished to draw more comprehensive conclusions on the scalability of our implementation, but we were unfortunately limited by the size of the data that we chose. Another limitation comes from the data being specific to time series from flight sensor recordings with a very drastic anomaly – the loss of engine power – making our thesis a great example of proof-of-concept of such framework and workflow for the network discovery and anomaly detection aspect, but not as generalizable to other data or other types of anomalies as we would have liked. We also focused in this work more on the forming of the similarity networks rather then using detailed network analysis on the results.

On top of our limitations that open up avenues for future work, there are also opportunities to further explore the parameters of the three methods. Specifically with SSH, we see it as a potential way to produce very sparse networks specialized to a type of anomaly or similarity by changing the form of the sketching vector and shingle formation. For ABC, exploration of the OR group formations could be proposed, along with the extension to going beyond a binary representation. The number of distance calculations could also possibly improve with an increase in number of hash tables and the related increase in accuracy. SAX has the variable *word length*, which we did not spend much time investigating in this thesis. The non-metric nature of the SAX distance measure might be interesting to explore for other types of data. Another possible extension is to test LSH on prediction of anomalies, which has a great need in the application area.

With LSH being used in so many different fields and types of tasks, we would hope to see our work leading to even further exploration of LSH in the areas of data compression, where LSH could be used as part of the procedure for "fingerprinting" data, with a type of parallel framework allowing faster and scalable use of LSH. Other applications of LSH of interest are using LSH for machine learning [Chen et al., 2019a] or combining LSH with other nearest neighbours techniques for even better results.

Another area where LSH could be of use is in systems. Latest ML approaches used for the detection of side channel attacks also use similarity analysis of time series data from monitored devices. LSH may also be effective at filling the gap in this field for applications working effectively with time series, that have correlation-based measures and have low false positive rates for anomaly detection [Alam et al., 2021].

Similarity-related tasks have not been easy to perform in data mining. However, the solution does not need to be complex to work effectively. Combining simple steps to form an approach like locality sensitive hashing has shown over and over again to be widely successful in many application areas and tasks, such as nearest neighbour search, and, as shown in this thesis, in network discovery and anomaly detection. Using parallel computing to even further improve the time of completing these tasks expands the usage of available computational powers to the most data-dense fields with terabytes of data awaiting to be turned into useful information. Investigating a multitude of techniques to draw comparisons on, opens up opportunities to have the optimal choice of method and parameter setting for specific data and circumstance. Even without an existing perfect solution, additional computational power can result in deeper analysis of large collections of time series in both off-line and real-time applications.

Bibliography

Big data takes off but flight is just beginning. *Aviation Maintenance Magazine*, 2017. URL `https://www.avm-mag.com/big-data-takes-off-flight-just-beginning/`. visited on 25/11/2021.

Manaar Alam, Sarani Bhattacharya, and Debdeep Mukhopadhyay. Victims can be saviors: A machine learning–based detection for micro-architectural side-channel attacks. *ACM journal on emerging technologies in computing systems*, 17(2):1–31, 2021.

Dan Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John Owens, and Nina Amenta. Real-time parallel hashing on the gpu. *ACM transactions on graphics*, 28(5): 1–9, 2009.

Daichi Amagata and Takahiro Hara. Correlation set discovery on time-series data. In Sven Hartmann, Josef Küng, Sharma Chakravarthy, Gabriele Anderst-Kotsis, A Min Tjoa, and Ismail Khalil, editors, *Database and Expert Systems Applications*, pages 275–290, Cham, 2019. Springer International Publishing. ISBN 978-3-030-27618-8.

Mehmet Aydar and Serkan Ayvaz. An improved method of locality-sensitive hashing for scalable instance matching. *Knowledge and information systems*, 58(2):275–294, 2018, 2019.

Ahmet M. Aytekin and Tevfik Aytekin. Real-time recommendation with locality sensitive hashing. *Journal of intelligent information systems*, 53(1):1–26, 2019.

Gaël Beck, Tarn Duong, Mustapha Lebbah, Hanane Azzag, and Christophe Cérin. A distributed approximate nearest neighbors algorithm for efficient large scale mean shift clustering. *Journal of Parallel and Distributed Computing*, 134:128–139, 2019.

Kanishka. Bhaduri, Qiang Zhu, Nikunj. C. Oza, and Ashok. N. Srivastava. Fast and flexible multivariate time series subsequence search. pages 48–57. IEEE, 2010. ISBN 1550-4786.

Beidi Chen, Tharun Medini, James Farwell, Sameh Gobriel, Charlie Tai, and Anshumali Shrivastava. Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. 2019a.

Xuening Chen, Hanwen Liu, and Dan Yang. Improved lsh for privacy-aware and robust recommender system with sparse data in edge environment. *EURASIP Journal on Wireless Communications and Networking*, 2019(1):1–11, 2019b.

Tobias Christiani. Fast locality-sensitive hashing for approximate near neighbor search. *CoRR*, abs/1708.07586, 2017. URL `http://arxiv.org/abs/1708.07586`.

Eric Chu, Dimitry Gorinevsky, and Stephen Boyd. Detecting aircraft performance anomalies from cruise flight data. 04 2010. doi: 10.2514/6.2010-3307.

Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Marroquín. Searching in metric spaces. *ACM computing surveys*, 33(3):273–321, 2001.

Andrew C. Connolly, J. S. Guntupalli, Jason Gors, Michael Hanke, Yaroslav O. Halchenko, Yu-Chien Wu, Hervé Abdi, and James V. Haxby. The representation of biological classes in the human brain. *The Journal of neuroscience*, 32(8):2608–2618, 2012.

Fred Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

Santanu Das, Bryan Matthews, Ashok Srivastava, and Nikunj Oza. Multiple kernel learning for heterogeneous anomaly detection: algorithm and aviation safety case study. pages 47–56. ACM, 2010. ISBN 9781450300551, 1450300553.

Ian Davidson, Sean Gilpin, Owen Carmichael, and Peter Walker. Network discovery via constrained tensor analysis of fmri data. pages 194–202. ACM, 2013. ISBN 9781450321747, 1450321747.

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL `http://archive.ics.uci.edu/ml`.

Osman Durmaz and Hasan S. Bilge. Fast image similarity search by distributed locality sensitive hashing. *Pattern Recognition Letters*, 128:361–369, 2019.

Ana Elisa, Castro Sotos, Stijn Vanhoof, and Patrick Onghena. The transitivity misconception of pearson's correlation coefficient. *Statistics Education Research Journal*, 8:33–55, 01 2009.

Philippe Esling and Carlos Agon. Time-series data mining. *ACM computing surveys*, 45(1):1–34, 2012, 2013.

G. Gilbert. Distance between sets. *Nature (London)*, 239(5368):174–174, 1972.

John Goglia. How aircraft engine monitoring works and how it could help malaysia airlines mh370 crash investigators, 2014. URL https://www.forbes.com/sites/johngoglia/2014/03/13/aircraft-engine-monitoring-how-it-works-and-how-it-could-help-malaysia-air-370-crash-investigtors/?sh=1127a1697620. visited on 25/11/2021.

Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, New York, June 1949.

Hao Hu, Joey Velez-Ginorio, and Guo-Jun Qi. Temporal order-based first-take-all hashing for fast attention-deficit-hyperactive-disorder detection. pages 905–914. ACM, 2016. ISBN 1450342329, 9781450342322.

Tian Huang, Yongxin Zhu, Yafei Wu, and Weiwei Shi. J-distance discord: An improved time series discord definition and discovery method. In *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, pages 303–310, 2015. doi: 10.1109/ICDMW.2015.120.

Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 604–613, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919629. doi: 10.1145/276698.276876. URL `https://doi.org/10.1145/276698.276876`.

Omid Jafari, Preeti Maurya, Parth Nagarkar, Khandker M. Islam, and Chidambaram Crushev. A survey on locality sensitive hashing algorithms and their applications. 2021.

Humaira Kamal and Alan Wagner. An integrated fine-grain runtime system for MPI. *Computing*, 96(4): 293–309, 2014. ISSN 0010-485X. doi: 10.1007/s00607-013-0329-x. URL `http://dx.doi.org/10.1007/s00607-013-0329-x`.

Klaus Kammerer, Burkhard Hoppenstedt, Rüdiger Pryss, Steffen Stökler, Johannes Allgaier, and Manfred Reichert. Anomaly detections for manufacturing systems based on sensor data—insights into two challenging real-world production settings. *Sensors (Basel, Switzerland)*, 19(24):5370, 2019.

John Kaneshige, John Bull, and Joseph Totah. Generic neural flight control and autopilot system. *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2000. doi: 10.2514/6.2000-4281. URL `https://arc.aiaa.org/doi/abs/10.2514/6.2000-4281`.

Yongwook B. Kim. *Physiological time series retrieval and prediction with locality-sensitive hashing*. PhD thesis, Massachusetts Institute of Technology, 2017.

Yongwook B. Kim and Una-May O'Reilly. Large-scale physiological waveform retrieval via locality-sensitive hashing. volume 2015, pages 5829–5833, United States, 2015. IEEE. ISBN 1094-687X, 1557-170X.

Yongwook B. Kim, Erik Hemberg, and Una-May O'Reilly. Stratified locality-sensitive hashing for accelerated physiological time series retrieval. volume 2016, pages 2479–2483, United States, 2016. IEEE. ISBN 1557-170X.

Sven Kosub. A note on the triangle inequality for the jaccard distance. *Pattern recognition letters*, 120: 36–38, 2019.

Keon M. Lee and Kyung M. Lee. A locality sensitive hashing technique for categorical data. *Applied mechanics and materials*, 241-244:3159–3164, 2012.

Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Finding Similar Items*, page 68–122. Cambridge University Press, 2 edition, 2014. doi: 10.1017/CBO9781139924801.004.

Michael Levandowsky and David Winter. Distance between sets. *Nature (London)*, 234(5323):34–35, 1971.

Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, feb 1966. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. A symbolic representation of time series, with implications for streaming algorithms. Association for Computing Machinery (ACM), 2003.

Hong Liu, Lijun Xu, Enmin Song, Renchao Jin, and Chih-Cheng Hung. Automatic labeling of mr brain images through the hashing retrieval based atlas forest. *Journal of Medical Systems*, 43(8):1–12, 2019.

Chen Luo and Anshumali Shrivastava. Ssh (sketch, shingle, & hash) for indexing massive-scale time series. 2016.

Martin Längkvist, Lars Karlsson, and Amy Loutfi. A review of unsupervised feature learning and deep learning for time-series modeling. *Pattern recognition letters*, 42(1):11–24, 2014.

Bryan Matthews. Flight data for tail 668, 2012. URL `https://c3.ndc.nasa.gov/dashlink/resources/646/`. visited on 29/11/2021.

Bryan. Matthews, Ashok N. Srivsatava, John. Schade, D. Schleicher, Kennis Y. Chan, Richard Gutterud, and Mike Kiniry. Discovery of abnormal flight patterns in flight track data. 2013.

Bryan Matthews, David Nielsen, John Schade, Kennis Chan, and Mike Kiniry. Automated discovery of flight track anomalies. pages 4B3–1–4B3–15. IEEE, 2014. ISBN 2155-7195.

Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

Thomas L. Moore. Paradoxes in film ratings. *Journal of statistics education*, 14(1), 2006.

Yadong Mu and Shuicheng Yan. Non-metric locality-sensitive hashing. In *AAAI*, 2010.

Christine Negroni. Newer jets are giving airlines troves of data. *The New York times*, page B5, 2021. URL `https://www.nytimes.com/2021/04/20/business/airplanes-technology-data.html`. visited on 29/11/2021.

Nir Nissim, Omri Lahav, Aviad Cohen, Yuval Elovici, and Lior Rokach. Volatile memory analysis using the minhash method for efficient and secured detection of malware in private cloud. *Computers & Security*, 87:101590, 2019.

Themis Palpanas and Volker Beckmann. Report on the first and second interdisciplinary time series analysis workshop (itisa), 2016.

John Paparrizos, Chunwei Liu, Aaron J. Elmore, and Michael J. Franklin. Debunking four long-standing misconceptions of time-series distance measures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1887–1905, New York, NY, USA, 2020.

Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3389760. URL https://doi.org/10.1145/3318464.3389760.

Sarah Pilz, Florian Porrmann, Martin Kaiser, Jens Hagemeyer, James M. Hogan, and Ulrich Rückert. Accelerating binary string comparisons with a scalable, streaming-based system architecture based on fpgas. *Algorithms*, 13(2):47, 2020.

Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, USA, 2011. ISBN 1107015359.

David N. Reshef, Yakir A. Reshef, Hilary K. Finucane, Sharon R. Grossman, Gilean McEvan, Peter J. Turnbaugh, Eric S. Ladner, Michael Mitzenmacher, and Pardis C. Sabeti. Detecting novel associations in large data sets. *Science (American Association for the Advancement of Science)*, 334(6062):1518–1524, 2011.

Kexin Rong, Clara E. Yoon, Karianne J. Bergen, Hashem Elezabi, Peter Bailis, Philip Levis, and Gregory C. Beroza. Locality-sensitive hashing for earthquake detection: A case study of scaling data-driven science. 2018.

Vibro-Meter SA. Engine vibration and health monitoring systems, 2019. URL https://meggitt.com/wp-content/uploads/2019/10/Aero-EVM-EHM-Systems_v1-0_11-08.pdf. visited on 29/11/2021.

Tara Safavi, Chandra Sripada, and Danai Koutra. Scalable hashing-based network discovery. pages 405–414. IEEE, 2017.

Tara Safavi, Chandra Sripada, and Danai Koutra. Fast network discovery on sequence data via time-aware hashing. *Knowledge and information systems*, 61(2):987–1017, 2019.

Soumik Sarkar, Xin Jin, and Asok Ray. Data-driven fault detection in aircraft engines with noisy sensor measurements. *Journal of Engineering for Gas Turbines and Power*, 133:081602–1, 08 2011. doi: 10.1115/1.4002877.

Anshumali Shrivastava. Simple and efficient weighted minwise hashing. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper/2016/file/c2626d850c80ea07e7511bbae4c76f4b-Paper.pdf.

Svetlana Sodol. Parallel minhash with fgmpi, 2020. URL https://github.com/Caliatdrimme/Minhash. visited on 25/11/2021.

Svetlana Sodol. Parallel lsh with fgmpi, 2021. URL https://github.com/Caliatdrimme/LSH-fgmpi. visited on 25/11/2021.

Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proceedings of the VLDB Endowment*, 6(14):1930–1941, 2013.

Kanji Tanaka and Eiji Kondo. A scalable localization algorithm for high dimensional features and multi robot systems. pages 920–925. IEEE, 2008. ISBN 142441685X, 9781424416851.

Dean Teffer, Dean Teffer, Ravi Srinivasan, Ravi Srinivasan, Joydeep Ghosh, and Joydeep Ghosh. Adahash: hashing-based scalable, adaptive hierarchical clustering of streaming data on mapreduce frameworks. *International Journal of Data Science and Analytics*, 8(3):257–267, 2019.

Anurag Tripathi, Abhinav Gupta, Santanu Chaudhury, and Arun Singh. Image super resolution using distributed locality sensitive hashing for manifold learning. *Multimedia Tools and Applications*, 78(18): 25673–25684, 2019.

Fernando Turrado García, Fernando Turrado García, Luis J. García Villalba, Luis J. García Villalba, Ana L. Sandoval Orozco, Ana L. Sandoval Orozco, Francisco D. Aranda Ruiz, Francisco D. Aranda Ruiz, Andrés Aguirre Juárez, Andrés Aguirre Juárez, Tai-Hoon Kim, and Tai-Hoon Kim. Locating similar names through locality sensitive hashing and graph theory. *Multimedia Tools and Applications*, 78(21):29853–29866, 2019.

Carlos Varela. Too many airplane systems rely on too few sensors, 2019. URL `https://phys.org/news/2019-04-airplane-sensors.html`. visited on 25/11/2021.

Mengni Wang, Yuanxiang Li, Yuxuan Zhang, and Lei Jia. Spatio-temporal graph convolutional neural network for remaining useful life estimation of aircraft engines. *Aerospace Systems*, 4(1):29–36, March 2021. doi: 10.1007/s42401-020-00070-x.

Jiagao Wu, Lu Shen, and Linfeng Liu. Lsh-based distributed similarity indexing with load balancing in high-dimensional space. *The Journal of Supercomputing*, 76(1):636–665, 2020.

Biwei Yan, Jiguo Yu, Meihong Yang, Honglu Jiang, Zhiguo Wan, and Lina Ni. A novel distributed social internet of things service recommendation scheme based on lsh forest. *Personal and Ubiquitous Computing*, 2019.

Chin-Chia M. Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang A. Dau, Zachary Zimmerman, Diego F. Silva, Abdullah Mueen, and Eamonn Keogh. Time series joins, motifs, discords and shapelets: a unifying view that exploits the matrix profile. *Data mining and knowledge discovery*, 32 (1):83–123, 2018.

Chenyun Yu, Lintong Luo, Leanne L. Chan, Thanawin Rakthanmanon, and Sarana Nutanong. A fast lsh-based similarity search method for multivariate time series. *Information Sciences*, 476:337–356, 2019.

Juan Zamora, Marcelo Mendoza, and Héctor Allende. Hashing-based clustering in high dimensional data. *Expert Systems With Applications*, 62:202–211, 2016.

Boyu Zhang, Xianglong Liu, and Bo Lang. *Fast Graph Similarity Search via Locality Sensitive Hashing*, pages 623–633. Springer International Publishing, Cham, 2015. ISBN 0302-9743.