Frequent Graph Mining for Data Streams

by

Ranran Bian

Supervisors: Yun Sing Koh, Gill Dobbie

Department of Computer Science
The University of Auckland
New Zealand
February 2015

A thesis submitted in partial fulfilment of the requirements for the degree of Master in Science, The University of Auckland, 2015.

Abstract

Frequent graph mining is a challenging task that extracts novel and useful knowledge from graph data. The problem becomes even more challenging when the information comes from data streams which evolve in real-time. Additionally, concept drift detection is needed for the problem domain.

In this thesis we present an approach for adaptively mining frequent graph patterns on time-varying streams. Our approach extends an existing graph batch mining framework by implementing and integrating a state-of-the-art change detector. The approach works on coresets of closed frequent subgraphs, compressed representations of graph sets and uses the change detector to address potential concept drifts. In our approach, we mine and monitor the concept drifts of the coresets of closed frequent subgraphs.

An evaluation study on large scale datasets compares the performance between our approach and a change-adaptive algorithm in the existing graph batch mining framework. The experiments process different real-world chemical molecular and social network graph datasets with varying severity of artificial drifts.

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge, contains no previously published materials or content from any authors from universities or institutions of higher learning, except where due acknowledgements are made.

Yours Sincerely,
Ranran Bian

Acknowledgments

First and foremost, I would like to thank my supervisors, Dr. Yun Sing Koh and Professor Gillian Dobbie for the support and advice they have given me through my Masters degree. They have influenced me into researching in the field of data mining and have been my role models throughout my entire research year.

I would also like to thank my parents for their support and encouragement for pursuing a higher level of education. In addition, I would like to thank my dearest friends who encouraged me to strive towards my goal.

Last but not the least, I would like to thank the Department of Computer Science, not only for providing the equipment and facilities to undertake this research, but also for giving me the opportunity to meet so many knowledgeable and interesting research peers.

Ranran Bian
Auckland
September 10, 2014

Contents

List of Figures

Chapter 1

INTRODUCTION

Nowadays, data stream mining has become one of the hottest research topics in data mining due to the large quantity of data generated every day. For example, the 2015 Chinese New Year Gala watched by billions of people around the world, encouraged its audiences to send WeChat messages during the five consecutive hours of TV program. According to the big data statistics provided by the program, more than 80 millions messages were received within the first hour.

Mining of data streams has flourished also due to applications using streaming data from various sources, such as heart rate monitors. A large number of commercial and scientific applications use patterns that are more sophisticated than frequent itemsets and sequential patterns. Such complicated patterns go beyond sets and sequences, toward trees and graphs. Being a general data structure, graphs have become increasingly important in modeling sophisticated structures and their interactions, with broad applications including chemical informatics, bioinformatics, computer vision, video indexing, text retrieval, and Web analysis [19].

Currently, some state-of-the-art databases have already embedded automatic frequent sequential pattern matching techniques into the system to reduce some tedious workload for database administrator. It is likely that their next step will be to integrate automatic frequent graph pattern matching technique into the database systems. In large social networks, a huge number of people are connected to each other, making graph mining increasingly valuable as the graph data structure can represent different communities in the large social network and the connections and communications between people within each community. By performing frequent graph mining in large social network, we can monitor the changes that happen to the communities as well as the inner-relationships of people inside each community.

1.1 Background

Among different types of graph patterns, frequent substructures are very basic patterns that can be discovered in a graph dataset. Some frequent substructure mining methods have been developed in previous research. Dehaspe et al.'s approach [37]

discovers frequent substructures by applying inductive logic programming to predict chemical carcinogenicity. SUBDUE was proposed by Holder et al. [38] for approximate substructure pattern mining on the basis of background knowledge and minimum description length. On top of these methods, there are two basic types of approaches for discovering frequent substructures: an Apriori-based approach and a pattern-growth approach [19].

Apriori-based frequent subgraph mining methods share similar characteristics with Apriori-based methods for finding frequent itemsets. The search for frequent substructures starts with substructures of small "size", and continues in a bottom-up approach. At each iteration, the size of newly found frequent subgraphs is increased by one. These new subgraphs are initially produced by joining two similar but slightly different frequent substructures that were found previously. The frequency of the newly generated graphs is then examined. Apriori-based graph pattern mining algorithms include Inokuchi et al.'s AGM [5], Kuramochi and Karypis's FSG [4], and Vanetik et al.'s edge-disjoint path-join algorithm [26].

Normally, the Apriori-based methods have considerable overhead when joining two size-k frequent subgraphs to produce size-(k+1) graph candidates. For the purpose of avoiding such overhead, pattern-growth-based frequent subgraph mining algorithms have been developed, including Nijssen and Kok's Gaston [39], Huan et al.'s SPIN [40], Huan et al.'s FFSM [41], Yan and Han's gSpan, Borgelt and Berthold's MoFa. Unlike the Apriori-based methods, the pattern-growth-based substructure pattern mining algorithms extend a frequent subgraph by adding a new edge, in every possible permutation.

Besides the frequent subgraph mining algorithms, some constraint-based subgraph mining algorithms have also been proposed. Yan and Han researched closed subgraph mining with the proposal of the algorithm, CloseGraph, as an extension of gSpan and CloSpan [42]. Huan et al. studied coherent subgraphs mining [43]. For mining relational graphs, Yan et al. [44] proposed two algorithms, CloseCut and Splat for mining exact dense frequent subgraphs in a collection of relational graphs.

1.2 Motivation

The majority of the current data mining research is on frequent itemsets and sequential patterns; some work has been done on frequent graph mining but there has been very little work done in the field of frequent graph mining in non-stationary data streams.

Our first research motivation is to explore more in this unexplored area.

Frequent graph mining in non-stationary data streams introduces a new set of challenges, which do not exist in a static environment, most notably the increased huge demands and constraints placed on running time and memory usage throughout the theoretically infinite sets of graphs. The challenges and constraints motivate us to develop a technique for the stated problem with optimal time and memory consumptions.

Nowadays, a large number of applications require the effective discovery, deep understanding of ongoing changes of graph data. Change graph pattern mining quantifies and describes the difference between snapshots of graph data streams taken at different points in time. It is crucial in many applications to discover, understand, and assess ongoing changes early. Thus, change mining [45]-[49] is vital in domains where early detection of emerging trends is crucial to allow users to make decisions.

1.3 Problem statement

Our research concentrates on developing an efficient and effective technique to find changes in frequent graphs in data streams. The technique needs to tackle the constraints and challenges placed by stream mining, which are time consumption, memory usage and change detection speed.

1.4 Objective and goal

In our research, we integrated a state-of-the-art change detector into an existing graph batch mining framework. The algorithm in our approach and a change-adaptive algorithm in the existing framework have both been deployed on chemical molecular and social network data represented in the form of graphs. When we compare the performance of the two algorithms, the three key evaluation metrics used measure: run time, memory usage consumption and change detection speed. By creating various severity of artificial drifts or changes in different datasets, we are able to control the amount and positions of drifts that occur in our data streams so that the two techniques' change detection speeds can easily be monitored. In addition, the time and memory consumed for completing the mining processes are also recorded for both techniques. The main objective and goal of our research is to perform an investigation on both our and existing algorithms and explore which one provides the best balance among the three key evaluation metrics when mining frequent graphs for data streams.

1.5 Contributions

We state three main contributions that address research gaps that appear in existing work and techniques for frequent graph mining in data streams. The first contribution is two-fold: (1) The existing change-adaptive technique uses ADWIN [22] as a change detector. The performance comparison done on the technique only underlined the speed of change detection between algorithms, we attempt to explore additional variables that are crucial to evaluating the usability of a frequent graph miner in data streams. (2) The efficiency and reliability of the change detector used by a frequent graph miner is one of the largest factors that determines its efficiency. We integrate a new state-of-the-art change detector, SEQDRIFT [23], into an existing frequent graph miner and perform a detailed performance comparison between the new and old change detectors.

For the second main contribution, we expand the input and output data structures of the frequent graph miner which only processes chemical molecular graph datasets. For our work, we wish to analyze data streams that are derived from large social networks. Hence, we generate a data converter that allows us to convert normal text representations of graphs into readable inputs for the frequent graph miner, therefore improving its functionalities and enabling us to process social network graph datasets.

For our final contribution, we perform deeper analysis on outputs produced by frequent graph miner. Currently, the outputs produced only notify users with the number of frequent graphs that were found, instead, we wish to derive meaning from these numbers by finding patterns, trends and points of interest which we can explain with confidence.

1.6 Thesis structure

The thesis will be structured as follows: The current chapter provides a general introduction to our topic of interest and the thesis itself. Chapter 2 will provide preliminary concepts and explore related work done in the field of graph mining. Chapter 3 details the background of our work, more specifically, data structures and representations that were used in our research. Chapter 4 describes our approach on the subject, how we implemented our techniques and what we hoped to accomplish. Chapter 5 is our experimental evaluation, we provide detailed analysis of our results and compare it with past research. Finally, we will conclude with Chapter 6 and provide a summary of this thesis and some research directions that could be pursued in the future.

Chapter 2

RELATED WORK

In this chapter, we explore related work of frequent graph mining for data streams. Figure 2.1 presents a diagram of the concepts involved. We first start by describing some preliminary concepts in Section 2.1. We then describe Frequent Graph Mining Techniques (FGMT) and Frequent Graph Mining Techniques For Data Streams (FGMTFDS) in Sections 2.2 and 2.3 respectively. Finally, in Section 2.4 we will study work done for Change Detection Techniques (CDT).

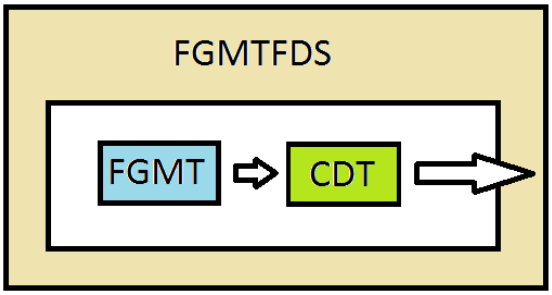


Figure 2.1: Underlying mechanisms for frequent graph mining for data streams.

In Section 2.3, we discuss four frequent graph mining techniques for data streams (FGMTFDS). By comparing and contrasting the inputs and outputs of the four FGMTFDS, we clarify expected mining outcomes in our problem domain. Section 2.2 focuses on discussing pattern-growth frequent graph mining techniques (FGMT), including CloseGraph algorithm and MoSS algorithm, which correspond to two implemented versions of Bifet's graph batch mining framework [13] (Section 2.3.1), serving as two separate underlying mining engines. When mining in a data stream environment, it is common to integrate drift detection techniques into the process for the purpose of quick adaptation to abrupt, moderate or gradual changes in the data distribution. Hence, as shown by the first arrow in Figure 2.1, mining outputs generated by FGMT are passed onto a Change Detection Technique (CDT).

The change detection process for graph mining is one of the three core ideas of Bifet's graph batch mining framework (Section 2.3.1). The other two core ideas are coresets of representation for mining outcomes and batch mining, which are both described in

detail in Section 2.3.1. For each batch , results generated by the specific CDT are passed onto Bifet's technique (Section 2.3.1) for combining and processing with results from previous/next batch as shown by the second arrow in Figure 2.1. One thing to notice is that change detectors discussed in Section 2.4 can take in either prequential (0 or 1) or real numbers. In the context of graph mining, a separate change detector instance is created for managing the support of every single closed frequent graph mined, which results in the untraditional analysis of real number inputs.

2.1 Preliminaries

This section introduces some core concepts that will be referenced throughout the thesis.

2.1.1 Support of an itemset

Definition 1. ***Itemset Support:*** *Let $I = \{i_1, i_2, ..., i_m\}$ be a set of literals, called items. Let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. We say the support for an itemset, i_x, is the proportion of transactions T in the database D that contains i_x [33].*

Given the market-basket transactions in Figure 2.2 as an example, an itemset is defined as a collection of one or more items [27]. For instance, {Bread, Diaper, Milk} is one itemset. The support count or the absolute support is the frequency of occurrence of an itemset, hence for Figure 2.2, the support count or absolute support of {Bread, Diaper, Milk} is two. Relative support is defined as the fraction of transactions that contain an itemset. For instance, relative support of {Bread, Diaper, Milk} is equal to its absolute support divided by the total number of transactions, which is $\frac{2}{5}$. We define a frequent itemset as an itemset whose support is greater than or equal to a user-defined minimum support threshold.

2.1.2 Closed and maximal frequent itemsets

Definition 2. ***Frequent Itemsets:*** *Let $I = \{i_1, i_2, ..., i_m\}$ be a set of literals, called items. We denote F to be the set of all frequent itemsets given $F = \{X | X \subseteq I$ and support(X) $\geq$ min-support$\}$ where support(X) is the support of X and min-support is a user-defined minimum threshold.*

Definition 3. ***Closed Frequent Itemsets:*** *For the set of all frequent items, F, we define set of all closed itemsets, C, to be $C = \{X | X \in F$ and $\nexists Y \supset X$ with support(X) = support(Y) and such that $Y \in F$ $\}$.*

TID	Items
1	Bread, Milk
2	Beer, Bread, Diaper, Eggs
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Bread, Coke, Diaper, Milk

Figure 2.2: Market-Basket Transactions [27].

Definition 4. ***Maximal Frequent Itemsets:*** *For the set of all frequent items, F, we define set of all maximal itemsets, M, to be $M = \{X | X \in F$ and $\nexists Y \supset X$ such that $Y \in F\}$.*

An itemset is closed frequent if none of its immediate frequent supersets has the same support as the itemset (Definition 3) [33]. An itemset is maximal frequent if none of its immediate supersets is frequent (Definition 4) [33]. Figure 2.3 shows the relationships between frequent itemsets, closed frequent itemsets and maximal frequent itemsets.

Figure 2.4 presents an example of these concepts. From the dataset on the left hand side of Figure 2.4 and the minimum threshold of 50%, we derive the frequent itemsets and its support on the table to the right. Using the last line of the frequent itemset table as an example, we can see that itemset cde is both a maximal and closed itemset. We see that there exists no proper supergraphs for cde and also by extension, there are no proper supergraphs of cde that have the same support, hence satisfying our definition in Definition 3 and 4.

2.1.3 Apriori algorithm for association rule mining

The Apriori algorithm, proposed by Agrawal [24], is the concept of mining association rules. The concept of association rules can be described using market-basket analysis, where occurrences of an item can be predicted by the appearance of particular items in the same transaction. An association rule is an implication expression of the form $X \rightarrow Y$, where X and Y are itemsets and the intersection of X and Y should be the empty set. For instance, in the transactions given in Figure 2.2, there may be the following association rule: {Diaper, Milk} $\rightarrow$ {Beer}. Given a set of transactions T, the purpose of association rule mining is to find all eligible rules which have support $\geq$

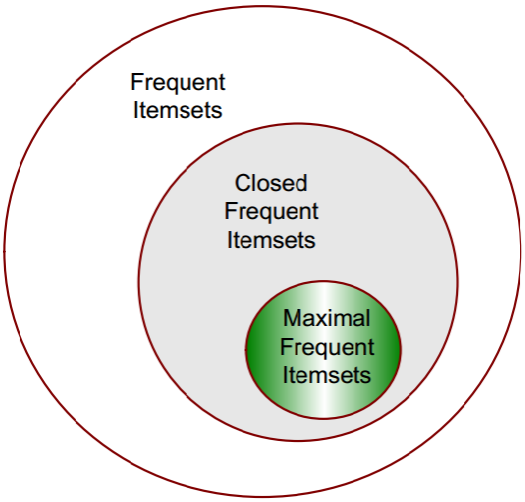


Figure 2.3: Relationship between frequent, closed frequent and maximal frequent itemsets [31].

d1	abce
d2	cde
d3	abce
d4	acde
d5	abcde
d6	bcd

Support	Frequent	Closed	Max
6	c	c	
5	e,ce	ce	
4	a,ac,ae,ace	ace	
4	b,bc	bc	
4	d,cd	cd	
3	ab,abc,abe be,bce,abce	abce	abce
3	de,cde	cde	cde

Figure 2.4: Example showing closed and maximal itemsets.[30]

user-defined minimum support threshold.

The Apriori principle is used to reduce the number of candidates. It states that if an itemset is infrequent, then all of its supersets must also be infrequent. The Apriori Algorithm works as follows:

2.1.4 Support of a graph

The support of a graph G is the fraction of graphs in the entire dataset that contains G. This can be used to filter out graphs with low frequency of occurrence, meaning we can set a minimum threshold (the minimum support) for which graphs need to appear for us to be interested in them. For graph dataset D, graph pattern P and threshold T, we

Algorithm 1 Apriori Algorithm

1: *Let k = 1*
2: *Generate frequent itemsets of length 1*
3: *Repeat until no new frequent itemsets are identified*
4: *Generate length (k+1) candidate itemsets from length k frequent itemsets*
5: *Prune candidate itemsets containing subsets of length k that are infrequent*
6: *Count the support of each candidate by scanning the database*
7: *Eliminate candidates that are infrequent, leaving only those that are frequent*

say P is frequent if it appears at least T times in D (Equation 2.1).

$$Freq(P) > T \tag{2.1}$$

2.1.5 Closed and maximal graphs

Given two graphs g and g', we say that g is a subgraph of g', or g' is a super-graph of g, if $g \preceq g'$. The symbol $\preceq$ is defined as a total order on graphs. Two graphs g, g' are said to be comparable if $g \preceq g'$ or $g' \preceq g$. Otherwise, they are incomparable. For the frequent graph G and graph dataset D, G is defined to be closed if there exists no super-graph of G that has the same support in D. Also, G is defined to be maximal if there exists no super-graph of G that is frequent in D.

These concepts are used to reduce the size of our output, which is one of the major limitations of data stream mining. Figure 2.5 shows the levels of encapsulation formed between the frequent graphs, closed frequent graphs and maximal frequent graphs. For the same support threshold, the set of closed frequent graphs contains all the information that its corresponding frequent graph set contains. However, this is not the case for maximal frequent graphs.

2.1.6 Apriori approach for frequent graph mining

Apriori-based frequent substructure mining is similar to the frequent itemset mining counterpart. The search starts with small graphs and proceeds in a bottom-up manner. Each iteration increments the size of newly discovered frequent substructures by one and these substructures are formed by joining existing frequent graphs already discovered [25]. Afterwards, the frequency of the newly formed graphs is checked. Apriori-based algorithms for frequent graph mining include an edge disjoint path-join algorithm by Vanetik et al. [26], FSG by Kuramochi and Karypis [4], and AGM by Inokuchi et al. [5]. A vertex-based candidate generation method is adopted by the AGM algorithm, meaning that the substructure size is increased by one vertex at each

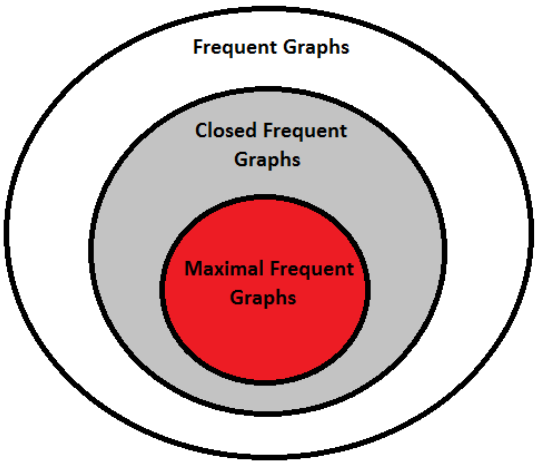


Figure 2.5: Frequent, Closed Frequent and Maximal Frequent Graphs.

iteration. The prerequisite for joining two size-k frequent graphs is that the two graphs must have the same size-$(k-1)$ subgraph. In AGM, the graph size is the number of vertices in a graph. The newly formed graph candidate contains the common size-$(k-1)$ subgraph and additional two vertices from the two size-k graphs. Since it is undetermined whether there is actually an edge connecting the additional two vertices, the AGM forms two candidates. Figure 2.6 shows how the two subgraphs are joined by two edges.

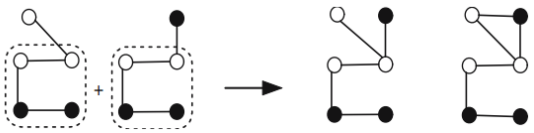


Figure 2.6: Two substructures joined by two chains [25].

2.1.7 Pattern-growth approach for frequent graph mining

The Apriori-based algorithms discussed in the previous section have considerable overhead when two size-$(k-1)$ frequent graphs are joined to generate size-k graph candidates. For the purpose of avoiding such overhead, the pattern-growth approach has been developed, which extends existing frequent graphs by adding an edge in every possible location. This algorithm is adapted in many graph miners, including ones we will discuss in Section 2.2. gSpan, a frequent graph mining algorithm that will be discussed in Section 2.2.1.1, uses an improved pattern growth algorithm that extends the right most path (a straight path from starting vertex v_0 to last, right-most vertex v_n).

This solves the potential problem of having the same graph discovered multiple times, which occurs using the standard pattern-growth approach.

2.2 Frequent graph mining techniques

Frequent graph mining is one of the main themes in data mining that uses structured representations of graphs to find recurring patterns within graphs and sub-graphs. Techniques in this field have been adopted and improved over the last decade, becoming more refined and time-efficient as we begin to see a stronger emphasis on exploring relevant or interesting parts of the datasets.

When dealing with small datasets, minor differences in graph traversal might not be noticeable, however when dealing with a theoretically infinite number of graphs of varying sizes, small adjustments or refinements could make or break an algorithm. In this section we explore techniques that find frequent graphs through depth-first search and breadth-first search methods. In depth-first search, we will study the pattern mining algorithm, gSpan, and the closed frequent graph finder, CloseGraph. While in breadth-first search we will look at the frequent molecule miner, MoFa and its Java extension MoSS. Finally, we will summarize and compare the strengths and weaknesses of each technique.

2.2.1 Depth-first search

The process of traversing through a tree using a depth-first search is widely known, we iterate as far down a branch and backtrack when needed. However, when dealing with graphs in data streams we are faced with time and memory constraints that do not exist in a static environment. This section details two graph-based pattern mining algorithms, the graph-based substructure pattern-mining (gSpan) [2] and the closed frequent graph mining (CloseGraph) [3].

2.2.1.1 gSpan

Yan and Han [2] introduce the graph-based pattern mining algorithm, gSpan. Unlike previous algorithms [4, 5], gSpan maps each graph it discovers into a subscript (or DFS code) and from this they are able to build lexicographic ordering among these codes. Graphs (b) to (d) in Figure 2.7 are isomorphic graphs to (a). We see that the orientation of the graph will influence the DFS codes created and therefore also its search efficiency. This can be seen from each tree's right-most path, with (b) and (c) iterating through three nodes while (d) iterates through four.

The concept of forward and backward edges are also introduced by the authors, as seen in Figure 2.7. An edge is stated to be a forward edge if it is traversed through in the DFS (as shown by bold edges in Figure 2.7). They are edges that are contained within the DFS tree, whereas the backward edges are redundant edges that do not appear in our DFS tree.

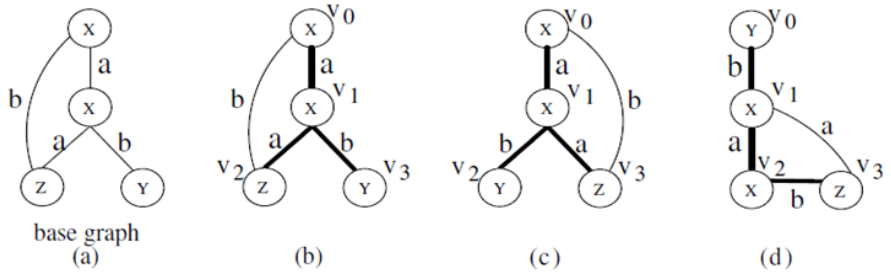


Figure 2.7: DFS Subscripting [3].

These different orientations can be represented as DFS codes that denote vertices involved in each iteration of the DFS and using this representation, gSpan creates parent-child relationships between DFS codes that are extensions of another DFS code, i.e the child. This technique defines the gSpan algorithm as we can find all frequent descendants of a graph by looking at the DFS Code Tree (Figure 2.8) and using this, the performance and efficiency was shown to be considerably faster than previous algorithms, which will be explained later.

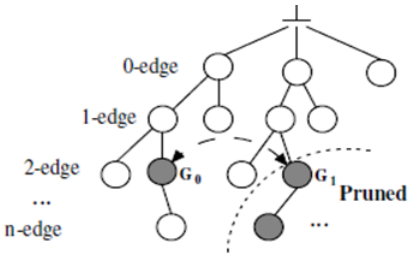


Figure 2.8: DFS Code Tree [2].

The gSpan algorithm itself starts by sorting and removing infrequent edges and vertices from the graph set (GS) and re-labels them in the order of descending frequency. All frequent 1-edge graphs are then separated into a set (S) and sorted into DFS lexicographic order. Each of these 1-edge graphs are then grown to find all its descendants, which is used to shrink all graphs in GS, this step is vital as it reduces the complexity and number of graphs, hence speeds up the mining procedure. This

process runs recursively and terminates when all 1-edge graphs and their descendants have been exhausted.

gSpan is shown to have some notable advantages over previous algorithms [4, 5]. First, it does not perform candidate generation on its graphs or perform any false candidate tests like Apriori algorithms. This saves a lot of computation time and memory. Secondly, each iteration of the mining process will cause the dataset to constantly shrink, which again reduces the search space.

2.2.1.2 CloseGraph

One of the greatest challenges in graph mining is that a labelled graph with n-edges could contain 2^n frequent sub-graphs. This poses an issue since most of the sub-graphs are redundant and do not provide any valuable information. A later publication by the same authors [3] addresses this issue by eliminating redundancy and only mines closed graphs. As an extension to their previous work with gSpan, CloseGraph attempts to check if graphs satisfy certain conditions and if true, all of their descendants will not need to be computed.

CloseGraph introduces the concept of equivalent occurrence. To explore this idea, the concepts of occurrence and extendable occurrence must first be explained. Definition 5 states that the occurrence of a graph, g, is the total number of isomorphic occurrences of g in dataset, D. To present this idea, we use the base graph in Figure 2.7 and the database in Figure 2.9 as our example. The base graph g appears in the dataset D three times in total (as shown by the coloured outlines). We see two occurrences in $g1$ and one occurrence in $g2$, giving us the occurrence value of three, $I(g, D)$=3.

Definition 5. ***Occurrence:*** *Given g and D = $\{G_1, G_2, ..., G_n\}$, the occurrence of g in D is the sum of the number of subgraph isomorphisms of g for every graph in D, i.e $\sum_{i=1}^{n} \varphi(g, G_i)$ where $\varphi(g, G_i)$ represents the number of possible subgraph isomorphisms of g in G_i. This is denoted by $I(g, D)$.*

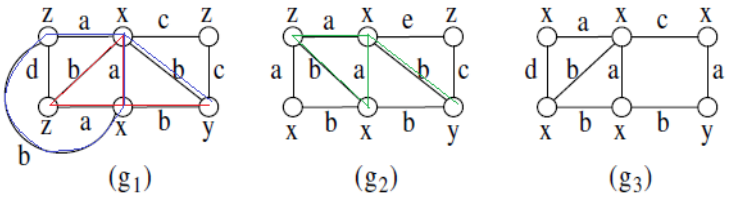


Figure 2.9: Dataset with occurrences of base graph from Figure 2.7 outlined [3].

The extended occurrence, as explained in Definition 6, states that given $g' = g \diamond_x e$ (g' is g with an additional edge), if f is a subgraph isomorphism of g and f' is a subgraph isomorphism of g', then for some p, p is a subgraph isomorphism of g in g', $\forall v, f(v) = f'(p(v))$, we define f to be extendable and f' an extended subgraph isomorphism from f.

Definition 6. ***Extended occurrence:*** *Given $g' = g \diamond_x e$ and $D = \{G_1, G_2, ..., G_n\}$, the extended occurrence of g' in D with respect to g is the sum of the number of extendable subgraph isomorphisms of g in every graph among D, i.e $\sum_{i=1}^{n} \phi(g, g', G_i)$, denoted by $L(g, g', D)$.*

Finally, we introduce the idea of equivalent occurrence, shown in Definition 7. If g' is an extension of g with an additional edge and the occurrence of g is equivalent to the extended occurrence of g' with respect to g, then we state that g and g' have an equivalent occurrence. This equivalence means that all descendants of the super-graph must apply the same rules as itself and we will not need to grow g' any further, allowing us to perform an early termination.

Definition 7. ***Equivalent occurrence:*** *Given $g' = g \diamond_x e$ and D, if I(g,D) = L(g,g',D). We say that g and g' have an equivalent occurrence, which means whenever g occurs in D, g' occurs.*

The authors have also noted that there are rare cases where early termination can lead to loss of graph patterns. Figure 2.10 shows an example of this. If Figure 2.10 (1) and (2) are graphs in a dataset with the minimum support of two. Then since $y - b - x$ (vertex y connected to vertex x with edge b) always follows $x - a - y$ then we would terminate the extending $x - a - y$ pattern as described by equivalent occurrence above. However, we would miss the pattern in Figure 2.10 (3) if we terminate the sub-graph $x - a - y$, meaning we would have lost a pattern frequency, making the results inaccurate.

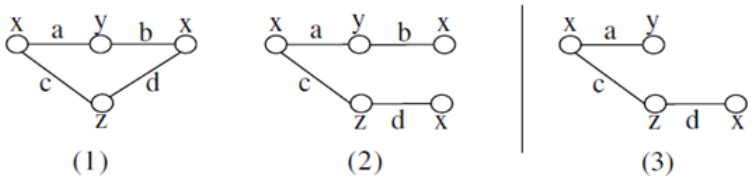


Figure 2.10: Failure of Early Termination [3].

To remedy this, the authors have included functions in CloseGraph to detect failure of early termination. Figure 2.11 shows an example of this where graphs $g2$ and $g3$ grow from $g1$. We notice that both $g2$ and $g3$ are not frequent (as their support is smaller than 2). However, we can see that both $g2$ and $g3$ introduces a common edge $z - d - x$ but $g3$ introduces a new vertex while $g2$ does not. Hence by breaking the $y - b - x$ edge (shown in $g4$), graphs can be grown into what we can see in Figure 2.10 (3).

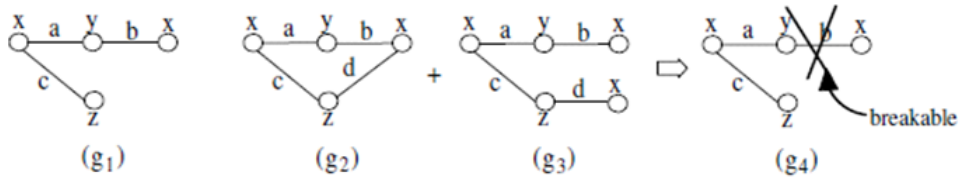


Figure 2.11: Detection of Failure of Early Termination [3].

2.2.2 Breadth-first search

This section describes MoFa and Moss [6, 7], miners that find common molecular fragments in chemical compounds. The breadth-first search, level-wise approach is preferred here due to the complex nature of molecular bonds and atomic structures of compounds.

2.2.2.1 MoFa and MoSS algorithms

An alternative approach in frequent graph mining is shown by Borgelt and Berthold [6, 7]. MoSS (Molecular Substructure Miner) is a Java extension of the MoFa (Molecular Fragment Miner) used to find frequent patterns in molecular substructures or chemical fragments in databases such as chemical compound descriptors or labelled graphs. Apart from its default processing mode, the algorithm also incorporates the gSpan and CloseGraph algorithms described in the previous section. MoSS reads in structured molecular descriptors (graphs) in SMILES format (as shown in Figure 2.12) and finds all frequent substructures that appear within a user-defined minimum support. We will describe the SMILES molecular representation in Chapter 3.

Figure 2.12: Chemical compound and SMILES representation [7].

Inspired by the Apriori and Eclat algorithms [9], MoSS transforms the molecule sequences into search trees in order to find frequent patterns. The algorithm starts by selecting a core structure to traverse from (usually a single atom). This structure is then expanded upon by introducing a single bond for each iteration. Naturally, this forms

our search tree and we can see it being a similar approach used in item-sets, with the number of bonds substituting the number of items. An example of frequent substructures forming can be seen in Figure 2.13. The atom S is used as the core structure with the molecular dataset being on the left hand side. From there, we expand it by a single element (either N, S or O in this case) increasing the number of bonds as we populate and check for frequencies and traverse down the search tree. The frequencies of the substructures are shown on the right of Figure 2.13.

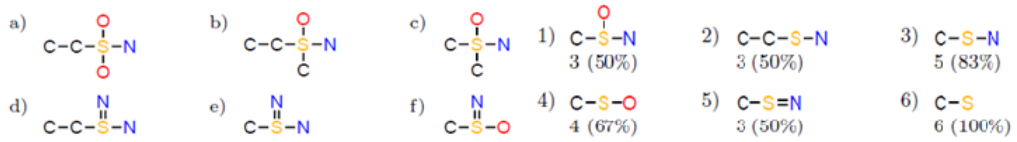


Figure 2.13: Chemical compound and SMILES representation [7].

MoSS puts a large emphasis on search tree pruning, meaning simplification of the tree to reduced search space over time. Search tree pruning can be triggered in multiple ways, as described by the authors [10]:

- *Support based Pruning:* Subtrees of the search tree will be pruned if they do not meet the required support.
- *Size based Pruning:* Search tree is pruned when the number of atoms in a fragment reaches a user-defined threshold.
- *Structural Pruning:* Pruning to avoid repetitions of substructures in the search tree, however this is not perfect.

Advanced pruning strategies have been noted by the authors [7] in an attempt to improve upon the structural pruning technique stated above, as it could not eliminate all redundant searches. This introduced alternatives such as:

- **Closed Molecular Fragments:** A molecular substructure is considered closed if there are no super-structures that have the same support. Reducing the search for only closed fragments will reduce the size of the output while not losing any information as all frequent fragments can be constructed from closed frequent ones. This strategy was adopted but not noted in [6].
- **Equivalent Sibling Pruning:** As suggested by its name, equivalent sibling pruning checks whether two sibling nodes in the search tree represent the same substructure. Consider Figure 2.14. If the benzene ring (the 6 joint carbon atoms) was embedded into the core molecule, it could be embedded in 12 unique orientations

(one in each position of the C atom and in two different directions). This would produce twelve identical fragments that only differ in the position of the bond, hence the smallest should be chosen and the rest made redundant.

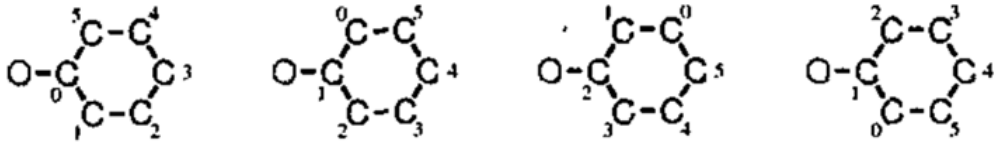


Figure 2.14: Equivalent sibling pruning (Benzene ring) [7].

- **Perfect Extension Pruning:** Perfect extension pruning draws upon the idea that there sometimes exists a large common fragment in all non-redundant molecules. The notion is that the traditional canonical form extensions will sometimes restrict us from extending into sub-fragments that are common among all molecules in the dataset (for example $S-C-N$ in Figure 2.15). If we can postpone extending to fragments that are not universally common to the molecules in the dataset then we could avoid traversing through multiple branches of the tree. The authors have noted two different variations of perfect extension pruning technique [11]:

  1. *Partial Perfect Extension Pruning:* We note from Figure 2.16, that if we extend $S-C$ to $S-C-N$ we would be unable to reach the $O-S-C-N$ state on the left side of the tree. Canonical form extension prevents us from extending using the S atom because atoms with higher atomic numbers will have a higher priority to be extended. These problems only occur to the left of the perfect extension branch, hence the fragments on the right can be pruned without loss because they can be reached by all the perfect extension fragments.
  2. *Full Perfect Extension Pruning:* Extending from the idea of Partial Perfect Pruning, the algorithm transforms fragments into code-words that contain bonding structures and atom orientation so pruning can occur on the left side of the perfect extension without loss (shown in Figure 2.17).

2.2.3 Summary

This section presents algorithms that use the two opposite spectrums of tree traversal. For the depth-first search study we looked at gSpan [2] and CloseGraph [3] algorithms. We noted the use of DFS subscripts and the problem of early termination, including the

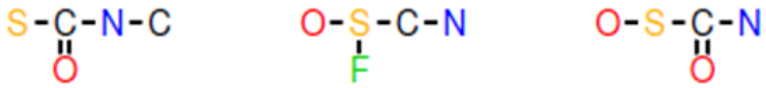


Figure 2.15: Sample molecules [11].

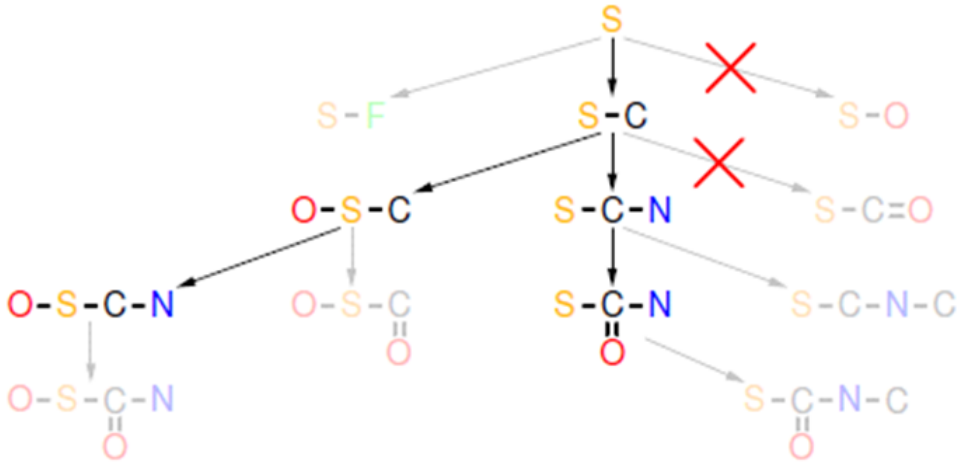


Figure 2.16: Partial perfect pruning [11].

algorithms' ability to detect and avoid such cases. For breadth-first search we explored MoSS [7], a frequent graph miner for chemical compounds. We studied the extensive pruning techniques used to reduce search space which included the equivalent sibling and perfect extension pruning techniques [10].

Currently, there are two basic approaches to the frequent graph mining problem: an Apriori-based approach and a pattern-growth approach. As briefly explained in Section 2.1, the Apriori-based approach for frequent graph mining suffers from having considerable overhead when two size-k frequent graphs are joined to generate size-(k+1) graph candidates, which is however, effectively avoided by the three pattern-growth techniques discussed in this section [19].

2.3 Frequent graph mining techniques for data streams

As the field of frequent graph mining continues to grow, we start to see it being used in more complex real-world scenarios. Graph mining in data streams has a new set

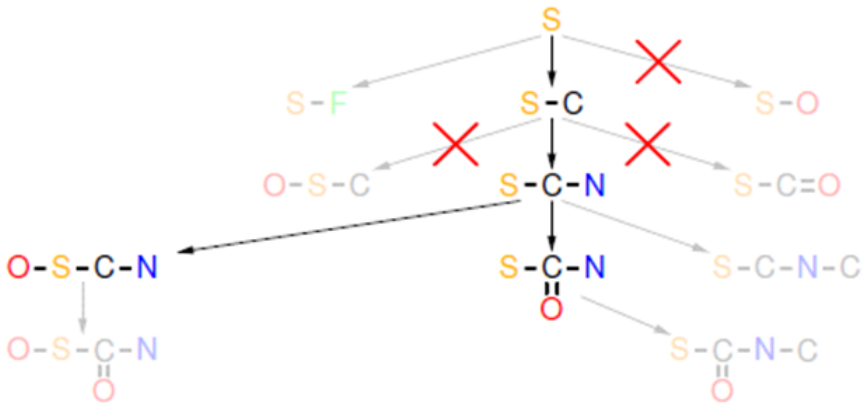


Figure 2.17: Full perfect pruning [11].

of challenges, most notably the increased demands and constraints placed on memory, time and the changes that occur through theoretically infinite data streams. To the best of our knowledge, there exists only four techniques for mining frequent graphs in data streams. This section will describe each one and draw comparisons between them.

2.3.1 Bifet's graph batch mining framework

Closed frequent graph mining algorithms on static data such as CloseGraph [3] and MoSS [7] operate under the assumption that there is a finite set of stationary data with unlimited resources for processing and storage. However, working with closed frequent graphs in data streams presents new challenges. We must ensure that the algorithm is efficient enough to produce output in a timely manner, so we need some way to simplify the input we obtain and our algorithm needs to be adaptable to changes that appear in the data stream. Bifet et al. present one of the first novel approaches to closed frequent graph mining in data streams. The paper introduces three distinct closed frequent graph mining modes, incremental mode (IncGraphMiner), sliding window mode (WinGraphMiner) and adaptive to change mode (AdaGraphMiner).

As noted before, detection and adaptation to change is a new challenge for data stream mining, where the input data is unpredictable and to account for this the authors make several extensions to the generic frequent graph mining algorithms. Weighted graphs are an extension to the standard algorithms. Through introducing weights to their graphs, the authors use compressed graph representations in their batch mining methods. These weights are used to define our graphs, where a graph g is closed if none of its proper supergraphs has the same weighted support as itself.

On the processing side, the authors integrate the concept of coresets into their algorithm. Coresets are small subsets of a larger problem P that when solved, provide an approximate solution to P. In our case, the outputs obtained from closed frequent graphs can be used as a representation for the outputs we obtain from frequent graphs as well. One thing to note is that all three modes of Bifet's approach are vertex-based miners and put heavy emphasis on the values of vertices rather than the edges (which for general graphs we simply assign the random value x, y or z).

2.3.1.1 Incremental miner (IncGraphMiner)

This section presents a comparison between non-incremental miner CloseGraph and the introduced incremental miner IncGraphMiner. Figure 2.18 presents the pseudocode of both algorithms. We see that the CloseGraph algorithm checks for right-most extensions of a subgraph g, which if exists, will check if it meets the minimum support threshold. If there exists a proper supergraph of g that meets the minimum support threshold and shares the same support value as g then g is not in the closed frequent graph set.

IncGraphMiner extends from CloseGraph but rather than taking inputs directly from dataset D, D is split into multiple even-sized batches that perform CloseGraph mining operations in chunks. When every pair of batches has finished mining, it is joined as a union to compute the relative support of the graphs (as shown in Figure 2.18 $CORESET$ method, lines 1 and 2).

```
CLOSEGRAPH(g, D, min_sup, S)
    Input: A graph g, a graph dataset D, min_sup.
    Output: The frequent graph set S.

 1  if g ≠ CANONICAL_REPRESENTATIVE(g)
 2     then return S
 3  isClosed ← TRUE
 4  C ← ∅
 5  for each g' that is a one step right-most
        extension of g
 6        do if support(g') ≥ min_sup
 7              then insert g' into C
 8           if support(g') = support(g)
 9              then isClosed ← FALSE
10  if isClosed = TRUE
11     then insert g into S
12  for each g' in C
13        do S ← CLOSEGRAPH(g', D, min_sup, S)
14  return S
```

```
INCGRAPHMINER(D, min_sup)
    Input: A graph dataset D, and min_sup.
    Output: The frequent graph set G.

1  G ← ∅
2  for every batch b_t of graphs in D
3        do C ← CORESET(b_t, min_sup)
4           G ← CORESET(G ∪ C, min_sup)
5  return G

CORESET(b_t, min_sup)
    Input: A graph dataset b_t, and min_sup.
    Output: The coreset C.

1  C ← CLOSEGRAPH(b_t, min_sup)
2  C ← COMPUTE_RELATIVE_SUPPORT(C)
3  return C
```

Figure 2.18: Non-incremental miner (left). Incremental miner (right). [13].

2.3.1.2 Sliding-window miner (WinGraphMiner)

WinGraphMiner differs to IncGraphMiner in that it maintains a sliding window of size W. When items arrive from batches, we update the recent closed frequent graphs using the coresets obtained. Under the circumstance that the window is full, we delete the oldest batch in the sliding window by introducing negative support. While the window is not full we perform IncGraphMiner on the next batch.

2.3.1.3 Delta value

The delta value, δ is a parameter we adjust to control our output to either closed frequent graphs or maximal frequent graphs and it also enables us to control the level of compression in the output closed frequent graphs. Figure 2.19 shows the relationship between number of graphs (y axis) and the δ value (x axis). The example shows a (0.4,δ)-coreset with a minimum support of 40%. When δ=0 we can see an output of approximately 650 closed frequent graphs, whereas when δ=1 we see an output of 140 maximal frequent graphs.

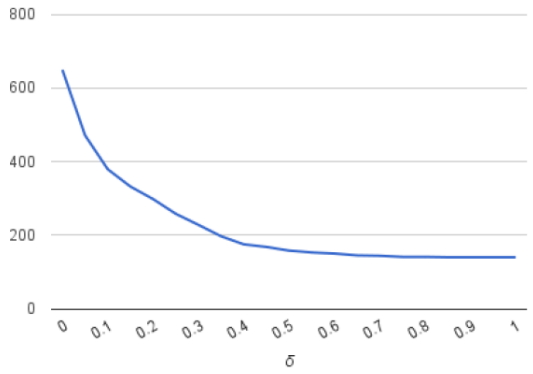


Figure 2.19: Number of graphs in a (0.4,δ)-coreset [13].

2.3.1.4 Adaptive miner (AdaGraphMiner)

The last mining mode extends from the previous two and differs in that it is adaptable to changes in the data stream and maintains a set of the current closed frequent graphs (Figure 2.20). There are two separate modes to this mining mode:

1. AdaGraphMiner-ADWIN mode: Uses an adaptive sliding window to maintain batches of graphs, where one single ADWIN instance monitors changes within the window and when a change is detected in a sub-session, it replaces the oldest batch with the new one.

2. AdaGraphMiner-ADAADWIN mode: Uses a separate ADWIN instance for managing the support of every closed frequent graph.

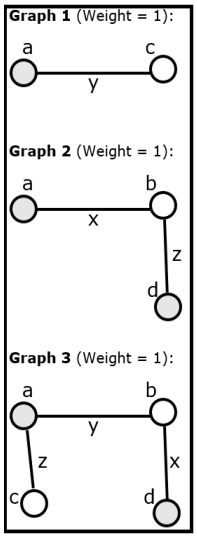
```
AdaGraphMiner(D, Mode, min_sup)
    Input: A graph dataset D, mode Mode and min_sup.
    Output: The frequent graph set G.

1   G ← ∅
2   Init ADWIN
3   for every batch b_t of graphs in D
4         do C ← Coreset(b_t, min_sup)
5            R̄ ← ∅
6            if Mode is Sliding Window
7               then Store C in sliding window
8                    if ADWIN detected change
9                       then R̄ ← Batches to remove
                                  in sliding window
                                  with negative support
10           G ← Coreset(G ∪ C ∪ R̄, min_sup)
11           if Mode is Sliding Window
12              then Insert # closed graphs into ADWIN
13              else for every g in G update g's ADWIN
14  return G
```

Figure 2.20: Pseudocode for AdaGraphMiner mode [13].

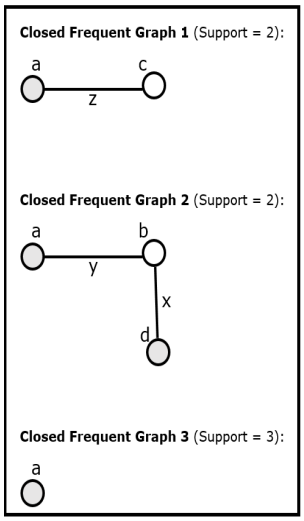
2.3.1.5 Experimental evaluation

Using batches of 10,000 instances and a minimum support of 40%, $\delta = 0$, 50 Giga-bytes of memory, the authors performed experiments to compare the non-incremental miners (such as CloseGraph and MoSS) and the incremental mode, IncGraphMiner. For small NCI databases (with only 250,000 graph instances), the IncGraphMiner was shown to use more time but less memory. However, for large ChemDB datasets (with almost 4 millions graph instances), the IncGraphMiner outperformed its counterparts in all aspects.

A comparison of the adaptation to change between the three modes of Bifet's approach shows that the incremental mode, IncGraphMiner, is the slowest to adapt to changes as it does not have any forgetting mechanism. When comparing the other two modes, the AdaGraphMiner-ADWIN mode had a slower response as there is only one ADWIN instance used to detect changes in the whole data stream. AdaGraphMiner-ADAADWIN mode on the other hand maintains a separate ADWIN instance for each closed frequent graph, hence was able to respond to changes much faster.



(a) Example of general graph weighted transaction dataset.



(b) Coreset with minimum support 50% for the transaction dataset of (a).

Figure 2.21: Bifet's approach input and output for general graphs.

2.3.1.6 Two types of input data for Bifet's graph batch mining framework

The output for Bifet's approach uses the pattern growth approach to find all complete closed frequent graphs of the dataset, given that they satisfy the user-defined threshold and are closed. Figure 2.21 is an example of general graphs represented in the NEList format which we will describe in detail later in Section 3.2. Figure 2.21 (b) shows a (50%,δ=0)-coreset, meaning we have a coreset of closed frequent graph which satisfies minimum support threshold of 50% and δ value of 0. Since the NEList format is vertex based, we assign arbitrary values to the edges. From this weighted general graph input we obtain the output shown in Figure 2.21 (b). To clarify this, Closed Frequent Graph 1, G, shown in Figure 2.21 (b) is contained in Graph 1 and Graph 3 of Figure 2.21 (a). This explains why the absolute support value for G is two. As the total number of graph instances is three, the relative support for G is 67%. Since the relative support is higher than the user-defined threshold (50% in this example), G is considered a frequent graph and since there is no frequent proper supergraph of G that shares the same support as G we can declare that G is a closed frequent graph.

Likewise, for the chemical molecular graphs represented in the SMILES format (described in detail in Section 3.1) we see that the third closed frequent graph G $(C - C - S - N)$ of Figure 2.22 (b) appearing in three of the six input chemical molecular graphs of Figure 2.22 (a). Hence, the relative support value of G is equivalent to the user-defined support threshold (50% in this example). In addition, no proper supergraph of

Transaction Id	Graph	Weight
1		1
2		1
3		1
4		1
5		1
6		1

(a) Example of molecular graph weighted transaction dataset [13].

Graph	Support
	6
	6
	3
	3
	3
	4
	5

(b) Coreset with minimum support 50% for the transaction dataset of (a) [13].

Figure 2.22: Bifet's approach input and output for chemical molecular graphs.

G was found to share the same support value as G. As a result, G was added to our closed frequent graph coreset (Figure 2.22 (b)).

2.3.2 Frequent subgraphs in dynamic networks

Deletion and insertion of edges and vertices in graph-based representations is common in many real-world applications, changes in data can occur rapidly. Wackersreuther et al. [14] takes a different approach to frequent graph analysis. Instead of finding patterns within static graphs, the authors focus on changes that occur within the graphs, or dynamic graphs. To better understand the work, we first describe two concepts.

Definition 8. ***Time Series:*** *Given a sequence G_{ts} of n graphs $\{G_1, ..., G_n\}$ with $G_i = \{V_i, E_i\}$ for $1 \leq i \leq n$, we define G_{ts} to be a time series of graphs if $V_1 = V_i$ for all $1 \leq i \leq n$. G_i is the i-th state of G_{ts} and A_i is the adjacency matrix of the i-th state. In time step i, labels L_i are assigned to nodes and edges.*

Definition 9. ***Dynamic Graphs:*** *Given a time series of graphs G_{ts} with n states, the dynamic graph $DG(G_{ts})$ of G_{ts} is defined as $DG(G_{ts}) = (V_{DG}; E_{DG}; \lambda)$, where $V_{DG} = V_i$ for all $1 \leq i \leq n$ and $E_{DG} = \bigcup_{i=1}^{n} E_i$. The mapping $\lambda : E_{DG} -> L \cup \epsilon$ maps each edge e in E_{DG} to a string λ(e) of length n, referred to as the edge existence string of e. Let us denote the i-th character of λ(e) as λ(e(i)). i(e) = ϵ if e does not exist in state i of G_{ts}. If e does exist in state of G_{ts}, then λ(e(i)) = L_i(e).*

The authors present a solution to show changes made to graphs in a volatile data stream. Figure 2.23 shows an example of three different time series of a graph and its summary (or the union) on the right-hand side. We see each of the instances in the time series presents a graph with different edges connecting the same set of vertices, this represents the changes to our graph through addition and deletion of edges and we can summarize the changes in a single dynamic graph (the right-most graph in Figure 2.23) where the attribute on the edges represents the value of the edge in each time-bin and ϵ represents the absence of an edge.

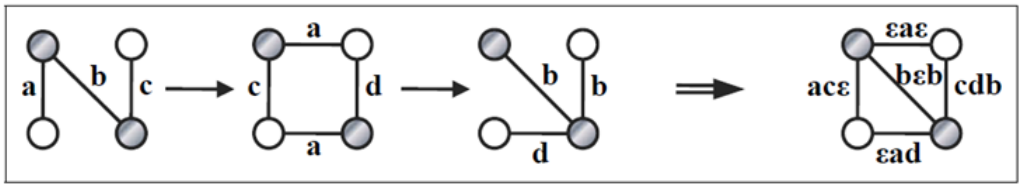
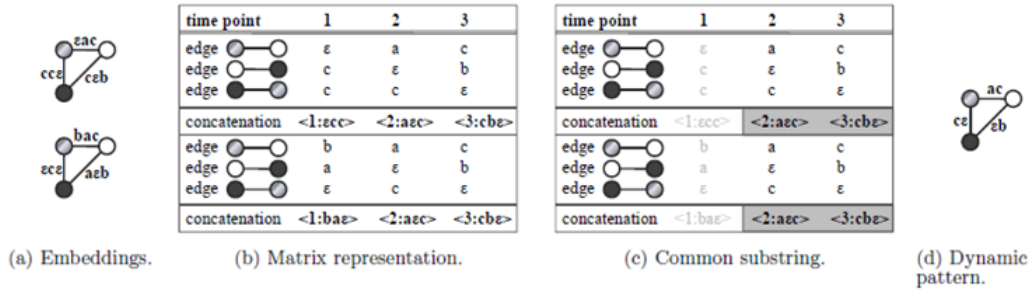


Figure 2.23: Graph transformation and summary [14].

.

Dynamic frequent subgraph discovery is separated into two stages. First by finding frequent subgraphs in a union graph using common frequent graph mining algorithms. Second, by searching for dynamic patterns from the resulting static dataset. Figure 2.24 gives an overview of the process behind the algorithm. The static embeddings (a) from a frequent subgraph S is input. For each embedding S_i of S, the edges are ordered by canonical labeling and then stored in a matrix as shown in (b). Each index of the matrix (S_i) contains a character which is concatenated into a string representation. Frequent substring discovery is performed on all resulting substrings by finding all common substrings (c), thus giving us a dynamic pattern embedding of S (d).



(a) Embeddings. (b) Matrix representation. (c) Common substring. (d) Dynamic pattern.

Figure 2.24: Transition from static to dynamic embeddings [14].

.

The dynamic algorithm was used to analyze data for a time series of yeast gene ex-

pression levels (represented as labeled graphs). The authors compute the frequencies of significant static subgraphs which is used to form the value p, the probability of it occurring in a random graph of the same distribution. This is then used to deem if a dynamic pattern is significant. Finally, a dynamic frequent subgraph will only be considered significant and processed if both its associated frequent subgraphs and dynamic patterns are significant.

2.3.3 Dense pattern mining

Data from the web and telecommunication applications can often be represented as a long sequence of edges. From these streams of edges, we can obtain valuable information by finding frequent and dense patterns from the large and dense graphs we obtain in the underlying interactions. Agarwal et al. [15] presents an algorithm for finding these dense graph structures within a graph stream through the use of a probabilistic model. The model is then used to create a summarization of graph streams which can be used for further mining.

The dense pattern mining algorithm is separated into two phases. In the first phase, the algorithm determines the correlated node patterns (patterns whose *pattern affinity* is greater than the user-defined threshold). The second phase attempts to find the subsets which satisfy the edge-density constraint with a user-specified threshold.

The data obtained is then mapped into a binary table with rows corresponding to graph identifiers and columns as node identifiers. For each graph entry, the node value is set to 1 if it is included. Hence, using the min-hash approach, the authors can determine all entries in a given subset by finding columns with the value 1, allowing them to implicitly sample sets of columns which have at least one entry that has the value 1.

This set of min-hash values will constantly be maintained and updated throughout the stream and once coherent node patterns have been found, it can be used to determine the dense patterns in the data. A new *graph fragment* database is formed using transaction sets generated from the min-hash table, which is created from graph fragments that contain subsets of nodes and the edges that connect all nodes in the subset.

2.3.4 StreamFSM

A defining characteristic of real-world data is that they contain information about nodes and edges as well as being dynamic in nature. Changes in data can include:

1. *Addition* of edges or nodes to the graph over time.
2. *Deletion* of previously existing edges or nodes.
3. *Modification* of edges and nodes attributes in the network.

These changes in the graph can be represented as either snapshots of static graphs, as seen in Section 2.3.2, or representing the graph as a stream of edge and node updates. Ray, Holder and Choudhury [12] presented an alternative approach to finding subgraphs in large dynamic graphs using the latter approach. StreamFSM finds the current set of frequent subgraphs in a dynamically labeled graph by looking at regions that have been changed due to batch updates.

The main graph G and the map of the enumerated subgraphs C are initialized as empty sets. For every batch of updates U_i we create a new set of graph transactions for it and then add it to the main graph G. For each edge e found in the current batch of updates, a region of the graph is extracted around it by finding the 1-hop neighborhood of the endpoints of e (the number of edges will not exceed a user-defined threshold to avoid a star-shaped pattern which increases computation time). At this point, all edges that have been extracted are marked and we add the sample to the list of transactions T. This process is repeated for every batch U_i and C is constantly updated to reflect the current counts of subgraphs.

The authors evaluate their algorithm by looking at several factors, using two artificial datasets, a Twitter dataset and a HETREC dataset. They ask the question, can the algorithm provide reports on subgraphs in a timely manner? To evaluate this, StreamFSM was executed with varying neighborhood sample parameters. From Figure 2.25, showing the relationship between average batch running time and the number of neighbours, we can see a constant running time for all datasets except for HETREC, with it reaching an optimal point around five neighbors.

Comparisons between StreamFSM and other miners were also made, most notably between the two publicly available large graph miners, GERM [17] and SUBDUE [18]. Table 2.1 shows the results of finding frequent subgraphs from the Twitter dataset and we see StreamFSM out performing the GERM algorithm in time and accuracy. SUBDUE on the other hand finished faster but produced fewer results than StreamFSM.

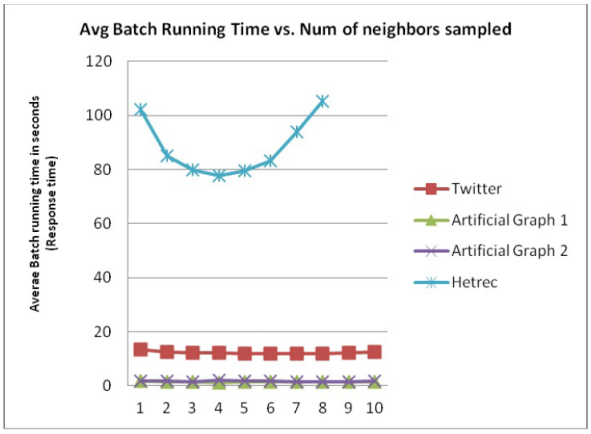


Figure 2.25: Average batch running time vs. Number of neighbors sampled [12].

.

This is because the Twitter dataset is difficult for frequent subgraph miners to process as it contains only one type of node label and two types of edge labels.

Table 2.1: Comparison between StreamFSM, GERM and SUBDUE [12].

StreamFSM	GERM	SUBDUE
Finished in 4817 sec with max neighbors set to 10 and found 23 freq subgraphs.	Did not finish in more than 8 hrs found 9 freq subgraphs.	Finishes in 1305 seconds, but only gives 2 edge freq subgraphs.

.

2.3.5 Summary

The techniques described in this section all work on frequent graph mining in a dynamic environment. To summarize:

1. AdaGraphMiner mode of Bifet's approach (Bifet et al. [13]): finds closed frequent graphs in weighted graph datasets and is adaptable to changes in the data stream.
2. Dynamic Graph Miner (Wackersreuther et al. [14]): creates dynamic embeddings of graphs by analysing edge connections of graphs in different timeslots.
3. Dense Pattern Miner (Agarwal et al. [15]): passes graphs into a binary min-hash map for further analysis.
4. StreamFSM (Ray, Holder and Choudhury [12]): populates graphs by batches and only updates sections of the graph by the edges seen in the update.

We see that each of these approaches works effectively with dynamic data, each uses its own techniques to simplify complex inputs, outputs and computations. Recurring

techniques were used. All four take input in batches, reduce the representation of graphs to simpler or more manageable forms (min-hash table in [15] and substring representations in [14]) and reduce their search space by eliminating uninteresting or infrequent subgraphs.

However, we notice that there are a few major differences between the techniques. First we notice that the input and output of Bifet's approach is vertex centralized, the edges on the other hand can be randomized. Figure 2.21 (a) and Figure 2.21 (b) show that the values of the vertices are used to find the frequent graphs and we assign random values (i.e. x, y, z) to the edges, whereas for the Dynamic Graph Miner we see a large emphasis on edges. The values of edges (whether existent or not) are used to construct graph fragments (as shown in Figure 2.23 and Figure 2.24). Hence, the focus of the inputs may differ between each algorithm.

The output for Bifet's approach uses pattern growth method to find all complete closed frequent graphs in the dataset, given that they satisfy the user-defined minimum support threshold and are closed. We see that for the given input shown in Figure 2.21 (a), we obtain the three closed frequent graphs shown in Figure 2.21 (b): closed frequent graph 1 appears in graphs 1 and 3, closed frequent graph 2 appears in graphs 2 and 3 while closed frequent graph 3 appears in all three given input graphs. However, this is not the case for other techniques such as the dense pattern mining algorithm in Section 2.3.3, where the algorithm outputs the fragments of a frequent graph and maintains the frequent patterns in a min-hash map.

To conclude, we note the common traits among the algorithms described in this section but we also compare their differences in the input, usage and applications in frequent graph mining. We will compare some of the techniques discussed in this section with our approach in Chapter 4.

2.4 Change detection techniques

Many real world applications deal with non-static data streams. Hence, it is crucial to integrate effective and efficient change detectors into those applications. This section first looks at four basic categories of change detection techniques and reviews the core ideas of those techniques from a high level. Following the general overview, Section 2.4.2 discusses the ADWIN algorithm. Section 2.4.3 is about One Pass Concept Change Detection for Data Streams (a.k.a SEQDRIFT). Since we implemented SEQDRIFT as the change detector in our approach, it is more logical to describe key ideas of the SEQDRIFT algorithm in Chapter 4. For this chapter we only compare and contrast

ADWIN and SEQDRIFT in Section 2.4.3.

2.4.1 Overview

Sebastiao and Gama [20] present a survey on recent change detection techniques. The authors studied the algorithms that used drift detection methods to detect abrupt, moderate or gradual changes in data distributions. The four algorithms presented in this paper are:

1. **Statistical Process Control (SPC)**: Constantly monitors for error during streaming observations. When the error exceeds the first (lower) threshold, the system changes to warning mode and stores the time of the observation. From this point, if the error reverts back then the threshold is canceled, or else if it exceeds the second threshold, it declares a change in distribution, and drift detection is declared.

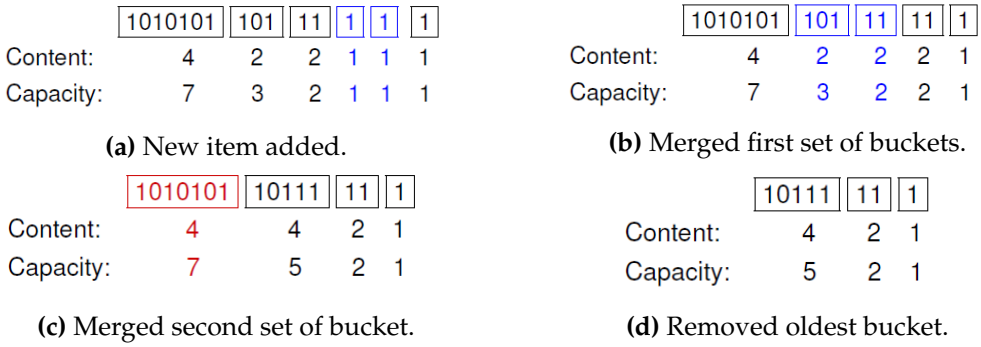
2. **Adaptive Windowing (ADWIN)**: The adaptive windowing approach keeps a sliding window W which receives input from the data stream and is used to compare with two sub-windows of W, W_1 and W_2. When the difference in the sub-windows average is high enough we drop the window W and assign a new distribution based on W_1 and W_2. A detailed explanation will be given in Section 2.4.2.

3. **Fixed Cumulative Windowing Schemes (FCWM)**: This method starts by constructing histograms using the Partition Incremental Discretization algorithm (PiD) [21]. This algorithm was designed to process histograms from high-speed data streams and change detection is identified when a reference window (distribution observed in the past) and current window (distribution of most recent data) exceeds a threshold when compared using the Kullback-Leibler divergence.

4. **Page Hinkley Test (PHT)**: PHT detects changes through the change in the average of a Gaussian signal. The value is then passed onto a cumulative variable which tracks the difference between observed values and its mean up to the current time slot.

2.4.2 ADWIN

As mentioned in Section 2.4.1, ADWIN is a change detection algorithm that utilizes a sliding window to find changes in data distribution. The algorithm manages the window size by growing it when no change occurs and shrinking it when we detect change in data, thus maintaining the windows size to balance between variance and reaction time. This approach keeps window W in memory and updates it with O($logW$). Because of this low time and memory usage, ADWIN can be used as a



(a) New item added.

(b) Merged first set of buckets.

(c) Merged second set of bucket.

(d) Removed oldest bucket.

Figure 2.26: Example of ADWIN sliding window with $M = 2$ [22].

learning algorithm that can create many instances of itself to maintain and update the statistics from which it builds its model.

The process of ADWIN and the sliding window W can be seen in Figure 2.26 with the defined M value (the value that controls the memory used and cutpoints checked) as two. For this figure, capacity indicates the size of each bucket while content means the frequency of ones the bucket contains. The first two diagrams, (a) and (b), show the insertion of a new item and since our M value is two, which means we cannot have more than two buckets that has the same content number and we must merge exceeding buckets giving us the resulting window in (c). Finally, since we detected a change, we remove the oldest bucket from the sliding window, giving us (d).

Although our example only shows a prequential representation, as mentioned in our introduction, we can also use real values by changing our content to be the sum of all real values and restrict the capacity to the power of two.

ADWIN keeps a sliding window W with the most recently read x_i. Let $\hat{\mu}_W$ (in Figure 2.27) denote the (observed) average of the elements in W, and μ_t (in Figure 2.27) denote the (unknown) average of μ_t for $t \in W$. As shown in the Figure 2.26, since we remove entire buckets when change is detected, there will be less computation, complexity and it will be less expensive than deleting single items. However, by deleting entire buckets we would obtain more jagged graphs in the case of gradual changes as ADWIN is more adaptable to abrupt changes. This can be seen in Figure 2.27 where the left diagram shows the quick and sharp adaption of window size W due to a sudden drop in μ_t whereas the diagram on the right displays a gradual decrease of μ_t with a jagged pattern for window size W.

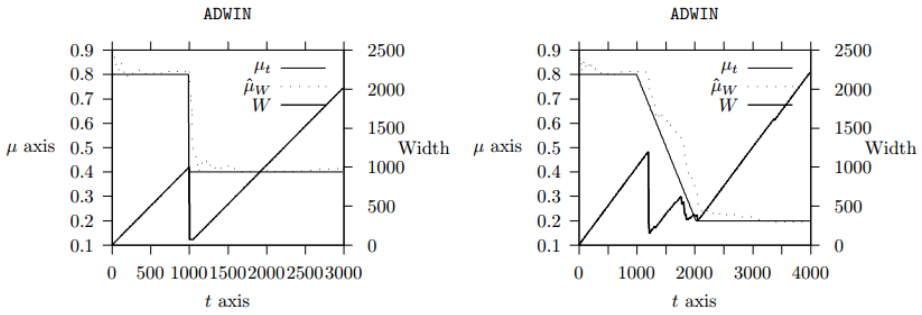
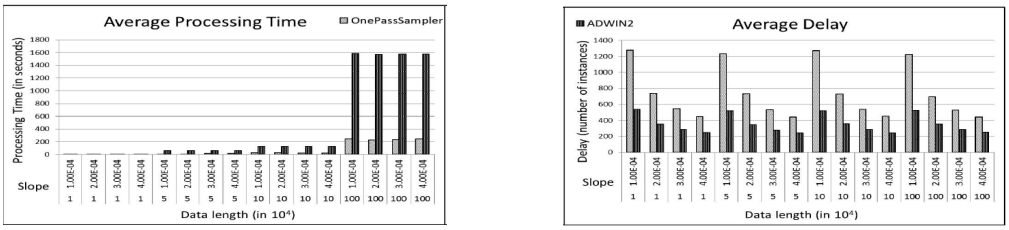


Figure 2.27: Output of ADWIN with abrupt(left) and gradual(right) change [22].



(a) Average processing time.

(b) Delay time.

Figure 2.28: Comparative Change Detection Performance of SEQDRIFT and ADWIN [23].

2.4.3 SEQDRIFT

Sakthithasan et al. [23] present the change detector SEQDRIFT (One pass concept sampler concept change detector) and draw comparisons to the ADWIN algorithm. The authors noted two major differences: Firstly, the policy in determining cutpoints is different. A cutpoint, in ADWIN, lies in the boundary between brackets meaning for N buckets it will have $N - 1$ possible cutpoints. SEQDRIFT on the other hand never re-examines previous boundaries for cutpoints and only examines the boundary between the collection of blocks and the newly arrived block, hence allowing us to search for cuts in a single pass.

The second difference lies in the estimation strategy for assessing data segments. ADWIN uses exponential histograms for estimating means whereas SEQDRIFT uses a random sampling algorithm to determine the means. Exponential Histograms could cause inaccuracies as buckets that are too small could give inaccurate results. The results show that SEQDRIFT produces competitive false positive and true positive rates to ADWIN, hence SEQDRIFT is shown to out-perform ADWIN in a broad spectrum of efficiency.

Furthermore, a comparative study on the performance of SEQDRIFT and ADWIN was conducted. Figure 2.28 (a) illustrates that ADWIN is much slower at processing streams than SEQDRIFT, with the difference being greater as the stream becomes longer. As explained earlier, ADWIN performs repeated scans of the histogram to find every possible combination of cuts, while SEQDRIFT does one single pass through the window to assess each block and check if they are sufficiently different from the previous blocks in its memory buffer.

However, we can also see from Figure 2.28 (b) that ADWIN has a better mean delay count than SEQDRIFT. This is due to SEQDRIFT needing a larger window to decide whether the newly arrived block is different enough compared to previous blocks. The authors noted that the difference in delay time will be reduced with increasing gradient of change and by changing the parameters for warning levels and block size limits in SEQDRIFT.

2.4.4 Summary

This section discusses an overview of recent change detection techniques. Sebastiao and Gama [20] presents a summary of recent change detection algorithms and draws comparisons between them. Bifet and Gavalda [22] compare the new ADWIN algorithm with its older versions, detailing improvements in performance. Sakthithasan et al. [23] contrast the SEQDRIFT algorithm with ADWIN and compare differences in finding cutpoints, assessing data and the efficiency of both algorithms.

2.5 Conclusion

In this chapter, we have presented existing work from three fields that are closely related to our research. As shown in Figure 2.1, the three research fields form an interconnected work-flow for mining frequent graphs in a data stream environment. In addition, we have provided some necessary preliminary concepts in Section 2.1 for understanding the research area. This chapter, together with the next chapter, provide foundational knowledge for our work.

Chapter 3

BACKGROUND

This chapter provides detailed explanations of the data representations used in our experiments, which are considered as background concepts of our research. We introduce two types of datasets: chemical molecular dense datasets in SMILES format and social network sparse datasets in NEList format (Table 3.1). The formats of the two types of datasets are not exchangeable due to the following reasons: SMILES is a specially designed language for representing only chemical atomic elements and bonds. The language itself has very strict syntax, any inputs that are not part of the language's predefined chemical atomic elements or bonds will not be considered valid and represented properly in SMILES. Normally, social network datasets have vertexes that are not chemical atomic elements which makes it impossible to represent social network datasets in SMILES format. On the other hand, it is also difficult to represent chemical molecular datasets in NEList format due to the complexity of chemical bonds which are not properly predefined and supported by the NEList format. As indicated by (Table 3.1), we categorize chemical molecular datasets as dense datasets and social network datasets as sparse datasets. We provide our explanations and proof for these categorizations in Section 3.3.

Sections 3.1 and 3.2 describes the graph input formats SMILES and NEList respectively. We compare the differences between the dense datasets in SMILES format and sparse dataset in NEList format in Section 3.3.

Table 3.1: Two types of input datasets for our experiments.

Dataset Type	**Density/Sparsity**	**Representation Format**
Chemical Molecular	Dense	SMILES
Social Network	Sparse	NEList

3.1 SMILES

SMILES (simplified molecular-input line-entry system) [28] is a structured line notation for representing chemical compounds and reactions. Based on principles of molecular graph theory, SMILES allows specification of chemical compounds ranging from

simple to the most complex in a succinct and natural form and most importantly, in a form that can be understood and processed by computer systems.

Due to the complexity of representing a 2D chemical structure as a text string, the SMILES notation is enforced by several rules to maintain its accuracy and validity.

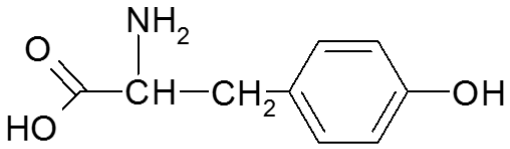
1. *Atom:* Atoms within the organic subset $\{B, C, N, O, P, S, F, Cl, Br,$ and $I\}$ are represented by their respective atomic symbols, whereas atoms not within the subset require square brackets e.g [Au], hydrogen atoms H are normally implied unless specified otherwise. Atoms with formal charges (positive or negative) must also be specified through the use of square brackets e.g [OH^-].

2. *Bond:* Single, Double, Triple and Aromatic bonds are represented by the symbols -, =, # and : respectively. Single and aromatic bonds are usually omitted. The SMILES notation in Figure 3.1 omits single bonds and shows the branching double bond to an O atom.

3. *Branches:* Atoms in a branch are represented through enclosed parenthesis. Figure 3.1 shows an example of three branches, one to a double bond to the O atom, one to the cyclic structure and one from the branched cyclic structure to a O atom.

4. *Cyclic Structures:* Cyclic structures are represented by breaking a single bond or aromatic bond in the ring. The designated ring opening bonds have a digit followed immediately from the atom. Figure 3.1 shows the cyclic structure c1ccc(O)cc1, which represents a benzene ring with a branching O atom.

5. *Aromatic Structures:* Aromatic structures are implied through the use of lower case letters. Figure 3.1 shows an example of the cyclic and aromatic structure, Benzene ring through c1ccc(O)cc1.



NC(Cc1ccc(O)cc1)C(=O)O

Figure 3.1: Chemical compound (above) and SMILES notation example (below) [29].

3.2 NEList

NEList is a format used to represent a dataset of general graphs, using multiple line descriptors to represent vertexes and edges in a graph. Figure 3.2(left) shows an

example NEList representation of a graph (Figure 3.2(right)) with four nodes and three edges. The starting value of our list indicates what type of value we want to represent, the value "v" indicates that we want to introduce a new vertex to our graph, the second value in the line is the vertex number while the value is the vertex label, the vertexes must be numbered consecutively starting with 1 for each graph. A starting letter of "e" indicates that an edge is added. The two following values after "e" indicate the two vertexes we are linking which is followed by a edge label that is randomly assigned to x, y or z. The value after a "g Graph" starting value is the graph label or descriptor which can be used to uniquely identify and label each graph in our general graph input dataset. Finally, the values after "x" is used to indicate if the graph will be processed or not, although this condition is not used in our experiments.

```
v 1 a
v 2 b
v 3 c
v 4 d
e 1 2 x
e 1 3 z
e 2 4 y
g Graph 3
x 0
```

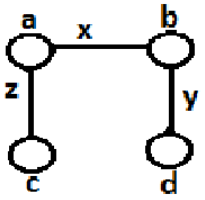


Figure 3.2: NEList format (left) and corresponding graph (right).

Currently, the majority of research on graph mining techniques deal with chemical molecular datasets represented in SMILES format. However, since we are interested in applying frequent graph mining techniques to large social network analysis, we also use social network graphs in NEList format as input datasets for our experiments.

The social network graph datasets used for our experiments were obtained from the Stanford Large Network Dataset Collection (SNAP) website [32]. These real-world datasets are input into NEList, categorized into two different formats which are shown in Figures 3.3 and 3.5. In these figures, we present three levels of representation, the first level (3.3 and 3.5) is a text representation of the graph, the second level (3.4 (a) and 3.6 (a)) is the corresponding NEList representation of the graph and finally, the third level (3.4 (b) and 3.6 (b)) shows the visual representation of the graph.

1. *From-to Node Input:* Figure 3.3 shows a snapshot of part of the From-to node input format. The first value in each line signifies the value of the "from node" of an

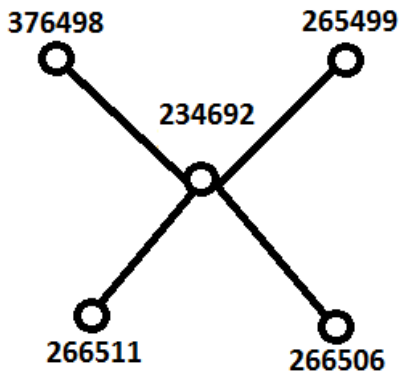
edge while the second value is the "to node" of the edge. We see from Figure 3.3 that we create four edges from the "234692" node, giving us the NEList format in Figure 3.4 (a), which correspond to the bipartite graph shown in Figure 3.4 (b).

```
2  # Youtube
3  # Nodes: 1134890 Edges: 2987624
4  # FromNodeId    ToNodeId
5  234692  265499
6  234692  266506
7  234692  266511
8  234692  376498
9  1157560 1157561
10 1157560 1157562
11 1157561 1157563
12 1157564 1157567
13 1157570 1157571
14 1157570 1157572
15 1157570 1157573
```

Figure 3.3: From-to node input format

```
v 1 234692
v 2 265499
v 3 266506
v 4 266511
v 5 376498
e 1 2 y
e 1 3 z
e 1 4 y
e 1 5 y
g Graph 143652
x 0
```

(a)



(b)

Figure 3.4: (a) NEList output of Figure 3.3 and (b) visual representation of the graph.

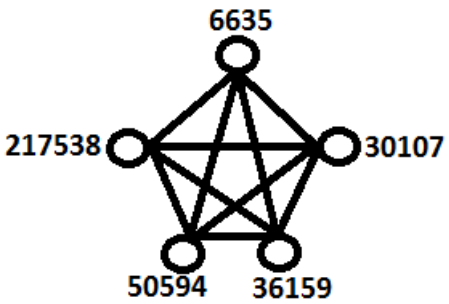
2. *Single Line Community Input:* The Single line community format, as shown in Figure 3.5, uses one line of values, separated by tabs, to produce complete graphs (graphs where every distinct pair of vertexes are connected by an edge). Consider the highlighted line of input in Figure 3.5. We produce its NEList format representation in Figure 3.6 (a) and the complete graph representation in Figure 3.6 (b). Note that the graph number for the Single line community format corresponds to the line number of the input dataset.

```
3159  203594  226899  270113  396305  441404  488105
3160  64740   132757  189195  274880  542176  542577
3161  20041   250801  520621
3162  63835   172852  241554
3163  65457   206353  244897
3164  214252  519950
3165  65457   206353  244897
3166  111786  232062
3167  65457   206353  244897
3168  251488  362996
3169  21692   347888
3170  6635    30107   36159   50594   217538
3171  90395   151045  301601  457060
3172  204470  204479
3173  299493  329558  368838
3174  142755  227793  289924
3175  177400  538433
3176  34079   545016  545603  546068  547980
3177  103248  251749  270289  356975  468885
```

Figure 3.5: Single line community input format

```
v 1 6635
v 2 30107
v 3 36159
v 4 50594
v 5 217538
e 1 2 y
e 1 3 x
e 1 4 z
e 1 5 y
e 2 3 z
e 2 4 z
e 2 5 z
e 3 4 x
e 3 5 y
e 4 5 z
g Graph 3170
x 0
```

(a)



(b)

Figure 3.6: (a) NEList output of Figure 3.5 and (b) visual representation of the graph.

3.3 Differences between dense dataset and sparse dataset

We are interested in a stream of possibly infinite set of graphs, G. However, this is not realistic and instead we choose to use a finite subset of G which gives us a point of disclosure. As for G, we use both dense dataset and sparse dataset in our experimental evaluations. This section provides a direct comparison between the two types of input datasets.

Due to the nature of the datasets, we use chemical molecular datasets represented in SMILES format (as described in Section 3.1) as our dense datasets. Whereas, general graph datasets represented in NEList format (as described in Section 3.2) are used as our sparse datasets. The sizes of these two types of datasets differ greatly despite them sharing the same number of total instances. The size of dense datasets in SMILES format average around 15MB as they contain fewer entries of long strings representing chemical compounds. On the other hand, the size of sparse datasets in NEList format range between 500MB to as large as 6GB due to the format using hundreds of lines of text to denote a graph that can be represented in SMILES in a single line.

The fundamental difference between dense and sparse datasets is the number of vertex values it can select from. The two key components of a graph are the vertexes and edges, however since our technique is vertex based, we focus primarily on values that appear in vertexes rather than edges. We define datasets to be dense if the number of possible vertex values we can select from is limited. For example, datasets in SMILES format have less than one hundred possible vertex values, which are the atomic symbols of atoms in an molecule. Additionally, there is a limited number of nodes and there are many chemical compounds, hence the resulting datasets are dense. Whereas datasets that have a large range of vertex values are defined as sparse datasets. An example of this is shown in social networking datasets where each user is represented by an unique value. When comparing dense and sparse datasets with the same number of vertex instances, we would expect to see small numbers of densely connected nodes for dense datasets and large numbers of loosely connected nodes for sparse datasets.

Another indirect difference we noticed between the two datasets is the difference in the optimal range of support values for users to obtain sensible results. Since dense datasets have a limited range of node instances, we require large support values (40%) to prevent every subgraph from being frequent. Whereas for sparse datasets we use small values (0.1% to 1%) so we will be able to obtain frequent graphs from expansive datasets even with few repeating occurrences.

3.4 Summary

This chapter explores some of the core ideas and concepts behind our work. We present two different types of input data, chemical molecular data and social network data, which are represented in SMILES format and NEList format respectively. In Section 3.1, we describe the applications and notations of SMILES while in Section 3.2 we present the NEList format and its two different input data formats, From-to node and Single line community. We also introduce three levels of a graph representation in this section

with NEList format being our second level. Finally, we conclude this chapter by presenting the differences between chemical molecular dense dataset and social network sparse dataset.

Chapter 4

OUR APPROACH

In this chapter, we describe our approach and contributions to frequent graph mining in data streams. Corresponding to the first figure in Chapter 2, Figure 4.1 presents our high level integration of the change detector, SEQDRIFT, into Bifet's graph batch mining framework [13]. The diagram presents the key elements in the framework; Firstly, the underlying mining engine, MoSS (details in Section 2.2.2.1.), which performs frequent substructure mining on molecular datasets. Secondly, the change detector, ADWIN (details in Section 2.4.2), that detects changes in frequent graphs. In our approach, we replace the change detector ADWIN with SEQDRIFT, allowing SEQDRIFT to monitor changes in data streams.

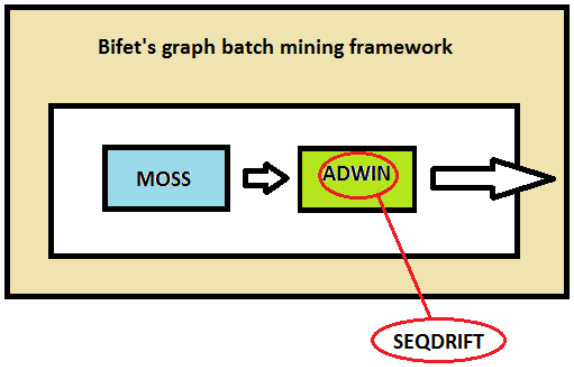


Figure 4.1: Integration of SEQDRIFT into AdaGraphMiner.

The structure of this chapter is as follows: Section 4.1 details the benchmark technique, AdaGraphMiner-ADAADWIN. We provide detailed examples explaining the strengths of the algorithm. Section 4.2 lists the research gaps we have found from existing research and the contributions we make to address these gaps. Section 4.3 explores the change detector, SEQDRIFT in detail, including how the algorithm operates to detect changes. Section 4.4 explains how we created the SeqDriftGraphMiner mode by integrating the change detector, SEQDRIFT into Bifet's graph batch mining framework [13] (Section 2.3.1). Section 4.5 shows how we convert the general graph formats: From-to node format and Single line community format to NEList format. Finally, Section 4.6

outlines the in-depth experimental analysis we have performed on the output of closed frequent graphs. We then introduce the concepts of real drift and internal change and examine the differences between them.

4.1 AdaGraphMiner-ADAADWIN mode

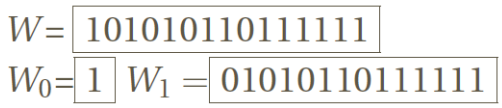
Bifet et al. [13] presented a comparison on performance between four different modes of their batch mining framework. The AdaGraphMiner-ADAADWIN mode was shown to have out-performed other modes in terms of how quickly it was able to adapt to changes in the distribution of graphs in an evolving stream scenario, hence we will be using this mode as our benchmark technique.

Previously, we discussed Bifet's graph batch mining framework in detail in Section 2.3.1 with the core concepts of AdaGraphMiner-ADAADWIN described in Section 2.3.1.4. Since the performance of change detectors is one of our major research focus, we explore ADWIN further in this section, as it is the change detector used by the AdaGraphMiner-ADAADWIN mode.

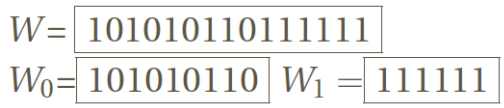
ADWIN uses a sliding window to store data from the input stream. The window shifts in size, growing when distributions remain stable and shrinking when change is detected. This process allows ADWIN to create balance between variance of data and its ability to react to change. ADWIN tunes itself to the changes it sees in the data stream, therefore it does not require input parameters or user moderation. In terms of efficiency when operating using prequential values (0 or 1), ADWIN can compute the number of true instances (1) in linear time $O(1)$. It will find all cutpoints in $O(logW)$ time and has the worst-case processing time of $O(logW)$.

Algorithm 2 shows a general outline of the ADWIN algorithm and we use Figure 4.2 as an example to complement it. The sliding window W is initialized and we perform mean comparisons at each point of time t (Figure 4.2 (a)). For each time slot t, we perform comparisons between the sub-window with traversed time slots W_0 and the sub-window which has unprocessed elements $\mid \hat{\mu}_{W0} - \hat{\mu}_{W1} \mid$ (Figure 4.2 (b)), we drop elements from the tail of W (Figure 4.2 (c)).

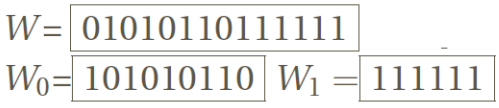
Although we use prequential values for our examples, it is difficult to represent social networking datasets in this form. When using real numbers such as the support values of our graphs, we maintain buckets of two elements, capacity and content. The content bucket stores the sum of the support values we wish to compute and the capacity is the number of elements we are computing but rounded up to be a power of two.



(a) Initialize sliding window W and sub-windows W_0 and W_1.



(b) Change detected.



(c) Drop elements from tail.

Figure 4.2: Example of ADWIN sliding window

Algorithm 2 ADWIN: Adaptive Windowing Algorithm

1: *Intialize Window* W
2: *for each* $t > 0$
3: *do* $W \leftarrow W \cup \{x_t\}$ (i.e., add x_t *to the head of* W)
4: ***repeat*** *Drop elements from the tail of* W
5: ***until*** $|\ \hat{\mu}_{W_0} - \hat{\mu}_{W_1}\ | \geq \epsilon_{cut}$ *holds*
6: *for every split of* W *into* $W = W_0 \cdot W_1$
7: *output* $\hat{\mu}_W$

4.2 Research gaps

This section presents the gaps of Bifet et al.'s [13] research and underlines the contributions that we've made to address these issues. For our three core contributions, we integrated a new change detector, SEQDRIFT, to Bifet's graph batch mining framework, expanded the framework to provide functionalities that allow us to analyze general graph input format and finally, we provide deeper analysis for outputs produced by miner.

4.2.1 First research gap

1. Bifet et al. compared the delay time of AdaGraphMiner-ADAADWIN mode with two other modes, IncGraphMiner mode and AdaGraphMiner-ADWIN mode. The results showed that AdaGraphMiner-ADAADWIN mode was comparatively

faster at detecting changes that occur in the graph stream. However, the run time and memory performance of the AdaGraphMiner-ADAADWIN mode was not evaluated, which are metrics that we are interested in exploring.

2. The AdaGraphMiner-ADAADWIN mode uses ADWIN as the change detector. However, better-performing change detectors have been developed in recent years and we are interested in exploring the performance of these new change detectors for closed frequent graph mining in data streams. We will replace ADWIN with SEQDRIFT in Bifet's graph batch mining framework and compare the performance in terms of running time, memory, delay time and note if there are any significant trade-offs between the change detectors.

4.2.2 Second research gap

The input for Bifet's graph batch mining framework has all been chemical molecular graphs in the SMILES format. However, we are interested in applying frequent graph mining techniques to large social network analysis. For this research gap, we attempt to convert social network datasets to the form of general graph representations in the NEList format as an input for Bifet's graph batch mining framework.

4.2.3 Third research gap

Bifet's research focused only on number of closed frequent graphs as an output. However, we are interested in providing a deeper analysis on the closed frequent graphs we obtain. For the last research gap, we will examine our outputs to extract patterns, trends and points of interest.

4.3 SEQDRIFT

SEQDRIFT [23], as introduced in Section 2.4.3, is a change detector that processes data sequentially, allowing for cheaper and faster computations with lower memory requirements than some recent change detectors. Corresponding to the first research gap as described in Section 4.2.1, we implemented SEQDRIFT and integrated it into the Bifet's graph batch mining framework, giving us a new mode named SeqDriftGraphMiner. We will then compare SEQDRIFT and ADWIN (Section 2.4.2) throughout this section and present the results we obtained for performance comparison between AdaGraphMiner and SeqDriftGraphMiner in Chapter 5 to see if SEQDRIFT is overall a better change detector.

4.3.1 Concept change detection

Given two samples S_1 and S_2 with μ_1 and μ_2 as the mean of the samples respectively. We state the null hypothesis H_0 is valid when $\mu_1 = \mu_2$, showing that the samples S_1 and S_2 are from the same distribution. The alternative hypothesis H_1 is valid when $\mu_1 \neq \mu_2$ and we can declare that S_1 and S_2 arrive from different distributions (Definition 10 [23]).

Definition 10. ***Concept change detection:*** *Let $S_1 = (x_1, x_2, .., x_m)$ and $S_2 = (x_{m+1}, .., x_t)$ with $0 < m < t$ be two samples of instances from a data stream where μ_1 and μ_2 are the mean of the samples respectively and distributions of the data stream $D = \{D_1, D_2, .., D_n\}$ where D_i is a distribution in the graph stream and $1 \leq i \leq n$. Hypothesis H_0 is true when $\mu_1 = \mu_2$ which implies $D_{S1} = D_{S2}$ where $\{D_{S1}, D_{S2}\} \in D$. Hypothesis H_1 is true when $\mu_1 \neq \mu_2$ which implies $D_{S1} \neq D_{S2}$.*

4.3.2 Bernstein Bound

SEQDRIFT requires well-established bounds between our random samples and the true population. The Hoeffding inequality has been widely used in many algorithms, including ADWIN. However, it is shown to overestimate the probability of large deviations of data in datasets with small variances [34]. Hence, SEQDRIFT uses the Bernstein bounds which provides tighter bounds for difference between the random sample and the true population. The Bernstein equality states the following:

$$Pr\left(\left|\frac{1}{n}\sum_{i=1}^{n} X_i - E[X]\right| > \epsilon\right) \leq 2\exp\left(\frac{-n\epsilon^2}{2\hat{\sigma}^2 + \frac{2}{3}\epsilon(c-a)}\right) \tag{4.1}$$

where $X_1, .., X_n$ are independent random variables, $E[X]$ is the expected value or population, ϵ is the cutpoint threshold, and $X_i \in [a, c]$ where a and c are random real numbers greater than zero and $\hat{\sigma}$ is the sample variance [23].

4.3.3 Algorithm overview

SEQDRIFT accumulates data instances into blocks of a user-defined size, b, differing from the parameter-free change detector, ADWIN. When using SEQDRIFT, the input data instances consist of a binary sequence of bits where 1 denotes a misclassification error and 0 denotes a correct classification. SEQDRIFT uses blocks of data instances as the basic unit as it is more efficient to test for concept changes than checking at the arrival of every instance.

As an example, suppose blocks B_1 and B_2 arrive at time t_1. SEQDRIFT would attempt to detect if concept change has occurred at the boundary between the two input blocks,

$B_1 \mid B_2$, to test for our hypothesis H_1, which can have two outcomes (as shown in Algorithm 3):

1. If H_1 is rejected, then we concatenate our two blocks into one single block, B_{12} and continue checking the boundaries between the existing block and the next block, $B_{12} \mid B_3$. For this comparison, we derive the sample mean of a random block of the concatenated block (size $2b$) of size b. This sample mean is again compared with the mean of the introduced block, B_3, which tests for hypothesis H_1 once again. This process continues until H_1 is accepted or we reach the end of our steam.

2. If H_1 is accepted at any boundary, then we declare concept change. We then remove the block or concatenated blocks on the left side of the boundary and replace it with the input block on the right. In the case of the given example, we would discard B_1 and shift B_2 to the left side.

Algorithm 3 SEQDRIFT algorithm

Given user-defined block size b
Given data stream D =$\{B_1, B_2 .. B_n\}$ *where size of* B_1 *and* B_x *is b and* $1 < x \leq n$
Initialize the left sub-window as the first block, $L = B_1$
for each block B_x *in D*
    $if(H_1(L \mid B_x) == True)$
        $L = B_x$
    $else$
        $L = L + B_x$

4.3.4 Random sampling

The random sampling process used by SEQDRIFT provides several benefits. Firstly, the sampling process on the left sub-window provides faster computation times compared to ADWIN. Both sub-windows are of equal size b, hence we retain the same level of robustness. Additionally, using the average value as comparison between blocks produces smooth variation in the data, making it more adaptable to noise compared to ADWIN.

The process of random sampling is required for the computation of the Bernstein bound. In a data stream environment, data instances in the same locality may not always be independent of each other as changes made at a instance could affect its following instances. Random sampling reduces the risk of dependency as we do not

always sample successive instances.

In circumstances where the difference in mean between the two sub-windows are significant not on the basis of drift confidence ($1 - \delta_{drift}$) but on the basis of warning confidence ($1 - \delta_{warning}$), we trigger a warning state where the sliding window scheme is used for the right sub-window, adding new data blocks to the right side instead of the left.

4.3.5 Memory management

SEQDRIFT manages memory through the use of arrays to store blocks of data. The array data structure provides fast access to specified data blocks and is used to represent the left and right sub-windows. When a new data block arrives, the data is stored in the right sub-window, which is compared with the left sub-window for significant differences. If no such difference exists (H_0), the data in the right sub-window is moved to the left sub-window, forming sets of homogeneous blocks.

An advantage SEQDRIFT has over ADWIN is its memory efficiency. SEQDRIFT concatenates and discards blocks of instances as it processes the stream while ADWIN is required to backtrack, maintain the history of previous candidates and cutpoints. Additionally, SEQDRIFT only requires the current state of its left and right sub-windows, ADWIN on the other hand, retains information on past candidates in the form of exponential histograms.

4.3.6 Change detection algorithms

The core algorithms used to detect changes are shown in Figure 4.3. Let S_l and S_r denote the left and right sub-windows respectively. The algorithm shown in Figure 4.3(a) is the function that determines if changes occur given the mean values $\hat{\mu_l}$ and $\hat{\mu_r}$ from S_l and S_r respectively, the difference threshold ϵ_{change} and the warning threshold $\epsilon_{warning}$. In the event that we detect change, that is if $(\hat{\mu_r} - \hat{\mu_l}) > \epsilon_{change}$, then we execute the process shown in Figure 4.3 (b), which transfers the content of S_r into S_l. When a warning state is triggered, the sample size increases which increases the precision in sampling, allowing the algorithm to be more sensitive to slow gradual changes.

4.4 SeqDriftGraphMiner mode

This section discusses how we implemented the change detector, SEQDRIFT, and integrated it into Bifet's graph batch mining framework. Initially, we create a SEQDRIFT

```
Input: μ̂_l, μ̂_r, ε_Change, ε_Warning
Output: Change || Warning || Internal
if ε_Warning ≤ |μ̂_l − μ̂_r| then
    if ε_Change ≤ |μ̂_l − μ̂_r| then
        if μ̂_r > μ̂_l then
            return Change;
        end
        return Internal;
    end
    return Warning;
end
return None;
```

(a) GetDriftType() [23].

```
Input: An instance(Ins), BlockSize, S_l,
       S_r
Output: True/False
Increment the instance counter;
S_l = S_r ∪ {Ins};
if At the block boundary then
    ChangeType = GetDriftType();
    if (DriftType is Change or Internal) then
        Remove all elements from S_l;
        Copy all elements of S_r to S_l;
        Remove all elements from S_r;
        Set SampleSize to BlockSize;
        if (DriftType is Change) then
            return True;
        end
        return False;
    end
    else if (DriftType is Warning) then
        Double the sample size;
        return False;
    end
    Copy all elements of S_r to S_l;
    SampleSize = BlockSize;
    return False;
end
```

(b) IsDrift() [23].

Figure 4.3: SeqDrift Algorithm.

instance for every closed frequent graph we obtain from each batch. For each SEQDRIFT instance, we use the parameter values set by the authors [23] which are set as follows:

1. Block size: 200
2. Left reservoir size: 1200
3. Right reservoir size: 200

The block size was set to 200 as it was shown to be optimal across a range of different gradients [23], the buffer of the left sub-window is set to a default value of six blocks while the buffer on the right sub-window is equivalent to one sample. We attempt to join the coresets produced from two consecutive batches by iterating through the following four steps for the purpose of monitoring changes that occur:

1. Step one: We create two maps, one that contains closed frequent graph instances and support values in the current batch and one for the previous batch. For the current batch, each closed frequent graph is checked using its SEQDRIFT instance and for each change detected, we increment a change detection counter C by one.
2. Step two: We find the difference in support between the current and previous batches by looking at the collection of SEQDRIFT instances. We then update graphs with support of 0 in the last batch.

3. Step three: We iterate through all closed frequent graphs in our coreset and remove any graphs that do not satisfy the minimum support defined.

4. Step four: We then update SEQDRIFT instances on new graphs that only appear in the current batch but not the previous batch. Finally, we calculate the number of closed frequent graphs based on the coreset we have updated using the collection of SEQDRIFT instances.

From the process described above, SEQDRIFT is shown to use different mechanisms compared to ADWIN when detecting changes. The first major difference is how each change detector determines cutpoints of processed data. From Figure 4.4, we see that ADWIN stores arriving graphs in buckets before adding them to the memory buffer, n buckets of graphs will therefore require ADWIN to examine $n - 1$ boundaries for possible cut points. SEQDRIFT, on the other hand, adds newly arrived blocks from the right reservoir to the collection in the left reservoir. Using this method, SEQDRIFT only needs to examine the boundary between the block collection and successive incoming blocks which can be performed through a single pass of the memory buffer.

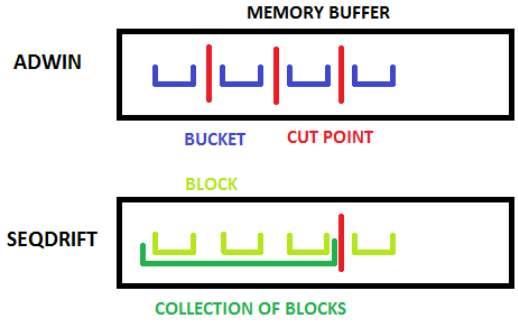


Figure 4.4: Determining cutpoints using ADWIN and SEQDRIFT.

The second major difference lies on how each change detector calculates the mean of the data segments. ADWIN maintains a exponential histogram to estimate mean values, this process is both computationally expensive and could yield inaccurate results when processing smaller buckets. SEQDRIFT uses a random sampling strategy on its left reservoir when calculating the difference in means, this process is both time efficient as there is no backtracking and prevents bias that occurs when evaluating the mean of blocks that are in the same graph distribution.

4.5 Processing general frequent graphs

Bifet et al.'s research focused on frequent graph mining on chemical molecular datasets. Although these types of datasets are commonly used to evaluate frequent graph miners, we wish to expand the applications of the miner so it can process general graphs in NEList format (as explained in Section 3.2). We have implemented two Java methods to convert From-to node format and Single line community format input data into NEList format separately. To address the research gap discussed in Section 4.2.2, we enhanced Bifet's graph batch mining framework so that it takes in input in NEList format, and produces graph mining outputs also in the same form.

4.5.1 From-to node format to NEList

Algorithm 4 shows pseudocode that converts From-to node format general graphs into NEList format. We begin by initializing the variables C, I and N shown in Line 2. For each line of the From-to node format file we delimit it into two variables, where n_1 represents the "from node" and n_2 represents the "to node". If n_1 is not equal to I and C is True, we know this is the first appearance of node instance n_1 and we set I to n_1, making n_1 the new current node evaluated and C to False so we know the recurring instances will not be the first instance of node n_1.

If n_1 is equal to I then this implies we are still processing node instance n_1 and that node n_1 has a connection to node n_2. We simply add n_2 to our list of connected nodes, N.

If n_1 is not equal to I and C is False then we know that we've encountered a new node instance. We begin the process of processing all nodes in N, by creating and writing each node instance to an output file, O, in NEList format and then create edges between n_1 and each n_2 in N. Finally, we write the graph number and the focus level of the graph to O and revert C to True (as we begin a new node instance) and empty N. This node instance gets passed down to the first condition we described and we start a new node instance n_1.

4.5.2 Single line community format to NEList

The Single line community conversion is similar to the From-to node approach. For each line of the Single line community input file, we delimit the line into array list N by tabs. For each item in N, we create and write the node instance to the output file, O, in NEList format. We then create a nested for loop, where j iterates n times, where n is the size of N, and k iterates $j - 1$ times for each iteration of j. For each iteration of

Algorithm 4 From-to node format to NEList conversion

Given From-to node format file F and output NEList file O
Initalize C as True, current node id I as null and node array list N as empty
For each line of F
    Delimit line by tab into the From-node id, n_1 and To-node id n_2.
    If $n_1 \neq I$ AND C == False
        Create n_1 node instance and write to O
        For each item, i, in N
            Create node instance of i and write to O
        For each item, i, in N
            Create edge instance between i and n_1 and write to O
        Write the graph number and a focus level to O
        C = True
        Set N as empty
    If $n_1 \neq I$ AND C == True
        $I = n_1$
        C = False
    else if n_1 == I
        Add n_2 to N

k, we create an edge instance between the node instances stored in the j-th index of N and k-th index of N. This creates a complete graph where each node has a connection to each other.

Algorithm 5 Single line community format to NEList conversion

Given Single line community format file F and output NEList file O
Initalize node array list N as empty
For each line of F
    Delimit line by tabs into N
        For each item, i, in N
            Create node instance of i and write to O
        for(j = 0; j <size of N; j++)
            for(k=j+1; k <size of N; k++)
                Create edge instance between N[j] and N[k] and write to O
    Write the graph number and a focus level to O

4.6 Analysis on output frequent graphs

Corresponding to the third research gap as described in Section 4.2.3, we performed detailed analysis on output frequent graphs. Bifet et al. [13] only examined the number of closed frequent graphs. However, we want a deeper analysis to better understand the results produced. We noticed two possible types of drifts in the output: Real drift and internal change.

4.6.1 Real drift

Definition 11. ***Real Drift:*** *For the set of graph instances $G = \{g_1, g_2, ..., g_n\}$ where g_i represents the i-th instance for $1 \leq i \leq n$. We define real drift as the change of the number of closed frequent graphs, caused by changes in the graph instance distribution at instance g_i where g_0 to g_{i-1} contain closed frequent graphs from dataset, D_1 and g_i to g_n contains closed frequent graphs from dataset, D_2, given that $D_1 \neq D_2$.*

To demonstrate this, we present Figure 4.5 which has the following parameters: the minimum support threshold is set to 40%, the number of instances per batch is set to 10,000 and we use the joint set of two datasets G_1 and G_2. G_1 contains 250,000 instances and produces 43 closed frequent graphs per batch as output, G_2 has the same number of instances and produces 29 closed frequent graphs per batch. Hence, we can expect G_1 and G_2 to individually produce a linear pattern when we mapped into number of closed graph vs. graph instance line graphs as shown in Figure 4.5.

The joint set G_1G_2 however produces a different result to the individual datasets. We expect a stable number of graphs produced by the first 250,000 instances which is produced from the same distribution in dataset G_1. At the 260,000 graph instances, the data from a second dataset G_2 is introduced. However, since this is only the first batch that contains graphs from G_2, it is not enough to influence the closed frequent graph coreset output. At 270,000 graph instances, we see a sudden spike in the number of closed frequent graphs, this is due to the current coreset having the combination of closed frequent graphs from both datasets, G_1 and G_2.

However, at the 280,000 to 290,000 graph instances range, we see a dramatic decrease in the number of closed frequent graphs, caused by the closed frequent graphs from G_1 being removed as they do not appear in dataset G_2, hence losing its property of being frequent and its place in the coreset. From 300,000 graph instances and onwards, we see a fairly stable linear pattern which represents the closed frequent graphs that exist only in G_2.

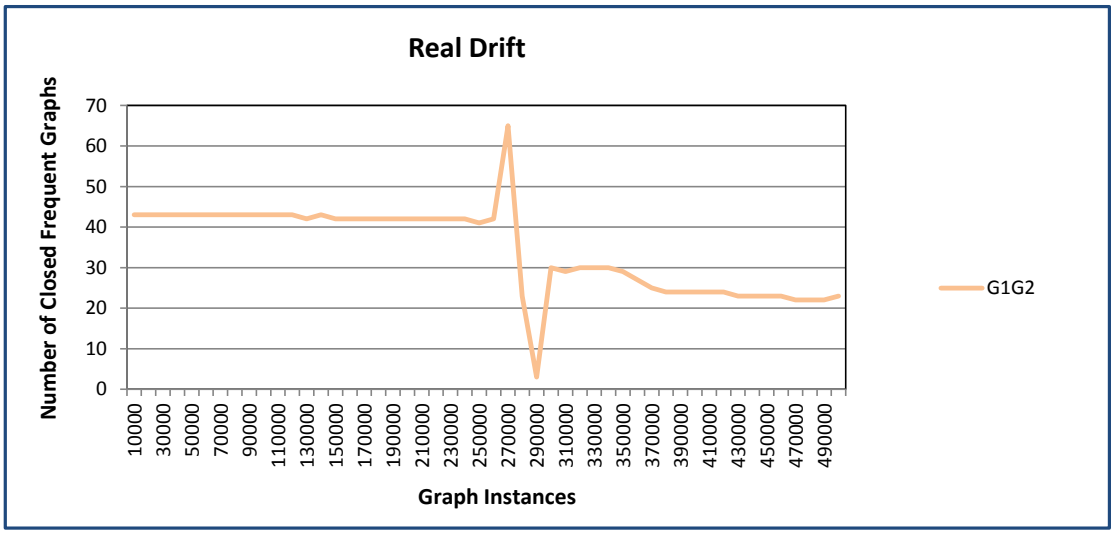


Figure 4.5: Relationship between number of closed graphs and graph instances for joint set G_1G_2 with minimum support of 40% and batch size of 10,000.

4.6.2 Internal change

Definition 12. ***Internal Change:*** *Let be a set of graph instances $G = \{g_1, g_2, ..., g_n\}$ where g_i represents the i-th instance for $1 \leq i \leq n$. We define internal change as the change to the number of output closed frequent graphs at g_i where the algorithm performs early termination of graphs in the coreset at instance g_{i-1} in a stationary graph data stream.*

The graph dataset G_3 from Figure 4.6 contains 240,000 graph instances and produces 43 closed frequent graphs per batch as output with minimum support set as 40% and the batch size being 10,000. The graph shown in Figure 4.6 presents the relationship between number of closed frequent graphs and graph instances for G_3.

From 1 to 220,000 graph instances, we see a consistent linear pattern at the 43 closed frequent graphs point. Figures 4.7 and 4.8 show the coreset of closed frequent graphs at 210,000 and 220,000 graph instance respectively. However, at 230,000 graph instances we see a loss of one closed frequent graph from our coreset (shown in Figure 4.9). Although it is difficult to deduce from the line graph perspective, this change is significant as the closed frequent graph, C, was removed from the output coreset. C was removed because its support value became equivalent to one of its proper supergraph's support. Hence, C is no longer a closed frequent graph at graph instance 230,000 which explains why it was not included in the coreset of closed frequent graphs. This phenomenon is similar to the equivalent occurrence technique as described in Section 2.2.1.2.

Inherited from the essence of the Apriori approach, all proper supergraphs must have a lower or equal support value to its subgraphs. Hence, the removal of C in the coreset affects its corresponding proper supergraphs and we see at the 240,000 graph instance point, an abrupt decrease in the number of closed frequent graphs caused by the removal of all proper supergraphs of C and only two graphs which do not have the subgraph C remain (shown in Figure 4.10).

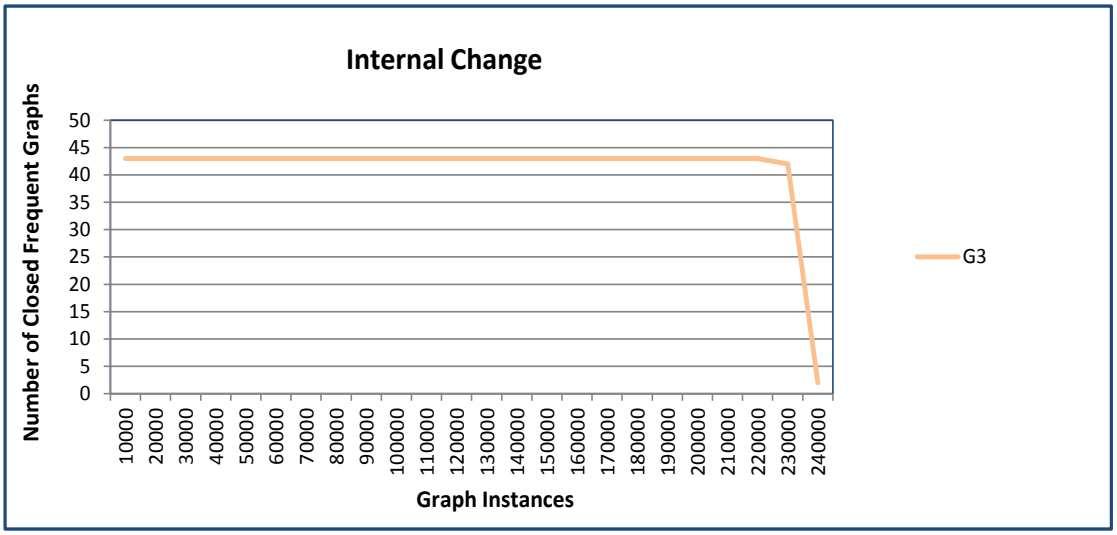


Figure 4.6: Relationship between number of closed graphs and graph instances for the graph set G_3 with minimum support of 40% and batch size of 10,000.

4.7 Conclusion

This chapter underlines the existing techniques that we have based our research and experiments on, the research gaps we have found and finally, our approach to address those research gaps that we've found, including the implementation of SeqDriftGraphMiner, an extension to Bifet's graph batch mining framework. We then introduce the concepts of real drift and internal change.

```
instances = 210000
id,description,nodes,edges,s_abs,s_rel,c_abs,c_rel
1,O(-C)-C,3,2,2294011,1092.3862,0,0.0
2,O-C-C,3,2,2295282,1092.9915,0,0.0
3,O-C,2,1,2330125,1109.5834,0,0.0
4,O=C-N,3,2,2277582,1084.5629,0,0.0
5,O=C-C-C,4,3,2283313,1087.2919,0,0.0
6,O=C-C,3,2,2341908,1115.1943,0,0.0
7,O=C,2,1,2347061,1117.6481,0,0.0
8,O,1,0,2394588,1140.28,0,0.0
9,N(-C)-C-C-C,5,4,2276454,1084.0258,0,0.0
10,N(-C)-C-C,4,3,2322223,1105.8204,0,0.0
11,N(-C)-C,3,2,2338235,1113.4452,0,0.0
12,N-C-C-C,4,3,2288697,1089.8557,0,0.0
13,N-C-C,3,2,2337865,1113.269,0,0.0
14,N-c(:c):c,4,3,2272493,1082.1395,0,0.0
15,N-c:c:c,4,3,2274047,1082.8795,0,0.0
16,N-c:c,3,2,2275296,1083.4742,0,0.0
17,N-C,2,1,2359099,1123.3805,0,0.0
18,N,1,0,2370374,1128.7495,0,0.0
19,C(-C)-C-C,4,3,2313027,1101.4414,0,0.0
20,C(-C)-c:c,4,3,2273041,1082.4005,0,0.0
21,C(-C)-C,3,2,2367024,1127.1543,0,0.0
22,c1(-C):c:c:c:c:c:1,7,7,2326451,1107.8339,0,0.0
23,c(-C)(:c:c):c:c:c,7,6,2326936,1108.0648,0,0.0
24,c(-C)(:c:c):c:c,6,5,2328138,1108.6371,0,0.0
25,c(-C)(:c):c:c:c:c,7,6,2328299,1108.7139,0,0.0
26,c(-C)(:c):c:c:c,6,5,2329894,1109.4734,0,0.0
27,c(-C)(:c):c:c,5,4,2330864,1109.9352,0,0.0
28,c(-C)(:c):c,4,3,2331788,1110.3752,0,0.0
29,c(-C):c:c:c:c:c,7,6,2328854,1108.9781,0,0.0
30,c(-C):c:c:c:c,6,5,2330887,1109.9462,0,0.0
31,c(-C):c:c:c,5,4,2331557,1110.2653,0,0.0
32,c(-C):c:c,4,3,2333059,1110.9805,0,0.0
33,c(-C):c,3,2,2336617,1112.6748,0,0.0
34,C(-C)=C,3,2,2291932,1091.3962,0,0.0
35,C-C,2,1,2405286,1145.3743,0,0.0
36,c1:c:c:c:c:c:1,6,6,2356465,1122.1262,0,0.0
37,c(:c:c):c:c:c,6,5,2356673,1122.2252,0,0.0
38,c(:c:c):c:c,5,4,2360508,1124.0514,0,0.0
39,c(:c):c:c,4,3,2360531,1124.0624,0,0.0
40,c(:c):c,3,2,2363974,1125.7019,0,0.0
41,c:c,2,1,2364344,1125.878,0,0.0
42,C=C,2,1,2294335,1092.5405,0,0.0
43,C,1,0,2409838,1147.5419,0,0.0
```

Figure 4.7: The coreset of closed frequent graphs for the graph set G_3 at graph instance 210,000.

```
instances=220000
id,description,nodes,edges,s_abs,s_rel,c_abs,c_rel
1,O(-C)-C,3,2,2514018,1142.7355,0,0.0
2,O-C-C,3,2,2515343,1143.3378,0,0.0
3,O-C,2,1,2551686,1159.8573,0,0.0
4,O=C-N,3,2,2496885,1134.9478,0,0.0
5,O=C-C-C,4,3,2502859,1137.6632,0,0.0
6,O=C-C,3,2,2563977,1165.4441,0,0.0
7,O=C,2,1,2569351,1167.8868,0,0.0
8,O,1,0,2618925,1190.4204,0,0.0
9,N(-C)-C-C-C,5,4,2495722,1134.4191,0,0.0
10,N(-C)-C-C,4,3,2543444,1156.111,0,0.0
11,N(-C)-C,3,2,2560145,1163.7023,0,0.0
12,N-C-C-C,4,3,2508475,1140.216,0,0.0
13,N-C-C,3,2,2559759,1163.5269,0,0.0
14,N-c(:c):c,4,3,2491633,1132.5604,0,0.0
15,N-c:c:c,4,3,2493237,1133.2896,0,0.0
16,N-c:c,3,2,2494527,1133.8759,0,0.0
17,N-C,2,1,2581907,1173.5941,0,0.0
18,N,1,0,2593668,1178.94,0,0.0
19,C(-C)-C-C,4,3,2533852,1151.7509,0,0.0
20,C(-C)-c:c,4,3,2492199,1132.8177,0,0.0
21,C(-C)-C,3,2,2590174,1177.3518,0,0.0
22,c1(-C):c:c:c:c:c:1,7,7,2547854,1158.1155,0,0.0
23,c(-C)(:c:c):c:c:c,7,6,2548360,1158.3455,0,0.0
24,c(-C)(:c:c):c:c,6,5,2549613,1158.915,0,0.0
25,c(-C)(:c):c:c:c:c,7,6,2549782,1158.9918,0,0.0
26,c(-C)(:c):c:c:c,6,5,2551445,1159.7477,0,0.0
27,c(-C)(:c):c:c,5,4,2552457,1160.2078,0,0.0
28,c(-C)(:c):c,4,3,2553421,1160.6459,0,0.0
29,c(-C):c:c:c:c:c,7,6,2550360,1159.2545,0,0.0
30,c(-C):c:c:c:c,6,5,2552481,1160.2186,0,0.0
31,c(-C):c:c:c,5,4,2553180,1160.5364,0,0.0
32,c(-C):c:c,4,3,2554747,1161.2487,0,0.0
33,c(-C):c,3,2,2558458,1162.9354,0,0.0
34,C(-C)=C,3,2,2511849,1141.7495,0,0.0
35,C-C,2,1,2630083,1195.4923,0,0.0
36,c1:c:c:c:c:c:1,6,6,2579160,1172.3455,0,0.0
37,c(:c:c):c:c:c,6,5,2579377,1172.4441,0,0.0
38,c(:c:c):c:c,5,4,2583377,1174.2623,0,0.0
39,c(:c):c:c,4,3,2583401,1174.2732,0,0.0
40,c(:c):c:c,3,2,2586992,1175.9054,0,0.0
41,c:c,2,1,2587378,1176.0809,0,0.0
42,C=C,2,1,2514355,1142.8887,0,0.0
43,C,1,0,2634831,1197.6505,0,0.0
```

Figure 4.8: The coreset of closed frequent graphs for the graph set G_3 at graph instance 220,000.

```
instances = 230000
id,description,nodes,edges,s_abs,s_rel,c_abs,c_rel
1,O(-C)-C,3,2,2744021,1193.0526,0,0.0
2,O-C-C,3,2,2745402,1193.6531,0,0.0
3,O-C,2,1,2783245,1210.1066,0,0.0
4,O=C-N,3,2,2726186,1185.2982,0,0.0
5,O=C-C-C,4,3,2732402,1188.0009,0,0.0
6,O=C-C,3,2,2796044,1215.6713,0,0.0
7,O=C,2,1,2801640,1218.1044,0,0.0
8,O,1,0,2853261,1240.5482,0,0.0
9,N(-C)-C-C-C,5,4,2724986,1184.7765,0,0.0
10,N(-C)-C-C,4,3,2774663,1206.3752,0,0.0
11,N(-C)-C,3,2,2792054,1213.9365,0,0.0
12,N-C-C-C,4,3,2738250,1190.5435,0,0.0
13,N-C-C,3,2,2791652,1213.7617,0,0.0
14,N-c(:c):c,4,3,2720770,1182.9435,0,0.0
15,N-c:c:c,4,3,2722424,1183.6626,0,0.0
16,N-c:c,3,2,2723754,1184.2408,0,0.0
17,N-C,2,1,2814715,1223.7892,0,0.0
18,N,1,0,2826961,1229.1135,0,0.0
19,C(-C)-C-C,4,3,2764675,1202.0326,0,0.0
20,C(-C)-c:c,4,3,2721354,1183.1974,0,0.0
21,C(-C)-C,3,2,2823322,1227.5312,0,0.0
22,c1(-C):c:c:c:c:c:1,7,7,2779255,1208.3717,0,0.0
23,c(-C)(:c:c):c:c:c,7,6,2779782,1208.6008,0,0.0
24,c(-C)(:c:c):c:c,6,5,2781087,1209.1682,0,0.0
25,c(-C)(:c):c:c:c:c,7,6,2781263,1209.2448,0,0.0
26,c(-C)(:c):c:c:c,6,5,2782994,1209.9974,0,0.0
27,c(-C)(:c):c:c,5,4,2784048,1210.4557,0,0.0
28,c(-C)(:c):c,4,3,2785052,1210.8922,0,0.0
29,c(-C):c:c:c:c:c,7,6,2781865,1209.5065,0,0.0
30,c(-C):c:c:c:c,6,5,2784073,1210.4666,0,0.0
31,c(-C):c:c:c,5,4,2784801,1210.7831,0,0.0
32,c(-C):c:c,4,3,2786432,1211.4922,0,0.0
33,c(-C):c,3,2,2790297,1213.1726,0,0.0
34,C(-C)=C,3,2,2741763,1192.0709,0,0.0
35,C-C,2,1,2994822,1302.0966,0,0.0
36,c1:c:c:c:c:c:1,6,6,2811854,1222.5452,0,0.0
37,c(:c:c):c:c:c,6,5,2812080,1222.6434,0,0.0
38,c(:c:c):c:c,5,4,2816246,1224.4548,0,0.0
39,c(:c):c:c,4,3,2816271,1224.4657,0,0.0
40,c(:c):c,3,2,2820010,1226.0913,0,0.0
41,c:c,2,1,2820411,1226.2656,0,0.0
42,C=C,2,1,2744373,1193.2057,0,0.0
```

Figure 4.9: The coreset of closed frequent graphs for the graph set G_3 at graph instance 230,000.

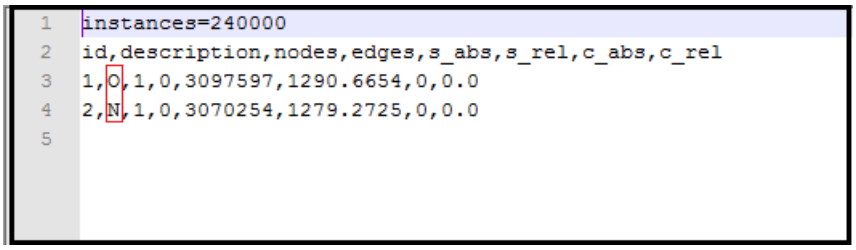


Figure 4.10: The coreset of closed frequent graphs for the graph set G_3 at graph instance 240,000.

Chapter 5

EXPERIMENTAL EVALUATION

In this chapter, we describe the results we obtained when performing experimental evaluation on chemical molecular dense datasets in SMILES format and social network sparse datasets in NEList format. We first define a concatenated dataset as a dataset that is formed when we join two or more original datasets with a interleaving mechanism which we will describe in detail later in Section 5.2.3.1. The concatenated datasets that imitate data streams have varying severities of artificial drifts created by three levels of slope values (small, medium and large). A slope value is a user-defined parameter to control gradients of changes in a produced data stream. This chapter presents a performance comparison between our technique, SeqDriftGraphMiner, with the benchmark technique, AdaGraphMiner-ADAADWIN.

The three key evaluation metrics we consider when evaluating the two frequent graph mining techniques are the running time, memory usage and the delay time, we perform evaluation on all concatenated datasets with the two techniques while focusing on these three factors. The time parameter presents the time taken to process our concatenated dataset, memory indicates the amount of RAM used for processing at each point of our graph instances and the delay time is the difference between the time when changes in a data stream occur and the time when the two techniques detect these changes. Please note that for simplicity, we refer to AdaGraphMiner-ADAADWIN as AdaGraphMiner in this chapter.
The chapter will be structured as follows: Section 5.1 describes the experimental setup and parameter settings for all of our experiments. Section 5.2 presents the dataset setup for both chemical molecular dense datasets and social network sparse datasets. Our dataset setup section includes detailed descriptions of each real-world or synthetic dataset used, mechanism for creating varying severity of artificial drifts within different concatenations of our datasets, positions of different drift points in our concatenated datasets. Finally, Sections 5.3 and 5.4 detail the experimental results we obtained when processing chemical molecular dense datasets and social network sparse datasets respectively. The results produce three evaluation metrics: time, memory and delay time which are presented in three separate corresponding subsections.

5.1 Experimental setup

This section describes machine and facility used in our experiments and parameter settings for all of our experiments.

5.1.1 General experiment setup

All programs were implemented in Java Standard Edition. Due to the large size of input and output files and long computation times, it would take too long for us to process our imitation data streams with the two techniques using conventional methods. Hence, we utilise the clustering services NZeI (New Zealand eScience Infrastructure) set up by the University of Auckland for our experiments and analysis. NZeI provides high speed computing services for registered researchers and by using the provided processing power and storage space, we reduce the limitations of processing time and file size for our experiments.

The cluster is a network of computers, each with several processors which contains several cores. Since each computation in our experiment requires a core to process, by having several dozen cores we speed up the total running time of our experiments considerably.

5.1.2 Parameter settings

Table 5.1 presents a summary of the parameter settings used in our experiments. When performing closed frequent graph mining on concatenated dense datasets we used a minimum support threshold of 40%, batch size of 10,000 and delta value of 0 (for details on delta values, please refer to Section 2.3.1.3). While for our three concatenated sparse datasets, which we will describe in detail separately in the next section, we used the following parameter settings:

1. *Concatenated Sparse dataset 1:* Our experiments perform closed frequent graph mining on the first sparse dataset using a minimum support of 1%, batch size of 1000 and delta value of 0. Please note that the batch size used for this dataset is 1000 which is ten times smaller than batch sizes used for all other concatenated datasets in our experiments. Since the number of total graph instances of this dataset is much smaller than other datasets, we normalize the difference by using smaller batch size.

2. *Concatenated Sparse dataset 2:* The second sparse dataset uses a minimum support of 0.15%, batch size of 10,000 and delta value of 0 when we use it to perform closed frequent graph mining.

3. *Concatenated Sparse dataset 3:* The experiments perform closed frequent graph mining on the third sparse dataset using a minimum support of 0.1%, batch size of 10,000 and delta value of 0.

Table 5.1: Parameter settings for concatenated datasets used.

Dataset	Minimum Support	Batch Size	Delta Value
Dense Dataset	40%	10,000	0
Sparse Dataset 1	1%	1000	0
Sparse Dataset 2	0.15%	10,000	0
Sparse Dataset 3	0.10%	10,000	0

5.2 Dataset setup

In this section, we describe how we setup the datasets used in our experiment. Section 5.2.1 provides detailed information on the four real-world chemical molecular dense datasets we used while Section 5.2.2 describes the real-world and synthetic social network sparse datasets. The first subsection of Section 5.2.3 introduces our interleaving mechanism for creating varying severities of artificial drifts in the concatenated datasets using three levels of slope values, the next two subsections describe drift points in concatenated chemical molecular dense datasets and social network sparse datasets respectively.

5.2.1 Descriptions of the four real-world chemical molecular dense datasets used

The four dense datasets in our experimental evaluation are all real world chemical molecular datasets in the SMILES format. As described in Section 3.1, SMILES uses a set of strict rules and notations to represent chemical compounds in a string sequence. The four dense datasets are independent of each other, ensuring that they are not subsets of one another, hence they all have unique graph instances. Additionally, each dataset listed below will be replicated to obtain a total of 250,000 graph instances. The four modified real world datasets are available to download at the following link: `https://www.cs.auckland.ac.nz/research/groups/kmg/MonicaBian.html`

1. *Pool A:* Pool A is a dataset retrieved from the open National Cancer Institute (NCI) database. It is a SMILES dataset built from organic samples submitted to the institute for testing. It is "open" as it is retrieved from the publicly accessible domain of the database. The number of total instances is 250,000 and the number of closed frequent graphs generated per batch is 43.

2. *Pool B:* Pool B is a public ChemDB dataset consisting of a collection of molecules retrieved from varying public resources and is translated into SMILES format for processing. The number of total instances is 250,000 and the number of closed frequent graphs generated per batch is 29.
3. *Pool C:* Pool C is a molecular dataset retrieved from a chemical informatics database. It is based on melting points of tested molecules and drug compounds. Number of total instances is 250,000 and the number of closed frequent graphs generated per batch is 35.
4. *Pool D:* Pool D is retrieved from the same database as Pool C. The dataset is based on testing the aequeous solubility of chemical compounds. The number of total instances is 250,000 and the number of closed frequent graphs generated per batch is 9.

5.2.2 Descriptions of the social network sparse datasets used

In this section, we present descriptions of the real-world and synthetic social network datasets used for our experimental evaluation. We will provide details on from what applications is the dataset retrieved from, what are the expected features of the dataset and where the dataset originates from.

5.2.2.1 Descriptions of the three real-world social network datasets

The three sparse datasets in our experimental evaluation are all real-world social network datasets obtained from the SNAP website [32] and as described in Section 3.2, we represent these kinds of graphs in the NEList format. The datasets are derived from large social networks, namely Youtube, Amazon and Friendster, each having a large user-base that interacts with each other. Through the use of friend-lists (list of people a user has tagged as a "friend") or user-defined groups (groups of users that are connected due to a common purpose), we are able to find groups of interconnected users. The three real-world social network datasets represented in the NEList format are available to download at the following link: `https://www.cs.auckland.ac.nz/research/groups/kmg/MonicaBian.html`.

Youtube dataset: Youtube is a large multi-media sharing website with integrated social networking features. Users can create customized accounts with preferences, link with friends and form user-defined groups which can be publicly available for other users to join. The total number of graph instances in the dataset is 15,000 and the data is provided by Mislove et al. [35]. Original text representation of the Youtube dataset is in the Single line community input format (Section 3.2).

Amazon dataset: This dataset was produced by crawling the Amazon website. The data is based on the feature "Customers Who Bought This Item Also Bought" which indicates products that are purchased together, much like the concept of association rule discussed in Section 2.1.3. If a product i is frequently co-purchased with product j, then we create an undirected edge from i to j in the graph. The total number of graph instances in this dataset is 100,000. Original text representation of the Amazon dataset is in the From-to node input format (Section 3.2).

Friendster dataset: Before being reformed as a online gaming website, Friendster was a social networking site where users can "friend" other users, thus forming friendship edges in the network with each other. Like Youtube, Friendster users can create social groups which other members can join, forming an inter-connected network of users. The total number of graph instances in the dataset is 250,000 and the data is provided by and available at the Web Archive Project [36]. Original text representation of the Friendster dataset is in the Single line community input format (Section 3.2).

5.2.2.2 Descriptions of the three synthetic social network datasets

For the convenience of visually identifying change points from the line graphs we create, we generate synthetic datasets from the three real-world social network datasets. A synthetic dataset, S, is a replication of the original real-world social network dataset, A, with the exception that each original vertex, v, is replaced by another synthetic vertex, v'. v' is produced by concatenating a synthetic value, r, with v. Algorithm 6 demonstrates how a synthetic dataset is generated from the original dataset. Each original vertex, v, in A is concatenated by r to generate synthetic vertex v' which is written to the output synthetic dataset S. This creates two versions of datasets with completely different graph instances but share the same distribution patterns. Hence, the synthetic dataset mirrors the characteristic of the original social network dataset. Since our underlying mining engine is vertex centralized, we are able to identify the delay time simply by looking at the difference between actual and expected beginning positions of a recurring pattern. In the next section we will discuss in detail how we use these two versions of the datasets. The three NEList format synthetic datasets are generated separately from the original Youtube, Amazon and Friendster datasets. These datasets are available to download at the following link: `https://www.cs.auckland.ac.nz/research/groups/kmg/MonicaBian.html`

Algorithm 6 Synthetic Algorithm

1: *Given dataset, A, and output synthethic dataset, S*
2: *Initialize synthetic value, r*
3: *For each vertex, v, in A*
4: *Create synthetic vertex, v', by concatenating r with v*
5: *Write v' to S*

5.2.3 Creating artificial drifts within real-world and synthetic datasets

This section details how we create artificial drift points with our slope values. We provide detailed examples on how artificial drifts are formed to join datasets and how we applied it to our chemical molecular dense and social network sparse datasets.

5.2.3.1 Creating artificial drifts using three levels of slope values

To simulate a non-stationary data stream, we are required to concatenate our real-world or synthetic datasets together with varying severities of artificial drifts. In this section, we describe the interleaving mechanism used for creating artificial drifts in our concatenated datasets. Although this concept can also be applied to social network sparse datasets, we use the chemical molecular dense datasets Pool A and Pool B for our example.

The pseudocode shown in Algorithm 7 presents how an artificial drift in the concatenated datasets was generated. Given the chemical molecular dense datasets Pool A, A, and Pool B, B, a slope value s (as explained in the next paragraph) and the drift point position value p where $p < \mid A \mid$, meaning p occurs within Pool A, we produce the concatenated dataset Pool AB, AB, with an artificial drift. From line 4 of Algorithm 7, we add all graphs from A to AB up until the drift point position, p, which we begin to introduce graphs from B and the data begins to overlap between ($\mid A \mid$ - p + 1) of A and all of B. Starting from p, for the remaining number ($\mid A \mid$ - p + 1) of graphs in A and all graphs in B, we create the random number, r, which is a value between 0 and 1 and compare it to the threshold value t (which is set as 1 initially). If $r > t$ then we add a graph from the remaining number of graphs in A to AB, otherwise, we add a graph from B to AB, the slope value s is then decremented from t.

The varying severity of artificial drifts in our concatenated datasets were generated using different slope values, categorized as either small (1.0×10^{-6}), medium (2.0×10^{-6}) or large (3.0×10^{-6}) in our experiments. Figure 5.1 presents an example demonstrating the abrupt change (large slope value used) and gradual change (small slope value used) in (a) and (b) respectively.

The highlighted blue areas in Figure 5.1 (a) and Figure 5.1 (b) correspond to abrupt and gradual artificial drifts respectively indicating small or large gradients of change when transitioning from one dataset to another. Evidently, the major difference between the two types of drift is the size of the overlapping area. Since Figure 5.1 (a) has a considerably smaller blue area, the overlapping segment which contains a combination of graphs from A and B is also smaller, meaning that there is a quicker transit from A and B which is considered as a more abrupt change. Whereas Figure 5.1 (b) presents a larger overlapping area and therefore a more gradual transition from A to B. The difference between the two overlapping segments is caused by the different slope values, s, used. Since the threshold, t, is decremented by s for each graph instance we see after the drift point p, a large s would likely to cause our algorithm to exhaust all graphs from A faster than a small s as the faster decreasing t would be more likely to be smaller than r. In a word, the sizes of the slope values influence the gradients of change in our concatenated datasets.

Algorithm 7 Algorithm for Generating Artificial Drift

Given Pools A and B, slope value s and Drift Point p.
Initialize empty concatenated dataset Pool AB
Initialize threshold t as 1
for (p - 1) amount of times:
    Add graph from A to AB
for the remaining (| A + B | −p + 1) number of graph instances:
    Initialize r as a random number between 0 and 1
    if r > t and A is not empty
        Add graph from A to AB
    else:
        Add graph from B to AB
    t = t - s
Return AB

5.2.3.2 Artificial drift points in chemical molecular dense datasets

For our experiments on the chemical molecular dense datasets, we concatenated Pool A, A, Pool B, B, Pool C, C and Pool D, D, to get the complete imitated data stream. In previous section, we explained the interleaving mechanism for generating the first artificial drift which joins A and B. The other two drifts happen when we join AB and C forming Pool ABC, ABC, and when joining ABC with D, giving us Pool ABCD,

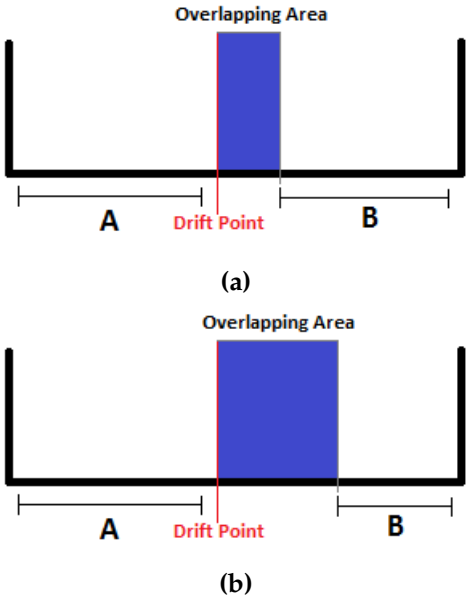


Figure 5.1: Abrupt (a) and Gradual (b) changes caused by different slope values.

$ABCD$, which both follow the same interleaving mechanism described previously. $ABCD$ is then used as the complete imitated data stream in our experiments. The total number of graph instances in $ABCD$ is 1,000,000. The first drift point of $ABCD$ starts at 245,000 graph instance, second drift point of $ABCD$ starts at 495,000 graph instance and the third drift point of $ABCD$ starts at 745,000 graph instance.

5.2.3.3 Artificial drift points in social network sparse datasets

Following the described interleaving mechanism used for chemical molecular dense datasets, we also use three levels of slope values to generate varying severities of artificial drifts in social network sparse datasets. A minor difference is that we only use two artificial drifts to concatenate three sparse datasets. The following are details of the three concatenated social network sparse datasets used for our experiments:

1. Concatenated sparse dataset1 - *YoutubeSyntheticYoutube:* Youtube + Synthetic + Youtube. The total number of graph instances is 45,000. First drift point starts at 14500 graph instance. Second drift point starts at 29500 graph instance.

2. Concatenated sparse dataset2 - *AmazonSyntheticAmazon:* Amazon + Synthetic + Amazon. The total number of graph instances is 300,000. First drift point starts at 95,000 graph instance. Second drift point starts at 195,000 graph instance.

3. Concatenated sparse dataset3 - *FriendsterSyntheticFriendster:* Friendster + Synthetic + Friendster. The total number of graph instances is 750,000. First drift

point starts at 245,000 graph instance. Second drift point starts at 495,000 graph instance.

Each of the concatenated sparse datasets described above are simply the original dataset joint with its synthetic dataset before joining once again with the original dataset. The artificial drift between two datasets are, like dense datasets, created by overlapping items from both datasets with a user-defined slope value, which in our experiments, we use small, medium and large values for generating gradual, moderate and abrupt changes respectively in our imitated data streams.

5.3 Experiments using chemical molecular datasets

As described in previous section, in order to obtain the concatenated set of our dense datasets, $ABCD$, which imitates non-stationary data stream, we use artificial drifts to link the four datasets. We generate ten datasets for each category of slope values (small, medium or large). The same configuration (including same slope value and datasets) is used to generate the ten datasets so they share similar characteristics, meaning they have the same severity of artificial changes. However, due to the random element included in the interleaving mechanism, the ten datasets have different artificial drifts, visually represented as the overlapping area in Figure 5.1. We then run experiments on the ten differently concatenated datasets to produce ten sets of output files. We minimize the randomness factor of generating concatenated dataset using the specific configuration by running multiple experiments on the ten datasets and getting average results from all the experiments performed. Afterwards, by batch processing the ten sets of output files generated for a specific slope value category, we obtain the average time, memory and delay time values which are then used to compare SeqDriftGraphMiner and AdaGraphMiner.

5.3.1 Time

The line graphs shown in Figures 5.2 to 5.4 represent a comparison of average time (in seconds) for AdaGraphMiner and SeqDriftGraphMiner. SeqDriftGraphMiner (represented by the green line), is shown to have consistently outperformed AdaGraphMiner in each graph batch. For small and medium slope values, we see noticeable small bumps in running time for AdaGraphMiner at each artificial drift point while SeqDriftGraphMiner remains fairly linear.

As described in Chapter 2 and by Sakthithasan et al. [23], the two algorithms use different estimation strategies when assessing data segments and determining cutpoints. Hence, we see an increasingly large difference in average running time and at the one

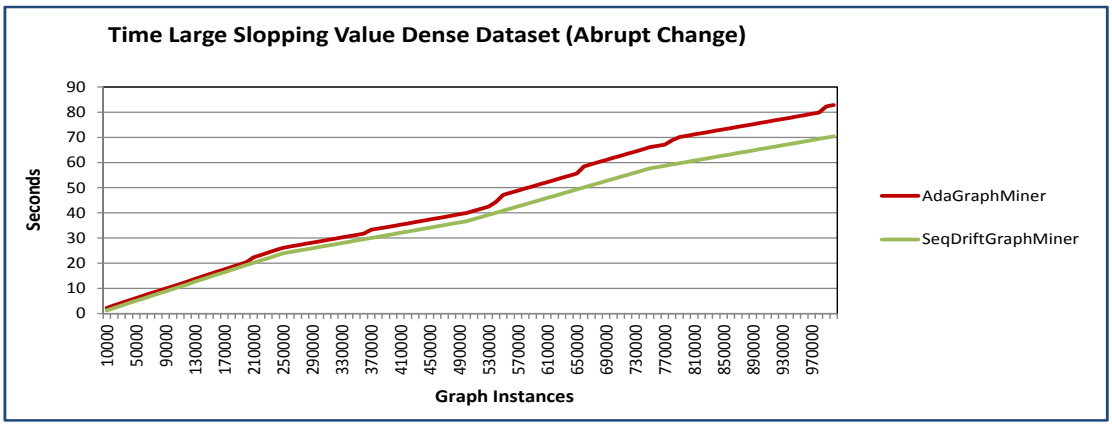


Figure 5.2: Time used for Dense Dataset with Abrupt Change.

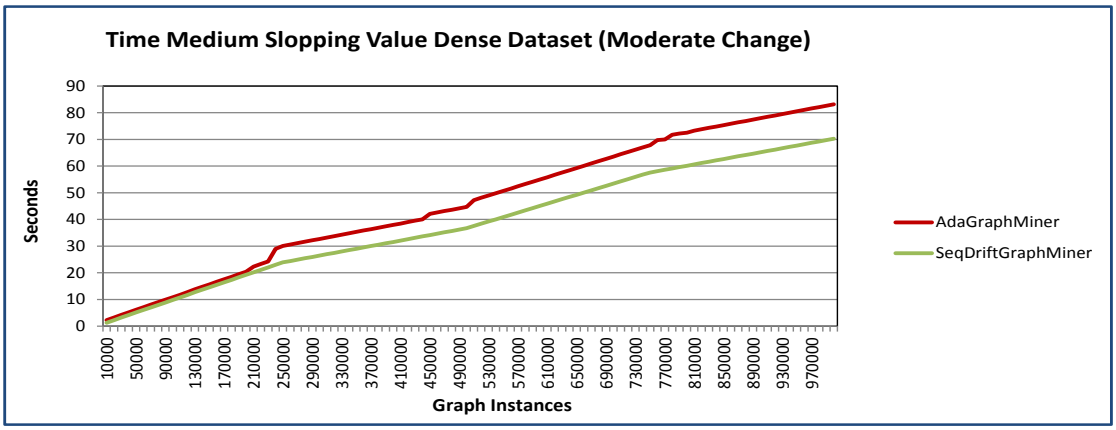


Figure 5.3: Time used for Dense Dataset with Moderate Change.

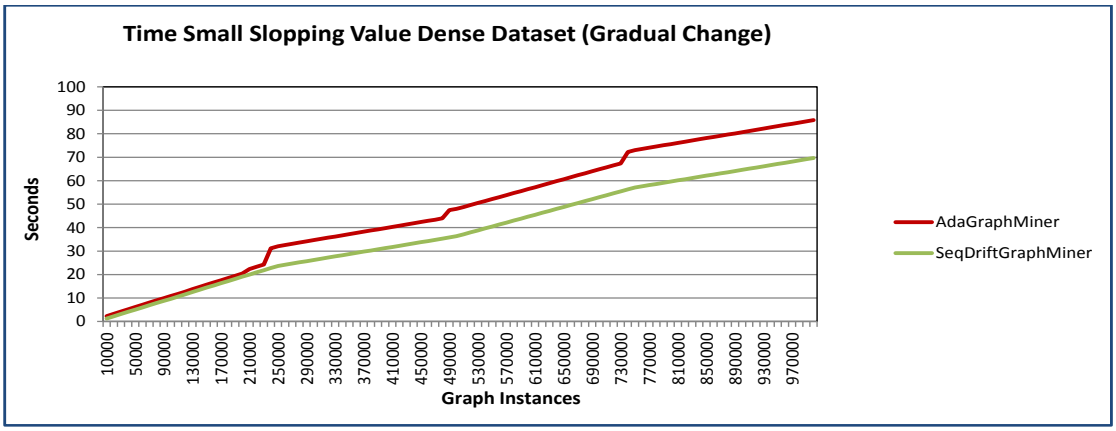


Figure 5.4: Time used for Dense Dataset with Gradual Change.

millionth molecular graph instance point, we see that SeqDriftGraphMiner being approximately 15% more efficient with a 15 seconds faster running time. We suspect that this difference will increase as more graph instances are introduced, which we will prove when evaluating sparse datasets in the Section 5.4.

5.3.2 Memory

The comparison in performance of SEQDRIFT and ADWIN by Sakthithasan et al. [23] showed that SEQDRIFT has lower computational overheads, however it did not explicitly state that SEQDRIFT requires comparatively less memory. Hence, we are interested in evaluating which algorithm requires less RAM to process. Figures 5.5 to 5.7 show SeqDriftGraphMiner outperforming AdaGraphMiner for all slope values in terms of RAM used. Notably from Figure 5.7, we see a significant difference in memory usage, close to 500MB, with SeqDriftGraphMiner being almost 20% more memory efficient than AdaGraphMiner.

5.3.3 Delay time

When comparing the delay time between AdaGraphMiner and SeqDriftGraphMiner, we are interested in analysing real drift but not internal change (Section 4.6). Figures 5.8 to 5.10 present line diagrams showing the number of closed frequent graphs in respect to the graph instance numbers. By finding how the number of closed frequent graphs change for each artificial drift point, we can determine how well each algorithm responds to change in a data stream.

1. *First Drift Point:* For all three slope values, the first drift point occurs at graph instance 245,000 and we notice a spike in the number of closed frequent graphs at the 270,000 graph instance point. At the 280,000 to 290,000 graph instance range, we see a sharp decrease, implying that change has been detected. This pattern is apparent for both algorithms, meaning they detect and respond to change at this drift point at the same time.

2. *Second Drift Point:* The second drift point occurs at 495,000 graph instances, we see both algorithms detecting artificial drift at 530,000 graph instances indicated by the increased number of closed frequent graphs. Hence, we see a similar delay time for both algorithms again.

3. *Third Drift Point:* The final drift point occurs at 745,000 graph instances and we see several interesting divergences between the two algorithms. When using medium slope value (as shown in Figure 5.9), SeqDriftGraphMiner was shown to have a higher delay time, detecting the artificial drift at graph instance 900,000 which is

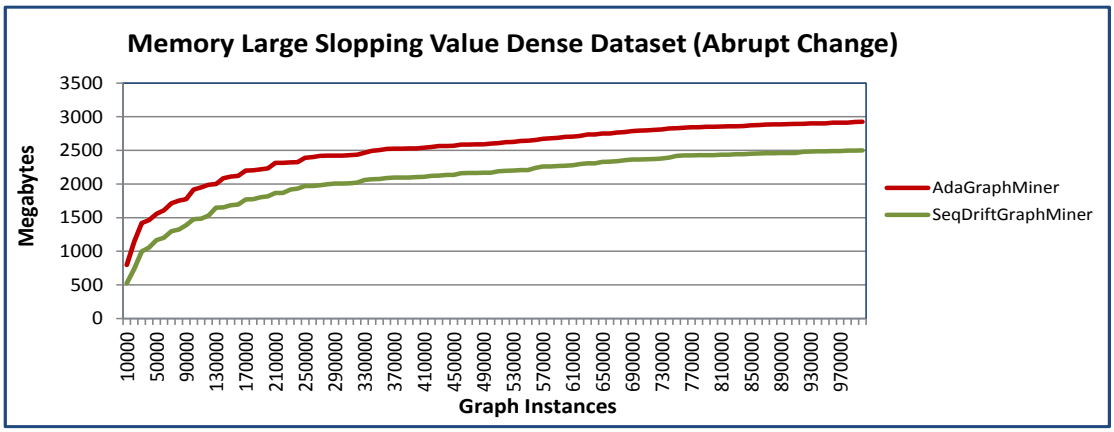


Figure 5.5: Memory used for Dense Dataset with Abrupt Change.

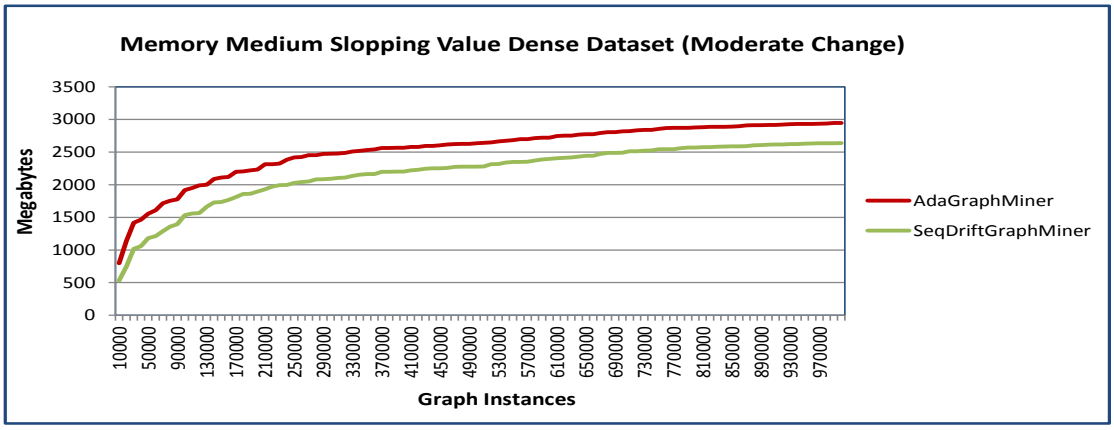


Figure 5.6: Memory used for Dense Dataset with Moderate Change.

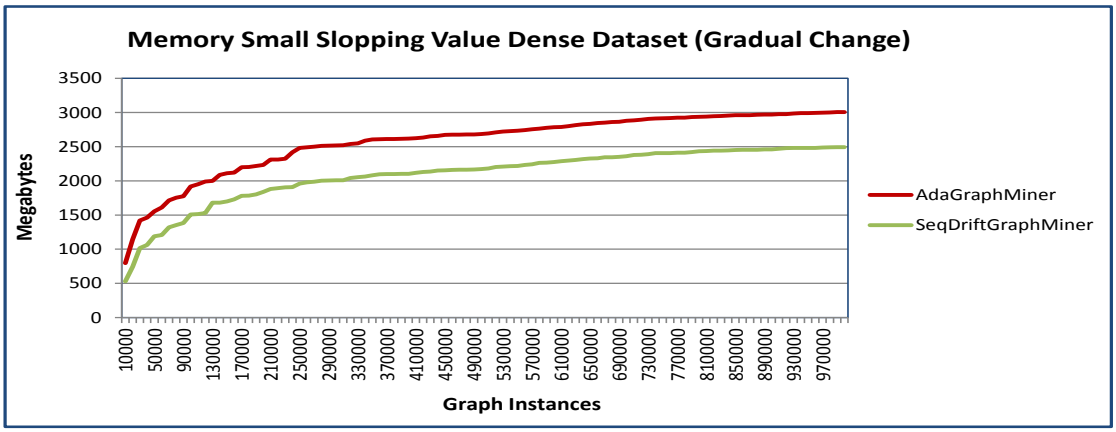


Figure 5.7: Memory used for Dense Dataset with Gradual Change.

five batches later than AdaGraphMiner. Additionally, for small slope value (Figure 5.10), we see AdaGraphMiner detecting change at 860,000 graph instances, while SeqDriftGraphMiner was unable to for the rest of the graph stream. However, it may be possible for SeqDriftGraphMiner to find this change if the concatenated dense dataset is extended beyond 1,000,000 graph instances.

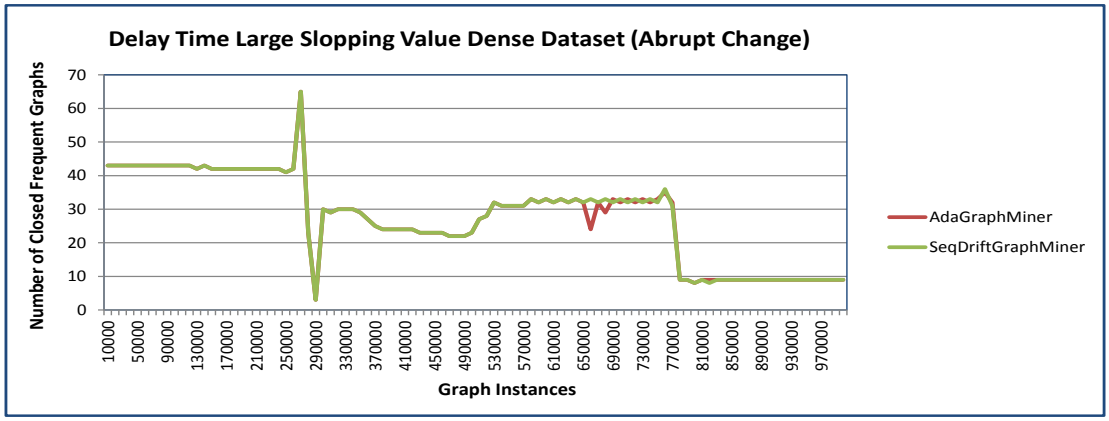


Figure 5.8: Delay Time for Dense Dataset with Abrupt Change.

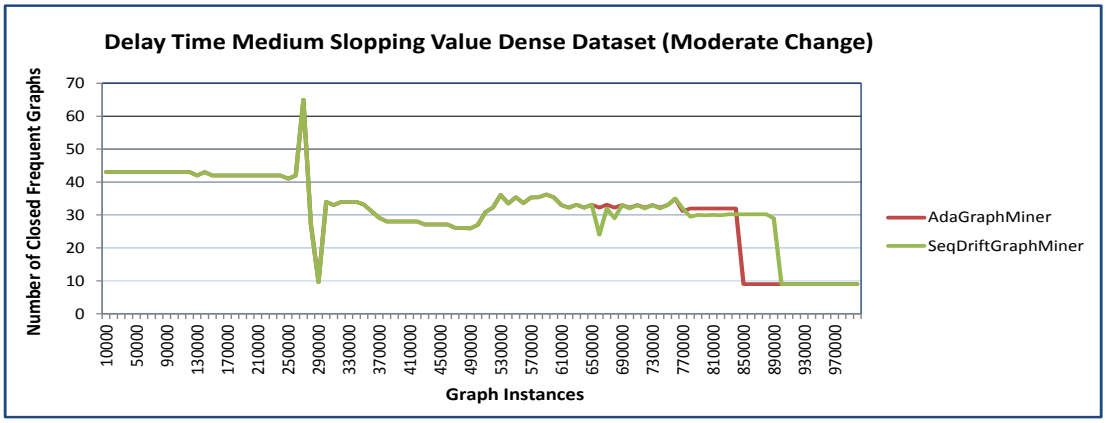


Figure 5.9: Delay Time for Dense Dataset with Moderate Change.

From the observations described above, we can conclude that SeqDriftGraphMiner has a higher delay time when compared to AdaGraphMiner, especially when the changes in the graph data streams are gradual. However, as stated by Sakthithasan et al. [23], SeqDriftGraphMiner's detection delay can be reduced if warning level and block size are adjusted appropriately. We would like to explore this further as our future work.

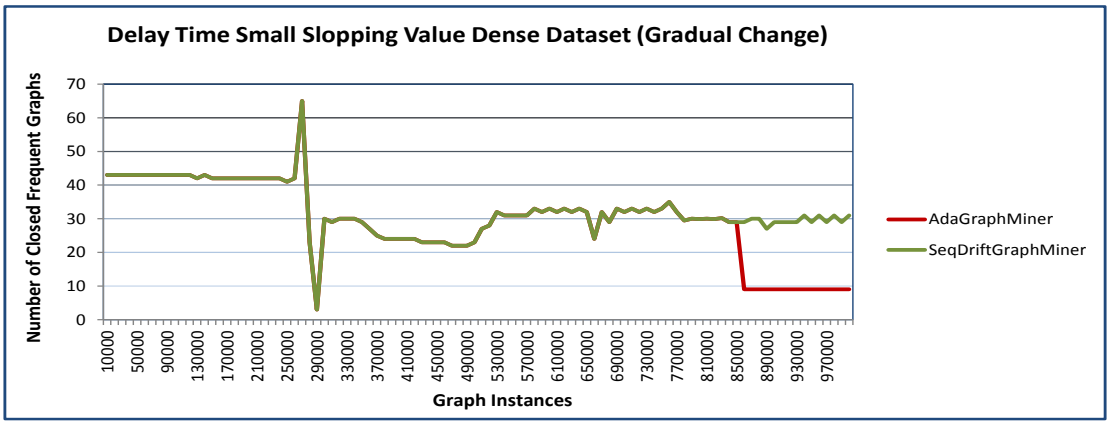


Figure 5.10: Delay Time for Dense Dataset with Gradual Change.

5.4 Experiments using social network datasets

Similar to experiments performed for the dense datasets, to obtain the concatenated set of our sparse datasets, original-synthetic-original, which imitates non-stationary data stream, we use artificial drifts to link the three datasets. We generate ten datasets for each category of slope values (small, medium or large). The same configuration (including same slope value and datasets) is used to generate the ten datasets so they share similar characteristics, meaning they have the same severity of artificial changes. However, due to the random element of the interleaving mechanism, the ten datasets have different artificial drifts, visually represented as the overlapping area in Figure 5.1. We then run experiments on the ten differently concatenated datasets to produce ten sets of output files. We minimize the randomness factor of generating concatenated dataset using the specific configuration by running multiple experiments on the ten datasets and getting average results from all the experiments performed. Afterwards, by batch processing the ten sets of output files generated for a specific slope value category, we obtain the average time, memory and delay time values which are then used to compare SeqDriftGraphMiner and AdaGraphMiner.

5.4.1 Recurring patterns for concatenated social network datasets (slope value = 0)

Figures 5.11 to 5.13 shows the patterns we produce when we concatenate original datasets with synthetic dataset using a slope value of 0. In this example, we are simply concatenating datasets together and we see the starting pattern immediately reoccurring at each artificial drift point. This shows that we are not creating artificial links between each dataset and these figures present the optimal results as we are not interleaving datasets at each drift point. Hence, we use the three figures we have presented

in this section as baseline to determine the delay times of SeqDriftGrapphMiner and AdaGraphMiner. The baseline provides the optimal delay times for our sparse datasets, which are just optimal results that are unlikely to be produced from our experiments because no matter how abrupt an artificial drift in our concatenated dataset is, it is still not quite the same as direct join of two datasets. However, the baseline can be used as a comparison metric when discussing delay times of both algorithms for all the three sparse datasets.

5.4.2 YoutubeSyntheticYoutube

In this section, we present the results obtained when evaluating running time, memory usage consumption and delay time for the YoutubeSyntheticYoutube dataset.

5.4.2.1 Time

Figures 5.14 to 5.16 presents a performance comparison in running time between AdaGraphMiner and SeqDriftGraphMiner. Both algorithms show an linear growth in running time with the exception of sudden spikes when they reach the drift points. These sudden spikes in running times are particularly distinct in large slope values, where we see each algorithm taking up to ten seconds to process the first batch when drifts are introduced. We see for all slope values (small, medium and large), SeqDriftGraphMiner outperforms AdaGraphMiner through all graph instances of the YoutubeSyntheticYoutube dataset. By the end of the stream, SeqDriftGraphMiner was shown to be approximately two seconds faster than AdaGraphMiner. However, the difference in running time increases as we process more graph instances, indicating that we should see a larger and more noticeable difference if we were to use a larger dataset.

5.4.2.2 Memory

Figures 5.17 to 5.19 presents the amount of memory used by SeqDriftGraphMiner and AdaGraphMiner when processing the YoutubeSyntheticYoutube dataset with all levels of slope values. We see a fairly linear patterns for both algorithms when we utilize medium and large slope values. For small slope values however, both algorithms exhibit unpredictable jagged patterns in running time.Figures 5.18 and 5.19 show that SeqDriftGraphMiner outperforms AdaGraphMiner at each point of graph instances. However, when using small slope values (Figure 5.17) we notice several interesting behaviours. Firstly, the memory used for both algorithms become the same at the first drift point despite SeqDriftGraphMiner using less memory beforehand. We see from the ten successive graph batches after the first drift point that AdaGraphMiner outperforms SeqDriftGraphMiner, using 100MB less memory at one point. Secondly, approaching the second drift point, we see SeqDriftGraphMiner shifting back to using

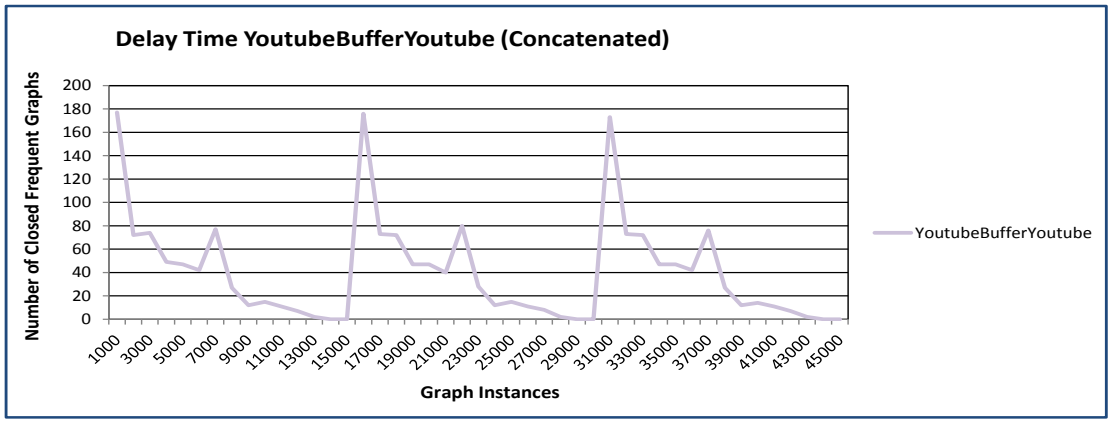


Figure 5.11: Delay Time of YoutubeSyntheticYoutube.

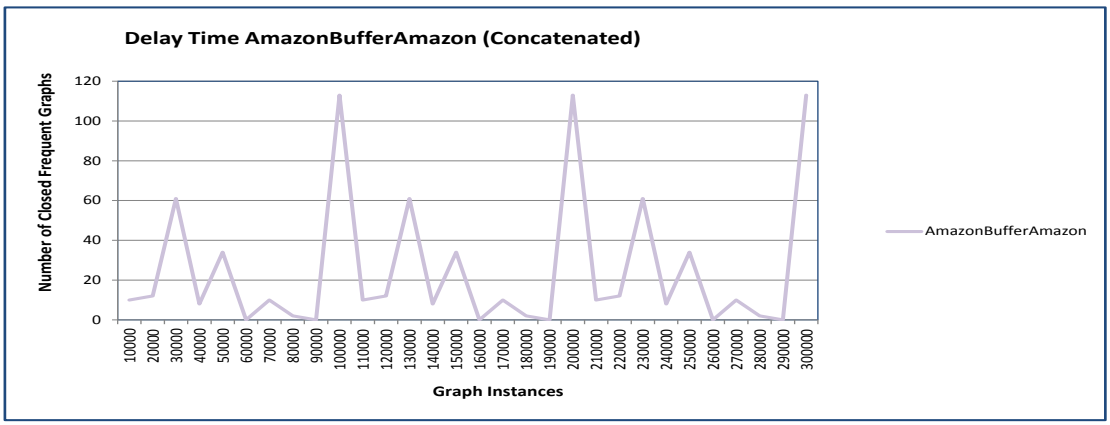


Figure 5.12: Delay Time of AmazonSyntheticAmazon.

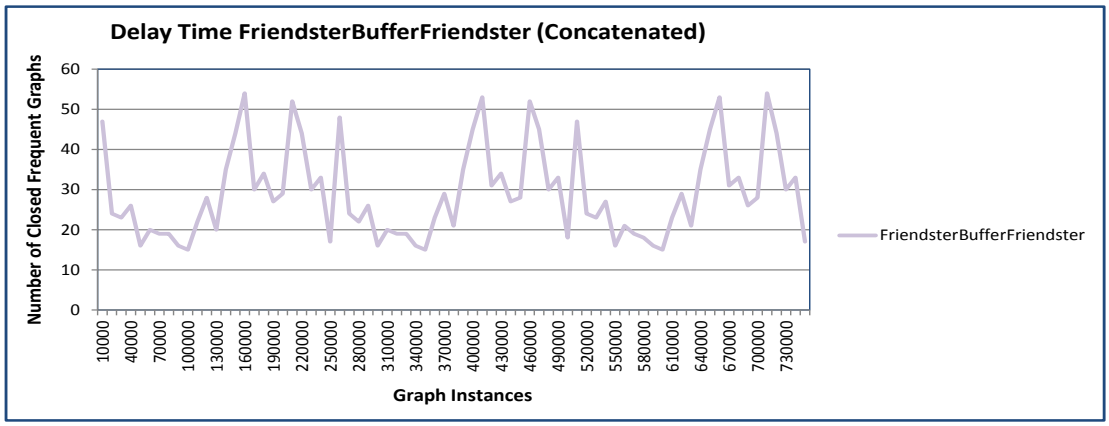


Figure 5.13: Delay Time of FriendsterSyntheticFriendster.

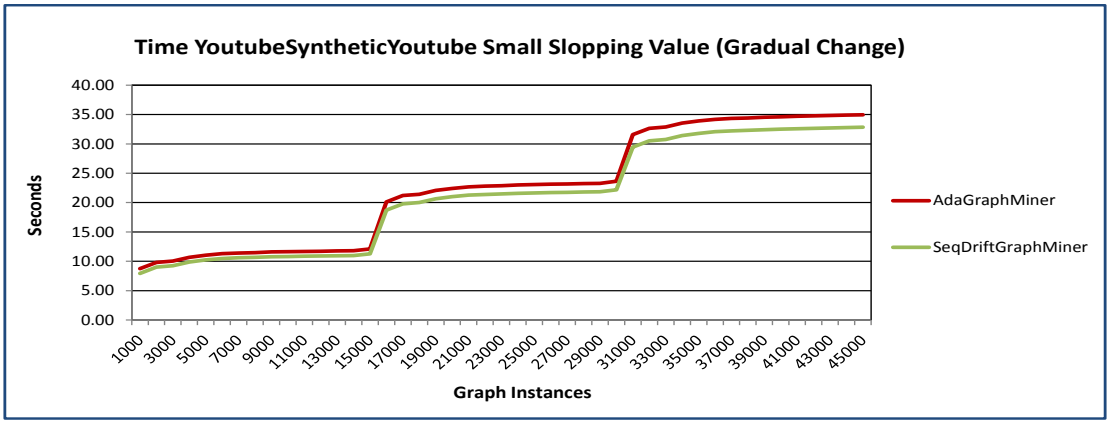


Figure 5.14: Running time of AdaGraphMiner and SeqDriftGraphMiner for YoutubeSyntheticYoutube with small slope value.

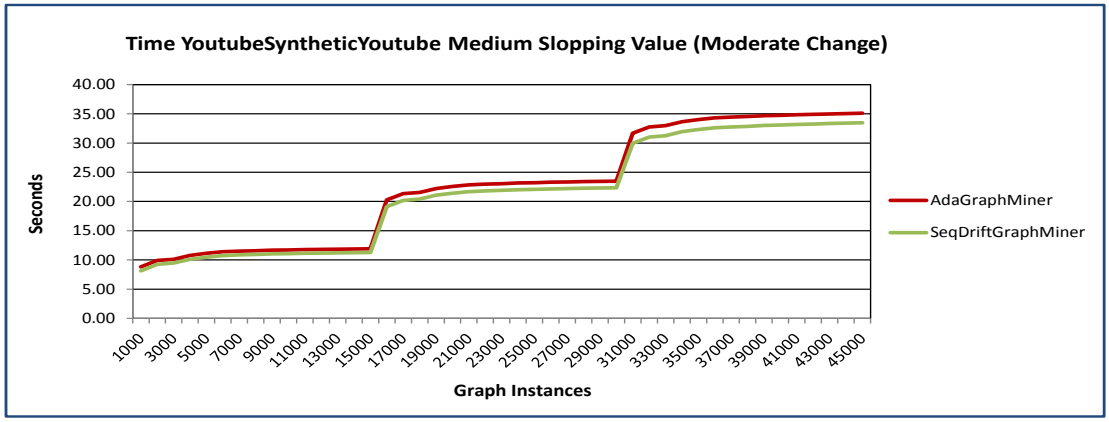


Figure 5.15: Running time of AdaGraphMiner and SeqDriftGraphMiner for YoutubeSyntheticYoutube with medium slope value.

less memory. During the second artificial drift, we notice a increasingly large difference in memory used, peaking at 250MB at the 31,000 graph instance point. The last strange phenomenon occurs at the 33,000 to 35,000 graph instances range, where the memory used for SeqDriftGraphMiner suddenly spikes before receding again.

5.4.2.3 Delay time

The delay time for SeqDriftGraphMiner and AdaGraphMiner when processing the YoutubeSyntheticYoutube dataset is shown by Figures 5.20 to 5.22. Both algorithms exhibit the same recurring patterns after each drift point as expected. For all slope values, we notice that AdaGraphMiner is able to detect the new closed frequent graphs approximately one batch faster than SeqDriftGraphMiner at each drift point.

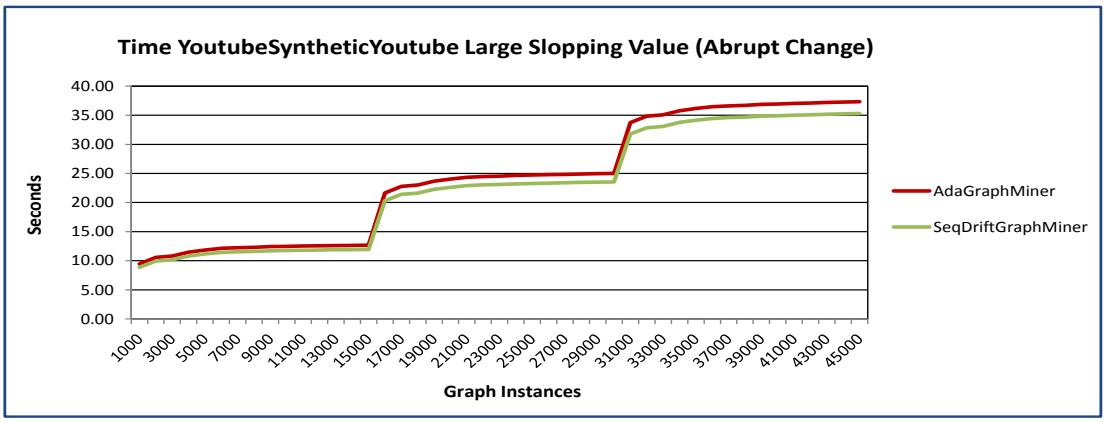


Figure 5.16: Running time of AdaGraphMiner and SeqDriftGraphMiner for YoutubeSyntheticYoutube with large slope value.

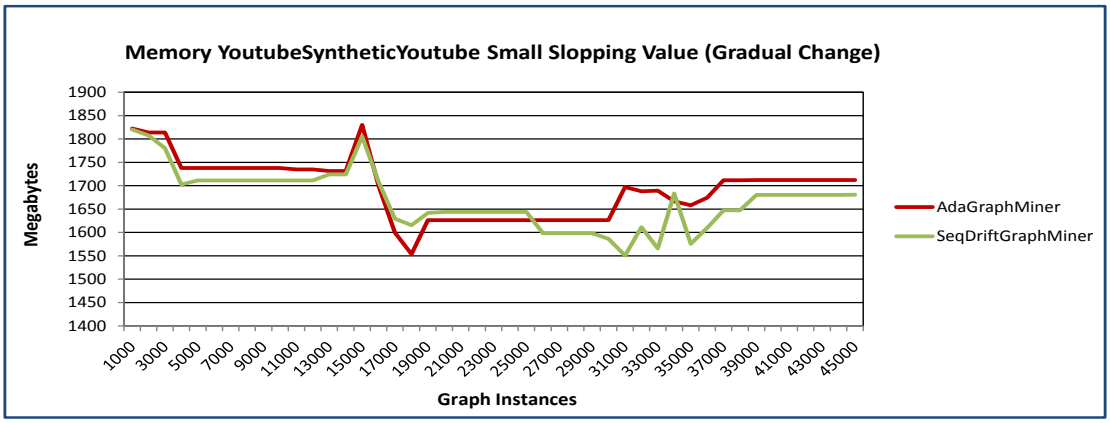


Figure 5.17: Memory used for AdaGraphMiner and SeqDriftGraphMiner for YoutubeSyntheticYoutube with small slope value.

5.4.3 AmazonSyntheticAmazon

In this section, we present the results obtained when evaluating running time, memory usage consumption and delay time for the AmazonSyntheticAmazon dataset.

5.4.3.1 Time

The running time for SeqDriftGraphMiner and AdaGraphMiner when processing the AmazonSyntheticAmazon dataset is shown in Figures 5.23 to 5.25. Both algorithms show an cubic growth pattern up until an drift point, in which the pattern resets as expected. Alike the YoutubeSyntheticYoutube dataset, we see SeqDriftGraphMiner outperforming AdaGraphMiner at all graphs instances and again, there is a increasing

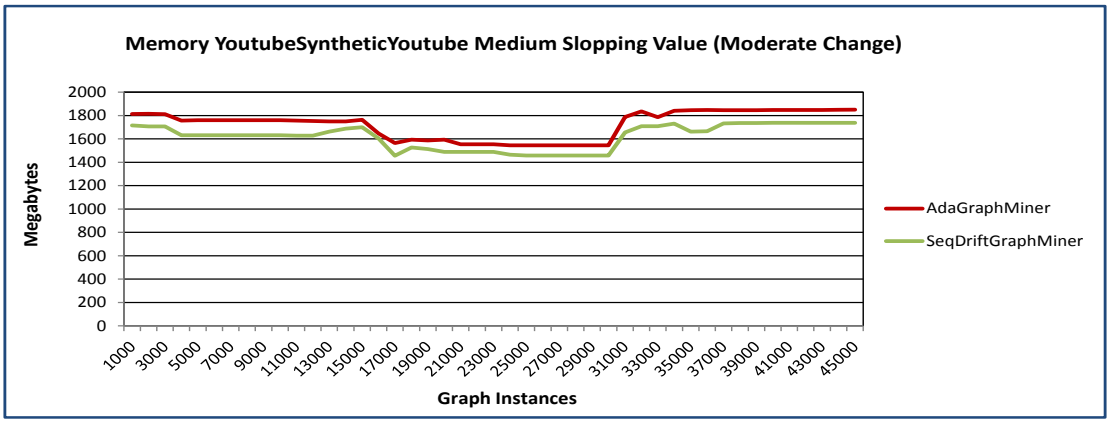


Figure 5.18: Memory used for AdaGraphMiner and SeqDriftGraphMiner for YoutubeSyntheticYoutube with medium slope value.

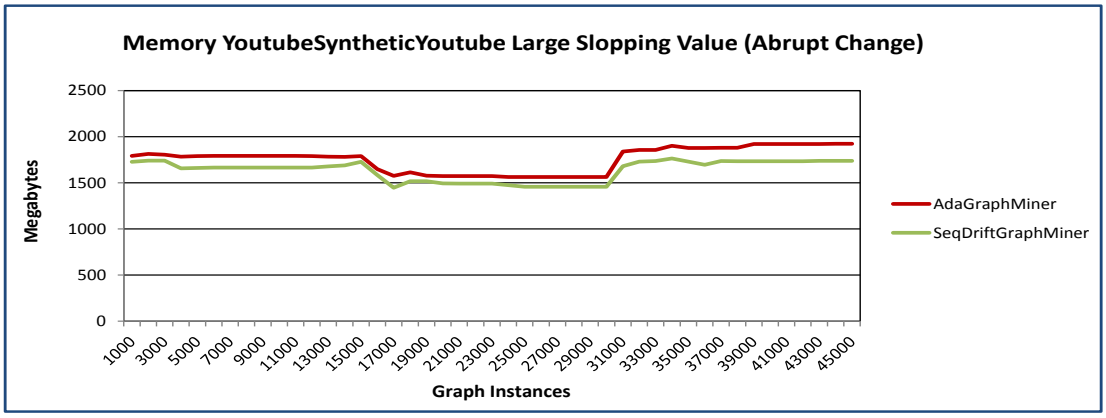


Figure 5.19: Memory Used for AdaGraphMiner and SeqDriftGraphMiner for YoutubeSyntheticYoutube with large slope value.

difference in running time between the two as more graphs instances are introduced with a five second difference shown at graph instance 30,000 for large slope values (Figure 5.25).

5.4.3.2 Memory

The memory used for AdaGraphMiner and SeqDriftGraphMiner when processing the AmazonSyntheticAmazon dataset is shown in Figures 5.26 to 5.28. Both algorithms show large spikes in memory used, peaking at the 80,000 graph instances point. For the remaining graph instances, we see a fairly linear pattern for AdaGraphMiner and a general decrease in running time for SeqDriftGraphMiner. Similar to the YoutubeSyntheticYoutube dataset, SeqDriftGraphMiner uses less memory then AdaGraphMiner

at all points in time and we see the difference increasing as we process more graph batches.

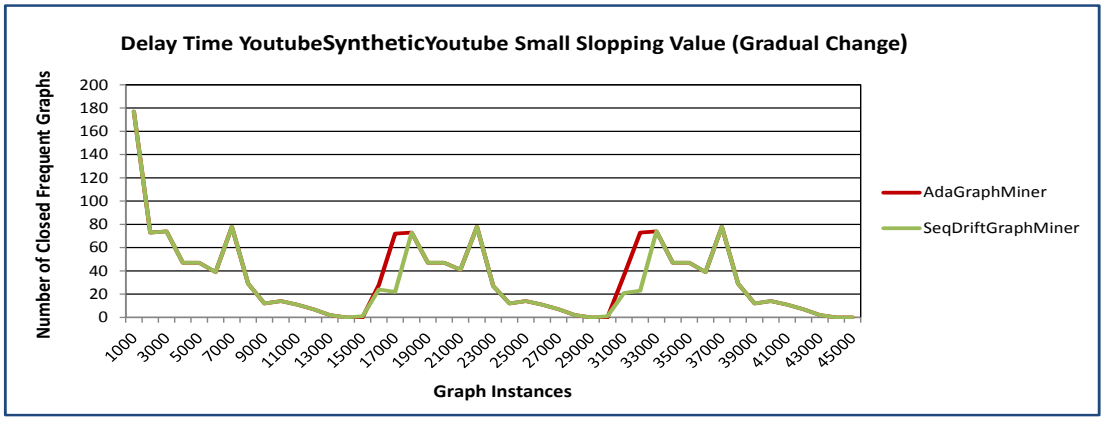


Figure 5.20: Delay Time of AdaGraphMiner and SeqDriftGraphMiner for YoutubeSyntheticYoutube with small slope value.

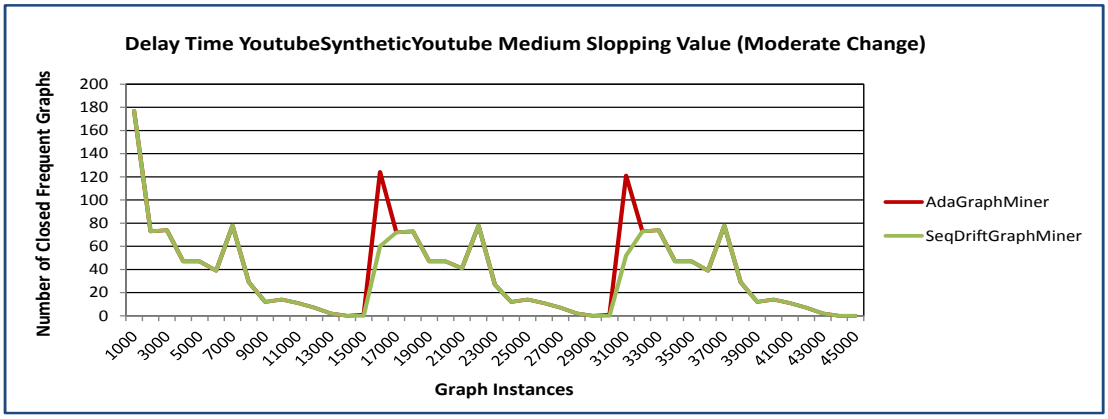


Figure 5.21: Delay Time of AdaGraphMiner and SeqDriftGraphMiner for YoutubeSyntheticYoutube with medium slope value.

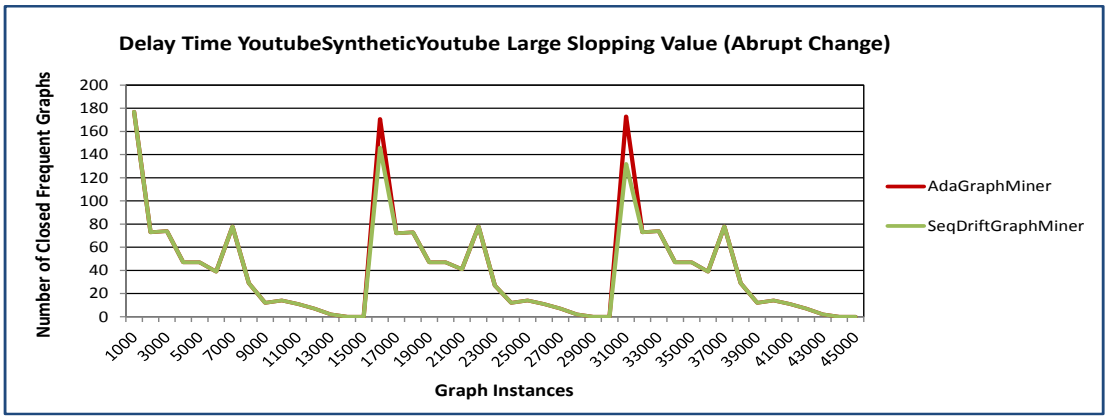


Figure 5.22: Delay Time of AdaGraphMiner and SeqDriftGraphMiner for YoutubeSyntheticYoutube with large slope value.

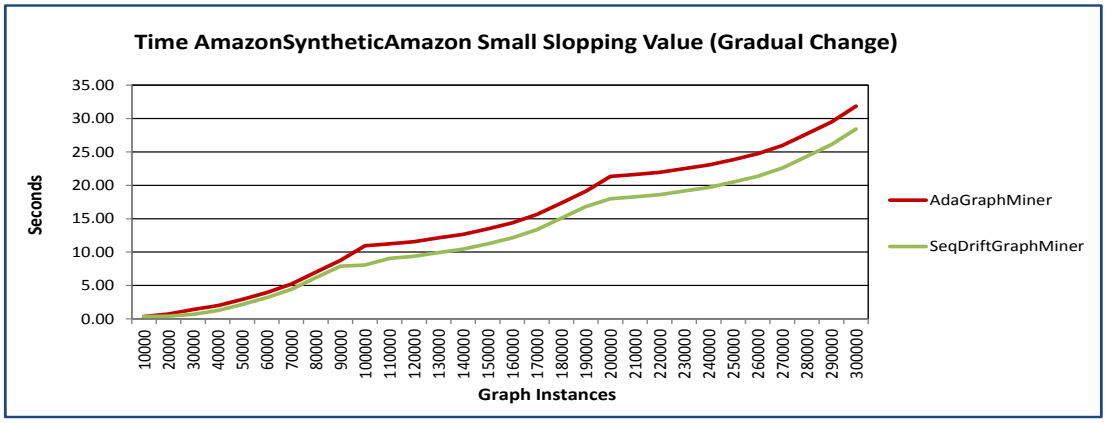


Figure 5.23: Running time of AdaGraphMiner and SeqDriftGraphMiner for AmazonSyntheticAmazon with small slope value.

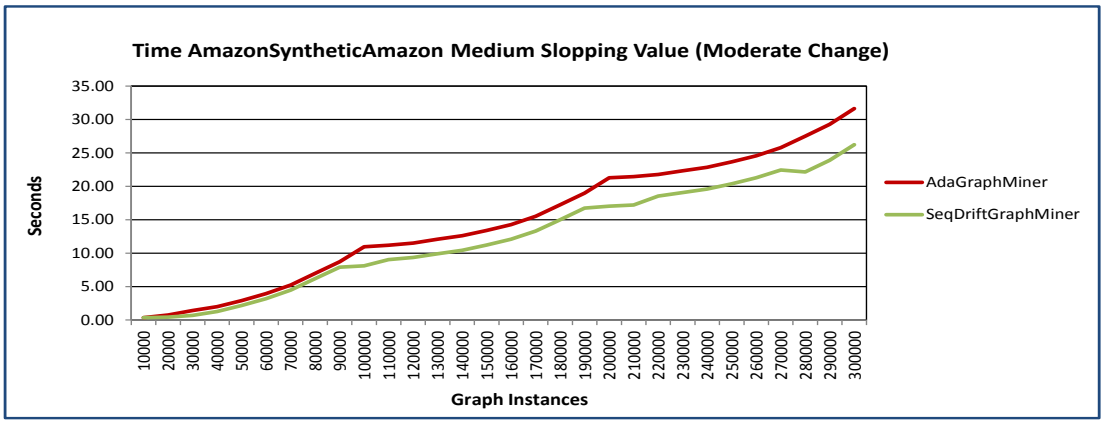


Figure 5.24: Running time of AdaGraphMiner and SeqDriftGraphMiner for AmazonSyntheticAmazon with medium slope value.

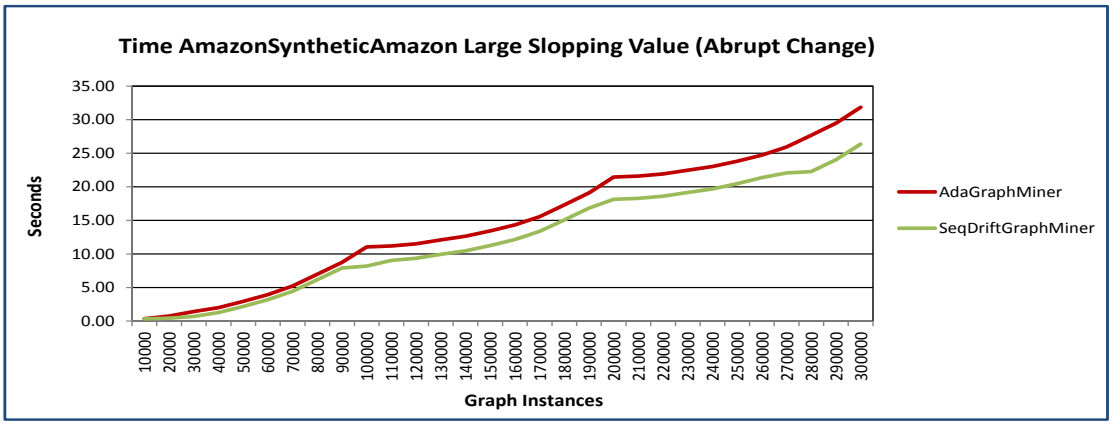


Figure 5.25: Running time of AdaGraphMiner and SeqDriftGraphMiner for AmazonSyntheticAmazon with large slope value.

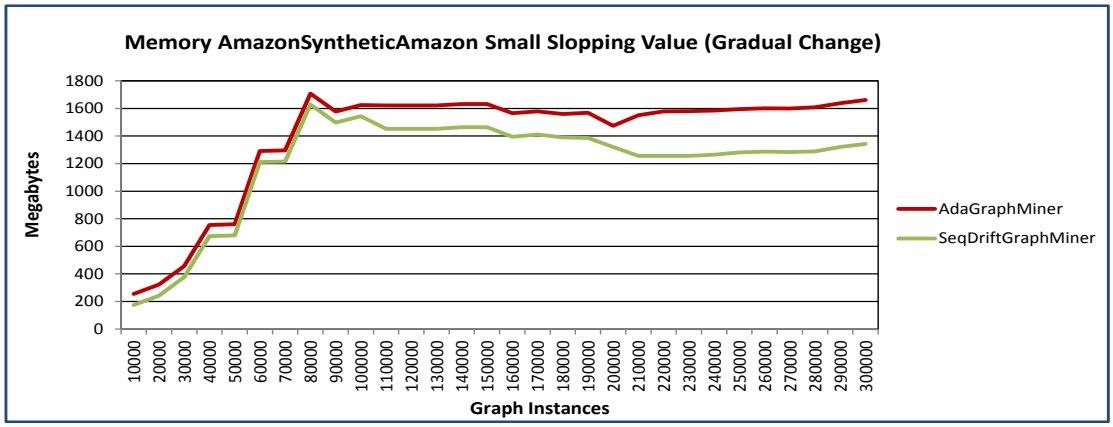


Figure 5.26: Memory used for AdaGraphMiner and SeqDriftGraphMiner for AmazonSyntheticAmazon with small slope value.

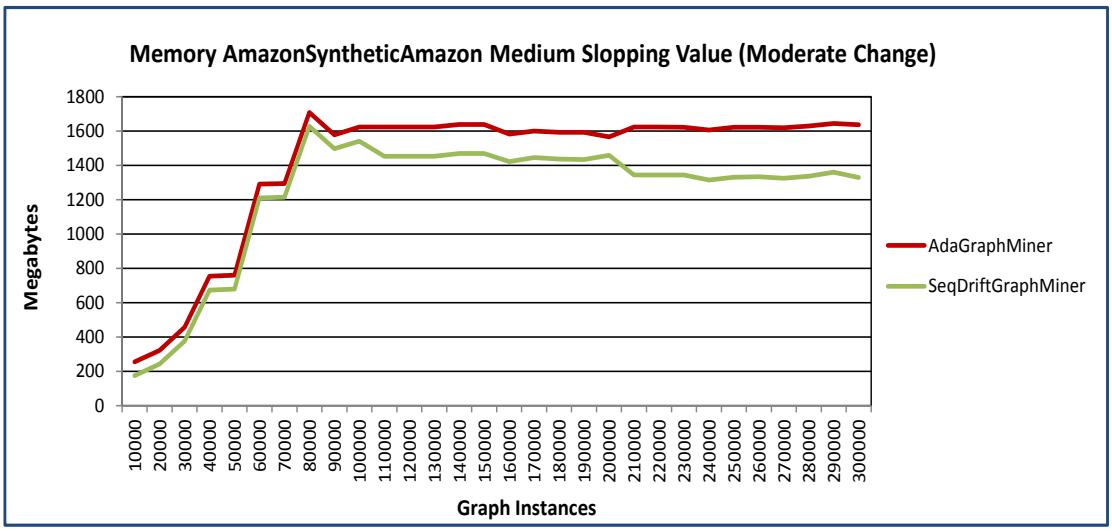


Figure 5.27: Memory used for AdaGraphMiner and SeqDriftGraphMiner for AmazonSyntheticAmazon with medium slope value.

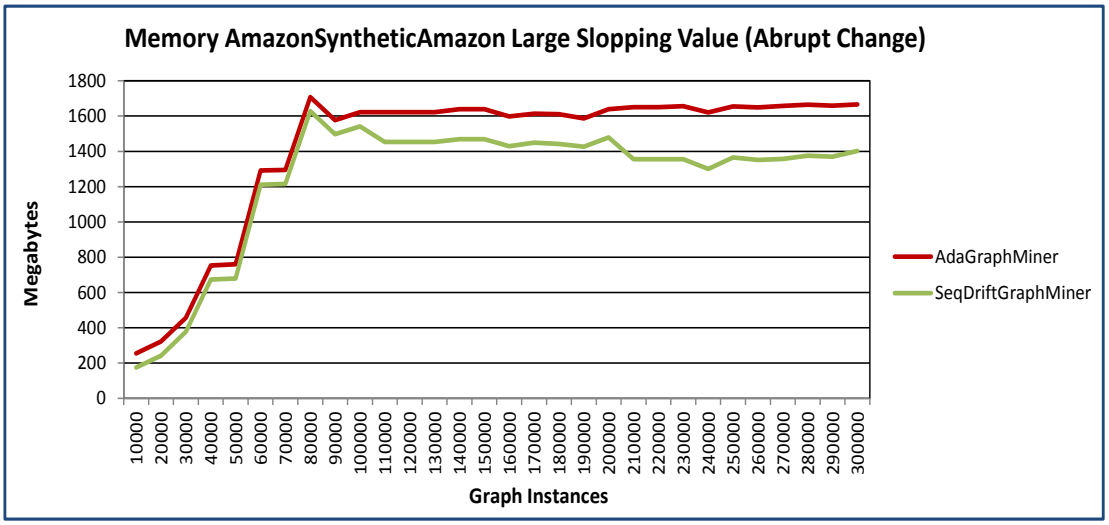


Figure 5.28: Memory Used for AdaGraphMiner and SeqDriftGraphMiner for AmazonSyntheticAmazon with large slope value.

5.4.3.3 Delay time

The delay time for SeqDriftGraphMiner and AdaGraphMiner is shown in Figures 5.29 to 5.31. Both algorithms exhibit the same recurring patterns at each drift point once again and AdaGraphMiner is shown to adapt to the changes faster than SeqDriftGraphMiner at the beginning of each drift point.

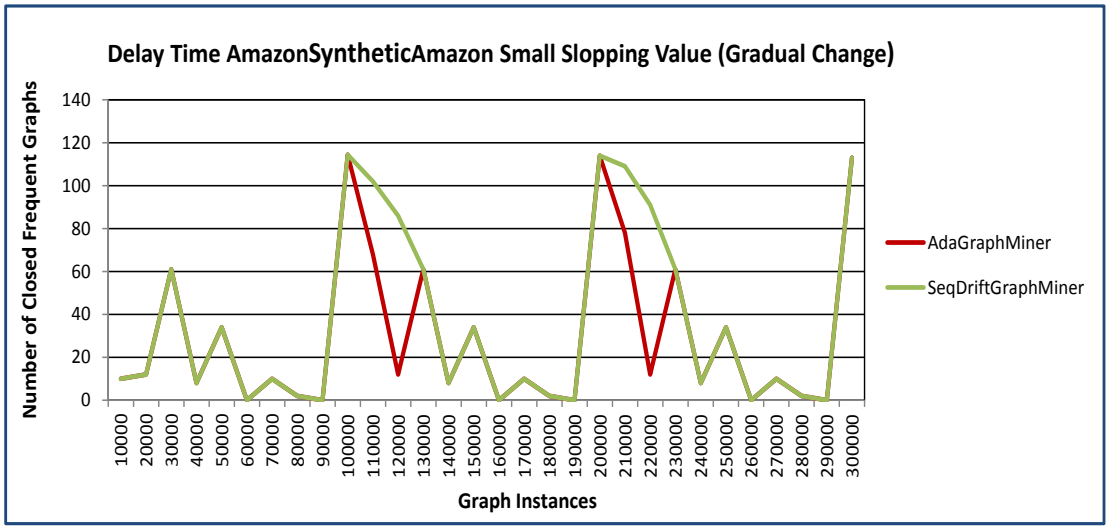


Figure 5.29: Delay Time of AdaGraphMiner and SeqDriftGraphMiner for AmazonSyntheticAmazon with small slope value.

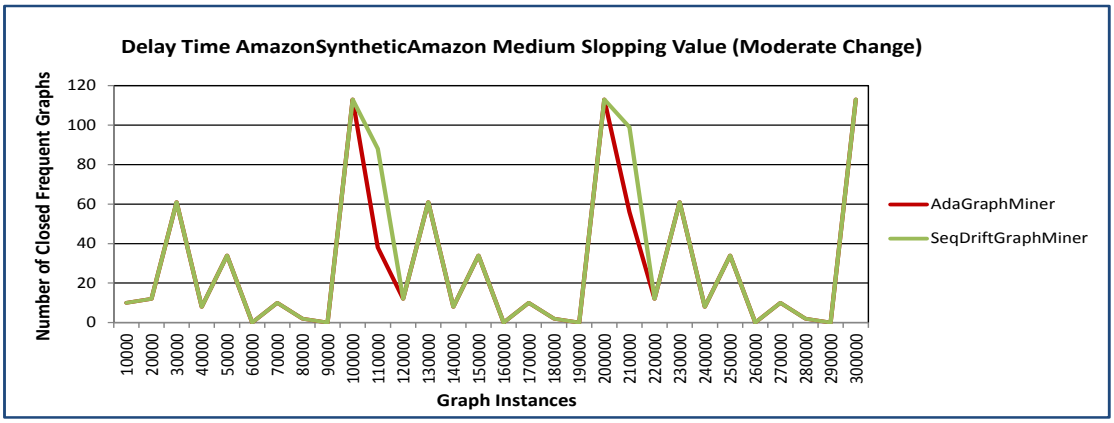


Figure 5.30: Delay Time of AdaGraphMiner and SeqDriftGraphMiner for AmazonSyntheticAmazon with medium slope value.

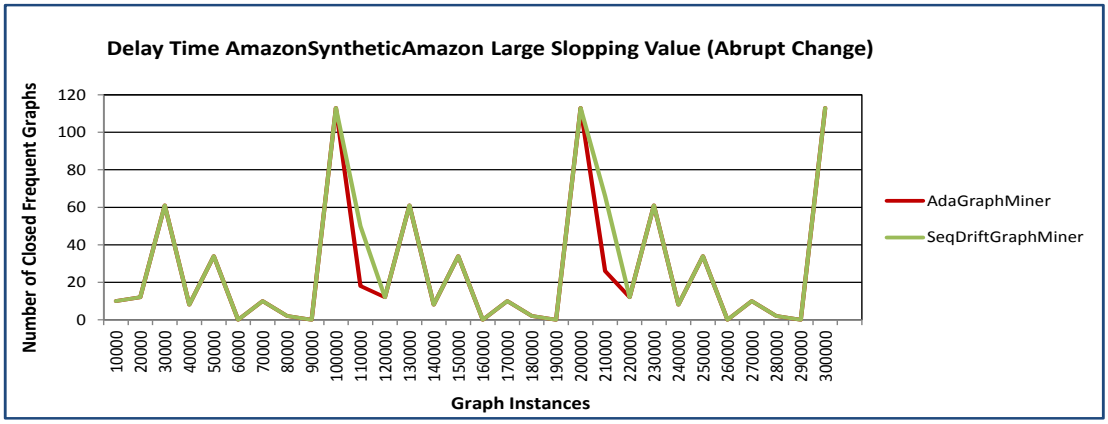


Figure 5.31: Delay Time of AdaGraphMiner and SeqDriftGraphMiner for AmazonSyntheticAmazon with large slope value.

5.4.4 FriendsterSyntheticFriendster

In this section, we present the results obtained when evaluating running time, memory usage consumption and delay time for the FriendsterSyntheticFriendster dataset.

5.4.4.1 Time

Figures 5.32 to 5.34 show the running time for AdaGraphMiner and SeqDriftGraphMiner when processing the AmazonSyntheticAmazon dataset for all gradients of change. Like the former two datasets, SeqDriftGraphMiner is shown to be faster than

AdaGraphMiner and both show a linearly pattern of growth in time. One thing to consider is that since the AmazonSyntheticAmazon dataset is much larger than the last two datasets, the difference in performance is much greater, up to the point where there is a 100 second difference (20% difference) between the two. This evidence supports the claim we made at the end of Section 5.5.1.1.

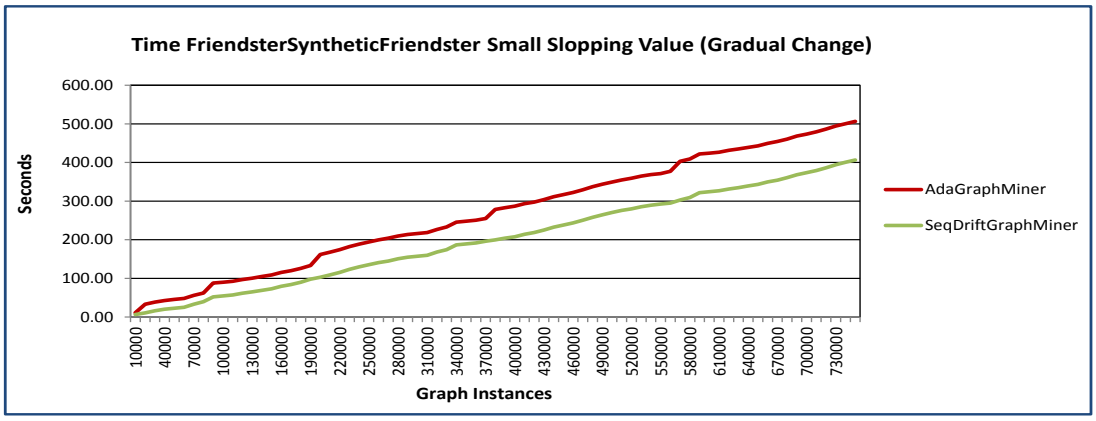


Figure 5.32: Running time of AdaGraphMiner and SeqDriftGraphMiner for FriendsterSyntheticFriendster with small slope value.

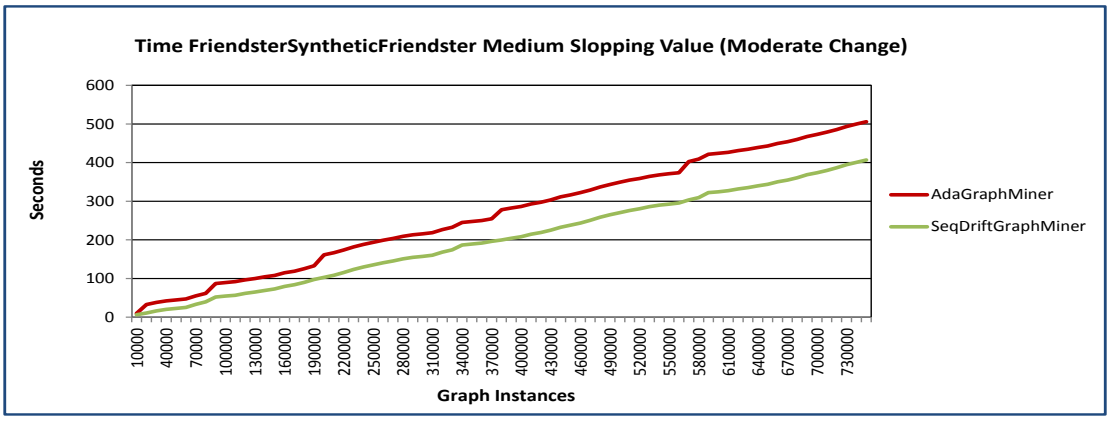


Figure 5.33: Running time of AdaGraphMiner and SeqDriftGraphMiner for FriendsterSyntheticFriendster with medium slope value.

5.4.4.2 Memory

The memory usage of the two algorithms is shown in Figures 5.35 to 5.37. Both algorithms exhibit the same behaviour, small bumps of increased memory usage until it levels off at around 45,000 graph instances. Again, SeqDriftGraphMiner is shown

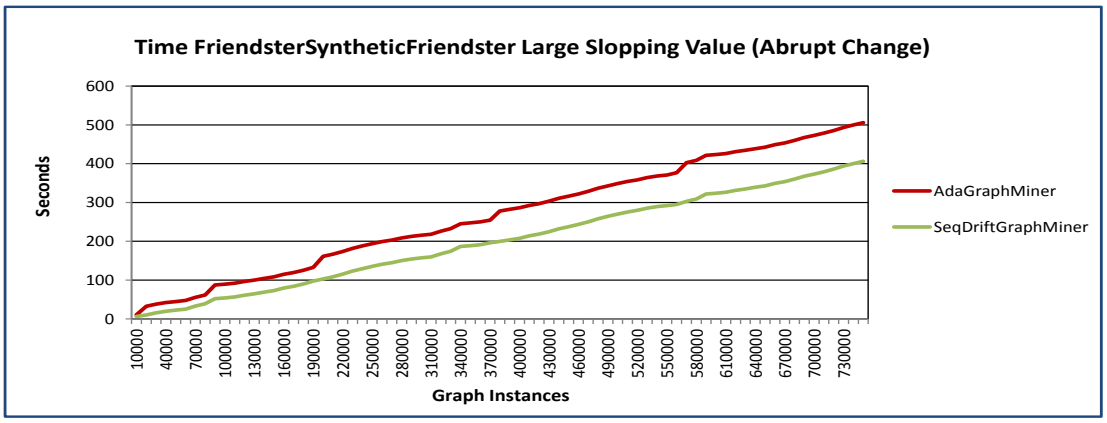


Figure 5.34: Running time of AdaGraphMiner and SeqDriftGraphMiner for FriendsterSyntheticFriendster with large slope value.

to used less memory than AdaGraphMiner, ranging from a few mebabytes to up to a gigabyte more efficient at one point.

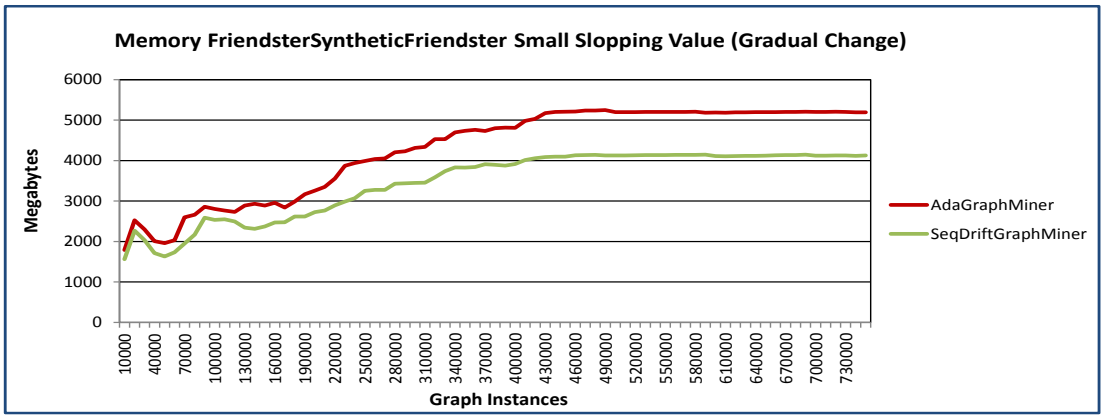


Figure 5.35: Memory used for AdaGraphMiner and SeqDriftGraphMiner for FriendsterSyntheticFriendster with small slope value.

5.4.4.3 Delay time

Finally, the delay time for the algorithms is shown in Figures 5.38 to 5.40. We see both algorithms exhibit the same patterns with the exception at the drift points (250,000 and 500,000 graph instances), where SeqDriftGraphMiner was unable to detect as many closed frequent graphs as AdaGraphMiner, indicating that it was slower to adapt to the changes in distribution.

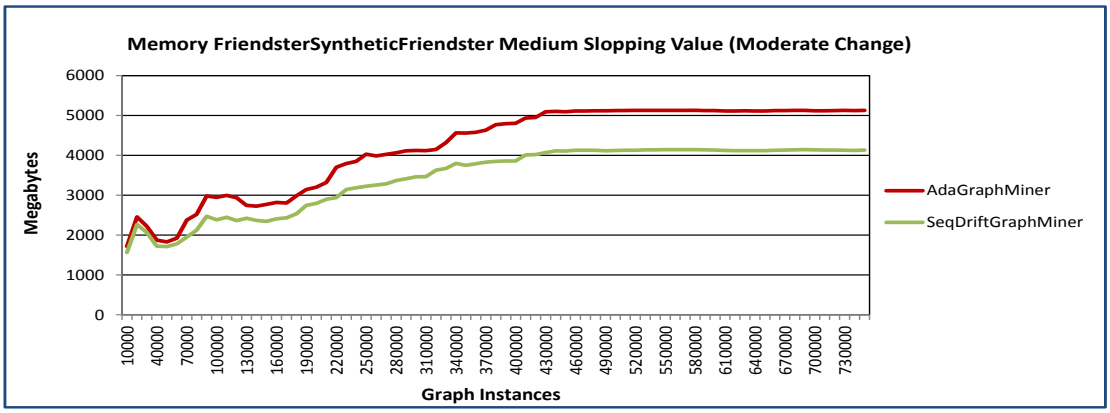


Figure 5.36: Memory used for AdaGraphMiner and SeqDriftGraphMiner for FriendsterSyntheticFriendster with medium slope value.

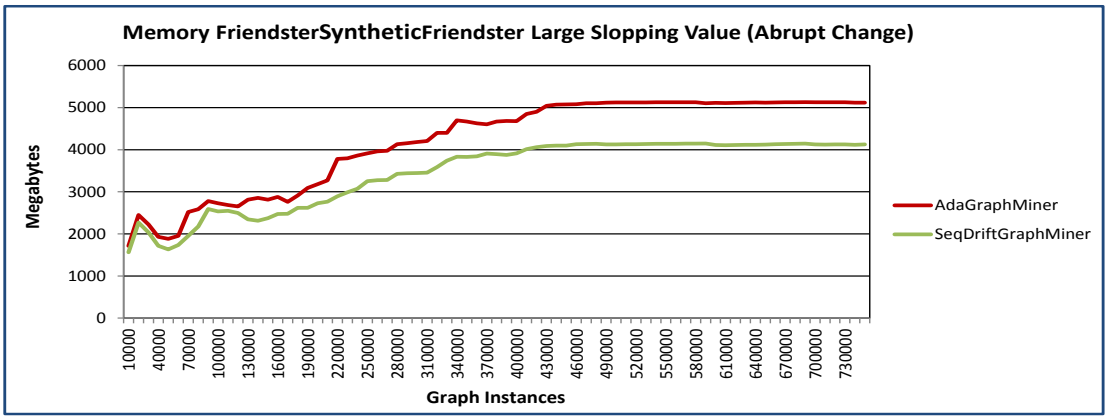


Figure 5.37: Memory Used for AdaGraphMiner and SeqDriftGraphMiner for FriendsterSyntheticFriendster with large slope value.

5.5 Conclusion

In this chapter, we presented an comparison on the performance of running time, memory used and delay time for our benchmark technique, AdaGraphMiner and our proposed technique, SeqDriftGraphMiner. Our results show, for both chemical molecular dense and social network sparse datasets, that SeqDriftGraphMiner is faster and requires less memory on average when processing various imitated non-stationary data streams. AdaGraphMiner was shown to detect changes for the majority of drift points faster than SeqDriftGraphMiner, thus having a smaller delay time. However, according to previous research, it is likely that the delay time of SeqDriftGraphMiner can be improved by having more appropriate parameter settings for its change detector, SEQ-

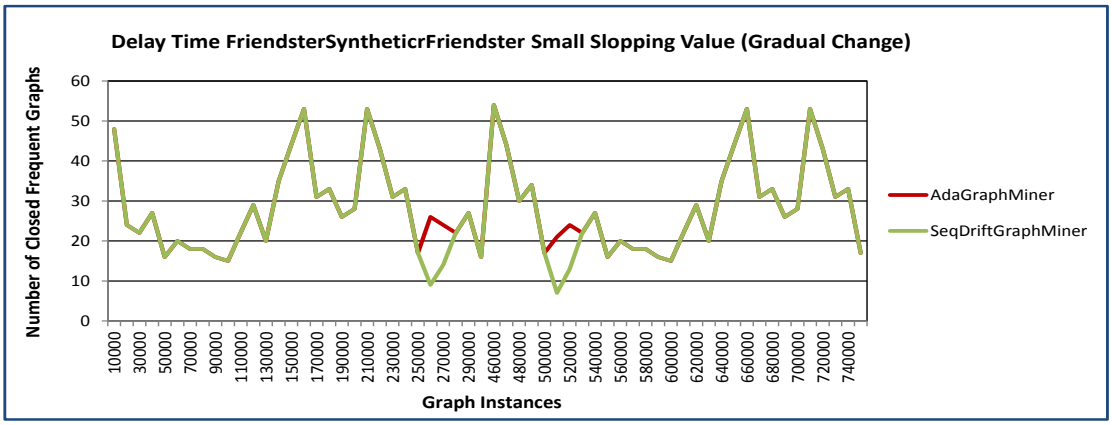


Figure 5.38: Delay Time of AdaGraphMiner and SeqDriftGraphMiner for FriendsterSyntheticFriendster with small slope value.

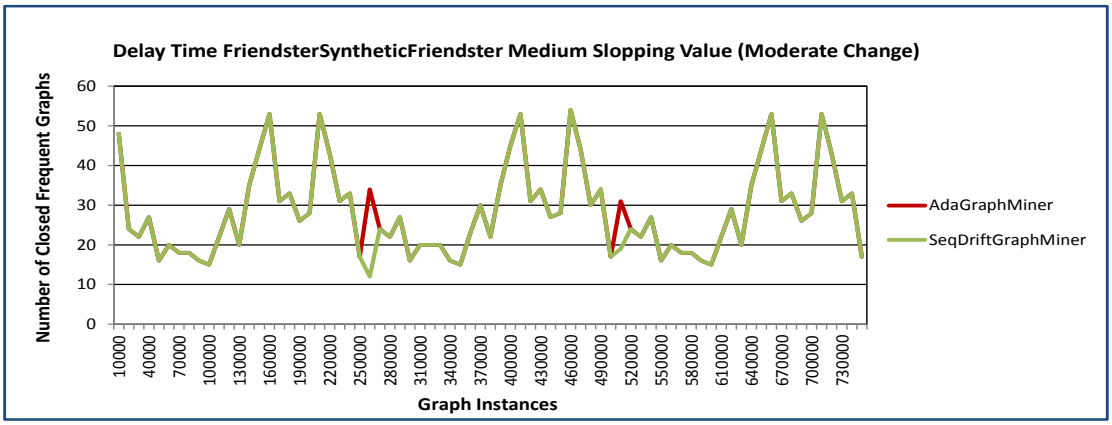


Figure 5.39: Delay Time of AdaGraphMiner and SeqDriftGraphMiner for FriendsterSyntheticFriendster with medium slope value.

DRIFT.

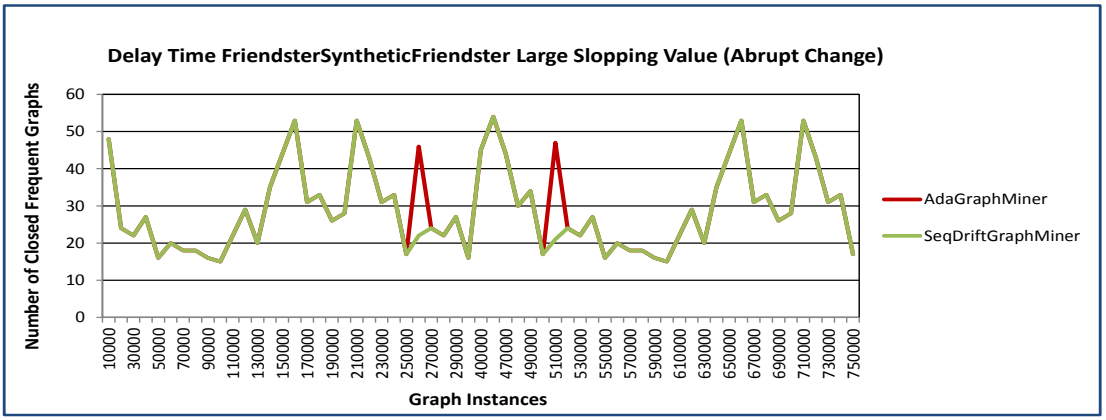


Figure 5.40: Delay Time of AdaGraphMiner and SeqDriftGraphMiner for FriendsterSyntheticFriendster with large slope value.

Chapter 6

CONCLUSIONS AND FUTURE WORK

This chapter contains three sections. The first section summarizes the research that we have undertaken and describes the contributions of our approaches. The second section presents some limitations of our research. The last section discusses potential research directions for the future.

6.1 Summary and Contributions

The focus of our research is on closed frequent graph mining for data streams. Performing frequent graph mining in a non-stationary data stream environment introduces many challenges such as constraints in running time, storage space, computation speed and adaption to changes. Dealing with these constraints, Bifet's graph batch mining framework [13] (Section 2.3.1) contains several modes that allow it to perform closed frequent graph mining on chemical molecular graphs through the use of incremental mining (IncGraphMiner), sliding window mining (WinGraphMiner) and most notably, adaptive mining (AdaGraphMiner). We focus primarily on the AdaGraphMiner-ADAADWIN mode (Section 2.3.1.4) which uses separate ADWIN instances to manage support values of each closed frequent graph found. We chose the AdaGraphMiner-ADAADWIN mode (Section 2.3.1.4) as our benchmark technique because the mode was proven in previous research to be the fastest mode to adapt to changes in the graph data stream.

Our research proposes three main contributions that improve upon existing research on frequent graph mining for data streams. For the first contribution, we expand the range of variables used for performance comparison for AdaGraphMiner-ADAADWIN mode in previous research work. Performance analysis will now include running time of the algorithm and the amount of memory used. Furthermore, we implement and integrate the state-of-the-art change detector, SEQDRIFT (Section 4.3) into Bifet's graph batch mining framework, resulting in a new mode SeqDrift-GraphMiner. We compare the performance of the two modes, SeqDriftGraphMiner and AdaGraphMiner-ADAADWIN using different datasets varying the severity of the drift. For dense datasets, the SeqDriftGraphMiner mode is shown to have outperformed the AdaGraphMiner -ADAADWIN mode in terms of time and memory

for all gradients of changes. However, the delay time of SeqDriftGraphMiner is relatively higher, especially for datasets with gradual drift, where changes were not even detected up until the end of the data stream. For sparse datasets, the overall experiment results are similar to the dense datasets, SeqDriftGraphMiner has worse detection delay when compared to AdaGraphMiner-ADAADWIN. The delay times of SeqDriftGraphMiner increases with decreasing gradient of change. This is a tradeoff of SeqDriftGraphMiner for having less computation overheads, running time and memory usage as compared to AdaGraphMiner-ADAADWIN. One fact we observe from the experiments of the sparse datasets is that the gaps between the two processing times and memory usages become wider as the number of graph instances increase.

Our second contribution expands the input formats of Bifet's graph batch mining framework (Section 2.3.1) which, as noted before, only takes chemical molecular graphs in SMILES format (Section 3.1). Since we are interested in performing analysis on large social networks, we integrate functionalities that allow Bifet's graph batch mining framework to process general graphs in NEList format (Section 3.2). Most publicly available social networking datasets are in either From-to node general graph (Section 4.5.1) or Single line community (Section 4.5.2) formats. However, to maintain consistency and simplicity we convert and categorize all general graphs into NEList format using algorithms described in Section 4.5.

For our final contribution, we perform an in-depth analysis on outputs produced by closed frequent graph miners. Bifet's graph batch mining framework only produces the number of closed frequent graphs in the coreset, not providing any valuable information other than times which the frequencies change. We wish to derive meaning from the numbers by analyzing patterns, trends and points of interest that appear. By performing analysis on output coresets, we find two major different types of change when mining closed frequent graphs in data streams and name them Real Drift and Internal Change. The formal definitions of these two types of change are provided in Section 4.6.

6.2 Limitations

1. For our first contribution, the parameter settings used for the change detector, SEQDRIFT, results in better time and memory performance, however SeqDriftGraphMiner produces higher delay times as compared to the AdaGraphMiner-ADAADWIN. As mentioned in Section 2.4.3, the experiments conducted show that by adjusting the warning level and block size of SEQDRIFT, we reduce the difference in delay time when compared to ADWIN. However, we have not tried

different parameter settings of SEQDRIFT to see whether without sacrificing running time and memory usage, the delay time of SeqDriftGraphMiner can be improved especially when changes exist in a graph mining scenario.

2. Our third contribution focuses on analyzing the mining outcomes of closed frequent graphs as this area has never been explored in previous work. In Section 4.6, we discuss the differences between real drift and internal change in the outputs and present detailed examples of these two concepts. In fact, we noticed some other interesting patterns and phenomenon that appear in the mining outcomes during our experiments which we would like to explore further in the future.

3. From our algorithm, we see that our approach performs more efficiently when the distribution of graphs is changing rapidly rather than when changes arrive homogeneously. The reason behind this is that we monitor the entire graph data stream when we perform mining, which may be suitable for streams with rapidly-changing distributions but will not be efficient when analyzing streams that are comparatively stable. An alternative approach to effectively mine stable streams is to focus solely on significant/hub nodes that reoccur frequently.

4. Change mining is crucial in domains where early detection of emerging trends is vital to allow users to make decisions in anticipation rather than in reaction. The technique we developed can effectively and efficiently detect shifts in graph data streams. However, at this stage our technique is not able to analyze already detected changes for anticipating any emerging trends in the distribution of graphs in an evolving stream scenario.

6.3 Future Work

Frequent graph mining for data streams is a relatively new research area in the data mining field. Although we have done some work, there is still a lot more that can be explored on the topic. Below we provide some potential future research directions:

1. To alter the parameter settings of SEQDRIFT and perform evaluations on performance with different combinations of settings.

2. To provide deeper understanding in this field and produce more sophisticated analysis on output patterns to better understand what they represent.

3. As stated above, significant node mining may yield more efficient and effective outcomes when data coming in is homogeneous. In addition, in some applications of social network analysis, hub nodes/graphs are probably the only frequent

patterns that people are interested in finding. Hence, in the future we would like to develop a new approach where we monitor the concept drifts of the hub nodes within the distributions of graphs and only mine closed frequent graphs when concept drifts of those hub nodes are triggered.

Bibliography

[1] Bifet, A., Holmes, G., Pfahringer, B., & Gavald, R. (2011). Mining frequent closed graphs on evolving data streams. In: Proceeding of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 591-599

[2] Yan, X., & Han, J. (2002). gSpan: graph-based substructure pattern mining. In: Proceeding of the 2002 International Conference on Data Mining (ICDM02), Maebashi, Japan, pp 721-724

[3] Yan, X., & Han, J. (2003). CloseGraph: mining closed frequent graph patterns. In: Proceeding of the 2003 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD03), Washington, DC, pp 286-295

[4] Kuramochi, M., & Karypis, G. (2001). Frequent subgraph discovery. In: Proceeding of the 2001 International Conference on Data mining (ICDM01), San Jose, CA, pp 313-320

[5] Inokuchi, A., Washio, T., & Motoda, H. (2000). An apriori-based algorithm for mining frequent substructures from graph data. In: Proceeding of the 2000 European Symposium on the Principle of Data Mining and Knowledge Discovery (PKDD00), Lyon, France, pp 13-23

[6] Borgelt, C., & Berthold, M. R. (2002). Mining molecular fragments: finding relevant substructures of molecules. In: Proceeding of the 2002 International Conference on Data Mining (ICDM02), Maebashi, Japan, pp 211-218

[7] Borgelt, C., Meinl, T., & Berthold, M. R. (2004). Advanced pruning strategies to speed up mining closed molecular fragments. In: Proceeding of the 2004 IEEE International Conference on Systems, Man and Cybernetics, Vol 5, pp 4565-4570

[8] Klemettinen, M., Mannila, H., Ronkainen, P., Toivonen, H., & Verkamo, A. I. (1994). Finding interesting rules from large sets of discovered association rules. In: Proceeding of the Third International Conference on Information and Knowledge Management, pp 401-407

[9] Zaki, M. J., Parthasarathy, S., Ogihara, M., & Li, W. (1997). New algorithms for fast discovery of association rules. In: Proceeding of the Third International Conference on Knowledge Discovery and Data Mining, Vol 97, pp 283-286

[10] Borgelt, C., Meinl, T., & Berthold, M. (2005). Moss: a program for molecular substructure mining. In: Proceeding of the First International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations, pp 6-15

[11] Borgelt, C., & Meinl, T. (2006). Full perfect extension pruning for frequent graph mining. In: Proceeding of the Sixth IEEE International Conference on Data Mining, pp 263-268

[12] Ray, A., Holder, L. B., & Choudhury, S. (2014). Frequent Subgraph Discovery in Large Attributed Streaming Graphs. In: Proceeding of the Third International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, pp 166-181

[13] Bifet, A., Holmes, G., Pfahringer, B., & Gavald, R. (2011). Mining frequent closed graphs on evolving data streams. In: Proceeding of the Seventeenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 591-599

[14] Wackersreuther, B., Wackersreuther, P., Oswald, A., Bhm, C., & Borgwardt, K. M. (2010). Frequent subgraph discovery in dynamic networks. In: Proceeding of the Eighth Workshop on Mining and Learning with Graphs, pp 155-162

[15] Aggarwal, C. C., Li, Y., Yu, P. S., & Jin, R. (2010). On dense pattern mining in graph streams. In: Proceeding of the Very Large Data Bases Endowment, pp 975-984

[16] Borgelt, C. (2007). Canonical forms for frequent graph mining. In Advances in Data Analysis, pp 337-349, Springer Berlin Heidelberg

[17] Berlingerio, M., Bonchi, F., Bringmann, B., & Gionis, A. (2009). Mining graph evolution rules. In Machine Learning and Knowledge Discovery in Databases, pp 115-130, Springer Berlin Heidelberg

[18] Holder, L. B., Cook, D. J., & Djoko, S. (1994). Substructure discovery in the subdue system. In: Proceeding of the AAAI94 Workshop Knowledge Discovery in Databases (KDD94), Seattle, WA, pp 169-180

[19] Han, J., Cheng, H., Xin, D., & Yan, X. (2007). Frequent pattern mining: current status and future directions. *In Data Mining and Knowledge Discovery*, *15*(1), pp 55-86

[20] Sebastiao, R., & Gama, J. (2009). A study on change detection methods. In: Proceeding of the Fourth Portuguese Conference on Artificial Intelligence, pp 353-364

[21] Gama, J., & Pinto, C. (2006). Discretization from data streams: applications to histograms and data mining. In: Proceeding of the 2006 ACM Symposium on Applied Computing, pp 662-667

[22] Bifet, A., & Gavalda, R. (2007). Learning from Time-Changing Data with Adaptive Windowing. In: Proceeding of SIAM International Conference on Data Mining, pp 443-448

[23] Sakthithasan, S., Pears, R., & Koh, Y. S. (2013). One pass concept change detection for data streams. In Advances in Knowledge Discovery and Data Mining, pp 461-472, Springer Berlin Heidelberg

[24] Agrawal, R., Imieliski, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, *22*(2), pp 207-216

[25] Han, J., Cheng, H., Xin, D., & Yan, X. (2007). Frequent pattern mining: current status and future directions. *In Data Mining and Knowledge Discovery*, *15*(1), pp 55-86

[26] Vanetik, N., Gudes, E., & Shimony, S. E. (2002). Computing frequent graph patterns from semistructured data. In: Proceeding of IEEE International Conference on Data Mining, pp 458-465

[27] Koh, Y. S. (2007). Generating sporadic association rules (Doctoral thesis, University of Otago, Dunedin, New Zealand). Retrieved from http://www.otago.ac.nz/library/index.html

[28] Weininger, D. (1988). SMILES, a chemical language and information system. Introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences*, *28*(1), pp 31-36

[29] Wild, D. (2004). SMILES2 [PowerPoint slides]. Retrieved from www.indiana.edu/ chem-info/gw/SMILESNEW2.ppt.

[30] Bifet, A. (2011). Mining Frequent Closed Graphs on Evolving Data Streams [PowerPoint slides]. Retrieved from http://www.slideshare.net/abifet/mining-frequent-closed-graphs-on-evolving-data-streams, pp 18-19

[31] Gouda, K., & Zaki, M. (2001). Efficiently mining maximal frequent itemsets. In: Proceeding of IEEE International Conference on Data Mining, pp 163-170

[32] Stanford Large Network Dataset Collection. (n.d.). Retrieved from https://snap.stanford.edu/data/index.html

[33] Zaki, M. J. (2009). Closed itemset mining and non-redundant association rule mining. In Encyclopedia of Database Systems, pp 365-368, Springer US

[34] Bifet, A., & Gavalda, R. (2007). Learning from Time-Changing Data with Adaptive Windowing. In SIAM International Conference on Data Mining, pp 443-448

[35] Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P., & Bhattacharjee, B. (2007). Measurement and analysis of online social networks. In: Proceeding of the Seventh ACM SIGCOMM Conference on Internet Measurement, pp 29-42

[36] Friendster Social Network Dataset: Friends. (2011). Retrieved from https://archive.org/details/friendster-dataset-201107

[37] Dehaspe, L., Toivonen, H., & King, R. D. (1998). Finding Frequent Substructures in Chemical Compounds. In: Proceeding of the 1998 International Conference on Knowledge Discovery and Data Mining, pp 30-36

[38] Holder, L. B., Cook, D. J., & Djoko, S. (1994). Substucture Discovery in the SUBDUE System. In: Proceeding of the AAAI94 Workshop Knowledge Discovery in Databases, pp 169-180

[39] Nijssen, S., & Kok, J. N. (2004). A quickstart in frequent structure mining can make a difference. In: Proceeding of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 647-652

[40] Huan, J., Wang, W., Prins, J., & Yang, J. (2004). Spin: mining maximal frequent subgraphs from graph databases. In: Proceeding of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 581-586

[41] Huan, J., Wang, W., & Prins, J. (2003). Efficient mining of frequent subgraphs in the presence of isomorphism. In: Proceeding of the Third IEEE International Conference on Data Mining, pp 549-552

[42] Yan, X., Han, J., & Afshar, R. (2003). CloSpan: Mining closed sequential patterns in large datasets. In: Proceeding of the 2003 SIAM International Conference on Data Mining, pp 166-177

[43] Huan, J., Wang, W., Bandyopadhyay, D., Snoeyink, J., Prins, J., & Tropsha, A. (2004). Mining protein family specific residue packing patterns from protein structure graphs. In: Proceeding of the Eighth Annual International Conference on Research in Computational Molecular Biology, pp 308-315

[44] Yan, X., Zhou, X., & Han, J. (2005). Mining closed relational graphs with connectivity constraints. In: Proceeding of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, pp 324-333

[45] Yang, Q., & Wu, X. (2006). 10 challenging problems in data mining research. *International Journal of Information Technology & Decision Making*, *5*(04), pp 597-604

[46] Liu, B., Hsu, W., Han, H. S., & Xia, Y. (2000). Mining changes for real-life applications. In Data Warehousing and Knowledge Discovery, pp 337-346, Springer Berlin Heidelberg

[47] Dong, G., & Li, J. (1999). Efficient mining of emerging patterns: Discovering trends and differences. In: Proceeding of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 43-52

[48] Tsai, C. Y., & Shieh, Y. C. (2009). A change detection method for sequential patterns. *Decision Support Systems*, *46*(2), pp 501-511

[49] Song, H. S., kyeong Kim, J., & Kim, S. H. (2001). Mining the change of customer behavior in an internet shopping mall. *Expert Systems with Applications*, *21*(3), pp 157-168