Incremental Algorithm for Discovering Frequent Subsequences in Multiple Data Streams

Reem Al-Mulla, University of Sharjah, UAE
Zaher Al Aghbari, University of Sharjah, UAE

ABSTRACT

In recent years, new applications emerged that produce data streams, such as stock data and sensor networks. Therefore, finding frequent subsequences, or clusters of subsequences, in data streams is an essential task in data mining. Data streams are continuous in nature, unbounded in size and have a high arrival rate. Due to these characteristics, traditional clustering algorithms fail to effectively find clusters in data streams. Thus, an efficient incremental algorithm is proposed to find frequent subsequences in multiple data streams. The described approach for finding frequent subsequences is by clustering subsequences of a data stream. The proposed algorithm uses a window model to buffer the continuous data streams. Further, it does not recompute the clustering results for the whole data stream at every window, but rather it builds on clustering results of previous windows. The proposed approach also employs a decay value for each discovered cluster to determine when to remove old clusters and retain recent ones. In addition, the proposed algorithm is efficient as it scans the data streams once and it is considered an Any-time algorithm since the frequent subsequences are ready at the end of every window.

Keywords: Any-Time Algorithm, Clustering Subsequences, Data Streams, Frequent Subsequences, Incremental Algorithm

1. INTRODUCTION

In recent years, many new applications emerged that generate data streams. Examples of applications that generate data streams are: financial applications, network monitoring, web applications, sensor networks, etc. (Tjoede & Taniar, 2005; Goh & Taniar, 2004). Unlike traditional static databases, data streams are continuous, unbounded in size, and usually with high arrival rate.

The nature of data streams poses some requirements when designing an algorithm to mine them such as finding the frequent subsequences. For example, since data streams are unbounded in size and have high arrival rate, algorithms are allowed only one look at the data.
This means that algorithms for data streams may not have the chance to revisit the data twice. To solve this problem a buffer is used to collect the data temporarily for processing. A sliding window model (Zhu & Shasha, 2002) can be used to buffer $n$ values of a data stream. Also the algorithm should be incremental, which means that the algorithm does not recompute the results after every window, but rather it only updates and builds on computed results of previous windows.

In this paper, we investigate finding frequent subsequences in multiple data streams. The approach of the proposed algorithm for finding frequent subsequences is by clustering subsequences of a data stream. A subsequence is considered to be frequent if the number of similar subsequences in a cluster is above a threshold value called support. Due to the challenging characteristics of data streams (continuous, unbounded in size, and usually with high arrival rate), the proposed algorithm is incremental, efficient and any-time algorithm. That is at the end of every window, the proposed algorithm does not recompute the clustering results of similar subsequences however it updates the previous clustering results. Therefore it employs a decay value for each discovered cluster to determine when to remove old clusters and retain recent ones. In addition, the proposed algorithm is efficient as it scans the data streams once and also it is considered an Any-time algorithm since the clustering results of frequent subsequences are readily available at the end of every window.

Finding frequent subsequences, or clusters of subsequences, can be used in many applications. For example, Network monitoring to discover common usage patterns, exploring common stocks’ trend in financial markets, which will lead to good prediction of their future behavior, discovering web click patterns on websites would help website administrators in more efficient buffering and pre-fetching of busy web pages and in the placement of advertisements, and finding the load pattern on busy servers would assist system administrators in placing a more efficient load balancing scheme. Applications like the aforementioned ones and the lack of efficient and incremental algorithms for finding frequent subsequences motivated us to do this work.

Although there are many works on mining frequent itemsets over transactional data streams, little is done on mining frequent subsequences over streaming real-valued data. Also most of the works dealt with single a data stream, while the proposed algorithm deals with multiple data streams. The main contributions of this paper are:

- The proposed algorithm is incremental because clustering results of a current window is built on results of previous windows and also it employs a decay value to remove old frequent subsequences and retain the most recent frequent ones.
- The proposed algorithm is any-time algorithm since the clustering results of frequent subsequences are readily available at the end of every window.
- The proposed algorithm is an exact algorithm since no approximation for the data is used.
- The proposed algorithm is designed to be executed in parallel for multiple data streams.

The rest of this paper is organized as follows. Section 2 discusses the related work. In Section 3, we present some background information and formally define the problem and propose a solution. The proposed algorithm is presented in Section 4. In Section 5, we discuss the results of our experiments and show the feasibility of our approach. Finally, we conclude the paper in Section 6.

2. RELATED WORK

Finding frequent subsequences in data streams has received the attention of many researchers in the data mining community. One of the early works in designing incremental algorithms for mining frequent itemsets in data streams is the one presented by Manku and Motwani (2002).
They introduced the lossy counting algorithm, which produces the frequent itemsets over the entire history of the data stream. The lossy counting algorithm inspired a number of researchers. For example, Li, Lee, and Shan (2004) used the estimation method for the support value in a lossy counting algorithm to produce a single pass algorithm for frequent itemsets mining. Their algorithm uses the prefix tree as a compact data structure for the frequent subsequences. Wong and Fu (2006) dealt with the problem of mining top \( K \) frequent itemsets by designing an algorithm based on a lossy counting method. Their algorithm would let the user specify the size of the result instead of specifying the support threshold.

Using the prefix tree as a data structure for maintaining the frequent itemsets in transactional data, has attracted many researchers. In Chang and Lee (2003), the authors used the prefix tree with a mechanism that gives less weight for old transactions. Jin and Agrawal (2005) developed a compact data structure for storing the frequent itemsets. This data structure benefited from the prefix tree which gives a compact representation for the frequent itemsets, and benefited from the hash table which allows the deletion of the itemsets when they no longer needed. Other researchers, such as Mozafari, Thakkar, and Zaniolo (2008), were interested in using the FP-tree as the data structure to maintain the frequent itemsets. The FP-tree is an extension form of the prefix tree.

A graph structure for maintain the frequent itemsets was proposed by Naganthan and Dhanaseelan (2008). Li, Ho, and Lee (2009) proposed an algorithm for mining frequent closed itemsets using transaction sensitive sliding window. In a transaction sensitive sliding window model, the data captured in the window for processing is decided by a completed transaction. Raissi, Poncelet, and Teisseire (2007) were interested in finding the maximal frequent itemsets. Jiang (2006) proposed an algorithm with in-memory data structure for mining closed frequent itemset over data streams. An algorithm for mining the temporal high utility itemsets was introduced by Chu, Tseng, and Liang (2008). In Yu, Chong, Lu, Zhang, and Zhou (2006), the authors used a false-negative approach instead of a false-positive one to reduce the amount of consumed memory.

Lin, Hsueh, and Hwang (2008) claimed that using fix support threshold is not realistic and they developed an algorithm that would allow the user to change the support threshold after evaluating the produced results. Silvestri and Orlando (2007) introduced an algorithm that uses an interpolation method to infer the support of some itemsets that were infrequent in the past time windows, but are frequent in the current one. They used this method because keeping counter for each item would be very costly in term of memory consumption. Chu, Tseng, and Liang (2009) were interested in keeping track of the itemsets that are non-frequent in the current sliding window but may be frequent in the coming ones.

As mining data streams consumes a lot of the computational resources like the CPU capacity and memory, a number of researchers paid attention to this problem. Dang, Ng, Ong and Lee (2007) used load shedding to automatically shed the unprocessed data when an overloaded CPU case is discovered. To save the memory used in mining the frequent itemsets, Li and Lee (2009) proposed a bit-sequence representation for the items.

All the works are interested in finding frequent itemsets where the order of items is not important (Ashrafi, Taniar, & Smith, 2007; Raahemi & Mumtaz, 2010). On the other hand, a number of researchers (Laur, Symphor, Nock, & Poncelet, 2007; Ashrafi, Taniar, & Smith, 2007; Welzker, Zimmermann, & Bauckhage, 2010) focused on the problem of mining sequential subsequence over data streams where the order of items is important. To reduce the number of discovered subsequences, Raissi, Poncelet, and Teisseire (2006) found the maximal sequential subsequences over data streams. Instead of using the support value, the researchers in Barouni-Ebrahimi and Ghorbani (2007) developed a frequency rate equation. If the frequency rate
of a sequence is greater than the frequency rate specified by the user, then the sequence is considered to be frequent.

The aforementioned works are dealing with a single stream. Sun, Papadimitriou, and Faloutsos (2006) are interested in finding frequent subsequences over multiple data streams. Otey, Parthasarathy, Wang, and Veloso (2004) designed parallel methods for mining the frequent itemsets for distributed data streams; they considered the communication overhead in their algorithms. Mining sequential subsequences over multiple data streams were studied by Chen, Wu, and Zhu (2005). They incorporated prior knowledge about the data distribution to improve the mining process. Most of the works above dealt with transactional streaming data. However, we propose to find frequent subsequences in multiple streams with continuous data values.

3. BACKGROUND AND NOTATION

Before formally defining the problem to be solved in this paper and its proposed solution, we briefly present a background on data streams.

3.1. Data Streams

A data stream is a collection of ordered items that arrive in a continuous manner, with high arrival rate, and has unbounded size. These characteristics raise many issues when designing algorithms for data streams (Jiang & Gruenwald, 2006). One of these issues is the need for incremental algorithms for mining data streams. Thus, there is no need to recompute the mining result as new data arrives; instead the result should be updated based on the old mining results.

Another issue when processing data streams is the period of the data that is most applicable for the application. This question is answered by choosing the right window model. According to Zhu and Shasha (2002) there are three kinds of window models to use when dealing with data streams. These models are the landmark window model, the damped window model, and the sliding window model. The choice of the window model decides the period of time that the data will be taken from. In the landmark window model, data can be taken from any time point called landmark till the current point. When using the damped window model, the older data is given less weight than the newer one, and thus gradually the effect of older data is decreased. The sliding window model is used when the interest is only in the current data.

The high arrival (incoming) rate of a data stream challenges the resources available to process it, like the CPU, memory, etc. That is the higher the arrival rate of the data, the faster the consumption of memory (Jiang & Gruenwald, 2006). According to Gaber, Krishnaswamy, and Zaslavsky (2003) the high incoming rate problem can be solved through two solutions. First one is the input and output rate adaption. In this solution, the input data stream is adapted to the available resources by selecting subset of it instead of processing it as a whole. Different techniques like sampling, aggregation, and load shedding can be used to select the subset of the stream to be processed. To adapt to the output rate, measurements such as available memory, time, and data rate should be taken into consideration. The second solution is to use approximate algorithms. These algorithms only have one look at the data, and they produce the results with some margin of error.

The nature of data streams makes it necessary to enforce some restrictions when designing data mining algorithms (Bhatnagar, Kaur, & Mignet, 2009; Golfarelli & Rizzi, 2009). Since there is a huge amount of data, some researchers chose to represent the data with summary information. Many techniques are used to summarize the data. These techniques include Wavelets, Discrete Fourier Transform, Piecewise Linear Representation, etc. Since the summarization techniques produce approximations of the original data, the solutions produced by these techniques are approximate ones. Thus, in this paper we propose an exact solution based on the original data.
3.2. Notations and Definitions

In this section, we introduce the definitions and notation used in explaining the purposed algorithm for finding frequent subsequences in multiple data streams. A data stream, $S$, is formally defined as follows:

**Definition 1**: A data stream, $S$, is an unbounded sequence of items arriving at fixed interval. $S = s_0, s_1, s_2, \ldots, s_\infty$.

Each item, $s_i$, of $S$ is a real valued number. The proposed algorithm uses the Sliding Window model to buffer the incoming data items of a data stream. When a window is full, the proposed algorithm processes the items in the window, $w$, to find the frequent subsequences.

**Definition 2**: A window, $w$, is a subset of the stream from time $t$ to $t + w - 1$, where $w = s_t, s_{t+1}, \ldots, s_{t+w-1}$.

A subsequence is a subset of a data stream within a window. Each subsequence has a length $l$, which is between a minimum length $h$ and a maximum length $m$.

**Definition 3**: A subsequence, $s$, is a subset of the window of length $l$, where $h \leq l \leq m$.

The proposed algorithm finds the frequent subsequences by means of clustering the subsequences. Two subsequences are placed in the same cluster if they are neighbors in the subsequences space.

**Definition 4**: A subsequence, $s_n$, is considered a neighbor of another subsequence $s_j$ starting at time $t$ and has length $l$, if the distance between them is $d(s_n, s_j) \leq \Theta$.

A subsequence is considered frequent if the number of its neighbors, $\eta$, in a cluster is greater or equal to some threshold, $\tau$. Otherwise, the subsequence is considered non-frequent and thus ignored.

**Definition 5**: A frequent subsequence, $FS$, is considered frequent in a window if the number of its neighbors, $\eta \geq \tau$.

Table 1 lists the symbols used in the proposed algorithm.

3.3. Problem Definition

In this paper, we address the problem of finding frequent subsequences in multiple data streams. We assume that these data streams are synchronized, that is they have the same arrival rate. A stream has unbounded size, and consists of real numbers that arrive in a specific rate. Formally, given a set of input data streams $\xi = S_1, S_2, \ldots, S_p$ our algorithm finds subsequences that are frequent, ($FS$), over all data streams.

Due the nature of data streams, the proposed algorithms should be:

- Incremental, that is the algorithms computes the current results based on previously computed ones without the need to recompute the result from the whole history of a data stream.
- Efficient, by scanning the data streams only once.
- Any time, that is the results can be readily retrieved after every window without having to recompute it on demand.

3.4. Proposed Solution

Finding frequent subsequences, $FS$s, is challenging because of the nature of the data streams. Due to the unbounded nature of the data streams, we employ a sliding window model to retrieve a set of $w$ values from each data stream. Then, we find $FS$s by clustering the subsequences of data streams. A subsequence $s$ in a window is considered frequent if it has enough number of neighbors ($\eta \geq \tau$). That is if the number of subsequences in a cluster is greater or equal than $\tau$. A subsequence $s$ enters the neighborhood of another subsequence $s_j$ if the distance between the two is less than a threshold, $d(s, s_j) < \Theta$. Keep the clusters up-to-date, we employ a decay
value, $\delta$, with each subsequence, to be able to remove old subsequences. The $\delta$ variable makes the algorithm incremental, by only keeping the most recent FSs, which are in subsequent windows without the need to recompute FSs from the whole history of data streams.

4. DISCOVERING FREQUENT SUBSEQUENCES

In this paper, we address the problem of finding FSs in multiple data streams. We assume that the data streams are synchronized and elements of data streams arrive sequentially at a specified arrival rate.

4.1. Algorithms

The main algorithm for finding FSs starts when a window becomes full. Thus, our solution consists of two algorithms. The first algorithm, called BufferDataStreams() (Algorithm 1), which collects the elements of a data stream, $S$, till the buffer (equivalent to one window, $w$) becomes full (see line 3). When $w$ is full, the FindFrequentSubsequences() algorithm (Algorithm 2) is called for each stream to process the current window (line 3-5). At the end of BufferDataStreams() algorithm, the Lists of all data streams are added into one global link list, called Glist, that contains all the FSs. Thus, Algorithm 1 and Algorithm 2 are applied to every data stream.

In this paper we applied the idea of monotonicity property used by the Apriori algorithm (Tan, Steinbach, & Kumar, 2005) on data streams to reduce the number of frequency computation of subsequences. That is, the algorithm omits the frequency computation of subsequences that have non-frequent subsets. This leads to less invocation to the Euclidean distance function, and as a result less execution time. Figure 1 shows the effect of applying the monotonicity property in our algorithm. If subsequence $s_{t, l_{\text{min}} + 1}$ is not frequent, then all its superset subsequences are not frequent and thus are not processed.

The approach we are using to find the FSs is a clustering approach, so while we are explaining the FFS algorithm (Algorithm 2), we may use the terms frequent subsequence and cluster interchangeably. A cluster is a frequent subsequence with its neighbors. The first subsequence arriving into the cluster is considered
the representative of the cluster. The algorithm begins by setting the number of neighbors, $\eta$, for every subsequence to 0, so that the value of $\eta$ of every subsequence is not affected by the results of the previous window (lines 1-3). The algorithm starts extracting the subsequences of length $l$, where $h \leq l \leq m$. We discuss how these lengths are determined later in this section. Then, the FFS algorithm checks the subsets of every subsequence $s_{t, l}$ of time $t_{\text{arrive}}$ and length $l$, if a subset is frequent, it finds the subsequence in the list of frequent subsequences, $List$, that has the minimum distance with $s_{t, l}$. Non-frequent subsequences are ignored. The minimum Euclidean distance is stored in $s_{t, l\text{Min}}$ (line 6-7). Only non-trivial matches (Keogh & Lin, 2005) are considered.

Definition 6: A Trivial Match: A trivial match of a subsequence $s_{t, l}$ of time $t_{\text{arrive}}$ and length $l$, is the one that overlaps with it.

Figure 2 gives an example of trivial matches of a subsequence. A subsequence under consideration is bounded by a bold-line rectangle. Two overlapping subsequences are bounded by thin-line rectangles (one on the left and another on the right). Both of these overlapping subsequences are considered trivial matches of the bold-line subsequence.
The FSS algorithm, line 8 checks if the $s_{t,lMin}$ distance from $s_{t,l}$ is less than the threshold, $\Theta$; if so, it then checks if the extracted subsequence is a trivial match with the last neighbor of the subsequence in the $s_{t,lMin}$ cluster (line 9). If the last neighbor subsequence, $LN$, of the subsequence $s_{t,lMin}$ and the new subsequence, $s_{t,l}$, makes a trivial match, then choose the one with the smaller distance from $s_{t,lMin}$ (lines 10-11). Otherwise $s_{t,l}$ is placed in the $s_{t,lMin}$ cluster and $\eta$ of this cluster is incremented by one (lines 14-15). If the $s_{t,lMin}$ distance from $s_{t,l}$ is not less than $\Theta$, $s_{t,l}$ starts a new cluster (line 18).

The number of neighbors $\eta$ of every subsequence of length $l$ in $List$ is checked, before moving to the subsequences with larger lengths to apply the monotonicity property. That is if $\eta$ of the subsequence $s_{t,l}$ is less than the minimum number of neighbors, $\eta < \tau$, then this subsequence is considered non-frequent in the current window and thus the algorithm decrements the decay value $\delta$ of the cluster, or $FS$, (lines 22-24). If $\delta$ of a subsequence reaches -1, this means that the subsequence is becoming non-frequent for the current window (this cluster, or subsequence, has decayed and thus considered non-frequent in the current window), and it is removed from $List$ (lines 25-26). Otherwise if $\eta \geq \tau$, $\delta$ of a subsequence is incremented, which means that the subsequence is frequent for the current window (line 29). We used JAVA threads to execute the FFS algorithm in parallel for each stream.

We explain some of the parameters affecting the number of produced frequent subsequences. These parameters are the threshold value $\Theta$, the support value $\tau$, the decay value $\delta$, and the subsequence length, $l$.

**The Threshold Value ($\Theta$)**

Choosing the threshold value, which decides if a subsequence is a neighbor to another subsequence, has a great impact on the produced $FS$s. A very small threshold value may result in too many false negatives. On the other hand, a very large $\Theta$ may result in too many false positives. The threshold used to decide if a subsequence is in the neighborhood of another subsequence is equal to (derived from the Euclidean distance). This means that under the Euclidean distance, for a subsequence of length $l$, the maximum difference allowed between the values of two correspondence elements in neighboring subsequences is equal to $C$, where $C$ is a user defined parameter. Thus, $\Theta$ is computed in terms of $l$ and $C$ as explained above.

**The Support Threshold ($\tau$)**

The support threshold, which is a user defined parameter, decides whether a subsequence has enough number of neighbors to be considered frequent in the current window.

**The Decay Value ($\delta$)**

Because the data is coming in a streaming fashion, we need to check if a cluster, or $FS$, is still frequent in each window. The $\delta$ decides whether...
Algorithm 2. FindFrequentSubsequence (FFS)

**Input:** The current window buffer, List contains the frequent subsequences computed from the previous window, and threshold value $\Theta$ to decide if a subsequence is a neighbor to another subsequence.

**Output:** List contains the frequent subsequences for the current window

1. for every($s_{i,j}$ in List) do
2.   Set the frequency count ($fc$) of every subsequence in the list to 0
3. end for
4. for every ($l$ from $h$ to $m$) do
5.   for every ($t$ from 1 to $w$) do
6.     If ($s_{i,t-1}$ is frequent) then
7.       $s_{i,lMin} = findMinimumDistance(s_{i,t}, List)$
8.       //trivial matches not considered
9.     if (d($s_{i,lMin}$, $s_{i,t}$) <= $\Theta$) then
10.       If ($s_{i,lMin}$ last neighbor, LN, is trivial match of $s_{i,t}$) then
11.         if (d($s_{i,lMin}$, $s_{i,t}$) < d($s_{i,lMin}$, LN)) then
12.           Replace LN with $s_{i,t}$ in $s_{i,lMin}$ cluster
13.         end if
14.       else
15.         $s_{i,t}$ belongs to $s_{i,lMin}$ cluster
16.       end if
17.     else
18.       $s_{i,t}$ is a new cluster
19.     end if
20.   end if
21. end for
22. for every(subsequence of length $l$ in List) do
23.   if ($\eta < \tau$) then
24.     decrement the $\delta$ of the cluster
25.     if ($\delta == -1$) then
26.       Remove $s_{i,j}$ from List
27.     end if
28.   else
29.     increment the $\delta$ of the cluster
30.   end if
31. end for
a subsequence is frequent in the current window or not. If number of neighbors, \( \eta \geq \tau \), then \( \delta \) is increased by 1. Otherwise the \( \delta \) is decreased by 1. When \( \delta \) reaches -1 the subsequence is removed from the list of frequent subsequences. This \( \delta \) parameter makes the algorithm incremental, by only keeping the most recent FSs, which are in subsequent windows without the need to recompute FSs from the whole history of data streams.

Figure 3 presents an example to show the affect of \( \delta \). Assuming \( \tau \) is set to 3, a) At the beginning of the algorithm, three subsequences \( s_1, s_2 \) and \( s_3 \) forming three clusters were found and their decay values is initialized to, \( \delta = 0 \). b) After finding 3, 3 and 1 neighbors for \( s_1, s_2 \) and \( s_3 \), respectively, the algorithm updates \( \delta.s_1=1 \), and \( \delta.s_2=1 \) since they are frequent (\( \tau \geq 3 \)) in this window and so, \( \delta.s_3=-1 \) since it is not frequent and thus removed. c) \( s_1 \) has no new neighbors, therefore the value of \( \delta \) is decreased by 1, while \( s_2 \) has three new neighbors and thus its \( \delta \) is increased by 1. d) \( s_1 \), has only one new neighbor in this window, so it is considered not frequent and its \( \delta \) is decreased to -1 and thus removed. Since \( s_2 \) has no neighbors in this window, its \( \delta \) value is decreased by 1. e) Again \( s_2 \) has no new neighbors, so its \( \delta \) is decreased by 1 to be 0, but this cluster will remain in the next window (window 5), because its \( \delta \) didn’t reach -1 yet.

**Definition 7:** Frequent Subsequence: A subsequence is called frequent in a window, if \( \delta \) is greater than zero, \( \delta \geq 0 \).

In Section 3.2 we defined a frequent subsequence as the one that has enough neighbors. However in definition 7 we redefine the frequent subsequence in relation with the decay value. The two definitions are not contradicting, but rather they complement each other. A subsequence will not reach a decay value greater than 0 unless it had enough number of neighbors in at least one window. If in the next windows it didn’t have enough number of neighbors but \( \delta \) didn’t reach -1, then it will be kept in the List of FSs.

**The Subsequence Length (l)**

\( h \) is the minimum subsequence length, and we left it as a parameter to be specified by the user. \( m \) is the maximum subsequence length and it is equal to \( w / (\tau + 1) \). This formula is derived from the requirement of our algorithm that for a subsequence to be frequent in a window of size \( w \) it should have at least \( \tau \) neighbors. So to make sure that all the subsequences of length \( m \) have enough number of neighbors in a window of size \( w \) we restricted the maximum subsequence length to the above formula.
4.2. Data Structure

One of the important issues in designing the algorithm is the choice of the data structure. In the first stage, each stream submits the FSs results to a linked list. So, each stream has its own local list \( \text{List} \). Each subsequence in \( \text{List} \) has linked list of neighbors. After each window the \( \text{List}s \) are submitted to a global linked list \( \text{GList} \). The subsequences in the global list are sorted in ascending order based on time. Our choice for the data structure was a linked list for both stages: local lists and global lists. This choice is justified because there are a lot of insertion and deletion to these lists during and after each window. Furthermore, the size of the results is not known in advance. Thus, for these reasons we chose a linked list structure to store the FSs.

4.3. Sliding Window Model

In Section 3.1, we discussed the window models used in data streams mining algorithm. One of the window models is the sliding window model which we are using in our algorithm. The sliding window model gives preference to the most recent data. But, as our algorithm is incremental, the FSs results of the current window are built on results of previous windows. The window size, \( uw \), is a user parameter, however the FFS algorithm uses a buffer of size \( w \). We extend the size of the window specified by the user to \( w = uw + (m-1) \). That is the buffer of \( \text{window}_i \) should includes the \( uw \) elements of \( \text{window}_i \) plus \( m-I \) elements from \( \text{window}_{i+1} \) before the buffer of \( \text{window}_i \) is processed by the FFS algorithm. These \( m-I \) extra elements will make sure that the last element in \( \text{window}_i \) has subsequences of all possible lengths.

Figure 4 shows the time in which the result of \( \text{window}_i \) is reported. Our algorithm is designed to be efficient enough to report the result of \( \text{window}_i \) before the next window gets all its elements. This is because data streams are continuous in nature and thus the proposed algorithms try to avoid the drop of new elements in the next window if they arrive before the \( \text{window}_i \) has not been completely processed. Thus, one of the goals of the proposed algorithms is to efficiently process online data streams to find FSs.

4.4. Complexity Analysis

The algorithm starts by passing through every subsequence in \( \text{List} \) to initialize its \( \eta \) to zero, so this operation is performed in \( O(L) \) where \( L \) is the \( \text{List} \) size. The subsequences are extracted from the window of size \( w \) starting from the subsequences with minimum length \( h \) to the subsequences with maximum length \( m \). Thus, the number of possible extracted subsequences is \( (m-h+1)w \). Each extracted subsequence is then compared to the subsequences in the \( \text{List} \) to find the closest one, that is the one with the minimum Euclidean distance, and each subsequence requires \( O(mL) \) to compute the distance (Actually it takes \( l \) instead of \( m \), and \( l \) varies from \( h \) to \( m \), but as we consider the worst case, we assumed that all the subsequences have the maximum length \( m \)). Before extracting larger subsequences, every subsequence of length \( l \) is checked for its frequency for the current window (Algorithm 2, line 28-38). This operation requires \( O(L^*(m-h+1)) \). The aforementioned operations are the main tasks performed on the extracted subsequences. There are some other operations that take constant time per extracted subsequence. We ignored these operations in the computation of the complexity.

Therefore, the complexity of the algorithm is: \( O(L) + O(((m-h+1)*w)\text{*(m*L))} + O(L^*(m-h+1)) = O(((m-h+1)*w)\text{*(m*L)})) \). Considering the worst length of subsequence, that is when \( h = 1 \), the complexity is \( O(m^2wL) \).

5. EXPERIMENTAL RESULTS

To evaluate the performance of the proposed algorithm, we conducted a set of experiments. In these experiments, we evaluate the purity of clustering, and the effect of the data arrival rate \( r \) (data incoming rate) on the performance under
different parameters. By evaluating $r$, we are measuring the speed performance of the system.

We implemented the dataset generator, and the algorithms in JAVA, JDK 6. We run the experiments on a PC running Windows XP, Intel Core 2 CPU 2GHz, and 2GB of RAM.

5.1. Description of the Dataset

To evaluate the proposed algorithms, we developed a data generator that produces multiple data streams. Each stream consists of real numbers. These numbers are generated based on some prototypes. A prototype is created by selecting random numbers falling in the range $[A, B]$. The prototypes can overlap if the overlapping-value $\alpha > 0\%$. Figure 5 explains the effect of $\alpha$ in the generation of the prototypes. In Figure 5, all the points falling in the range $[A, B]$, and the range $[C, D]$, are represented on a line. In case 1, where $\alpha = 0\%$, there is no overlap of values between the two ranges. In the second case, $\alpha = 30\%$, which means that 30% of the range of values $[A, B]$ is shared with 30% of the range of values $[C, D]$. The data streams of a cluster, which is represented by a prototype, are generated by tweaking the values of the prototype by adding a random number in the range of $[0..\nu]$, where $\nu$ is a parameter to simulate the density of clusters.

5.2. Evaluating FFS Algorithm

We performed two sets of experiments to evaluate the performance of FFS algorithm. The first set of experiments focuses on evaluating the effect of two parameters on the purity of clustering. These two parameters are the overlap value $\alpha$, which is the overlap between the generated values of the data streams in a cluster, and density $C$ of a cluster, which is the amount of change allowed between the corresponding values of a cluster’s members. The second set of experiments evaluates the effect of the arrival rate on the performance of the algorithm under different parameters.

5.2.1. Clustering Purity

We compute the purity of the clusters produced under different parameters. The purity of the clustering algorithm is a supervised measure for validating the clusters (Tan, Steinbach, & Kumar, 2005). We compute the purity of a cluster as follows:

$$ purity\text{ of cluster}(A) = \frac{\text{#correct neighbors of cluster } A}{\text{Total #neighbors for all clusters}} $$

Thus, the purity of a cluster is computed as the number of correct neighbors of the cluster divided by the total number of neighbors in all clusters. However, the purity of clustering the subsequences in a window, $w$, is the sum of the purities of the clusters that exist in the window. If there are $k$ clusters in $w$, the purity of clustering in $w$ is computed as follows:

$$ purity\text{ of clustering } w = \sum_{i=1}^{k} purity\text{ of cluster}(i) $$
To compute the purity of clustering a stream of subsequences over \( n \) windows, the purity would be the average of purities of clustering the \( n \) windows.

\[
purity \text{ of a stream} = \frac{1}{n} \sum_{i=1}^{n} purity \text{ of clustering } w_i
\]

In our experiments, the purity results are the average of running 100 streams under the same parameters and for each stream we computed the average of running 5 windows.

From the above formula it can be seen that the purity of a cluster is affected by how many subsequences are assigned to the wrong clusters.

### Purity vs. Overlap Value \( \alpha \) Between Clusters

In this experiment, we evaluated the effect of the overlap value \( \alpha \) between clusters on the purity, while fixing the other parameters as follows: \( w \) to 120, \( \tau = 3 \), \( C = 4 \) and the number of clusters = 4. From Figure 6 we notice that the larger the value of \( \alpha \), the less the purity we achieve, which agrees with our expectation. This is because an overlap between clusters causes some member subsequences of different clusters to have similar values. Thus, these overlaps result in assignments of the subsequences to the wrong clusters. That is the number of false positives and true negatives increase in the produced clusters.

### Purity vs. Cluster Density \( C \)

Figure 7 shows the effect of the cluster density \( C \), which is the amount of change allowed between the corresponding values of the members of a cluster, on the purity. While computing the purity vs. \( C \), we fixed the other parameters as follows: \( w \) to 120, \( \tau = 3 \), \( \alpha = 60\% \) and the number of clusters = 4. As the value of \( C \) increases the purity decreases. This is due to the fact that increasing \( C \) makes the cluster sparser and thus allows some far subsequences, which are not necessarily members of the current cluster, to join the cluster. These far subsequences may belong to other clusters.

### 5.2.2. Performance Experiments

The next sets of experiments are conducted to evaluate the speed performance of the FFS algorithm. To evaluate the speed performance we measure the value of arrival rate \( r \) that our algorithm can cope with under different parameters. In other words, we are measuring to what speed extent of the data incoming rate the proposed algorithm can cope with under different parameters. The value of \( r \) is affected by the amount of time needed to process a window and produce FSs.

If a new window is ready to be processed while the FFS algorithm is still processing the previous one the new window will be dropped, which causes a loss of data. The processing time is affected by the number of subsequences needed to be processed. In turn, the number of subsequences to be processed is affected by window size \( w \), number of prototypes, the support threshold \( \tau \) and the minimum subsequence length \( h \). Also, we compared the performance of two versions of the FFS algorithm: one using the Euclidean distance and the other using the Uniform Scaling (US) distance(Yankov,
Arrival Rate vs. Window Size

In this experiment we evaluate the effect of the window size $w$ on the arrival rate $r$, while fixing the other parameters as follows: $\alpha = 50\%$, $\tau = 3$, $C = 1.5$ and the number of prototypes = 4. We notice from Figure 8 that the larger the size of the window, the more time required for processing the window and thus the slower the required $r$. This is to avoid any drop of new data. Furthermore, the larger the size of the window, the larger the number of subsequences needed to be processed.

Arrival Rate vs. Number of Prototypes

The number of the prototypes represents the distribution of the data. This means that, as the number of the prototypes decreases, then over a fixed $w$, the number of subsequences that
belong to the same prototype in $w$ increases, and vice versa. Thus, decreasing the number of prototypes would result in forming clusters with more member subsequences and thus more frequent subsequences to be found. In this experiment we measure $r$ for different values of the number of prototypes (Figure 9), while fixing other parameters as follows: $\alpha = 50\%$, $\tau = 3$, $C = 1.5$ and $w = 120$. We compare the performance of the FFS algorithm when applying the monotonicity property and when not using it. When using the monotonicity property, as the number of prototypes increases, the number of FSs decreases. In addition, the use of the monotonicity property avoids processing the non-frequent subsequences, which results in reduction of the time required to process the window. In contrast, without using monotonicity property, $r$ that the FSS algorithm can cope with increases when the number of processed subsequences increases. This shows that using the monotonicity property allows the FFS algorithm to cope with faster arrival rates of streaming data.

**Arrival Rate vs. Support Threshold**

In this experiment we evaluate the effect of the support threshold $\tau$ on the arrival rate $r$, while fixing the other parameters $\alpha = 50\%$, $C = 1.5$, $w = 120$ and the number of prototype = 4. These experiments are conducted on two versions of the FFS algorithm: one using the monotonicity property and the other one without using the monotonicity property. Figure 10 shows that the general trend of both version of the FFS algorithm is that as $\tau$ increases, the proposed algorithm can cope with faster arrival rates. Again this is because the smaller $\tau$ we have, the more subsequences will satisfy the threshold condition and thus the larger number of subsequences to be processed. We notice from Figure 10 that using the monotonicity property makes FFS algorithm perform faster as the number of processed subsequences becomes less due to the pruning process of monotonicity property.

**Arrival Rate vs. Minimum Subsequence Length**

For a fixed size of $w$, the maximum subsequence length $m$ is fixed and depends on $w$ (see Section 4.1). We conducted a set of experiments to evaluate the effect of varying the size of minimum subsequence length $h$ on the arrival rate, while fixing $\alpha = 50\%$, $C = 1.5$, $w = 120$, $\tau = 3$ and the number of prototype = 4. Figure 11 shows that as $h$ increases, $r$ that the algorithm can cope with becomes faster. Since the num-
ber of subsequences that are extracted from a window is equal to \( m-h+1 \), it is obvious that as \( h \) increases the number of subsequences to be processed decreases. As a result, it is expected that the required \( r \) is faster.

Euclidean distance vs. Uniform Scaling distance

The arrival rate \( r \) that the FFS algorithm can cope with is calculated for different values of the support threshold \( \tau \), while fixing \( \alpha = 50\% \), \( C = 1.5 \), \( w = 120 \), \( \tau = 3 \), \( h = 3 \), and the number of prototype = 4. Figure 12 shows the results of a comparison between the performances of two versions of the FFS algorithm: one that uses the Euclidean distance that finds the distance between two subsequences with similar lengths, with another version that uses the Uniform Scaling, US, distance (Yankov, Keogh, Medina, Chiu, & Zordan, 2007). The US distance finds the distance between two subsequences with variable lengths. The results shows that the general trend is that when \( \tau \) is decreased the FFS algorithm can cope with slower \( r \). However, in the US version as \( \tau \) decreases, the \( r \) that the algorithm can handle degrades quickly (becomes
very slow), which means that the US version is not suitable for fast data streams.

6. CONCLUSION

We presented the FFS algorithm, which is incremental, any-time and exact algorithm, to find FSs in multiple data streams. The FFS algorithm benefits from the use of the monotonicity property to reduce the number of processed subsequences. By this property, subsequences, which have non-frequent subsets, are not considered frequent and thus ignored during the process of finding FSs. The FFS algorithm employs a decay value to timeout and removes older subsequences as they are considered non-frequent subsequences in the current window. In addition, the FFS algorithm is any-time algorithm as FSs are readily available at the end of every window of the data stream and it
is considered an exact algorithm since it works on the original data (no approximation).

We conducted extensive experiments to evaluate the FFS algorithm and show its feasibility. We evaluated the purity of clustering subsequences under different parameters. We noticed that the distribution of the data and the value of threshold have an impact on the purity of the clusters. We evaluated the arrival rate that the algorithm can handle under different parameters. We experimented with two versions of our proposed FFS algorithm, one with using monotonicity property, and one without using it. The one with using monotonicity property showed its performance superiority over the other one. Also we test our algorithm under two distance measures, the Euclidean distance and the Uniform Scaling distance and showed that the Euclidean distance version could handle faster arrival rates of data streams and thus more suitable for online applications.

REFERENCES


Reem Al-Mulla received her BSc degree in Computer Science from the University of Sharjah, UAE, in 2004, and the MSc degree in Computer Science from University of Sharjah, UAE, in 2010. She has been working as a lecturer in the Department of Computer Science, University of Sharjah, since 2010. Her research interests include databases, data mining and data stream management.

Zaher Al Aghbari received his B.Sc degree from the Florida Institute of Technology, Melbourne, USA, in 1987, and the MSc and PhD degrees in computer science from Kyushu University, Fukuoka, Japan, in 1998 and 2001, respectively. He was with at the Department of Intelligent Systems, Kyushu University, Japan, from 2001 to 2003. From 2003 to 2008, he was with the Computer Science Department, University of Sharjah, UAE. Since 2008 he has been the Chairperson for the Department of Computer Science, University of Sharjah, UAE. His research interests include multimedia databases, data mining, multidimensional indexing, distributed indexing, data streams management, image/video semantic representation and classification, and Arabic handwritten text retrieval.