A Fast Algorithm For Finding Frequent Episodes In Event Streams

Srivatsan Laxman*
Microsoft Research Labs India
Bangalore
slaxman@microsoft.com

P. S. Sastry
Indian Institute of Science
Bangalore
sastry@ee.iisc.ernet.in

K. P. Unnikrishnan
General Motors R & D Center
Warren
k.unnikrishnan@gm.com

ABSTRACT

Frequent episode discovery is a popular framework for mining data available as a long sequence of events. An episode is essentially a short ordered sequence of event types and the frequency of an episode is some suitable measure of how often the episode occurs in the data sequence. Recently, we proposed a new frequency measure for episodes based on the notion of non-overlapped occurrences of episodes in the event sequence, and showed that, such a definition, in addition to yielding computationally efficient algorithms, has some important theoretical properties in connecting frequent episode discovery with HMM learning. This paper presents some new algorithms for frequent episode discovery under this non-overlapped occurrences-based frequency definition. The algorithms presented here are better (by a factor of N, where N denotes the size of episodes being discovered) in terms of both time and space complexities when compared to existing methods for frequent episode discovery. We show through some simulation experiments, that our algorithms are very efficient. The new algorithms presented here have arguably the least possible orders of space and time complexities for the task of frequent episode discovery.

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Database Management—*Data mining*

General Terms

Algorithms

Keywords

Event streams, frequent episodes, temporal data mining, non-overlapped occurrences

*This work was carried out when Srivatsan Laxman was at Department of Electrical Engineering, Indian Institute of Science, Bangalore.

KDD'07, August 12–15, 2007, San Jose, California, USA.

1. INTRODUCTION

Frequent episode discovery [8] is a popular framework for temporal data mining. The framework is applicable on data available as a single long sequence of ordered pairs, (E_i, t_i), which are called as events. In each event, (E_i, t_i), E_i is referred to as an event type (which takes values from a finite alphabet, $\mathcal{E}$) and t_i is the time of occurrence of the event. The data, which is also referred to as an event sequence (or an event stream), is ordered according the times of occurrence. There are many applications where data appears in this form, e.g., alarm sequences in telecom networks [8], web navigation logs [1, 2], time-stamped fault report logs from manufacturing plants [6, 7], etc.

The framework of frequent episode discovery [8] can be used to mine temporal patterns from event streams. The temporal patterns, referred to as *episodes*, are essentially small, (partially) ordered collections of event types. For example, $(A \rightarrow B \rightarrow C)$ denotes a temporal pattern where an event type A, is followed (some time later) by a B and a C, in that order. When events of appropriate types appear in the data sequence, in the same order as in the episode, these events are said to constitute an occurrence of the episode. For example, in the data sequence $\langle (A,1),\ (D,2),\ (E,4),\ (B,5),\ (D,6),\ (C,10)\rangle$, the episode $(A \rightarrow B \rightarrow C)$ occurs once. An episode is considered interesting if it occurs "often enough" in the data. Stated informally, the framework of frequent episodes is concerned with the discovery of all episodes that occur often in the data. To do this, we need to define a frequency measure for episodes in the data. The data mining task is to find all episodes whose frequencies exceed a user-defined threshold. This paper presents a new and very efficient algorithm for frequent episode discovery.

The framework of discovering frequent episodes in event streams was introduced by Mannila, et al. [8]. They define the frequency of an episode as the number of windows (of prefixed width) on the time axis in each of which the episode occurs at least once. They propose a counting algorithm using finite state automata for obtaining the frequencies of a set of candidate episodes. The worst case time complexity of the algorithm is linear in the total time spanned by the event stream, the size of episodes and the number of candidates. The space needed by the algorithm is also linear in the size of episodes and the number of candidates. Some extensions to this windows-based frequency have also been proposed [2, 9]. There have also been some theoretical studies into this framework whereby one can estimate (or bound) the expected frequency of an episode in a data stream of a given length if we have a prior Markov or Bernoulli model for the

data generation process. Thus, if sufficient training data is available, we can first estimate a model for the data source, and then, on new data from the same source, can assess the significance of discovered episodes by comparing the actual frequencies with the expected frequency [3, 4, 10].

Recently, we have proposed [6, 7] a new notion for episode frequency based on the *non-overlapped* occurrences of an episode in the given data sequence. In [6], we have also presented an efficient counting algorithm (based on finite state automata) to obtain the frequencies for a set of candidate episodes. This algorithm has the same order of worst case time and space complexities as the windows-based counting algorithm of [8]. However, through some empirical investigations, it is shown that the non-overlapped occurrences-based algorithm is much more efficient in terms of the actual space and time needed, and that, on some typical data sets, it runs several times faster than the windows-based algorithm. It is also seen that our new frequency definition results in qualitatively similar kinds of frequent episodes being discovered (as in the case of windows-based frequency) and all the prominent correlations in the data come out among the top few frequent episodes under both frequency definitions [6]. Another important advantage of the non-overlapped occurrences count is that it facilitates a formal connection between discovery of frequent episodes and learning of generative models for the data sequence in terms of some specialized family of Hidden Markov Models [6]. This formal connection allows us to assess statistical significance of episodes discovered without needing any prior model estimation step (and thus obviating the need for any separate training data). Our formal connection also allows one to fix a frequency threshold automatically and in empirical studies, this automatic threshold is seen to be quite effective [6]. All this makes the non-overlapped occurrences count an attractive method for applications involving frequent episode discovery from event streams.

In this paper, we present a new algorithm for frequent episode discovery under the frequency count based on non-overlapped occurrences. The algorithm is significantly superior to that proposed in [6] both in terms of time and space complexities. The space complexity is same as number of candidates input to the algorithm and the time complexity is linear in the number candidates and the total number of events in the data stream. Unlike the existing algorithms for frequent episode discovery [6, 8], the time and space complexities do not depend even on the size of episodes being discovered. This is because our algorithm needs only one automaton per episode, while the other algorithms need N automata per episode, where N is the size of (or number of nodes in) the episodes being counted. We believe that our algorithm attains the minimum possible order of worst case time and space complexities for the frequent episode discovery process. Thus, our new algorithm is a very competitive alternative to all existing algorithms for frequent episode discovery.

The paper is organized as follows. Sec. 2 presents a brief overview of the frequent episode discovery framework. Sec. 3 presents the new frequency counting algorithms. We demonstrate the effectiveness and efficiency of our new frequency counting algorithms through some simulations in Sec. 4. In Sec. 5 we present the conclusions.

2. OVERVIEW OF FREQUENT EPISODES MINING FRAMEWORK

Formally, an episode, α, is defined by a triple, $(V_\alpha, \leq_\alpha, g_\alpha)$, where V_α is a collection of nodes, $\leq_\alpha$ is a partial order on V_α and $g_\alpha : V_\alpha \rightarrow \mathcal{E}$ is a map that associates each node in α with an event type from a finite alphabet, $\mathcal{E}$. When $\leq_\alpha$ represents a total order among the nodes of α, the episode is referred to as a serial episode, and when $\leq_\alpha$ is trivial (or empty), the episode is referred to as a parallel episode. Given an event sequence, $\langle (E_1, t_1), \ldots, (E_n, t_n)$, an occurrence of episode $\alpha = (V_\alpha, \leq_\alpha, g_\alpha)$ in this event sequence, is an injective map, $h : V_\alpha \rightarrow \{1, \ldots, n\}$, such that $g_\alpha(v) = E_{h(v)}$ for all $v \in V_\alpha$, and for all $v, w \in V_\alpha$ with $v \leq w$ we have $t_{h(v)} \leq t_{h(w)}$. Finally, an episode β is said to be a *subepisode* of α if all the event types in β appear in α as well, and if the partial order among the event types of β is the same as that for the corresponding event types in α.

As mentioned earlier, the episode's frequency is some measure of how often it occurs in the data. There are many ways to define episode frequency [2, 6, 8]. In the original framework of [8], the frequency of an episode was defined as the number of fixed-width sliding windows over the time axis that each contain an occurrence of the episode. In this paper, we consider the non-overlapped occurrences-based frequency definition proposed in [6]. Two occurrences of an episode are said to be *non-overlapped* if no event corresponding to one occurrence appears in between events corresponding to the other. *Definition 1* given below, formalizes this notion of non-overlapped occurrences, using the notation that was just introduced for episodes and their occurrences in an event stream.

DEFINITION 1. *Consider an N-node episode $\alpha = (V_\alpha, <_\alpha, g_\alpha)$ where $V_\alpha = \{v_1, \ldots, v_N\}$. Two occurrences, h_1 and h_2, of α are said to be* **non-overlapped** *if, either (i) $h_2(v_1) > h_1(v_j)\ \forall v_j \in V_\alpha$ or (ii) $h_1(v_1) > h_2(v_j)\ \forall v_j \in V_\alpha$. A collection of occurrences of α is said to be non-overlapped if every pair of occurrences in it is non-overlapped. The corresponding* **frequency** *for episode α is defined as the cardinality of the largest set of non-overlapped occurrences of α in the given event sequence.*

The standard approach to frequent episodes discovery is to use an Apriori-style level-wise procedure. Starting with frequent episodes of size 1, frequent episodes of progressively larger sizes are obtained (till there are no more frequent episodes at some level). Each level involves two steps – a candidate generation step and a frequency counting step. Candidate generation in the $(N+1)^{\text{th}}$ level, takes frequent episodes of size N and combines them in all possible ways to obtain a set of potential frequent episodes (referred to as candidate episodes) of size $(N+1)$. Candidate generation exploits the anti-monotonicity of episode frequency, i.e. frequency of an episode is bounded above by the frequencies of its subepisodes. Hence, whenever an episode has frequency greater than the user-defined threshold, all its subepisodes would have also met this frequency threshold criterion at previous levels in the algorithm. The frequency counting step obtains the frequencies for the candidate episodes (of a given size) using one pass over the data. This data pass is the main computationally intensive step in frequent episodes discovery. In the next section, we present some very efficient frequency counting algorithms under the

non-overlapped occurrences-based frequency. Before that, we first illustrate how the choice of definition for episode frequency has a direct bearing on efficiency of the frequency counting step of the frequent episode discovery process.

Consider the following example event sequence:

$$\langle (A,1),(A,2),(B,3),(A,7),(C,8), \\ (B,9),(B,10),(D,11),(C,12),(C,13)\rangle. \quad (1)$$

The occurrence of a serial episode may be recognized using a finite state automaton that accepts the episode and rejects all other input. For example, for the episode $(A \rightarrow B \rightarrow C)$, we would have an automaton that transits to state 1 on seeing an event of type A and then waits for an event of type B to transit to its next state and so on until it transits to its final state, when an occurrence of the episode is regarded as complete. Intuitively, the total number of occurrences of an episode seems to be a natural choice for frequency of an episode. However, counting all occurrences turns out to be very inefficient. This is because different instances of the automaton of an episode are needed to keep track of all its state transition possibilities. For example, there are a total of eighteen occurrences of the episode $(A \rightarrow B \rightarrow C)$ in the event sequence (1). We list four of them here:

1. $\{(A,1),(B,3),(C,8)\}$
2. $\{(A,1),(B,3),(C,12)\}$
3. $\{(A,1),(B,3),(C,13)\}$
4. $\{(A,1),(B,9),(C,12)\}$

On seeing the event $(A,1)$ in the event sequence (1), we can transit an automaton of this episode into state 1. However, at the event $(B,3)$, we cannot simply let this automaton transit to state 2. That way, we would miss an occurrence which uses the event $(A,1)$ but some other occurrence of the event type B later in the sequence. Hence, at the event $(B,3)$, we need to keep one instance of this automaton in state 1 and transit another new instance of the automaton for this episode into state 2. As is easy to see, we may need spawning of arbitrary number of new instances of automata if no occurrence is to be missed for an episode. Moreover, counting all occurrences renders candidate generation inefficient as well. This is because, when using total number of occurrences as the frequency definition, subepisodes may be less frequent than corresponding episodes. For example, in (1), while there are eighteen occurrences of $(A \rightarrow B \rightarrow C)$, there are only eight occurrences of the subepisode $(A \rightarrow B)$. So, under such a frequency definition, level-wise procedures cannot be used for candidate generation. Hence, the question now is what kind of restrictions on the class of occurrences will lead to an efficient counting procedure?

Recall that each occurrence, h, of episode α, is associated with a set of events $\{(E_{h(v_i)}, t_{h(v_i)}) : v_i \in V_\alpha\}$ in the data stream. Two occurrences, h_1 and h_2, of an episode α are said to be *distinct* if they do not share any events in the event sequence, i.e., if $h_1(v_i) \neq h_2(v_j)\ \forall v_i, v_j \in V_\alpha$. In the event sequence (1), for example, there can be at most three distinct occurrences of $(A \rightarrow B \rightarrow C)$:

1. $\{(A,1),(B,3),(C,8)\}$
2. $\{(A,2),(B,9),(C,12)\}$
3. $\{(A,7),(B,10),(C,13)\}$

It may appear that if we restrict the count to only distinct occurrences, we may get efficient counting procedures. However, while subepisodes now will certainly be at least as frequent as the episodes, the problem of needing unbounded number of automata remains. This can be seen from the following example. Consider the sequence

$$\langle (A,1),(B,2),(A,3),(B,4),(A,7),(B,8),\ldots\rangle. \quad (2)$$

In such a case, we may need (in principle) any number of instances of the $(A \rightarrow B \rightarrow C)$ automaton, all waiting in state 2, since there may be any number of events of type C occurring later in the event sequence. Hence there is a need for further restricting the kinds of occurrences to count when defining the frequency.

Definition 1 provides an elegant alternative for defining frequency of episodes based on *non-overlapped* occurrences. Two occurrences of an episode in an event sequence are non-overlapped if no event corresponding to one occurrence appears in between events corresponding to the other occurrence. In (1) there can be at most one non-overlapped occurrence of $(A \rightarrow B \rightarrow C)$, e.g., $\{(A,2),(B,3),(C,8)\}$ (since every other occurrence of $(A \rightarrow B \rightarrow C)$ in (1) overlaps with this one). Similarly, in (2) we need to keep track of only one of the pairs of event types A and B, since any other occurrence of $(A \rightarrow B \rightarrow C)$ will have to overlap with this occurrence. In general, there can be many sets of non-overlapped occurrences of an episode in an event sequence. For example, in the event sequence (1), $\{(A,2),(B,3)\}$ and $\{(A,7),(B,9)\}$ are two non-overlapped occurrences of the 2-node episode, $(A \rightarrow B)$. However, if we consider the occurrence $\{(A,2),(B,9)\}$, then there is no other occurrence that is non-overlapped with this one in the sequence (1). This means that the number of non-overlapped occurrences, by itself, is not well-defined. For this reason, we define the frequency in *Definition 1* as the cardinality of the *largest* set of non-overlapped occurrences.

In terms of automata, to count non-overlapped occurrences of an episode, only one automaton is needed. Once an automaton for the episode is initialized, no new instance of the automaton needs to be started till the one that was initialized reaches its final state. In Sec. 3.1, we present the associated frequency counting algorithm and show that, using just one automaton per episode, it is possible to count the maximal (or largest) set of non-overlapped occurrences for a serial episode. Finally we note that although the examples discussed above are all serial episodes, *Definition 1* prescribes a frequency measure for episodes with all kinds of partial orders (including the trivial partial order case of parallel episodes). In Sec. 3.2, we present an efficient algorithm for obtaining the non-overlapped occurrences-based frequency for parallel episodes and indicate later in a discussion how these may be extended to the case of counting episodes with general partial orders as well.

3. FAST COUNTING ALGORITHMS

This section presents some new frequency counting algorithms for frequent episode discovery. In Sec. 3.1, we first present an algorithm for counting non-overlapped occurrences of serial episodes. Then in Sec. 3.2, we show how the algorithm can be adapted to obtain the frequencies of parallel episodes.

3.1 Frequency counting algorithm for serial episodes

Counting serial episodes requires the use of finite state automata. Since we must count the frequencies of several episodes in one pass through the data, there are many automata that need to be simultaneously tracked. In order to access these automata efficiently they are indexed using a $waits(\cdot)$ list. The automata that are currently waiting for event type A can be accessed through $waits(A)$. Each element in the $waits(\cdot)$ list is an ordered pair like (α, j), indicating which episode the automaton represents and which state it is currently waiting to transit into. More specifically, $(\alpha, j) \in waits(A)$ implies that an automaton for α is waiting for an event of type A to appear in the data to complete its transition to state j. This idea of efficiently indexing automata through a $waits(\cdot)$ list was introduced in the windows-based frequency counting algorithm [8]. The list was used to manage up to N automata per N-node episode. The algorithm we present here requires just one automaton per episode, and is also time-wise more efficient.

The overall structure of the algorithm is as follows. The event sequence is scanned in time order. Given the current event, say (E_i, t_i), we consider all automata waiting for an event with event type E_i, i.e., the automata in the list $waits(E_i)$. Automata transitions are effected and fresh automata for an episode are initialized by adding and removing elements from appropriate $waits(\cdot)$ lists. In this respect, a temporary storage called *bag* is used, if it is found necessary to add elements to the $waits(\cdot)$ list over which we are currently looping. We present all algorithms as pseudo code. In the algorithms, N denotes the size of the episodes whose frequencies are being counted, $\alpha[j]$ is the event type corresponding to node j of episode α and $\alpha.freq$ is its current frequency count.

The strategy for counting non-overlapped occurrences is very simple. An automaton for an episode, say α, is initialized at the earliest event in the data sequence that corresponds to the first node of α. As we go down the data sequence, this automaton makes earliest possible transitions into each successive state. Once it reaches its final state, an occurrence of the episode is recognized and its frequency is increased by one. A fresh automaton is initialized for this episode when an event corresponding to its first node appears again in the data and the process of recognizing an occurrence is repeated. This way, for each episode, a set of non-overlapped occurrences is counted. Later in this section, we prove that this strategy yields the maximal set of non-overlapped occurrences. *Algorithm 1* gives the pseudo code for counting non-overlapped occurrences of serial episodes. In the description below we refer to the line numbers in the pseudo code.

Algorithm 1 requires the following inputs: the set of candidate episodes, the event stream and a frequency threshold. (Note that frequency threshold is given as a fraction of data length). The output of the algorithm is the set of frequent episodes (out of the set of candidates input to the algorithm). The $waits(\cdot)$ lists are initialized by adding the pair $(\alpha, 1)$ to $waits(\alpha[1])$, for each episode $\alpha \in \mathcal{C}$, (lines 1-4, *Algorithm 1*). The frequencies are initialized to zero (line 5, *Algorithm 1*) and the temporary storage, *bag*, is initially *empty* (line 6, *Algorithm 1*). Basically, one automaton for each episode is set waiting for the event type corresponding to its first node. The main loop in the algorithm (lines 7-20, *Algorithm 1*) looks at each event in the input sequence and makes necessary changes to the automata in $waits(\cdot)$. When processing the i^{th} event in the data stream, namely, (E_i, t_i), the automata in $waits(E_i)$ are considered. Every automaton (α, j) waiting for E_i is transited to its next state. This involves removing (α, j) from $waits(E_i)$ (line 10, *Algorithm 1*) and adding, either $(\alpha, j+1)$ or $(\alpha, 1)$ to the appropriate $waits(\cdot)$ list (lines 11-17, *Algorithm 1*). More specifically, if the automaton has not yet reached its final state, it waits next for $\alpha[j+1]$ i.e., $(\alpha, j+1)$ is added to $waits(\alpha[j+1])$. If instead, an automaton has reached its final state, then a new automaton for the episode is initialized by adding $(\alpha, 1)$ to $waits(\alpha[1])$. Note that since this process of adding to the $waits(\cdot)$ list is performed inside the loop over all elements in $waits(E_i)$ (i.e., loop starting line 9, *Algorithm 1*), it is inappropriate to add to this list from within the loop. Hence, as was mentioned earlier, we use a temporary storage called *bag*. Whenever we want to add an element to $waits(E_i)$ it is stored first in *bag* which is later emptied into $waits(E_i)$ after exiting from the loop (line 20, *Algorithm 1*). Finally, the episode frequency is incremented every time its automaton reaches the final state (lines 18-19, *Algorithm 1*). Since a new automaton for the episode is initialized only after an earlier one reached its final state, the algorithm counts non-overlapped occurrences of episodes.

Algorithm 1 Non-overlapped count for serial episodes

Require: Set $\mathcal{C}$ of candidate N-node serial episodes, event stream $s = \langle (E_1, t_1), \ldots, (E_n, t_n)) \rangle$, frequency threshold $\lambda_{\min} \in [0, 1]$

Ensure: The set $\mathcal{F}$ of frequent serial episodes in $\mathcal{C}$

```
 1: for all event types A do
 2:    Initialize waits(A) = φ
 3: for all α ∈ C do
 4:    Add (α, 1) to waits(α[1])
 5:    Initialize α.freq = 0
 6: Initialize bag = φ
 7: for i = 1 to n do
 8:    /* n is length of data stream */
 9:    for all (α, j) ∈ waits(E_i) do
10:       Remove (α, j) from waits(E_i)
11:       Set j' = j + 1
12:       if j' = (N + 1) then
13:          Set j' = 1
14:       if α[j'] = E_i then
15:          Add (α, j') to bag
16:       else
17:          Add (α, j') to waits(α[j'])
18:       if j = N then
19:          Update α.freq = α.freq + 1
20:    Empty bag into waits(E_i)
21: Output F = {α ∈ C such that α.freq ≥ nλ_min}
```

3.1.1 Space and time complexity

At any stage in the algorithm, there is only one active automaton per episode which means that there are $|\mathcal{C}|$ automata being tracked simultaneously. The maximum possible number of elements in *bag* is also $|\mathcal{C}|$. Thus, the space

complexity of *Algorithm 1* is $\mathcal{O}(|\mathcal{C}|)$. The initialization time is $\mathcal{O}(|\mathcal{C}| + |\mathcal{E}|)$, where $|\mathcal{E}|$ denotes the size of the alphabet. The time required for the actual data pass is linear in the length, n, of the data sequence. Thus, to count frequencies for all episodes in the set, $\mathcal{C}$, the time complexity of *Algorithm 1* is $\mathcal{O}(n|\mathcal{C}|)$.

The space required by the serial episode counting algorithms of both [8] and [6] are $\mathcal{O}(N|\mathcal{C}|)$, where $\mathcal{C}$ is a collection of N-node candidate episodes. The time complexity of the algorithm in [8] is $\mathcal{O}(\Delta T N|\mathcal{C}|)$, where ΔT denotes the total number of time ticks in the data sequence, while that for the algorithm in [6] is $\mathcal{O}(nN|\mathcal{C}|)$. Thus, both these algorithm suffer an increase in time complexity due to the size, N, of episodes being discovered. In addition to this, sometimes, when the time span of the event sequence far exceeds the number of events in it, the windows-based algorithm would take an even longer time (since $\Delta T \gg n$).

Thus, *Algorithm 1*, both time-wise and space-wise, is an extremely efficient procedure for obtaining frequencies of a set of serial episodes. In fact, it appears difficult to do better than this algorithm in terms of order complexities. This is because, at the least, we need to store and access all the candidate episodes in $\mathcal{C}$, and so space required cannot be less than $\mathcal{O}(|\mathcal{C}|)$. Similarly, at least one pass through the data is required for obtaining the episode frequencies, and in the worst case, at each event in the given event sequence, every candidate might require an update. Thus, it looks like $\mathcal{O}(n|\mathcal{C}|)$ is the best possible worst-case time complexity that can be achieved for counting frequencies of $|\mathcal{C}|$ candidates in a data sequence of n events.

3.1.2 Proof of correctness of Algorithm 1

Algorithm 1 uses only one automaton per episode and hence we wait for the first event type again only after one complete occurrence of the episode. Thus, it is clear that the occurrences of any episode counted by Algorithm 1 would be non-overlapped. Hence, to establish correctness of the algorithm, we have to only show that it counts maximum possible number of non-overlapped occurrences which is what we do in this subsection. Fix an N-node serial episode α. Let $\mathcal{H}$ be the (finite) set of *all* occurrences of α in the given event sequence. (We emphasize that $\mathcal{H}$ contains *all* occurrences of α, including overlapping ones as well as non-distinct ones that share events). Based on the definition of episode occurrence, it is possible to associate with each occurrence, $h \in \mathcal{H}$, a *unique* N-tuple of integers, $(h(v_1), \ldots, h(v_N))$. (Essentially, the events $\{(E_{h(v_1)}, t_{h(v_1)}), \ldots, (E_{h(v_N)}, t_{h(v_N)})\}$ constitute the occurrence.) The lexicographic ordering among these N-tuples, imposes a total order, $<_\star$, on the set $\mathcal{H}$. (The notation $h \leq_\star g$ will be used to denote that either $h = g$ or $h <_\star g$.) This orders the elements of $\mathcal{H}$ such that when $h \leq_\star g$, the occurrence times of events corresponding to these two occurrences must satisfy the following conditions. The first event corresponding to h never occurs later than that for g, i.e., $h(v_1) \leq g(v_1)$. Now, if $h(v_1) = g(v_1)$, then $h(v_2) \leq g(v_2)$. (If instead, $h(v_1) < g(v_1)$, then the remaining occurrence times for h and g need not satisfy any further constraints.) Again, if $h(v_1) = g(v_1)$ and $h(v_2) = g(v_2)$, then $h(v_3) \leq g(v_3)$, and so on.

Let f be the frequency count based on non-overlapped occurrences and let $\mathcal{H}_{no} = \{h_1, \ldots, h_f\}$ denote the sequence of non-overlapped occurrences of α that is counted by *Algorithm 1*, i.e., h_1 is the first occurrence of α that *Algorithm 1* counts, h_2 is the second, and so on. Clearly, $\mathcal{H}_{no} \subset \mathcal{H}$ and we have, $h_1 <_\star \cdots <_\star h_f$. *Algorithm 1* employs one automaton for α which makes earliest possible transitions into each of its states and a fresh automaton for α is initiated only after the current automaton reaches its final state. Thus, h_1, which is the first occurrence of α counted by *Algorithm 1*, is in fact the first occurrence possible for α in the data stream. Then, h_2, the second occurrence of α counted by *Algorithm 1*, is basically the earliest possible occurrence of α in the data stream after h_1 is completed. This gives us two important properties of the set, $\mathcal{H}_{no}$, that *Algorithm 1* counts;

A1-1 The occurrence $h_1 \in \mathcal{H}_{no}$ is such that, $h_1 <_\star h$ for all $h \in \mathcal{H}$, $h \neq h_1$. In other words, h_1 is the earliest occurrence of α in the data steam and hence is the first element of $\mathcal{H}$.

A1-2 For each $i = 1, \ldots, (f-1)$, the occurrence $h_i \in \mathcal{H}_{no}$ is overlapped with any other occurrence, $h \in \mathcal{H}$, if $h_i <_\star h <_\star h_{i+1}$. That is, h_{i+1} is the earliest occurrence after h_i which is non-overlapped with h_i. (Recall that f is the number of non-overlapped occurrences of α counted by *Algorithm 1*.)

Given an occurrence $h \in \mathcal{H}$ that appears after some $h_i \in \mathcal{H}_{no}$, we now ask the question, what can be the earliest occurrence after h that is non-overlapped with h? Since $h_i <_\star h$, and since h_i makes earliest possible transitions to its states, $h_i(v_N) \leq h(v_N)$. Now consider a later occurrence, $\bar{h}$, that is non-overlapped with h. We must have $\bar{h}(v_1) > h(v_N)$, and hence, $\bar{h}(v_1) > h_i(v_N)$. Thus, $\bar{h}$ is non-overlapped with h_i. But the first occurrence after h_i which is non-overlapped with h_i is h_{i+1}, because *Algorithm 1* effects the earliest possible transitions for the automaton. Thus, since $\bar{h}$ is also non-overlapped with h_i and is later than h_i, we must have $h_{i+1} \leq_\star \bar{h}$. We state this fact as a third property of the set, $\mathcal{H}_{no}$, below:

A1-3 Consider an occurrence, $h \in \mathcal{H}$, with $h_i \leq_\star h$ for some $i = 1, \ldots, f$. If $i < f$, and if $\bar{h} \in \mathcal{H}$ is any occurrence non-overlapped with h such that $h <_\star \bar{h}$, then $h_{i+1} \leq_\star \bar{h}$. Instead if $i = f$, then there is no occurrence $\bar{h} \in \mathcal{H}$ such that $h <_\star \bar{h}$ and h is non-overlapped with $\bar{h}$.

We use the above properties to establish maximality of H_{no} (and consequently the correctness of *Algorithm 1*). Assume that there is some other set of f' non-overlapped occurrences of α in the event sequence with $f' > f$. Let us denote this set by $\mathcal{H}' = \{h'_1, \ldots, h'_{f'}\}$ and we have $h'_1 <_\star \cdots <_\star h'_{f'}$. From **A1-1**, we have $h_1 \leq_\star h'_1$. If $f > 1$, from **A1-2** we know that h_2 is the earliest occurrence after h_1 that is non-overlapped with h_1. Thus, since $h_1 \leq_\star h'_1$, and since h'_2 is non-overlapped with h'_1, using **A1-3** we have $h_2 \leq_\star h'_2$. This way, by repeated application of **A1-2** and **A1-3**, we have $h_i \leq_\star h'_i$ for $i = 1, \ldots, f$. Now, since $h_f \leq_\star h'_f$, there can be no occurrence after h'_f that is non-overlapped with h'_f (again, using **A1-3**), implying that, if $f' > f$, then h'_{f+1} must overlap with h'_f, which contradicts our earlier assumption about $\mathcal{H}'$ being a set of f' non-overlapped occurrences of α. Thus, $f' \leq f$ and so f is indeed the maximum number of non-overlapped occurrences possible in the data stream for episode α. This proves that, $\mathcal{H}_{no}$, the set of occurrences counted by *Algorithm 1*, is the largest set of non-overlapped occurrences of α in the given event sequence.

3.2 Frequency counting algorithm for parallel episodes

The non-overlapped frequency definition (i.e.*Definition 1*) is applicable to episodes with all kinds of partial orders. In this section we present an algorithm for counting non-overlapped occurrences of parallel episodes.

An occurrence of a parallel episode simply requires event types corresponding all its nodes to appear in the event sequence, with no restriction on the order in which they appear. The difference when recognizing occurrences of parallel episodes (as compared to recognizing occurrences of serial episodes) is that there is no need to worry about the order in which events occur. Instead, we are interested in asking if each event type in the episode has occurred as many times as prescribed by the episode. For example, each occurrence of the 6-node parallel episode $\alpha = (AABCCC)$ is associated with a set of six events in the data sequence in which, two are of event type A, one is of event type B and the remaining three are of event type C (and it does not matter in which time order they appear).

Algorithm 2, presented below, obtains the non-overlapped occurrences-based frequencies for a set of candidate parallel episodes. As usual, we present the algorithm as a pseudo code and refer to it through line numbers in our description. *Algorithm 2* takes as inputs, the set of candidates, the data stream and the frequency threshold, and outputs the set of frequent episodes. The main data structure here is once again a $waits(\cdot)$ list - but it works a little differently from the one used earlier in Sec. 3.1. Each entry in the list $waits(A)$, is an ordered pair like, (α, j), which now indicates that there is a partial occurrence of α which still needs j events of type A before it can become a complete occurrence. The initialization process (lines 1-8, *Algorithm 2*) involves adding the relevant ordered pairs for each episode α into appropriate $waits(\cdot)$ lists. For example, episode $\alpha = (AABCCC)$ will initially figure in three lists, namely, $waits(A)$, $waits(B)$ and $waits(C)$, and they will have entries $(\alpha, 2)$, $(\alpha, 1)$ and $(\alpha, 3)$ respectively. There are two quantities associated with each episode, α, namely, $\alpha.freq$, which stores the frequency of α, and $\alpha.counter$, which indicates the number of events in the sequence that constitute the current partial occurrence of α.

As we go down the event sequence, for each event (E_i, t_i), the partial occurrences waiting for an E_i are considered for update (line 11, *Algorithm 2*). If, $(\alpha, j) \in waits(E_i)$, then having seen an E_i now, (α, j) is replaced by $(\alpha, j-1)$ in $waits(E_i)$ if sufficient number of events of type E_i for α are not yet accounted for in the current partial occurrence (lines 13-15, *Algorithm 2*). Note that this needs to be done through the temporary storage *bag* since we cannot make changes to $waits(E_i)$ from within the loop. Also, $\alpha.counter$ is incremented (line 12, *Algorithm 2*), indicating that the partial occurrence for α has progressed by one more node. When $\alpha.counter = |\alpha| = N$, it means that the N events necessary for completing an occurrence have appeared in the event sequence. We increment the frequency by one and start waiting for a fresh occurrence of α by once again adding appropriate elements to the $waits(\cdot)$ lists (lines 16-23, *Algorithm 2*).

3.2.1 Space and time complexity

Each $waits(\cdot)$ list can have at most $|\mathcal{C}|$ entries and so the space needed by *Algorithm 2* is $\mathcal{O}(N|\mathcal{C}|)$ (because there can be at most N distinct event types in an episode of size N). To analyze the time complexity, note that, some extra work needs to be done during initialization (as compared to the serial episode algorithms) to obtain the number of times each event type in an episode repeats (lines 4-5, *Algorithm 2*). This means the initialization time complexity is $\mathcal{O}(|\mathcal{E}| + N|\mathcal{C}|)$. The main loop, as usual, is over n events in the data, and any of the $waits(\cdot)$ loops, can at most be over $|\mathcal{C}|$ partial occurrences. Re-initialization of appropriate $waits(\cdot)$ lists whenever an occurrence is complete (lines 16-23, *Algorithm 2*) takes $\mathcal{O}(N)$ time. This re-initialization needs to be done at most $\frac{n}{N}$ times for each episode. Hence, the total worst case time complexity of *Algorithm 2* is $\mathcal{O}(n|\mathcal{C}|)$. The space complexity of the windows-based algorithm for N-node parallel episodes is $\mathcal{O}(N|\mathcal{C}|)$ and the time complexity is $\mathcal{O}(\Delta T|\mathcal{C}|)$, where ΔT denotes the number of time ticks in the data sequence. Thus, except for the fact that, sometimes, the time, ΔT, spanned by the data sequence can be much larger than the number, n, of events in it, the time and space complexities of the non-overlapped occurrences-based algorithm and the windows-based algorithm are identical.

Algorithm 2 Non-overlapped count for parallel episodes

Require: Set $\mathcal{C}$ of candidate N-node parallel episodes, event stream $s = \langle (E_1, t_1), \ldots, (E_n, t_n)) \rangle$, frequency threshold $\lambda_{\min} \in [0, 1]$
Ensure: The set $\mathcal{F}$ of frequent parallel episodes in $\mathcal{C}$

```
 1: for all event types A do
 2:    Initialize waits(A) = φ
 3: for all α ∈ C do
 4:    for each event type A in α do
 5:       Set a = Number of events of type A in α
 6:       Add (α, a) to waits(A)
 7:    Initialize α.freq = 0
 8:    Initialize α.counter = 0
 9: Initialize bag = φ
10: for i = 1 to n do
11:    for all (α, j) ∈ waits(E_i) do
12:       Update α.counter = α.counter + 1
13:       Remove (α, j) from waits(E_i)
14:       if j > 1 then
15:          Add (α, j − 1) to bag
16:       if α.counter = N then
17:          Update α.freq = α.freq + 1
18:          for each event type A in α do
19:             Set a = Number of events of type A in α
20:             if A = E_i then
21:                Add (α, a) to bag
22:             else
23:                Add (α, a) to waits(A)
24:          Reset α.counter = 0
25:    Empty bag into waits(E_i)
26: Output F = {α ∈ C such that α.freq ≥ nλ_min}
```

3.2.2 Correctness of Algorithm 2

Earlier, in Sec. 3.1, we proved that *Algorithm 1* always yields the maximum possible number of non-overlapped occurrences in the data. The basic idea was that, by making earliest possible transitions in the automata, we ensure that we track the largest number of non-overlapped occurrences available in the data, for each (serial) episode being

counted. As we have seen from *Algorithm 2*, there is no need for any automata when tracking non-overlapped occurrences of parallel episodes. However, *Algorithm 2* is similar to *Algorithm 1* in respect of how they both track occurrences in the data by recognizing the earliest possible events for *each node* of an episode. This strategy ensures that we will count the maximum number of parallel episodes. Since the arguments needed to show this formally follow the same lines as our proof for the case of serial episodes in Sec. 3.1, for the sake of brevity, we do not explicitly prove the correctness of *Algorithm 2* here.

3.3 Discussion

In this section, we have presented two new algorithms – one that obtains the non-overlapped occurrences-based frequencies for a set of serial episodes, and the other that obtains the same for a set of parallel episodes. *Algorithm 1*, which is the counting algorithm for serial episodes, requires just *one automaton per candidate episode*. This makes it an extremely efficient algorithm, both in terms of time and space, compared to all currently known algorithms [8, 6] for frequent serial episode discovery. We have also provided a proof of correctness for the algorithm to show that *Algorithm 1* indeed obtains the frequency of serial episodes as prescribed by *Definition 1*. The algorithm for parallel episodes (i.e. *Algorithm 2*) is also very efficient. We note that this is the first time an algorithm has been reported for obtaining the non-overlapped occurrences-based frequencies for parallel episodes. However, its space and time complexities are same as that for the windows-based counting algorithm for parallel episodes [8].

In general, *Definition 1*, is applicable to episodes with all kinds of partial orders. Any general partial order can be represented as a combination of serial and parallel episodes. For example, consider an episode having three nodes with event types, A, B and C. Let the partial order be such that both A and B must occur before C, but there is no restriction on the order among A and B. We can denote this episode as $(AB) \rightarrow C$. Such an episode is like a serial episode with two nodes, where the first node corresponds to a parallel episode, (AB), and the second node is C. Occurrences of such partial orders can be recognized using automata-type structures, where parallel episodes recognition needs to be used as a subroutine. Viewed like this, it is possible, in principle, to design algorithms for counting non-overlapped occurrences of episodes with general partial orders. However, more work is needed to transform this strategy into an efficient counting algorithm. Moreover, there is also a need to design efficient candidate generation strategies which can exploit the fact that the number of non-overlapped occurrences of an episode is never greater than that of any of its subepisodes. Therefore, developing algorithms for discovering frequent episodes with general partial orders under the non-overlapped occurrences-based frequency, would be a useful extension of the work presented here.

4. SIMULATIONS

We present results obtained on some synthetic data generated by embedding specific temporal patterns in varying levels of noise. The main objective of the experiments presented here is to empirically demonstrate the efficiency advantage of our new algorithm. The utility and effectiveness of the non-overlapped occurrences-based frequency in real applications have already been discussed in our earlier work [6, 7]. We had also shown through simulation experiments there, that our earlier algorithm for counting non-overlapped occurrences is itself faster than the windows-based algorithm of [8]. Here, we compare *Algorithm 1* with our earlier algorithm for counting non-overlapped occurrences (which was reported in [6] and to which we refer to in this section as *Algorithm 0*), as well as, with the windows-based frequency counting algorithm of [8] (which is referred to as *Algorithm W* in this section). We note that the frequency counts obtained for serial episodes using *Algorithm 1* of this paper are identical to those obtained using *Algorithm 0*, and hence, exactly the same set of frequent episodes would be output by both algorithms. However, it is not possible to directly relate the frequencies obtained using the windows-based algorithm (*Algorithm W*) with those obtained under the non-overlapped occurrences-based counting algorithms. In our simulation experiments, we found that the sets of frequent episodes obtained under both frequency definitions are qualitatively very similar. Hence, the goal of this section is mainly to demonstrate that the gains (by a factor of N, where N is the size of episodes) in order complexities of *Algorithm 1* over *Algorithm 0* and *Algorithm W*, translate to actual run-time gains as well. For the case of parallel episodes, the space and time complexities of our algorithm is same as that of the windows-based algorithm for parallel episodes reported in [8]. Further, since there is no other algorithm for counting non-overlapped occurrences of parallel episodes, we do not provide any comparative results for *Algorithm 2*.

By varying the control parameters of synthetic data generation, it is possible to generate qualitatively different kinds of data sets. In general, the temporal patterns that were picked up by our algorithm correlated very well with those that were explicitly inserted in the data even when these patterns are embedded in varying amounts of noise. Also, the sets of frequent episodes discovered were same as that discovered using our earlier algorithm of [6]. However, both in terms of memory as well as run-times our new algorithm is much more efficient.

4.1 Synthetic data generation

Each of the temporal patterns to be embedded (in the synthetically generated data) consists of a specific ordered sequence of events. A few such temporal patterns are specified as input to the data generation process which proceeds is as follows. There is a counter that specifies the current time instant. Each time an event is generated, it is time-stamped with this current time as its time of occurrence. After generating an event (in the event sequence) the current time counter is incremented by a small random integer. Each time the next event is to be generated, we first decide whether the next event is to be generated randomly with a uniform distribution over all event types (which would be called an *iid* event) or according to one of the temporal patterns to be embedded. This is controlled by the parameter ρ which is the probability that the next event is *iid*. If $\rho = 1$ then the data is simply *iid* noise with no temporal patterns embedded. If it is decided that the next event is to be from one of the temporal patterns to be embedded, then we have a choice of continuing with a pattern that is already embedded partially or starting a new occurrence of one of the patterns. This choice is also made randomly. It may

ρ	*Algo 1*	*Algo 0*	*Algo W*
0.0	1	1	1
0.2	1	1	1
0.4	1	1	1
0.6	1	1	1
0.8	1	1	1

Table 1: Ranks of α in frequency-sorted list of 4-node frequent episodes, for *Algorithm 1*, *Algorithm 0* and *Algorithm W*, on synthetic data with two patterns embedded in varying levels of noise. Data length is 50000 and number of event types is 50.

ρ	*Algo 1*	*Algo 0*	*Algo W*
0.0	12	12	12
0.2	1	1	1
0.4	1	1	1
0.6	1	1	1
0.8	1	1	1

Table 2: Ranks of β in frequency-sorted list of 3-node frequent episodes, for *Algorithm 1*, *Algorithm 0* and *Algorithm W*, on synthetic data with two patterns embedded in varying levels of noise. Data length is 50000 and number of event types is 50.

be noted here that due to the nature of our data generation process, embedding a temporal pattern is equivalent to embedding many episodes. For example, suppose we have embedded a pattern $A \rightarrow B \rightarrow C \rightarrow D$. Then if this episode is frequent in our event sequence then, based on the amount of noise, episodes such as $B \rightarrow C \rightarrow D \rightarrow A$ can also become frequent.

4.2 Effectiveness of frequency count based on non-overlapped occurrences

We now present some simulation results to show that the episodes discovered as frequent by *Algorithm 1* are same as that discovered by *Algorithm 0*. (This indeed must be the case, since *Algorithms 1 & 0* are essentially frequency counting algorithms under the same frequency definition.) The main difference between Algorithms 1 & 0 is that among any set of overlapped occurrences, *Algorithm 1* tracks the earliest among them, while *Algorithm 0* tracks the innermost among them. In this section, we illustrate this aspect empirically by considering some synthetic data generated by embedding two patterns in varying degrees of *iid* noise. The two patterns embedded are: $\alpha = (B \rightarrow C \rightarrow D \rightarrow E)$ and $\beta = (I \rightarrow J \rightarrow K)$. Data sequences with 50000 events each are generated for different values of ρ. The objective is to see whether these two patterns indeed appear among the sets of frequent episodes discovered (under both frequency counts), and if so, at what positions. The respective positions of α and β (referred to as their *ranks*) in the (frequency) sorted lists of 3-node and 4-node frequent episodes discovered are shown in Tables 1 & 2. For comparison, we also show the ranks obtained using the windows-based algorithm in the tables. As can be seen from the tables, for all data sets, the ranks of α and β are identical under all three algorithms.

Size of episodes	*Algo 1*	*Algo 0*	*Algo W*	Speed-up
1	0.36	0.35	28.25	78
2	0.40	0.37	29.54	74
3	0.44	0.85	29.75	67
4	0.45	1.53	30.52	67
5	0.46	2.22	31.48	68

Table 3: Run-times (in seconds) for *Algorithm 1*, *Algorithm 0* and *Algorithm W*, for a fixed number of candidate episodes but with different sizes of episodes. The last column records the speed-up factor of *Algorithm 1* with respect to *Algorithm W*. Data length is 50000, number of candidates is 500 and number of event types is 500.

No of candidates	*Algo 1*	*Algo 0*	*Algo W*	Speed-up
100	0.32	0.34	28.27	88
200	0.34	0.51	28.29	83
300	0.37	0.75	29.34	86
400	0.39	1.03	30.52	78
500	0.45	1.55	30.73	68

Table 4: Run-times (in seconds) for *Algorithm 1*, *Algorithm 0* and *Algorithm W*, for a fixed size of episodes but with different number of candidate episodes. The last column records the speed-up factor of *Algorithm 1* with respect to *Algorithm W*. Data length is 50000, size of episodes is 4 and number of event types is 500.

4.3 Run-time comparisons

Now we present some run-time comparisons to show that *Algorithm 1* runs faster than both *Algorithm 0* and *Algorithm W*. Recall that the worst-case time complexities of these algorithms are $\mathcal{O}(n|\mathcal{C}|)$ and $\mathcal{O}(nN|\mathcal{C}|)$. In this section, we empirically show that the better time complexity of *Algorithm 1* translates to significant advantages in terms of actual run-times as well.

We first show how run-times of the algorithms vary with the size of episodes being discovered. The frequency counting algorithms were presented with several sets of candidate episodes with each set containing the same number (500) of episodes. However, the size of the episodes in each set was different. Table 3 lists the comparison of run-times of the two algorithms for these sets of candidate episodes. The input data stream used was a 50000-long uniform iid event sequence over a large number (500) of event types. In such a sequence, all episodes of a given size would roughly have the same frequencies and hence the computation associated with all candidates in any given set would roughly be of the same order. It can be seen from the tables that *Algorithm 1* is always faster than both *Algorithm 0* and *Algorithm W*. It can also be seen from the table that, for *Algorithm 1*, the run-times do not increase much with the size of episodes, while for *Algorithm 0*, the run-times are roughly linear in the size of episodes.

Next we perform a similar experiment by varying number of candidates but keeping the size of episodes fixed. We

No of events	*Algo 1*	*Algo 0*	*Algo W*	Speed-up
10000	0.19	0.43	5.63	29
20000	0.25	0.72	12.47	49
30000	0.48	0.99	18.39	38
40000	0.52	1.33	25.95	49
50000	0.63	1.55	29.75	47

Table 5: Run-times (in seconds) for *Algorithm 1*, *Algorithm 0* and *Algorithm W*, for different lengths of data sequences. The last column records the speed-up factor of *Algorithm 1* with respect to *Algorithm W*. Number of candidates is 500, size of episodes is 4 and number of event types is 500.

ρ	*Algo 1*	*Algo 0*	*Algo W*	Speed-up
0.0	0.46	2.61	64.82	140
0.2	0.44	4.38	76.66	174
0.4	0.89	5.12	96.80	108
0.6	0.56	3.46	85.23	152
0.8	0.49	2.04	83.64	170

Table 6: Total run-times (in seconds) for *Algorithm 1*, *Algorithm 0* and *Algorithm W*, for synthetic data with two patterns embedded in varying levels of noise. The last column records the speed-up factor of *Algorithm 1* with respect to *Algorithm W*. Data length is 50000 and number of event types is 500.

consider sets with different number of 4-node candidate serial episodes. The corresponding run-times for the three algorithms are listed in Table 4. Here again, we see that *Algorithm 1*, both runs faster and scales better with increasing number of candidates, than *Algorithm 0* and *Algorithm W*. Similar results were obtained when we studied the effect of data length on the run-times of the various algorithms and the results are shown in Table 5.

Finally, we compare the overall run-times for frequent episode discovery based on all these algorithms. The algorithms were run on synthetic data generated by embedding two patterns in varying levels of iid noise. The frequency thresholds were chosen such that roughly the same number (100) of frequent 4-node episodes are output by both algorithms. The results are tabulated in Table 6. In all cases, it can be seen that *Algorithm 1* outperforms both *Algorithm 0* and *Algorithm W*.

5. CONCLUSIONS

In this paper, we have presented some new algorithms for frequency counting under the non-overlapped occurrences-based frequency for episodes. The new algorithms are, both space-wise as well as time-wise, significantly more efficient than the earlier algorithms reported in [6]. These algorithms arguably have the best space and time complexities for frequency counting of a given set of candidate episodes. This algorithmic efficiency, together with the theoretical properties presented in [6], make out a very strong case for using the non-overlapped occurrences-based frequency for frequent episode discovery in event streams.

In the frequency counting algorithms described in this paper we do not worry about the spread of events within an occurrence. In some applications, it may be necessary not to count an episode occurrence if the events constituting it are widely spread out. The windows-based frequency count of [8], for example, implements some kind of a time constraint on the occurrences of an episode, since the width of the window used is basically an upper bound on the time span of the occurrences (that are considered for the episode's frequency). However, the problem with such a scheme is that, while it eliminates widely spread out occurrences from contributing to the frequency count, it also artificially increases the frequency when occurrences are very compact.

Since our new frequency counting algorithms explicitly count episode occurrences, it is possible, when an application so requires, to incorporate an extra time constraint directly on the occurrences being counted [5]. An *expiry time* constraint can be used to define the extent to which events of an occurrence may be spread out in the event sequence. In case of serial episodes, the expiry time constraint is an upper bound on the time difference between the events in an occurrence corresponding to the first and last nodes of the episode. Incorporating such expiry time constraints does increase the space and time complexities of the algorithm a little bit. The counting strategy we now need closely resembles that of the algorithm for serial episodes that we proposed earlier in [6]. Even when such time constraints are prescribed, the algorithms for counting non-overlapped occurrences-based frequencies are both space-wise and time-wise very efficient, and compare favorably with the windows-based frequency counting scheme. We will address some of these issues in our future work.

Acknowledgment

This research was partially funded by GM R&D Center, Warren through SID, IISc, Bangalore.

6. REFERENCES

[1] M. J. Atallah, R. Gwadera, and W. Szpankowski. Detection of significant sets of episodes in event sequences. In *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM 2004)*, pages 3–10, Brighton, UK, 01-04 November 2004.

[2] G. Casas-Garriga. Discovering unbounded episodes in sequential data. In *Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'03)*, pages 83–94, Cavtat-Dubvrovnik, Croatia, 2003.

[3] R. Gwadera, M. J. Atallah, and W. Szpankowski. Reliable detection of episodes in event sequences. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM 2003)*, pages 67–74, 19-23 November 2003.

[4] R. Gwadera, M. J. Atallah, and W. Szpankowski. Markov models for identification of significant episodes. In *Proceedings of the 2005 SIAM International Conference on Data Mining (SDM-05)*, Newport Beach, California, April 2005.

[5] S. Laxman. *Discovering frequent episodes in event streams: Fast algorithms, connections with HMMs and generalizations*. PhD thesis, Dept. of Electrical

Engineering, Indian Institute of Science, Bangalore, India, Mar. 2006.

[6] S. Laxman, P. S. Sastry, and K. P. Unnikrishnan. Discovering frequent episodes and learning Hidden Markov Models: A formal connection. *IEEE Transactions on Knowledge and Data Engineering*, 17(11):1505–1517, Nov. 2005.

[7] S. Laxman, P. S. Sastry, and K. P. Unnikrishnan. Discovering frequent generalized episodes when events persist for different durations. To appear in *IEEE Transactions on Knowledge and Data Engineering*, 2007.

[8] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[9] N. Meger and C. Rigotti. Constraint-based mining of episode rules and optimal window sizes. In *Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'04), Pisa, Italy*, Sept. 2004.

[10] M. Regnier and W. Szpankowski. On pattern frequency occurrences in a Markovian sequence. *Algorithmica*, 22:631–649, 1998.