Abstract

In this paper we consider the process of discovering frequent episodes in event sequences. The most computationally intensive part of this process is that of counting the frequencies of a set of candidate episodes. We present two new frequency counting algorithms for speeding up this part. These, referred to as non-overlapping and non-intelevied frequency counts, are based on directly counting suitable subsets of the occurrences of an episode. Hence they are different from the frequency counts of Mannila et al [1], where they count the number of windows in which the episode occurs. Our new frequency counts offer a speed-up factor of 7 or more on real and synthetic datasets. We also show how the new frequency counts can be used when the events in episodes have time-durations as well.

1. INTRODUCTION

Development of data mining techniques for time series data is an important problem of current interest [1, 2, 3, 4, 5]. An interesting framework for temporal data-mining in the form of discovering frequent episodes in event sequences was first proposed in [1]. This framework has been found useful for analyzing, e.g. alarm streams in telecommunication networks, logs on web servers, etc. [1, 6, 7]. Further, in [2], the formalism for episode discovery was extended to incorporate time durations into the episode definitions. For many data sets, such as the manufacturing plant data analyzed in this paper, this extended framework provides a richer and more expressive class of patterns in the temporal data mining context. This paper describes two new algorithms useful in the general framework of frequent episode discovery.

This paper proposes two alternative measures for frequency of an episode. Both these are more closely related to the number of occurrences of episodes than the windows-based frequency count proposed in [1]. We show through some empirical investigations that our method of counting would also result in essentially the same kind of episodes being discovered as frequent. However, our frequency counting algorithms need less temporary memory and more importantly they speed up the process of discovering frequent episodes by a factor of 7 (or more) thus rendering the framework of episode discovery attractive in many more applications.

The following is a quick introduction to the framework proposed in [1]. The data is a sequence of events given by \((E_1, t_1), (E_2, t_2), \ldots\) where \(E_i\) represents an event type and \(t_i\) the corresponding time of occurrence of the event. All \(E_i\) belong to a finite set of event types. For example, the following is an event sequence containing seven events:

\[
\langle (A, 1), (B, 3), (C, 4), (E, 12), (A, 14), (B, 15), (C, 16) \rangle.
\]

Note that in all our examples, we use the set \(\{A, B, C, \ldots\}\) as the set of event types. An episode is an ordered tuple of event types\(^1\). For example, \((A \rightarrow B \rightarrow C)\) is a 3-node episode. An episode is said to occur\(^1\) in an event sequence if we can find events in the sequence which have the same time ordering as that specified by the episode. In the example sequence given by (1), an occurrence of the episode \((A \rightarrow B \rightarrow C)\) is constituted by the events \((A, 1), (B, 3), (C, 4)\).

Note that the events \((A, 14), (B, 3), (C, 4)\) do not constitute an occurrence because the episode demands that \(A\) precedes \(B\) and \(C\). Another occurrence of the episode here is \((A, 1), (B, 3), (C, 16)\). It is easy to observe that there are altogether four occurrences of this episode in the data sequence given by (1). A subepisode is a subsequence of the episode which has the same ordering as the episode. For example, \((A \rightarrow B), (A \rightarrow C)\) and \((B \rightarrow C)\) are the 2-node subepisodes of the 3-node episode \((A \rightarrow B \rightarrow C)\), while \((B \rightarrow A)\) is not. The frequency of an episode can be defined in many ways. However, for a frequency count to be reasonable, its definition must guarantee that any subepisode is at

\(^1\)In the formalism of [1], this corresponds to the serial episode.
A frequent episode is one whose frequency exceeds a user specified threshold. The procedure for discovering frequent episodes proposed in [1] is based on the same general idea as the Apriori algorithm [5]. First we compute all frequent 1-node episodes. Then these are combined in all possible ways to make candidate 2-node episodes. By calculating the frequencies of these candidates, obtain all frequent 2-node episodes. These are then used to obtain candidate 3-node episodes and so on. In our method we use the same procedure for generating candidate episodes as in [1].

2. FREQUENCY COUNTING

The main computationally intensive step in frequent episode discovery is that of calculating the frequency of sets of candidate episodes. This needs to be achieved using as few database passes as possible. In [1], Mannila, et. al., suggested a frequency count which is defined as the number of (fixed width) windows (over the data) in which the episode (whose frequency we are counting) occurs at least once. Multiple occurrences of the episode in the same window has no effect on this frequency measure. Since episodes are essentially temporal patterns their occurrence(s) can be recognized using finite state automata. For the windows-based count, Mannila, et. al. [1] present a fairly efficient algorithm that uses n automata per episode to count frequencies of n-node episodes.

The windows-based frequency count however, is not easily relatable to the general notion of frequency, namely, the number of occurrences of an episode in the data. Further, it is sensitive to a user-specified window width. Another windows-based frequency count has been recently proposed [8] in which the size of the window grows automatically to accommodate episode occurrences that are more spread out. In [1], Mannila et al. proposed an alternative frequency measure which is defined as the number of minimal occurrences of an episode. A minimal occurrence is essentially a window in which the episode occurs such that no proper sub-window of it contains an occurrence of the episode. However, the informal algorithm suggested in [1] for counting this frequency is very inefficient in terms of the temporary memory needed. (The space complexity of this method is of the order of the length of the data sequence).

Intuitively, the number of occurrences of an episode seems to be the most natural choice for its frequency. As proposed in [1], the occurrence of an episode in an event sequence may be recognized by 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 a automaton that transits to state \( 'A' \) 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. When this automaton transits to its final state we have recognized an occurrence of the episode. We need different instances of the automaton of an episode to keep track of all its state transition possibilities and hence count all its occurrences. If we want to count all occurrences, the number of automata can become unbounded. Consider again the example given earlier. When we see the event \( (A, 1) \) we can transit an automaton of this episode into state \( A \). When we see the next event \( (B, 3) \), we cannot simply let this automaton transit to state \( B \). That way, we would miss an occurrence which uses the event \( (A, 1) \) but some other occurrence of the event type \( B \). Hence, when we see \( (B, 3) \), we need to keep one automaton in state \( A \) and transit another instance of automaton for this episode into state \( B \). As is easy to see, we may need spawning of arbitrary number of new instances of automata if we do not want to miss counting any occurrence. Hence, the question now is how should we suitably restrict the class of occurrences so that we can find an efficient counting procedure.

Each occurrence of an episode is associated with a set of events in the data stream. We say two occurrences are distinct if they do not share any events. In the data sequence given by (1), there are only two distinct occurrences of \((A \rightarrow B \rightarrow C)\) though the total number of occurrences are four. It may appear that if we restrict the count to only distinct occurrences, then we do not need to spawn unbounded number of automata. However, this is not true, as can be seen from the following example. Consider the sequence \( \langle (A, 1), (B, 2), (A, 3), (B, 4), (A, 7), (B, 8), \ldots \rangle \). (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 the there may be any number of occurrences of the event type \( C \) later in the event sequence.

Hence there is a need for further restricting the kinds of occurrences to count when defining the frequency. We define two such frequencies below, one based on what we call non-overlapping occurrences and the other based on what is termed non-interleaved occurrences.

Two occurrences of an episode in an event sequence are called non-overlapping if any event corresponding to one occurrence does not happen to be in between events corresponding to the other occurrence. In (1) there are only two non-overlapping occurrences of \((A \rightarrow B \rightarrow C)\). In (2) we need to keep track of only the last pair of event types \( A \) and \( B \) if all we need to recognize are non-overlapping occurrences \((A \rightarrow B \rightarrow C)\). We note in passing here that this frequency count based on non-overlapping episode occurrences is also interesting because it facilitates a formal connection between frequent episode discovery and HMM learning [9].

Although, the idea of counting only non-overlapping occurrences is theoretically elegant and practically attractive, it does appear to constrain the occurrence possibilities for an episode in the event sequence. There is thus a need to find a way to count at least some overlapping occurrences as well. Towards this end we define non-interleaved occurrences of an episode.

Each occurrence of an episode is a 1-1 map from the nodes of the episode to events in the data sequence. For an episode \( \alpha \), we denote the number of nodes in it by \( |\alpha| \) and the ordered sequence of nodes of \( \alpha \) is referred to as \( v_1, v_2, \ldots \). Let \( h_1 \) and \( h_2 \) denote two different occurrences of \( \alpha \). Thus, \( h_1(v_i) \) denotes the event in the data sequence that corresponds to the node \( v_i \) of the episode in the occurrence represented by \( h_1 \). By \( h_1(v_i) < h_2(v_i) \), we mean that the event (in the data) corresponding to node \( v_i \) in occurrence \( h_1 \) has an
We illustrate the definition with an example. Consider an event sequence or of an episode earlier occurrence time than that of the event corresponding to node $v_j$ in occurrence $h_2$. Two occurrences, $h_1$ and $h_2$, of an episode $\alpha$ are said to be non-interleaved if either

$$h_2(v_j) > h_1(v_{j+1}) \forall j, 1 \leq j < |\alpha|,$$

or

$$h_1(v_j) > h_2(v_{j+1}) \forall j, 1 \leq j < |\alpha|.$$

We illustrate the definition with an example. Consider an event sequence

$$\langle (A, 1), (B, 3), (A, 4), (A, 7), (E, 12), (B, 14), (B, 15), (C, 16), (C, 18), (B, 19), (C, 23) \rangle.$$ (3)

Consider the same episode as earlier, namely, $\alpha = (A \rightarrow B \rightarrow C)$. Here, there are three distinct occurrences corresponding to: $(A, 1), (B, 3), (C, 16), (A, 4), (B, 14), (C, 18)$ and $(A, 7), (B, 15), (C, 23)$. Since there are only three occurrences of the event type $A$, we cannot have more than three distinct occurrences. Since any two occurrences are overlapping, there can be at most one non-overlapping occurrences of $A$. When counting non-interleaved occurrences, there can be a maximum of two occurrences: $(A, 1), (B, 3), (C, 16)$ and $(A, 4), (B, 19), (C, 23)$ (or alternatively, $(A, 1), (B, 3), (C, 18)$ and $(A, 4), (B, 19), (C, 23)$).

In terms of the automata that recognize each occurrence, this definition (of non-interleaved) means the following. An instance of the automaton for $\alpha$ can transit into the state corresponding to a node, say $v_2$, only if an earlier instance (if any) of the automaton has already transited into state $v_3$ or higher. Non-interleaved occurrences would include some overlapping occurrences though they do not include all occurrences.

We present two algorithms (Algorithm A and B respectively) for counting non-overlapping and non-interleaved occurrences. Both these use as many automata per episode as there are states in it which is same as the number needed for the window-based frequency count of [1]. Given a set of candidate episodes $\mathcal{C}$, the algorithms return the set of frequent episodes $\mathcal{F}$. Each frequent episode has frequency above a user specified threshold (which is an input to the algorithm). The other input is the data represented as $s = \langle s_1, t_1, s_2, t_2 \rangle$ where $s$ is the sequence of events with $T_s$ and $T_e$ being the start and end times of that sequence.

### 2.1 Non-overlapping occurrences

Algorithm A counts the number of non-overlapping occurrences. At the heart of this automata-based counting scheme is the $\text{waits}(\cdot)$ list. Since there are many candidate episodes and for each of which there are multiple occurrences, at any time there would be many automata waiting for many event types to occur. In order to traverse and access the automata efficiently, for each event type $A$, the automata that accept $A$ are linked together in the list $\text{waits}(A)$. The $\text{waits}(A)$ list contains entries of the form $(\alpha, j)$ meaning that an automaton of the episode $\alpha$ is waiting for event type $A$ as its $j^{th}$ event. That is, if the event type $A$ occurs now in the event sequence, this automaton would accept it and transit to the $j^{th}$ state. At any time (including at the start of the counting process) there would be automata waiting for the event types corresponding to first nodes of all the candidate episodes. This is how the $\text{waits}(\cdot)$ list is initialized. After that, every time an event type for which some automaton is waiting occurs, that automaton makes a transition and the $\text{waits}(\cdot)$ list is accordingly updated. In addition to the $\text{waits}(\cdot)$ list, we also need to store the initialization time (i.e. time at which the first event of the episode occurred) for each instance of the automaton (for a given episode).

It useful to have multiple automata in the same state, as they would only make the same transitions. It suffices to maintain the one that reached the common state last (when counting non-overlapping occurrences). Thus, we need to store at most $|\alpha|$ initialization times for $\alpha$. This is done by $\alpha: \text{init}[j]$ which indicates when an instance for $\alpha$ that is currently in its $j^{th}$ state, got initialized. If multiple instances transit to the same state, we remember only the most recent initialization time.

Algorithm A, at the instant of completion of any one occurrence of the episode, resets all other automata that might have been initialized for that episode. This ensures that any collection of overlapping occurrences increments the frequency for the episode by exactly one.

Further, it may be useful to prescribe an expiry time for episode occurrences, so that we do not count very widely spread out events as an occurrence of some episode. This condition is enforced in the algorithm by testing the time taken to reach the new state before permitting a transition into it. This expiry time condition is only an added facility in the algorithm and the condition can easily be dropped. It may be noted that the scheme of retaining only the latest initialization time for the automata in a given state is consistent with this expiry time restriction.

### Algorithm A: Non-overlapping occurrence count

**Input:** Set $\mathcal{C}$ of (candidate) episodes, event stream $s = \langle (E_1, t_1), \ldots, (E_n, t_n) \rangle$, frequency threshold $\lambda_{\text{min}}$

**Output:** The set $\mathcal{F}$ of frequent episodes in $\mathcal{C}$

```
1: /* Initialization */
2: Initialize $\text{bag} = \phi$
3: for all event types $A$ do
4: Initialize $\text{waits}(A) = \phi$
5: for all $\alpha \in \mathcal{C}$ do
6: Add $(\alpha, 1)$ to $\text{waits}(\alpha[1])$
7: Initialize $\alpha: \text{freq} = 0$
8: for $j = 1$ to $|\alpha|$ do
9: Initialize $\alpha: \text{init}[j] = 0$
10: /* Data pass */
11: for $i = 1$ to $|s|$ do
12: for all $(\alpha, j) \in \text{waits}(E_i)$ do
13: if $j = 1$ then
14: /* Initialize $\alpha$ automaton */
15: Update $\alpha: \text{init}[1] = t_i$
16: else
17: /* Transit $\alpha$ automaton to state $j$ */
18: Update $\alpha: \text{init}[j] = \alpha: \text{init}[j - 1]$
19: Reset $\alpha: \text{init}[j - 1] = 0$
20: Remove $(\alpha, j)$ from $\text{waits}(E_i)$
21: if $j < |\alpha|$ then
```
2.2 Non-interleaved occurrences

We next present Algorithm B, which counts the number of non-interleaved occurrences. There are two significant changes with respect to Algorithm A. The first is that the algorithm does not permit a transition into a particular state (except the first state) if there is already an instance of the automaton waiting in that state. In other words, while it still uses only one automaton per state, it does not forget an earlier initialization of the automaton until that has transited to the next state. The second change is that we do not reset all instances of an episode if one of it reaches the final state. This way we can count some overlapping occurrences (as needed), so long as a previous instance has transited at least one state more than the next instance.

We illustrate this counting scheme with an example. Consider the same sequence that was used earlier,

\[(A, 1), (B, 2), (A, 3), (B, 4), (C, 5), (B, 6), (C, 8)\].

Algorithm B will count two non-interleaved occurrences – the first with events \((A, 1), (B, 2), (C, 5)\), and the second with events \((A, 3), (B, 6), (C, 8)\). Note that, at time \(t = 4\), a second instance of the automaton for \((A \rightarrow B \rightarrow C)\) does not transit into state 2, since there is already one in that state due to the event \((B, 2)\). This second instance has to wait till \(t = 6\), by which time the earlier instance has transited to state 3.

It may be noted that there may be many sets of non-interleaved occurrences of an episode in the event sequence. This algorithm counts that set of non-interleaved occurrences, which includes the first occurrence of the episode in the event sequence.

**Algorithm B: Non-interleaved occurrence count**

**Input:** Set \(C\) of (candidate) episodes, event stream \(s = \{(E_1, t_1), \ldots, (E_n, t_n)\}\), frequency threshold \(\lambda_{\text{min}}\)

**Output:** The set \(\mathcal{F}\) of frequent episodes in \(C\)

1: /* Initializations */
2: Initialize \(\text{bag} = \phi\)
3: for all event types \(A\) do

4: Initialize \(\text{waits}(A) = \phi\)
5: for all \(\alpha \in C\) do

6: Add \((\alpha, 1)\) to \(\text{waits}(\alpha[j + 1])\)
7: Initialize \(\alpha.freq = 0\)
8: for \(j = 1\) to \(|\alpha|\) do
9: Initialize \(\alpha.init[j] = 0\)
10: /* Data Pass */
11: for \(i = 1\) to \(|s|\) do
12: for all \((\alpha, j) \in \text{waits}(E_i)\) do
13: if \(j = 1\) then
14: /* Initialize \(\alpha\) automaton */
15: Update \(\alpha.init[1] = t_i\)
16: Set \(\text{transformation} = 1\)
17: else
18: if \((t \neq \alpha.init[j - 1]) \&\&(\alpha.init[j] = 0)\) then
19: /* Transit \(\alpha\) automaton to state \(j\) */
20: Update \(\alpha.init[j] = \alpha.init[j - 1]\)
21: Set \(\text{transformation} = 1\)
22: Reset \(\alpha.init[j - 1] = 0\)
23: Remove \((\alpha, j)\) from \(\text{waits}(E_i)\)
24: if \(j < |\alpha|\) \&\& \(\text{transformation} = 1\) then
25: if \((\alpha[j + 1] = E_i)\) then
26: Add \((\alpha, j + 1)\) to \(\text{bag}\)
27: else
28: Add \((\alpha, j + 1)\) to \(\text{waits}(\alpha[j + 1])\)
29: if \(j = |\alpha|\) \&\& \(\text{transformation} = 1\) then
30: /* Recognize occurrence of \(\alpha\) */
31: Update \(\alpha.freq = \alpha.freq + 1\)
32: Reset \(\alpha.init[j] = 0\)
33: Empty \(\text{bag}\) into \(\text{waits}(E_i)\)
34: /* Output */
35: for all \(\alpha \in C\) such that \(\alpha.freq \geq n\lambda_{\text{min}}\) do
36: Add \(\alpha\) to the output \(\mathcal{F}\)

3. COUNTING GENERALIZED EPISODES

As was mentioned earlier in Sec. 1, the framework of frequent episode discovery has been generalized to incorporate time durations into the episode definitions [2]. In this generalized framework, the input data is a sequence of event types with (not one, but) two time stamps associated with it – a start time and an end time. Now, an episode is defined to be an ordered collection of nodes, with each node labelled not just with an event type but also with one or more time intervals. Thus, an episode in this new framework will be said to have occurred in an event sequence if the event types associated with the episode occur in the same order in the event sequence and if the duration (or dwelling time) of each corresponding event belongs to one of the intervals associated with the concerned node in the episode. For a complete treatment of the extended formalism the reader is referred to [2].

In this section, we will discuss how the new frequency counts presented in Sec. 2 extend to the case of these generalized episodes as well.

Introducing dwelling times into the episode definition does not alter the overall structure of the frequency counting algorithms. The same general counting strategy as earlier is used. Finite state automata are employed for keeping track of episodes that have occurred partially in the data while se-
Algorithm A1: Non-overlapping occurrence count for generalized episodes

**Input:** Set $C$ of (candidate) episodes, event stream $s = \langle E_1, t_1, \tau_1 \rangle, \ldots, \langle E_n, t_n, \tau_n \rangle$, frequency threshold $\lambda_{\text{min}}$, set $B$ of allowed time intervals

**Output:** The set $F$ of frequent episodes in $C$

```
1: /* Initializations */
2: Initialize bag = φ
3: for all event types $A$ and time intervals $\delta$ do
4: Initialize waits($A, \delta$) = φ
5: for all $\alpha \in C$ do
6: Add $(\alpha, 1)$ to waits($\alpha.g[1], \delta$) $\forall \delta \in \alpha.d[1]
7: Initialize $\alpha.freq = 0$
8: for $j = 1$ to $|\alpha|$ do
9: Initialize $\alpha.init[j] = 0$
10: /* Data Pass */
11: for $i = 1$ to $|s|$ do
12: Choose $D \in B$ s.t. $D.left \leq (\tau_i - t_i) \leq D.right$
13: for all $(\alpha, j) \in \text{waits}(E_i, D)$ do
14: if $j = 1$ then
15: /* Initialize $\alpha$ automaton */
16: Update $\alpha.init[1] = t_i$
17: else
18: /* Transit $\alpha$ automaton to state $j$ */
19: Update $\alpha.init[j] = \alpha.init[j - 1]$
20: Reset $\alpha.init[j - 1] = 0$
21: Remove $(\alpha, j)$ from waits($E_i, \delta$) $\forall \delta \in \alpha.d[j]$
22: if $j < |\alpha|$ then
23: if $\alpha.g[j + 1] = E_i$ and $D \in \alpha.d[j + 1]$ then
24: Add $(\alpha, j + 1)$ to bag
25: Add $(\alpha, j + 1)$ to waits($\alpha.g[j + 1], \delta$)
26: else
27: Add $(\alpha, j + 1)$ to waits($\alpha.g[j + 1], \delta$)
28: if $j = |\alpha|$ then
29: /* Recognize occurrence of $\alpha$ */
30: Update $\alpha.freq = \alpha.freq + 1$
31: Reset $\alpha.init[|\alpha|] = 0$
32: /* Remove partial occurrences of $\alpha$ */
33: for all $1 \leq k < |\alpha|$ do
34: Reset $\alpha.init[k] = 0$
35: Remove $(\alpha, k + 1)$ from waits($\alpha.g[k + 1], \delta$)
36: Remove $(\alpha, k + 1)$ from bag
37: Empty bag into waits($E_i, D$
38: /* Output */
39: for all $\alpha \in C$ such that $\alpha.freq \geq n\lambda_{\text{min}}$ do
40: Add $\alpha$ to the output $F$
```

4. RESULTS

This section presents some results obtained with the new frequency counting algorithms. We quote results obtained on both synthetically generated data as well as manufacturing plant data from General Motors.

The synthetic data is generated as some random time series containing temporal patterns embedded in noise. The temporal patterns that were picked up by our algorithms correlated very well with those that were used to generate the data even when these patterns are embedded in varying amounts of noise. Also, the set of frequent episodes discovered is essentially the same as that discovered using the algorithms form [1]. We also show that our algorithms results in a speed-up by a factor of 7 or more.

4.1 Synthetic data generation

Each of the temporal patterns to be embedded consists of a specific ordered sequence of events. A few such temporal patterns are specified as input to the data generation process which proceeds as is 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 start time. Further, another random integer is generated which is then associated as this event’s dwelling time. The end time of the event is now computed by adding the start and dwelling times and the current time is set to equal this end time. 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 \( \rho \) 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 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 and our expiry time constraints (in the counting algorithm), episodes such as \( B \rightarrow C \rightarrow D \rightarrow A \) can also become frequent.

### 4.2 Results on synthetic data

In this section we present a few of our simulation results to compare our algorithms (i.e., Algorithm A and Algorithm B) with the windows-based frequency count of [1] (which is referred to as Algorithm C in the tables below).

We first demonstrate the effectiveness of our algorithms by comparing the frequent episodes discovered when the event sequence is \( iid \) with those when we embed some patterns. Suppose we take \( \rho = 1 \). Then event types and dwelling times are chosen randomly from a uniform distribution. Hence we expect any sequence of, e.g., two events to be as frequent in the data as any other sequence of two events. Thus, if we are considering all 2-node episodes then most of them would have similar frequencies. If we increase the frequency threshold starting from a low value, initially most of the episodes would be frequent and, after some critical threshold, most of them would not be frequent. Now suppose we embed a few temporal patterns. Then some of its permutations and all their subepisodes would have much higher frequencies than other episodes. Hence if we plot the number of frequent (principle) episodes found versus frequency threshold, then, in the \( iid \) case we should see a sudden drop in the graph while in the case of data with embedded patterns, the graph should level off. Fig. 1 shows the plot of the number of 2-node frequent episodes discovered by the algorithm versus the frequency threshold in the two cases of \( iid \) data and biased data with the number of non-overlapping occurrences being the frequency count. Fig. 2 shows the same things for non-interleaved occurrences frequency count. For the biased data we put in two 4-node patterns so that sufficient number of 2-node episodes would be frequent. In both the graphs, the sudden transition in case of \( iid \) event sequences is very evident. Similar results were obtained for longer sequences as well. It is noted here that in the iid case, since all permutations of events are equally likely, the number of frequent episodes (that meet a low frequency threshold criterion) is much higher than in biased data.

In the next experiment we describe, data was generated by embedding two patterns in varying degrees of \( iid \) noise. The two patterns that we embedded, are as follows: (1) \( \alpha = (B \rightarrow C \rightarrow D \rightarrow E) \) and (2) \( \beta = (I \rightarrow J \rightarrow A) \). Data sequences with 5000 events each were generated for different values of \( \rho \), namely \( \rho = 0.0, 0.2, 0.3, 0.4 \) & 0.5. The objective is to see whether these two patterns indeed appeared among the set of frequent episodes discovered, and if so, at what position. Since \( \alpha \) is a 4-node pattern and \( \beta \) is a 3-node pattern, their respective positions (referred to as their ranks) in the (frequency) sorted 3-node and 4-node frequent episode sets discovered are shown in Tables 1–2. As can be seen from the tables, our frequency counts are as effective as the windows-based frequency proposed in [1].

We next compare the time complexity of the different algorithms by looking at the overall run times for frequent episode discovery. We have taken data sequences of length 5000 and chosen the frequency thresholds so as to get roughly the same number (around 50) of 4-node frequent episodes. We have varied \( \rho \) from 0.0 to 0.5. These results are shown in Table 3. The last column of the table gives the speed up achieved by Algorithm A in comparison with Algorithm C.

It is seen from the table that the algorithms based on counting the number of non-overlapping occurrences and the number of non-interleaved occurrences, run much faster than the windows-based frequency counting algorithm. This advantage comes from the fact that we are counting occurrences and not windows. In the windows-based count, in addition


<table>
<thead>
<tr>
<th>(\rho)</th>
<th>Algo A</th>
<th>Algo B</th>
<th>Algo C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.4</td>
<td>1</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>19</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Rank of \(\alpha\) in sorted 4-node frequent episodes set

<table>
<thead>
<tr>
<th>(\rho)</th>
<th>Algo A</th>
<th>Algo B</th>
<th>Algo C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0.2</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>0.3</td>
<td>6</td>
<td>29</td>
<td>5</td>
</tr>
<tr>
<td>0.4</td>
<td>5</td>
<td>27</td>
<td>7</td>
</tr>
<tr>
<td>0.5</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2: Rank of \(\beta\) in sorted 3-node frequent episodes set

to keeping track of new events that enter each sliding window, one also has to keep track of any events falling out of it at its left extremity. This has an additional temporary memory overhead as well since we need a list that stores for each time the automata which make transitions at that particular time. Moreover, this necessitates checking for new events and events that fall out at every time tick. In contrast, our occurrence-based counts need to act only every time a new event occurs in the sequence. This property translates to major run-time gains if the number of events is much less than the actual time span of the data sequence.

Also, the noise in the data, per se, has no effect on the runtime as can be expected from the algorithms. Through another set of simulations, it is observed that the run times for all three frequency counts increase roughly at the same rate with the number of events in the data sequence. This is shown in Table 4 where run times for data of length 5000 and 20000 events are compared. All the run times seem to scale roughly linearly with the data length (which is what we expect from the algorithms).

### 4.3 Results on GM data

This section describes some experiments on GM data. This data pertains to stamping plants that make various body parts of cars. Each plant has one or more stamping lines. The data is the time-stamped logs of the status of these lines. Each event is described by some breakdown codes when the line is stopped due to some problem or by a code to indicate the running status. The objective of analysis is to find frequent episodes that can throw light on frequent co-occurring faults.

In Section 4.2 we demonstrated how the graphs for the number of episodes discovered versus frequency threshold falls in characteristically different ways for iid data and data with some patterns embedded in them. We can use these to ask whether the GM data has any patterns of interest at all. In order to do this we plotted the number of frequent episodes discovered (as a function of frequency threshold) on the GM data. The number of different breakdown codes in these sequences were roughly around 25. So, for comparison we also plot the number of frequent episodes obtained on an iid sequence with 25 event types of length 50000 (The lengths of data slices from GM data we analyzed were also of the same order.). As before, plots were obtained for both the frequency counts. Fig. 3 gives the graph for the frequency count based on non-overlapped occurrences and Fig. 4 gives the graphs based on non-interleaved occurrences. These plots closely resemble the earlier plots on simulated data i.e. Fig. 1 and Fig. 2. Again this experiment was repeated for episodes of larger sizes too. We may infer from these that the GM data on which the algorithms were run indeed contained patterns with some strong temporal correlations quite unlike the case of iid data.

Our new frequency counts made fast exploration of the large data sets from GM plants feasible and some interesting temporal patterns were obtained.

### 4.4 Conclusions

From the results described, we can conclude that in all cases, our frequency counting algorithms result in a speed-up by a factor of 7 while delivering similar quality of output. It

<table>
<thead>
<tr>
<th>Length</th>
<th>Algo A</th>
<th>Algo B</th>
<th>Algo C</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>22</td>
<td>22</td>
<td>156</td>
<td>7.1</td>
</tr>
<tr>
<td>20000</td>
<td>87</td>
<td>117</td>
<td>625</td>
<td>7.2</td>
</tr>
</tbody>
</table>

Table 4: Run-times (in seconds) for different sequence lengths

Figure 3: Effect of frequency threshold (Non-overlapping occurrences) using GM data: 2-node episodes
may be recalled that all three frequency counts ensure that there is at most one automaton per state per episode during the frequency counting process. Thus, the space complexity of the three algorithms is of the same order. However, in terms of actual amount of temporary storage needed, the windows-based frequency count is a little more expensive because it has to keep the list beginsat which is not needed in the other two algorithms (See [1] for the details of why this list is needed in the windows-based frequency count).

5. DISCUSSION
Mining of interesting temporal patterns from data which is in the form of time series of events is an important data-mining problem. The framework of frequent episodes introduced in [1] is very useful for this purposes. In this paper we proposed two attractive alternatives to the somewhat non-intuitive windows-based frequency measure of [1].

Unlike in the static data-mining scenario, whether or not a temporal pattern occurs cannot be ascertained by looking at only one record at a time in a memoryless fashion. Hence for recognizing the occurrence of episodes we need finite state automata. In this context we have explained why it would be very inefficient if we want to count all occurrences of episodes. Based on the insight gained, we suggested two possible ways to restrict counting to only some specialized occurrences. Both of these frequency counts can be achieved with a reasonable and fixed number of automata per episode so that the space complexity is controlled. In particular, the number of automata needed here are the same as those needed in the windows-based count proposed in [1]. It is also shown through simulations that both these frequency measures are much more efficient in terms of the time taken and they are just as effective in discovering frequent episodes as the windows-based frequency counting algorithm. Thus, these new algorithms make the framework of frequent episode discovery attractive in many more applications.

6. REFERENCES