An Efficient Approach for Maintaining and Mining Frequent Episodes

Show-Jane Yen
Dept. of Computer Science & Information Engineering
Ming Chuan University
Taoyuan, Taiwan
sjyen@mail.mcu.edu.tw

Yue-Shi Lee
Dept. of Computer Science & Information Engineering
Ming Chuan University
Taoyuan, Taiwan
leeys@mail.mcu.edu.tw

***Abstract*—Data mining technology plays a crucial role in the area of data analysis. Mining frequent episodes stands out as a pivotal task within this domain, enabling users to forecast future events based on present occurrences. Conventional methods for discovering frequent episodes typically follow a hierarchical approach, involving the generation of candidate episodes and subsequent scanning of sequence data to count their frequency, which can be quite time-consuming, as it necessitates repeated scans of the sequence data and the search for candidate episodes.**

In this paper, we introduce a novel approach for episode mining in a data stream. Our method distinguishes itself by scanning newly added data to update existing frequent episodes, all without the need for scanning the original data or searching for candidate episodes. The experiments also show that our approach is more efficient compared to other existing methods.

Keywords—data mining, frequent episode, data stream

I. INTRODUCTION

Data mining aims to extract valuable insights from vast datasets. Mining frequent episodes [2, 3, 4, 5, 6, 7, 8, 9] involves the identification of frequent patterns in a series of events. Specifically, it seeks to determine the likelihood of one event occurring following another. This capability allows us to make predictions when an event occurs. For instance, in network error detection, frequent episodes can be employed to anticipate error occurrences. Likewise, in meteorology, they can aid in forecasting weather changes.

An event sequence, as depicted in Fig.1, consists of event types denoted by letters and the corresponding time points represented by numbers. An event sequence can be recorded as a combination of event types and their occurrence times. For example, the event sequence shown in Figure 1 can be recorded as (A,25)(B,27)(C,28)(A,30)(C,31). Therefore, an event sequence is represented as $\langle(a_1,t_1)\ (a_2,t_2)\ldots(a_n,t_n)\rangle$ $(n\geq 1)$, where $t_1<t_2<\ldots<t_n$ and $a_i\in E$ $(1\leq i\leq n)$, with E being the set of all possible events.

A sequence of events occurring in a specific order is referred to as an *episode*, which can be categorized into two main types: *Serial episodes* and *Parallel episodes*. In the case of serial episodes, the events are temporally related to one another. For instance, in Figure 2 (a), event B occurs after event A and before event C, denoted as <ABC>. Consequently, the two serial episodes <ABC> and <CAB> are distinct from each other. In contrast, parallel episodes involve events that have no temporal sequence relationship. Figure 2 (b) serves as an illustration of a parallel episode, in which all events occur without regard to their order of occurrence. This is represented as (ABC). Therefore, the two parallel episodes (ABC) and (CAB) are considered identical. The *length of an episode* refers to the number of events within the episode. For example, the length of the episode <ABC> is 3.

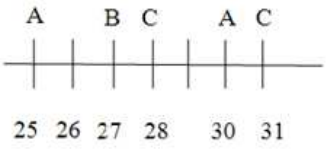

Fig.1 Example of event sequence

(a) Serial episode (b) parallel episode

Fig.2 Examples of episodes

The occurrences of an episode encompass the time span extending from the beginning to the end of the events comprising the episode [2, 5, 7, 8]. To illustrate, in Fig.1, the serial episode <AC> manifests two occurrences: (25, 28) and (30, 31), respectively. When considering any two occurrences for an episode E, which denote them as O1=(t_1, s_1) and O2=(t_2, s_2), if $t_1\leq t_2$ and $s_1>s_2$ or $t_1<t_2$ and $s_1\geq s_2$, it signifies that occurrence O1 contains occurrence O2. This suggests that the relationship between the events within E is comparatively distant and weak in occurrence O1, while the relationship between the events in occurrence O2 is closer and stronger. Therefore, O2 provides a more representative depiction of the temporal range of the events within E than O1.

If a particular occurrence, denoted as MO, of episode E does not contain any other occurrences of E, it is referred to as a *minimal occurrence* of E and is considered the most representative occurrence. For instance, in Fig.3, the minimal occurrence for the episode <BC> is (25, 27), which stands alone without containing any other occurrences. On the contrary, (21, 27) does not qualify as a minimal occurrence because it includes another occurrence (25, 27). Consequently,

there exist two distinct minimal occurrences for the episode <BC>: (25, 27) and (30, 33).

When the count of minimal occurrences for an episode in an event sequence satisfies a user-defined *min_count* threshold, the episode is called a *frequent episode* [2, 7, 8]. In the event sequence shown in Fig.3, suppose the min_count threshold is set to 2, <BC> is a frequent episode, which enables us to anticipate the occurrence of event C when event B occurs.

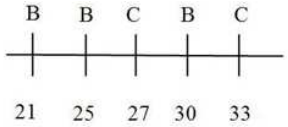

Fig. 3 Another example of event sequence

Given that events will persistently occur, the events within an event sequence will steadily accumulate over time, and the set of frequent episodes will evolve accordingly. To stay current with the latest frequent episodes, it becomes imperative to promptly identify these episodes as new events are added. Nevertheless, traditional methods for mining frequent episodes [2, 5, 6, 7, 8] entail the need to rediscover existing frequent episodes each time new events are introduced. This results in a considerable waste of time and undermines overall efficiency. Therefore, we explore methods for updating the existing episodes in response to newly added events in a data stream, obviating the necessity to revisit the original event sequence.

This paper presents a novel algorithm designed to swiftly identify the most recent frequent episodes as new events are introduced into the event sequence. Our approach employs a tree structure, housing each event in the initial-level nodes while reserving other nodes for the storage of frequent episodes. Upon adding an event, if the last event within a frequent episode matches the newly introduced event, an update action is triggered. By exclusively storing and handling frequent episodes and new events, our method not only accelerates the mining process but also substantially reduces memory consumption.

II. RELATED WORK

Mannila et al. [7] employed a sliding window approach to progressively extract frequent episodes from a sequence of events using the Apriori framework. To begin, the user needs to specify an appropriate window size to prevent excessive time gaps between events within a frequent episode. Following this, candidate episodes that encompass a single event can be generated. If the count of windows containing a candidate pattern reaches a user-defined threshold, it is identified as a frequent episode. To generate candidate episodes comprising k events, they are formed by combining with the frequent episodes that encompass k-1 events.

Mannila et al. [8] have expanded and refined their prior approach, introducing a novel algorithm that distinguishes between the extraction of serial episodes and parallel episodes. In the context of serial episodes, a state table is utilized to verify the presence of candidate episodes within the window. This is necessitated by the sequential relationships between events within a serial episode. Consequently, there are certain events that are initially awaited. To illustrate, consider the serial episode <ABC>. In this algorithm, the system first waits for the appearance of event A. As the window shifts, if the newly added event is A, it signifies the emergence of event A, and the system proceeds to anticipate event B. The arrival of the final event, C, signals that the window now contains <ABC>.

Conversely, in the case of parallel episodes, it is only imperative to count the number of events within the episode since there are no temporal dependencies among the events in the episode. However, it is essential to allocate a substantial amount of memory space for concurrently storing numerous state tables. Additionally, as this algorithm is based on the Apriori algorithm [1], it necessitates revisiting the original event sequence each time a new event is added. Furthermore, the window size has to be specified prior to frequent episode mining from an event sequence. When the window size is adjusted, it demands the re-mining of frequent episodes, such that this algorithm exhibits inefficiencies.

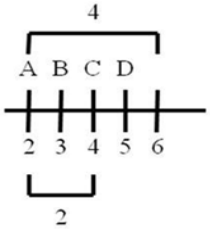

Fig. 4 Example of different window size

Casa-Garriga, et al. [2], introduced an innovative approach aimed at addressing the limitation of the previous method which necessitated the determination of a fixed window size. This constraint resulted in an inability to identify frequent episodes exceeding the length of the window. For instance, in Fig.4, if the window size is set at 4 time units, it allows the detection of the episode <ABCD>. Conversely, when the window size is restricted to 2 time units, the episode <ABCD> cannot be discovered. To overcome this challenge, they proposed the concept of an unbounded episode, incorporating user-defined time units between adjacent events. For instance, Fig.5 illustrates a serial episode <ABC> with a length of 3 and a window size of 2 time units. For a serial episode with a length k, the window size is extended to k-1 time units. As a result, the window size expands as the increasing length of frequent episodes. This adaptive window size ensures that frequent episodes are not missed due to the window size constraint. However, it's important to note that despite solving the window size issue, this method retains the Apriori framework's characteristics, which entail the substantial storage of candidate episodes and the need to re-examine the event sequence as new events are added.

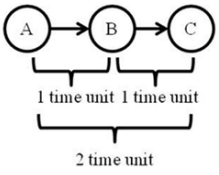

Fig.5 An example of unbounded episode

Mannila et al. [6] introduced the MINEPI algorithm, a pioneering approach that introduces the concept of minimal occurrence for the extraction of frequent episodes. Unlike the sliding window method, MINEPI is based on the Apriori algorithm [1], but it deviates from the process of initially defining a window size and then counting the number of windows containing the candidate episode within the event sequence. Instead, MINEPI directly computes the number of minimal occurrences within an event sequence.

However, it's important to note that each minimal occurrence for a candidate episode in an event sequence comes with distinct start and end times. This leads to the generation of numerous minimal occurrences for a single candidate episode, demanding a significant amount of memory space for storage. Additionally, MINEPI doesn't set a threshold to restrict the maximum duration, potentially resulting in extensive time intervals for minimal occurrences of a candidate episode. For instance, consider the example of <AB>: (5,105), where it indicates that after event A occurs at the fifth time point, event B follows after 100 time units. This extended time gap between the two events may not necessarily imply a meaningful relationship, but MINEPI includes such minimal occurrences in its calculations.

Xi Ma and colleagues [5] introduced the Episode Prefix Tree (EPT) and Position Pair Set (PPS) as improvement to the MINEPI algorithm which needs the storage of numerous candidate episodes. These two methods, EPT and PPS, eliminate the need to generate candidate episodes by focusing on mining frequent episodes based on episode prefixes. To illustrate, let's consider an event sequence depicted in Fig.6. For instance, if the prefix is <A>, the episodes and their occurrences within the event sequence are as follows: <AB>: (25,26), <AD>: (25,27), <AC>: (25,28), (25,31), (30,31), and <AA>: (25,30). Notably, occurrence (25,31) contains the occurrence (25,28) for the episode <AC>, therefore it non-minimal for this episode. Similarly, when the prefix is <AD>, the episodes and their occurrences in the event sequence are as follows: <ACD>: (25,28), (25,31) and <ADA>: (25,30). In this case, occurrence (25,31) contains occurrence (25,28) for the episode <ACD>, signifying it as a non-minimal occurrence for <ACD>.

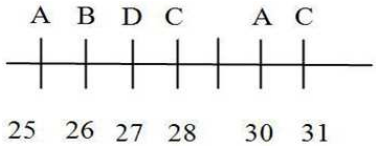

Fig. 6 Another example of event sequence

The EPT algorithm employs a prefix tree structure to extract frequent episodes, where each node in the tree keeps track of an event and the count of minimal occurrences for each episode. In the Episode Prefix Tree, if the number of minimal occurrences for the final node in a particular path fails to meet a user-specified threshold, then there's no need to further extend that path. This is because it's unable to discover additional frequent episodes within that specific path. The notable advantage of the EPT algorithm is that it eliminates the necessity to generate candidate episodes, leading to reduced memory consumption. However, it is necessary to repeatedly scan the event sequence when a path requires extending to a longer path.

Therefore, Xi Ma and colleagues [5] introduced an alternative method known as PPS to address the issue of repetitive event sequence scanning posed by the EPT algorithm. PPS initially identifies frequent episodes of length 1 within the event sequence in Fig.6, such as <A>, <B>, <C>, and <D>. These identified frequent episodes are then utilized as prefixes, combined with other frequent episodes of length 1. For instance, episodes <AA>, <AB>, <AC>, and <AD> are generated through this process. Employing a depth-first approach, each frequent episode of length k-1 serves as a prefix, and in combination with all frequent events to generate episodes of length k. If the count of minimal occurrences for a generated episode fails to meet the specified threshold, it is deemed non-frequent and subsequently eliminated. However, PPS encounters the challenge of continuous re-scanning of the event sequence if new events are continually added, which may hinder its ability to efficiently generate frequent episodes.

Mielikäinen [4] introduced a technique for mining frequent episodes in scenarios where events are continuously appended to the event sequence. This algorithm has the capability to dynamically update existing episodes upon the addition of an event. However, it does not take into account the concept of minimal occurrences. This method amalgamates all the existing episodes with the newly added event. For instance, if the current episodes comprise <A>, <C>, and <AC>, the addition of event B results in the generation of new episodes like <AB>, <CB>, and <ACB>. Nevertheless, when events are added continuously, this approach may lead to the storage of a substantial number of episodes, which can result in space wastage and a considerable drop in efficiency.

III. Our Approach

In this section, we describe our proposed algorithm SFET. The main storage structure of SFET is the Suffix Frequent Episode Tree (SFE Tree) which store all frequent episodes so far and their minimal occurrences. Starting from the root of the tree, there will be zero or more child nodes under each node. The child nodes of the root node store all the events in the event sequence, and each other node represents a frequent episode in which the order of the events is from the node to root node. In Fig. 7, there is node A under the root node and nodes B and C under node A, where node B represents frequent episode <BA> and node C represents frequent episode <CA>.

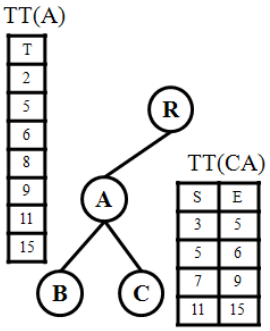

Fig. 7 The structure of SFE Tree

In the structure of SFE Tree, each event and frequent episode have a TT (Time Table) structure. The two fields S and E in TT(N) of a node N representing a frequent episode X record start time and end time of the minimal occurrences of the frequent episode X, respectively. The TT(e) of each event e has a field T which records the time point of the occurrence of the event e. Fig. 7 shows an example of TT structure of SFE Tree. In addition, each node also records the current number of minimal occurrences of the frequent episode in the node N, which is denoted as count(N).

When an event e is added to the event sequence and time point is t, if e does not exist in the child node of the tree root, SFET creates a child node e of the root, records time t in TT(e) of the node and set count(e) to 1. Otherwise, SFET adds time t to the time table TT(e) and accumulate count(e) by 1. If count(e) is equal to min_count, then e is a frequent event and e is combined with each frequent event x to form a candidate episode <xe> and the minimal occurrences are generated. The minimal occurrence generation method is as follows: For each time point t in TT(e), search from TT(x) to find the set of time points which is smaller than t and retrieve the largest time point t' in the set. Therefore, time point t' and t are the start time and end time of a minimal occurrence of <xe>, and count(xe) is accumulate by 1. If count(e) is greater than min_count, then x may already be a child node of e, that is, <xe> may already be a frequent episode. If x is already a child node of e, then retrieve the values u and v of S and E of the last record in TT(xe), and apply the minimal occurrence generation method to update TT(xe) from value u of TT(x) and value v of TT(e).

After generating all the minimal occurrences of <xe>, if count (xe)=min_count, then <xe> is a frequent episode. SFET creates child node x under node e and TT(xe), and retrieves the frequent episode by traversing SFE Tree from the child node x of root to the child node y of node x, that is <yx>. The two frequent episodes <yx> and <xe> can be joined to generate a length 3 candidate episode <yxe>, and TT(yxe) is generated as follows: For each time point t in TT(xe), search from the field E of TT(yx) to find the set of time points which is smaller than t and retrieve the largest time point p in the set. Suppose the corresponding start time of p in TT(yx) is t'. (t', t) is a minimal occurrence of <yxe>, and count(yxe) is accumulated by 1.

If count(xe)>min_count and there is no child node y under node x of <xe>, which means that <yxe> is not a frequent episode, SFET calculate minimal occurrences of <yxe> from TT(xe) and TT(yx) as above. If count(xe)>min_count and there is child node y under node x of <xe>, that is, <yxe> is already a frequent episode, then SFET does not re-generate TT(yxe) but updates original TT(yxe) as follows: SFET retrieves the two time points t_s and t_e from the two fields S and E of the last record in TT(yxe), and starts from the next record of t_s in field S of TT(yx) and the next record of t_e in field E of TT(xe) to calculate minimal occurrences of <yxe> and added to TT(yxe) according to the minimal occurrence generation method.

If <yxe> is a frequent episode, then SFET searches for length 3 frequent episodes ending with <yx> from SFE Tree, that is a frequent episode <zyx> formed by following each child node z of node y in the path x→y under the root, which can be joined with frequent episode <yxe> to generate a candidate episode <zyxe> whose minimal occurrences also can be generated as above. For a length k (k≥3) frequent episode <e_1 e_2… e_k>, SFET retrieves a frequent episode <y e_1 e_2…e_{k-1}> by following the path $e_{k-1}\rightarrow e_{k-2}\rightarrow\ldots\rightarrow e_2\rightarrow e_1$ under the root node and find the child node y of e_1, and the two length k frequent episodes can be joined to generate a candidate episode <y e_1 e_2… e_k>. SFET generate frequent episodes by the above way until there is no frequent episodes can be generated.

In the following, we illustrate our SFET algorithm by using a simple example. Assume that the event sequence is S = <BCABAB> and min_count = 2. When events B, C and A at time points 1, 2 and 3 arrive, the nodes B, C and A and their time table (TT) are created under the root node of SFE Tree. Since the count of the three events do not satisfy min_count, there is nothing to do. When the event B at time point 4 arrives, time point 4 is added to TT(B) and count(B)=min_count, which event B is a frequent event. Since current frequent event is only B, event B and event B can be joined to generate candidate episode <BB>, which is shown in Fig. 8.

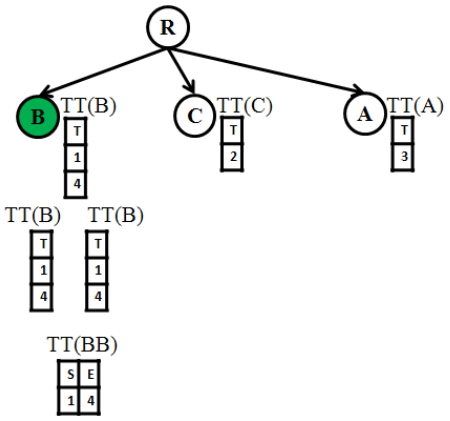

Fig. 8 SFE Tree after processing <BCAB> of event sequence S

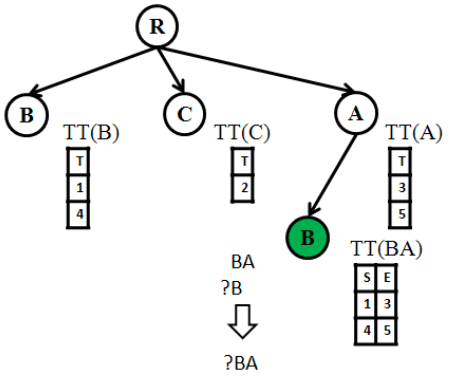

Fig.9 SFE Tree after event A arrives at time point 5

When event A in even sequence S at time point 5 arrives, time point 5 is added to TT(A) and event A is a frequent event since count(A) is equal to min_count 2. The two frequent events A and B can be joined to generate candidate episodes

<AA> and <BA>, and their minimal occurrences are {(3, 5)} and {(1, 3)(4, 5)}, respectively, in which <BA> is a frequent episode since count(BA) is equal to min_count 2. Therefore, SFET creates child B of node A and TT(BA), and then searches for child node of node B under the root node. Because there is no child node of node B, that is, there is no frequent episode ending with event B that can be joined with <BA>, which is shown in Fig. 9.

When event B in S at time point 6 arrives, which is added to TT(B), because count(B) > min_count and there is no child node of node B under the root, event B is joined with all the frequent events, which are <BB> and <AB> and their time tables are TT(BB)= {(1, 4),(4, 6)} and TT(AB)= {(3, 4), (5, 6)}. Because count(BB) and count(AB) are equal to min_count 2, <BB> and <AB> are newly generated frequent episodes. SFET creates child nodes B and A of the node B under the root node, and searches for length 2 frequent episodes to join with the two frequent episodes <BB> and <AB>. For frequent episode <BB>, because there are two child nodes B and A of node B under the root (Fig.10), that is, there are two frequent episodes <BB> and <AB>, which can be joined with <BB> to generate candidate episodes <BBB> and <ABB> and their minimal occurrences are {(3,6)} and {(4,6)}, respectively.

For frequent episode <AB>, because there is a child node B of node A under the root, that is, there is a frequent episode <BA>, which can be joined with <AB> to generate a candidate episode <BAB> and TT(BAB)={(1,4) (4,6)}. Since count (BAB) is equal to min_count 2, <BAB> is a newly generated frequent episode. Because there is no child node of node B in the path A→B under the root, that is, there is no length 3 frequent episode ending with <BA>, the frequent episode <BAB> cannot joined with the other frequent episodes. Therefore, all the frequent episodes generated from the event sequence S are <BB>, <AB>, <BAB> and <BA>. Fig. 10 shows the SFE Tree after processing all the events in event sequence S.

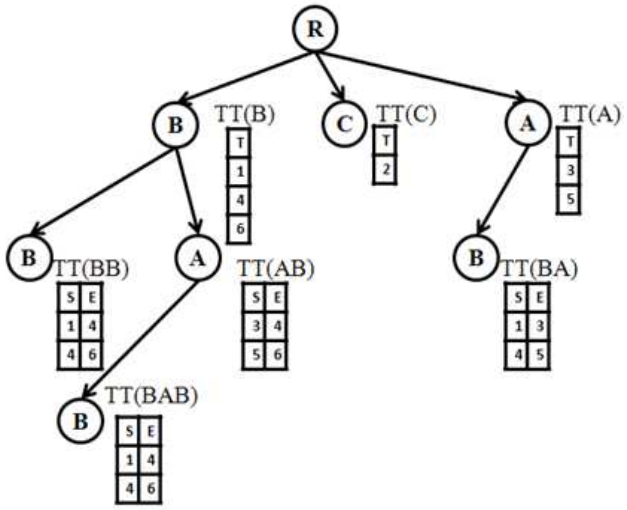

Fig.10 SFE Tree after processing all the events in S

IV. Experiments

We generate synthetic datasets for conducting our experiments. We first defined two parameters: Et, representing the number of events, and S, representing the average number of events in the event sequence. Next, we set the average length, denoted as E, of frequent episodes, and generated all possible frequent episodes in the set C. We first randomly select a frequent episode from the set C and inserting it into the event sequence S. We then made a random decision to either select another frequent episode or repeat the insertion of the same frequent episode into the event sequence. This process continued until the desired event sequence length was achieved. We then proceeded to compare the execution time and memory usage of our algorithm with that of the SE-stream algorithm [4], which is designed to discover frequent episodes within streams of events.

Due to the substantial storage requirements for non-frequent episodes and the generation of a significant number of candidate episodes, the SE-stream algorithm tends to consume excessive memory space and execution time when the event sequence becomes longer. To address this issue in our experiments, we standardized the values. We fixed the number of events (Et) and the potential set of frequent episodes (C) at 200 and 100, respectively, while maintaining the length of the event sequence (S) at 100. Furthermore, we specified varying average lengths for the frequent episodes: 3, 5, and 7. This enabled us to generate three distinct synthetic event sequences, denoted as E3S100, E5S100, and E7S100, respectively.

We initiated the process by selecting the initial 10 events from each of the three event sequences. Subsequently, we augmented the event sequences by adding 10 events at a time, and set a minimum count threshold of 4. Fig. 11, 12 and 13 illustrate the execution times for both algorithms as events were added in increments of 10 on the three datasets. Notably, as the event sequence grew to encompass 60 to 100 events, the execution time for SE-stream exhibited a sudden and substantial increase, which is attributed to a significant upsurge in the number of candidate episodes, leading to an extended search process.

In contrast, our SFET algorithm operates without the need to store candidate episodes, eliminating the necessity for time-consuming candidate searches. SFET's primary task involves updating nodes associated with the newly added events within the tree structure. Our approach ensures that the execution time remains consistent, regardless of any alterations in the event sequence length. As a result, our algorithm consistently outperforms SE-stream and offers superior stability in comparison.

The experiment in Fig.14 clearly demonstrates that SE-stream demands a greater amount of memory space compared to our SEFT algorithm. This discrepancy arises from the fact that SE-stream must accommodate a substantial volume of candidate episodes, whereas our tree structure only stores frequent episodes. Consequently, as the length of the event sequence extends, SE-stream generates an excessive number of candidate episodes. In contrast, SEFT merely requires additional storage space for SFE Tree to maintain frequent episodes, thereby causing the disparity in memory usage

between the two algorithms to widen as the event sequence length increases, which is shown in Fig.14.

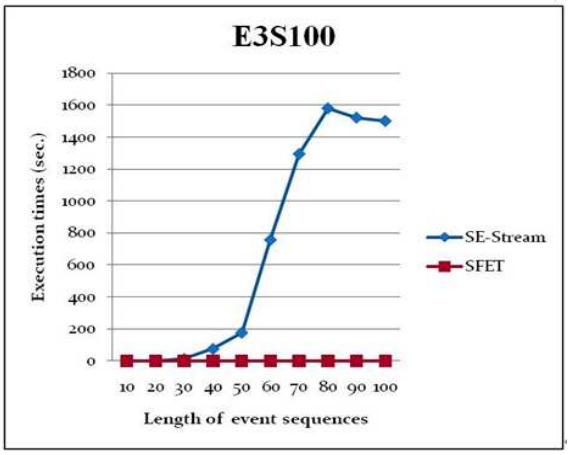

Fig.11 Execution time for the two algorithms on E3S100

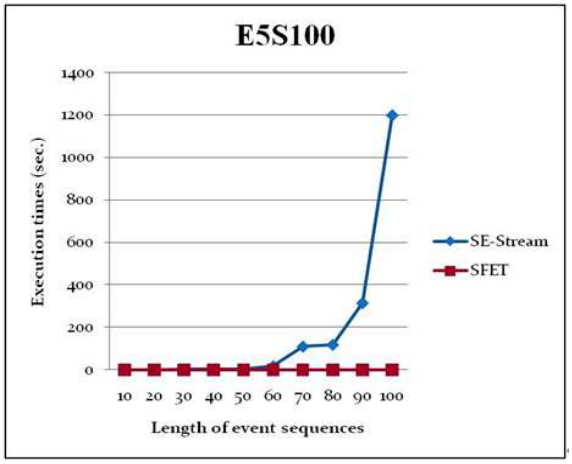

Fig.12 Execution time for the two algorithms on E5S100

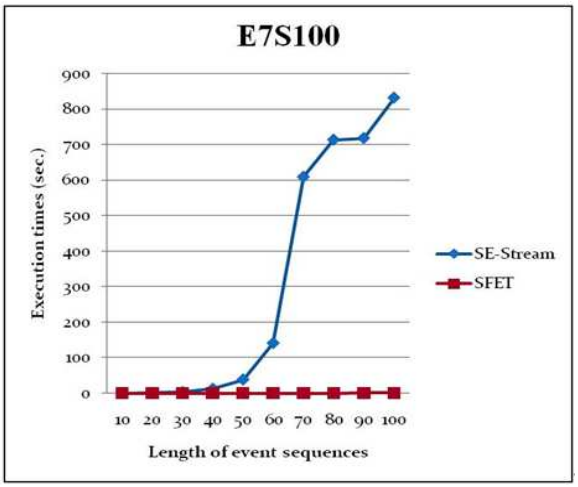

Fig.13 Execution time for the two algorithms on E7S100

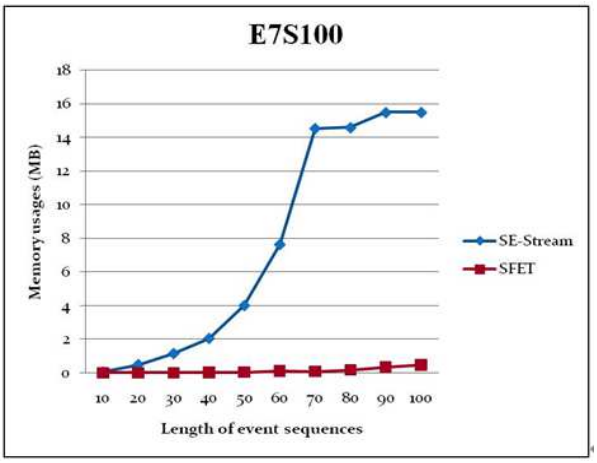

Fig.14 Memory usages for the two algorithms

V. CONCLUSIONS

In this paper, we present a novel algorithm designed for the extraction of frequent episodes when the events are continuously added into event sequence. Our approach stands out from existing algorithms in that it eliminates the need for searching candidate episodes and re-scanning the event sequence, it allows for direct updates to the nodes in our tree structure and generate frequent episodes upon the addition of a new event. The experiments show that our algorithm is remarkable efficiency in terms of both execution time and memory usage.

REFERENCES

[1] R. Agrawal, and R. Srikant, "Fast Algorithms for Mining Association Rules," Proc. of 20th International Conference on Very Large Databases, Santiago, Chile, pp. 487-499, September 1994.

[2] G. Casas-Garriga, Discovering unbounded episodes in sequential data, Knowledge Discovery in Databases:PKDD 2003, volume 2838 of Lecture Notes in Artificial Intelligence, pp. 83–94. Springer-Verlag, 2003.

[3] Wu, Cheng-Wei, et al. "Mining high utility episodes in complex event sequences." Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining. 2013.

[4] T. Mielikäinen, Discovery of serial episodes from streams of events, Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSBDM), pp. 447-448, 2014.

[5] X. Ma, H. PANG, K. L. TAN, Finding constrained frequent episodes using minimal occurrences, Proceedings of the Fourth IEEE International Conference on Data Mining, pp. 471-474, 2004.

[6] H. Mannila and H. Toivonen, Discovering generalized episodes using minimal occurrences, In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96), pp. 146-151, 1996.

[7] H. Mannila, H. Toivonen and A. I. Verkamo, Discovering frequent episodes in sequences, In Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95), pp. 210-215, 1995.

[8] H. Mannila, H. Toivonen and A. I. Verkamo, Discovering frequent episodes in sequences, Data Mining and Knowledge Discovery(DMKD) 1(3): 259-289, November 1997.

[9] Gan, Wensheng, et al. "Utility-driven mining of high utility episodes." 2019 IEEE International Conference on Big Data (Big Data). IEEE, 2019.