Computing (2020) 102:141–200
https://doi.org/10.1007/s00607-019-00734-3

A new efficient approach for extracting the closed episodes for workload prediction in cloud

Maryam Amiri[1] · Leyli Mohammad-Khanli[2] · Raffaela Mirandola[3]

Received: 9 October 2018 / Accepted: 6 June 2019 / Published online: 13 June 2019

Abstract
The prediction of the future workload of applications is an essential step guiding resource provisioning in cloud environments. In our previous works, we proposed two prediction models based on pattern mining. This paper builds on our previous experience and focuses on the issue of time and space complexities of the prediction model. Specifically, it presents a general approach to improve the efficiency of the pattern mining engine, which leads to improving the efficiency of the predictors. The approach is composed of two steps: (1) Firstly, to improve space complexity, redundant occurrences of patterns are defined and algorithms are suggested to identify and omit them. (2) To improve time complexity, a new data structure, called closed pattern backward tree, is presented for mining closed patterns directly. The approach not only improves the efficiency of our predictors, but also can be employed in different fields of pattern mining. The performance of the proposed approach is investigated based on real and synthetic workloads of cloud. The experimental results show that the proposed approach could improve the efficiency of the pattern mining engine significantly in comparison to common methods to extract closed patterns.

Keywords Closed episode · Cloud computing · Prediction · Pattern mining engine · Workload

✉ Leyli Mohammad-Khanli
l-khanli@tabrizu.ac.ir

Maryam Amiri
m-amiri@araku.ac.ir

Raffaela Mirandola
raffaela.mirandola@polimi.it

1 Department of Computer Engineering, Faculty of Engineering, Arak University, Arak 38156-8-8349, Iran

2 Faculty of Electrical and Computer Engineering, University of Tabriz, 29 Bahman Blvd, Tabriz, Iran

3 Dipartimento di Elettronica, Informazione e Bioingegneria Politecnico di Milano, Via Golgi 42, 20133 Milan, Italy

Mathematics Subject Classification 68T10 · 62-07

1 Introduction

Elasticity, one of the prominent features of cloud computing, is the degree of the system adaptability to workload changes by provisioning and deprovisioning resources automatically in a way that the allocated resources match the current demand [1,2]. The future demand prediction is the only practical and effective solution for the fast resources provisioning and the rapid elasticity implementation [3,4]. The most important challenges of the application prediction models are as follows [3]:

- *Complexity* Each prediction model needs computation resources to estimate future behavior of applications. The computation resources consumption of the prediction models should not be significant in comparison with the other applications. So, time and space complexities of the prediction model should be reasonable in a way that its deployment is affordable.
- *Pattern length* In most of the prediction models such as [4–10], the pattern length is fixed. In these models, using a sliding window, the extracted patterns have a predefined length. The constraint of the pattern length restricts the model to specific patterns and prevents the model from learning the other useful patterns. However, choosing the pattern length is a challenge. The pattern length should be selected in a way that the most popular patterns can be extracted and application behavior can be estimated accurately.

In our previous works, based on Sequential Pattern Mining (SPM), we proposed two new prediction models, called POSITING [11] and RELENTING [2]. As Fig. 1 shows, POSITING considers application behavior in the past, extracts behavioral patterns and stores them in the off-line pattern base. Based on the extracted patterns and the recent behavior of the application, POSITING predicts the future demand for different resources. In [2], we developed POSITING with the capability of online learning and proposed RELENTING. While RELENTING predicts the status of all the allocated resources, it also learns the new behavior of the application rapidly without gathering new data and retraining the model.

According to Fig. 1, the pattern mining engine is the core of the predictors. To improve mining efficiency and avoid information loss, a compressed set of patterns, called closed patterns, is extracted by the pattern mining engine [2,11]. A pattern is closed if none of its super-patterns have the same frequency as the pattern's [12]. The common approach for extracting the closed episodes under gap constraints is based on the hash table [12,13]. As our experiment results show this approach is very time-consuming if there are many candidate closed patterns.

The main goal of the paper is to improve time and space complexities of the pattern mining engine in a way that it could be employed in different fields such as [14–17] efficiently. To improve space complexity, we define redundant occurrences of patterns and prove that omitting them causes no information loss. Then, by using a new imProved RepresentatiOn of the Stream based on PointERs ($PROSPER$), we

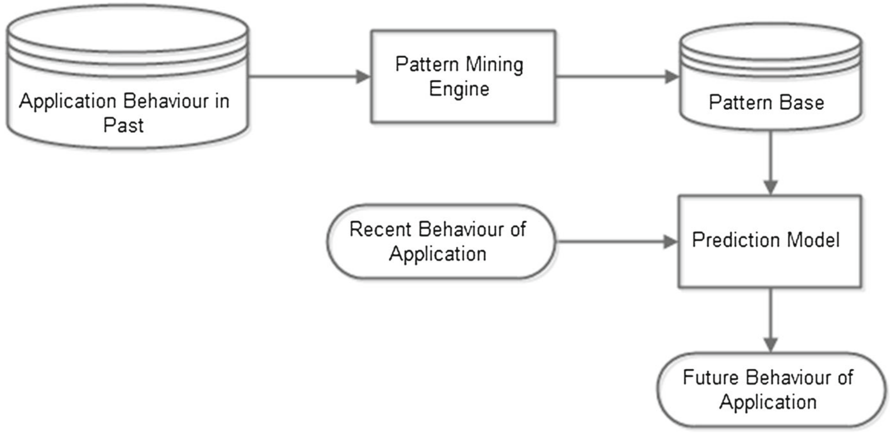


Fig. 1 The scheme of POSITING [11]

develop algorithms proposed in [11] to identify the redundant occurrences and omit them from the occurrence list of patterns. Thus, memory consumption improves. To improve time complexity, we introduce a new data structure, called closed pattern backward tree ($CPBT$) to extract the closed patterns directly. As it will be shown in the experiment results, $CPBT$ improves the efficiency of the pattern mining engine significantly when there are a huge number of patterns. So, the contributions of this paper are as follows:

- Since SPM is widely used in different fields, this paper presents a general approach to improve time and space complexities of the pattern mining engine.
- To improve space complexity, this paper proposes $PROSPER$, defines the redundant occurrences of patterns and suggests algorithms to identify and omit them. Thus, the length of the occurrence list of patterns decreases and memory consumption improves (Sect. 4).
- To improve time complexity, this paper introduces a new data structure, called $CPBT$, to store the closed patterns. Based on $CPBT$, depth-first search algorithms are presented to extract the closed patterns under gap constraints directly without storing/processing all of the candidate closed patterns (Sect. 5).
- The performance of the proposed approach is evaluated using both real and synthetic workloads and compared with the hashing based approach. According to the evaluation results, the proposed approach outperforms the hashing based approach significantly when there are a large number of the candidate closed patterns (Sect. 6).

The rest of the paper is organized as follows: The related works are discussed in Sect. 2. Section 3 introduces the main concepts of POSITING and RELENTING briefly. To improve space complexity, the redundant occurrences of patterns are discussed in Sect. 4. To improve time complexity, Sect. 5 introduces $CPBT$ and presents new algorithms to extract the closed patterns directly. Experimental results are presented in Sect. 6. Finally, the paper is concluded in Sect. 7.

2 Related work

The sequential patterns could capture causative chains of influences present in data. They are useful in many real-life applications [12,18,19] such as the prediction of system failures [20], ICT risk assessment and management [21] and mining web access patterns [22].

Events could be classified into two groups: time-point events and time-interval events. The time-point events are stamped with the time of the event occurrence. On the contrary, the time interval events describe the status of variables in time intervals. An episode is defined as a partially ordered collection of events that occur together [23]. The main goal of episode mining is to find the relationship between events [24].

Most of the research works focus on events stamped with the time-point and extend algorithms to extract their corresponding episodes. The time-interval events are considered in many applications such as health care, data network, financial applications [25,26] and cloud workloads [2,11]. The problem of mining the time-interval episodes is a relatively young research field [27]. Most of the research works ignore the time-interval events and the relationship between them [28]. So presenting algorithms with the capability of learning from such complex data is one of the most important and the most challenging topics in the field of data mining [27,29]. It is clear that mining the time-interval episodes from such data is more complicated [28]. This paper focuses on the time-interval events and proposes a general approach for mining the closed time-interval episodes efficiently. In the following section, we review some research works on the time-interval events briefly.

Winarko et al. [28] propose a new algorithm, ARMADA, for discovering temporal patterns from interval-based data. The authors extend MEMISP (MEMory Indexing for Sequential Pattern mining) [30] to mine frequent patterns. Their algorithm requires one database scan and does not require the candidate generation.

In [31], time-stamped multivariate data are converted into time interval-based abstract concepts by using the temporal abstraction (see Sect. 3.1). An algorithm, called KarmaLego, enumerates all of the patterns whose frequency is above a given support threshold. Moskovitch et al. [32] improve KarmaLego to handle thousands of symbol types.

Batal et al. [27] present Recent Temporal Pattern (RTP) mining to find predictive patterns for the event detection problems. At first, their approach converts the time series data into time-interval sequences of temporal abstractions. Then complex time-interval patterns are constructed by using temporal operators. The mining algorithm explores the space of temporal patterns in the level by level fashion. They also present the minimal predictive recent temporal patterns framework to choose a small set of predictive and non-spurious patterns.

In [33], the abstracted time series is used to find temporal association rules by generalizing Allen's rules [34] into a relation called PRECEDES. The user defines a set of complex patterns, which constitutes the basis of the construction of temporal rules. An Apriori-like algorithm looks for meaningful temporal relationships among the complex patterns.

Patal et al. [35] augment the hierarchical representation with count information to achieve a lossless representation. The hierarchical representation provides a compact mechanism to express the temporal relations among events. Based on this representation, an Apriori-based algorithm, called IEMiner (Interval-based Event Miner), is proposed to discover frequent temporal patterns. IEMiner employs two optimization strategies to reduce the search space. Finally, interval-based temporal patterns are used for classification.

Physiological conditions of patients are reported by using variables such as blood pressure and the heart rate [27,36,37]. Gosh et al. [37] propose an approach that combines sequential patterns extracted from multiple physiological variables and captures interactions between these patterns by Coupled Hidden Markov Models (CHMM).

Laxman et al. [38] present a pattern discovery framework that utilizes event duration information in the temporal data mining process. They incorporate event duration information explicitly into the episode structure and discover event duration-dependent temporal correlations in the data. Furthermore, they define "principal episodes", which are similar to closed episodes, and extract them based on the hashing approach.

As it is observed, most of the research works focus on mining frequent patterns. Mining frequent patterns might lead to extracting a huge number of patterns. To improve mining efficiency and avoid information loss, closed patterns are usually extracted [39]. Most of the works such as [12,39] focus on extracting closed episodes from the sequence of the time-point events. The common approach to extract closed episodes under gap constraints is based on the hash table [12,13]. This approach generates closed patterns under gap constraints in two steps [13]. In the first step, candidate closed patterns are extracted from frequent patterns. In the next step, they are considered and closed patterns are determined by using a hashing procedure with frequency as the key. In this step, all the candidates with the same frequency are hashed to the same bucket in the hash table. Among the candidate patterns which are hashed to the same bucket, those patterns for which a super-pattern with the same frequency is found, are discarded. As our experimental results show, this approach is not appropriate to extract the closed time-interval patterns for the small values of the frequency threshold because it leads to generating a huge number of candidate closed patterns. In this paper, we introduce a new data structure, called $CPBT$, to store the closed pattern. Based on $CPBT$, depth-first search algorithms are proposed to extract the closed patterns directly. As it will be shown in the experiment results, $CPBT$ improves the efficiency of the pattern mining engine significantly when there are a huge number of the candidate closed patterns.

3 An overview of the pattern mining engine of POSITING/RELENTING

In this section, we consider the structure of POSITING briefly. Firstly, the background concepts such as event, stream and episode are defined. Then, the episode occurrence is discussed. Finally, the pattern mining engine is explained concisely. We recommend that readers refer to [2,11] for more detail.

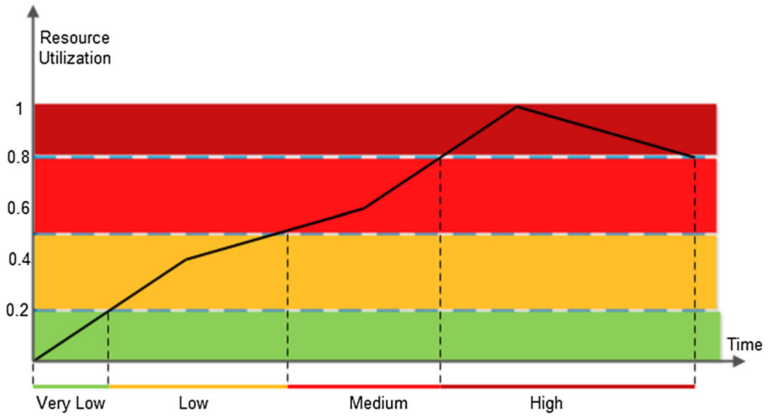


Fig. 2 Converting a time series into a symbolic (discretized) time series by the value abstraction that $Status = \{Very\ Low, Low, Medium, High\}$ and blue dashed lines show the border of the values [11,27] (colour figure online)

3.1 Background concepts

As Fig. 2 shows, POSITING converts the numeric time series of all the resources allocated to the application into a sequence of abstractions $\langle S_1[st_1, et_1], \ldots, S_n[st_n, et_n]\rangle$ where $S_i \in Status, 1 \leq i \leq n$ is an abstraction that holds from time st_i to time et_i and $Status$ is the abstraction alphabet. Let $Status = \{S_1, \ldots, S_M\}$ be a set of the abstract values and $ResourceType = \{R_1, \ldots, R_N\}$ be a set of all the resources allocated to the application. Without loss of generality, we define an arbitrary order on $ResourceType$, for example $R_1 < R_2 < \cdots < R_N$.

Definition 1 An **event** e_i is defined as a 4-tuple $\langle r_i, s_i, st_i, et_i\rangle$ that means the abstract value of $r_i \in ResourceType$ is $s_i \in Status$ from the start time st_i to the end time et_i. The span of the event $e_i = \langle r_i, s_i, st_i, et_i\rangle$ is $\Delta e_i = et_i - st_i > \epsilon$, where ϵ is a positive constant ($\epsilon \in \mathbb{Z}_{\geq 0}$). So the span of each event is at least $\epsilon + 1$ time slots.

All the discretized time series of the resources are represented as a multivariate stream. Note that the value of ϵ depends on the length of sampling intervals. In coarse grained sampling, ϵ is set to small values. For fine grained sampling, ϵ could be set to larger values.

Definition 2 A multivariate **stream** $E = \langle e_1, e_2, \ldots, e_n\rangle$, where n is the index of the latest observed event, is a sequence of events that are ordered according to their start time:

$$\forall e_i, e_j \in E\ that\ 1 \leq i < j \leq n : (st_i < st_j) or (st_i = st_j\ and\ r_i < r_j)$$

Definition 3 A **state** is an ordered pair of (r, s), where $r \in ResourceType$ and $s \in Status$. The Resource-Status (**RS**) is a set of all the possible states: $RS = \{(r, s)|\forall r \in ResourceType, \forall s \in Status\}$.

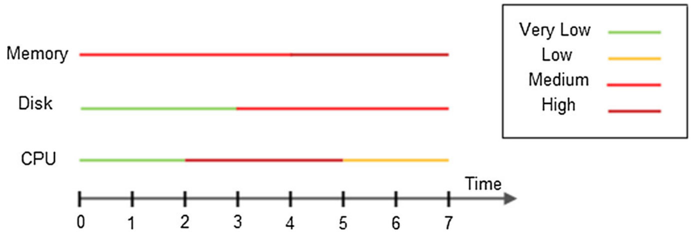


Fig. 3 An example of a multivariate stream with $ResourceType = \{CPU, Memory, Disk\}$ and $Status = \{Very\ Low, Low, Medium, High\}$ [27]

Example 1 Figure 3 shows a multivariate stream E with $ResourceType = \{CPU, Memory, Disk\}$ and $Status = \{Very\ Low, Low, Medium, High\}$. If the order on $ResourceType$ is defined as $CPU < Memory < Disk$, according to Definition 2, $E = \langle(CPU, Very\ \ Low, 0, 2), (Memory, Medium, 0, 4), (Disk, Very\ \ Low, 0, 3), (CPU, High, 2, 5), (Disk, Medium, 3, 7), (Memory, High, 4, 7), (CPU, Low, 5, 7)\rangle$.

If the span of events is large, they are decomposed based on the decomposition unit μ. For example the event $(Disk, Medium, 3, 7)$ with $\mu = 3$ is decomposed into two events $(Disk, Medium, 3, 6)$ and $(Disk, Medium, 6, 7)$. However, after decomposing the event e, the span of the last decomposed event might be less than ϵ. Here, to satisfy Definition 1, the latest and penultimate decomposed events merge together.

Inspired by the temporal relations defined in [27], we define two types of relations between events: **concurrent** and **consecutive**.

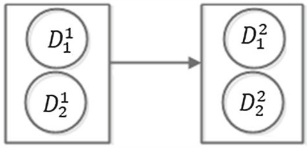
Definition 4 Given the stream $E = \langle e_1, \ldots, e_n\rangle$, two events e_i and e_j, $1 \leq i, j \leq n$, are **concurrent** iff $|st_i - st_j| \leq \epsilon$ and are **consecutive** iff $|st_i - st_j| > \epsilon$.

Mannila et al. [23] informally define an episode as a partially ordered collection of events that occur together [23]. Inspired by the definition of the episode in [23], we present a detailed definition of the episode based on our problem domain. Note that we use the terms "pattern" and "episode" interchangeably in this paper.

Definition 5 A Concurrent Nodes Group (CNG) $G = D_1 D_2 \cdots D_l$ is a group of nodes such that $\forall D_j, D_m \in G, 1 \leq j, m \leq l$, there is no partial order between D_j and D_m.

Definition 6 An episode α is defined as a directed acyclic graph $\alpha = (V_\alpha, \prec_\alpha, g_\alpha)$, where V_α is a set of nodes, $\prec_\alpha$ is a partial order on V_α and $g_\alpha : V_\alpha \rightarrow RS$ is a function that maps each node into one state. The episode α is composed of $k(> 1)$ CNGs in the form of $G_1 = D_1^1, D_2^1, \ldots, D_{l_1}^1$,…, $G_k = D_1^k, D_2^k, \ldots, D_{l_k}^k$ that:

1. $|G_i| = l_i$
2. $V_\alpha = \{D_1^1 \ldots, D_{l_1}^1, D_1^2 \ldots, D_{l_2}^2, \ldots, D_1^k \ldots, D_{l_k}^k\}$
3. $\forall D_j^i \in G_i, \forall D_n^m \in G_m, 1 \leq i < m \leq k, j \in \{1, \ldots, l_i\}, n \in \{1, \ldots, l_m\}$: $D_j^i \prec_\alpha D_n^m$



$V_\alpha = \{D_1^1, D_2^1, D_1^2, D_2^2\}$
$G_1 = D_1^1 D_2^1$
$G_2 = D_1^2 D_2^2$
$\|\alpha\| = |V_\alpha| = 4$

$g_\alpha(D_1^1) = (CPU, High)$
$g_\alpha(D_2^1) = (Memory, Medium)$
$g_\alpha(D_1^2) = (CPU, Low)$
$g_\alpha(D_2^2) = (Disk, Low)$

Fig. 4 The graphical representation of the episode $\alpha = (CPU, High)(Memory, Medium) \to (CPU, Low)(Disk, Low)$ [11]

4. $|CNG_\alpha| = k$
5. $G'_i = \{(r, s) \in RS | g_\alpha(v) = (r, s) \ \ , \forall v \in G_i\}$.

The episode α could be represented as a general form $\alpha = G'_1 \to G'_2 \to \cdots \to G'_k$.

Example 2 Consider the episode $\alpha = (V_\alpha, \prec_\alpha, g_\alpha)$ in Fig. 4. The set V_α contains four nodes. As it is shown, the function g_α maps the nodes into the states and $D_1^1 \prec_\alpha D_1^2$, $D_1^1 \prec_\alpha D_2^2$, $D_2^1 \prec_\alpha D_1^2$ and $D_2^1 \prec_\alpha D_2^2$. As a simple graphical notation, this episode is represented as $\alpha = (CPU, High)(Memory, Medium) \to (CPU, Low)(Disk, Low)$.

3.2 The episode occurrence

Informally, the occurrence of an episode in the stream means that the nodes of the episode have the corresponding events in the stream such that the partial order of the episode is preserved [40]. A frequent episode occurs often enough in the stream. Given a frequency threshold $\theta \in \mathbb{R}_{\geq 0}$, the goal of episode mining is to extract all the frequent episodes in the stream. We choose the Non-Overlapped (NO) frequency [41] to compute the frequency of episodes. Two occurrences h_1 and h_2 of the episode α are said to be non-overlapped if either "h_1 starts after h_2" or "h_2 starts after h_1" [41]. The NO frequency is computed by using minimal occurrences. A minimal occurrence is an occurrence that includes no other occurrences. So, $freq(\alpha)$ is the cardinality of a maximal NO set of minimal occurrences of the episode α in the stream [11,41]. In Sect. 4, we discuss how to compute the episode frequency based on the occurrences.

Definition 7 Given the episode α such that $|CNG_\alpha| = k$ and $1 \leq i \leq k$, for each occurrence of α, the **starting interval** of the occurrence of G_i, $[t_1^i, t_2^i]$, is:

$$t_1^i = \min\{st \text{ of the corresponding events of the nodes of } G'_i \text{ in the occurrence}\} \quad (3.1)$$

$$t_2^i = \max\{st \text{ of the corresponding events of the nodes of } G'_i \text{ in the occurrence}\} \quad (3.2)$$

Definition 8 Given the episode α such that $|CNG_\alpha| = k$, each occurrence O of α is determined as a sequence of the starting intervals of CNGs: $O = ([t_1^i, t_2^i]_{i=1}^k)$

Example 3 Consider the stream $E = \langle e_1 = (CPU, High, 0, 3), e_2 = (Memory, Medium, 0, 4), e_3 = (Network, Low, 0, 2), e_4 = (Disk, Medium, 0, 3), e_5 =$

$(Network, Medium, 2, 5)$, $e_6 = (CPU, 3, 5, Low)$, $e_7 = (Disk, Low, 3, 5)$, $e_8 = (Memory, Very\ Low, 4, 5)\rangle$. For $\epsilon = 0$, there is an occurrence of the episode α given in Example 2 in the stream E. The starting intervals of the occurrence of G_1 and G_2 are $[t_1^1, t_2^1] = [0, 0]$ and $[t_1^2, t_2^2] = [3, 3]$ respectively. So the occurrence is represented as $O = ([0, 0], [3, 3])$.

Dynamic resources allocation is based on virtualization techniques [42]. Based on time spent on booting VMs, patterns should be extracted from application behavior in a way that SLA is satisfied and energy wasting is avoided. Given the episode $\alpha = G'_1 \to G'_2 \to \cdots \to G'_k$ and an occurrence $O = ([w_1^i, w_2^i]_{i=1}^k)$ of α, if the time it takes to instantiate a new VM instance is $\delta(> \epsilon)$ time slots, the starting interval of G'_{i+1}, $1 \le i < k$, should begin after $\delta + w_2^i$. Thus, the resources manager has enough time to instantiate a new VM instance. On the other hand, if resources are allocated before occurring workload burstiness for a long time, energy and resources are wasted. According to the discretion of the resources manager and characteristics of the cloud data center, the gap constraint $\Delta(\ge \delta)$ determines that resources might be allocated at most $\Delta - \delta$ time slots before occurring workload burstiness. Therefore, the Valid Interval (VI) of G'_{i+1} is $VI([w_1^i, w_2^i], i + 1) = [w_2^i + \delta, w_2^i + \Delta]$ to satisfy QoS and SLA and avoid wasting energy. δ and Δ are called minimum internal gap and maximum internal gap respectively.

To compute the NO frequency of episodes under gap constraints, tracking the minimal occurrences of episodes is not enough [12]. So we introduced a new type of the occurrence, called the latest occurrence, to compute the NO frequency under gap constraints in [11]:

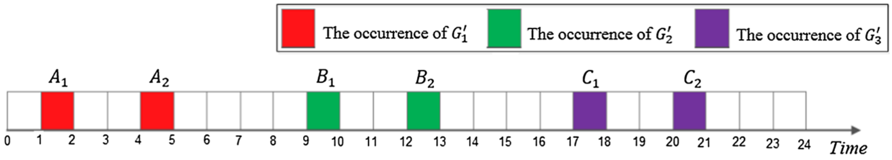
Definition 9 Given the episode $\alpha = G'_1 \to G'_2 \to \cdots \to G'_k$ and the internal gaps δ and Δ, an occurrence $O = ([w_1^i, w_2^i]_{i=1}^k)$ of α is a **valid occurrence** iff $\forall i, 1 \le i < k, w_2^i + \delta \le w_1^{i+1} \le w_2^i + \Delta$.

Example 4 Consider Example 3. If $\delta = 2$ and $\Delta = 3$, $VI([0, 0], 2) = [0+2, 0+3] = [2, 3]$. In addition, the occurrence O is valid because we have $0 + 2 \le 3 \le 0 + 3$.

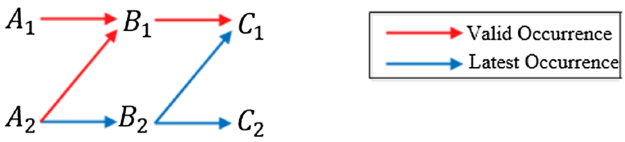
Definition 10 Given the episode $\alpha = G'_1 \to G'_2 \to \cdots \to G'_k$, if for a valid occurrence $O = ([t_1^i, t_2^i]_{i=1}^k)$ of α there exists no other valid occurrence $Q = ([w_1^i, w_2^i]_{i=1}^{k-1}, [t_1^k, t_2^k])$ of α such that $\exists j, 1 \le j < k, w_1^j > t_1^j$, it is said that O includes the **Latest Prefix Occurrence** (LPO).

Definition 11 Each valid occurrence of the episode α that includes LPO, is called the **Latest Occurrence** (LO). $LO(\alpha)$ is a set of all the latest occurrences of α.

Example 5 Given the episode $\alpha = G'_1 \to G'_2 \to G'_3$, $\epsilon = 1$, $\delta = 4$ and $\Delta = 7$, Fig. 5a shows the occurrences of G'_i, $i = 1, 2, 3$. Note that there are two occurrences for each G'_i: $A_1 = [1, 2]$ and $A_2 = [4, 5]$ are the occurrences of G'_1, $B_1 = [9, 10]$ and $B_2 = [12, 13]$ are the occurrences of G'_2 and $C_1 = [17, 18]$ and $C_2 = [21, 22]$ are the occurrences of G'_3. Figure 5b shows that four valid occurrences (note that each LO is also a valid occurrence) could be identified for α under gap constraints. According to Definition 11, the corresponding occurrences of the red lines in Fig. 5b are not LO. As the figure shows there are two LOs for α: (A_2, B_2, C_1) and (A_2, B_2, C_2).



(a) The occurrences of $G_i', i = 1, 2, 3$ of the episode $\alpha = G_1' \to G_2' \to G_3'$



(b) The valid occurrences and the LOs of the episode α based on the occurrences of $G_i', i = 1, 2, 3$

Fig. 5 Extracting LOs of the episode α based on the occurrences of its CNGs

In previous works, a superset of the minimal occurrences, called Minimal Prefix Occurrences (MPOs), is extracted to compute the NO frequency under gap constraints. An MPO with the span of $[ts, te]$ is a valid occurrence (satisfying gaps) such that there is no other valid occurrence that starts strictly after ts and ends at or before te [12].

Example 6 Consider Example 5 again. Assume there is only the occurrence A_2 for G_1'. In this case, according to the definition of MPO, there are three MPOs: $O_1 = (A_2, B_1, C_1)$, $O_2 = (A_2, B_2, C_1)$ and $O_3 = (A_2, B_2, C_2)$ for α. On the contrary, there are two LOs: $O_2 = (A_2, B_2, C_1)$ and $O_3 = (A_2, B_2, C_2)$ for α. This example shows that LOs of α are a subset of its MPOs.

Lemma 1 *Given the episode $\alpha = G_1' \to \cdots \to G_k'$, if $MPO(\alpha)$ is a set of all the minimal prefix occurrences of α, then $LO(\alpha) \subseteq MPO(\alpha)$.*[1]

Definition 12 Given the episode $\alpha = G_1' \to G_2' \to \cdots \to G_k'$, $LOList(\alpha)$ includes a 4-tuple $(t_2^{k-1}, t_1^k, t_2^k, t_1^1)$ for each occurrence $O = ([t_1^i, t_2^i]_{i=1}^k) \in LO(\alpha)$. $LOList(\alpha)[i]$ is the i-th member of $LOList(\alpha)$.

The focus of the following sections is on improvements to time and space complexities of the pattern mining engine.

3.3 The pattern extraction

The pattern mining engine constructs a pattern tree and extracts frequent patterns.

Definition 13 Given the episode $\alpha = G_1' \to G_2' \to \cdots \to G_k'$ and $(r, s) \in RS$, the **serial extension** of α with (r, s) is:

$$\alpha \oplus (r, s) = G_1' \to G_2' \to \cdots \to G_k' \to (r, s) \tag{3.3}$$

[1] The proof of lemmas and theorems could be found in "Appendix A".

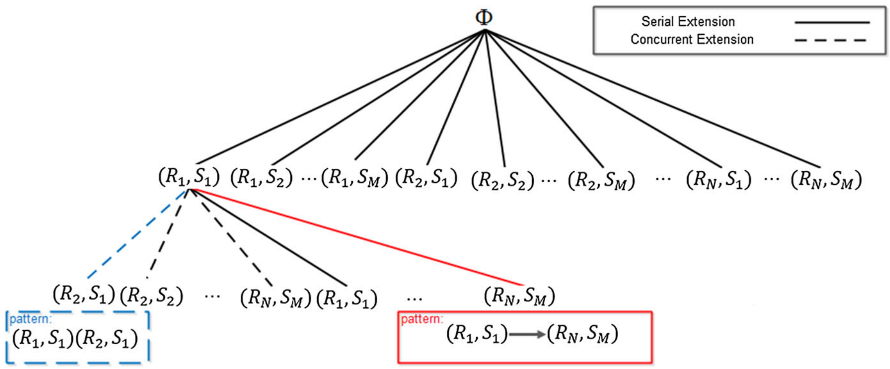


Fig. 6 A part of the lexicographic pattern tree [11]

Definition 14 Given the episode $\alpha = G'_1 \rightarrow G'_2 \rightarrow \cdots \rightarrow G'_{k-1} \rightarrow G'_k$ and $(r, s) \in RS$, the **concurrent extension** of α with (r, s) is:

$$\alpha \odot (r, s) = G'_1 \rightarrow G'_2 \rightarrow \cdots \rightarrow G'_{k-1} \rightarrow G'' \;\; that \; G'' = G'_k \cup (r, s) \quad (3.4)$$

The lexicographic tree (pattern tree) is constructed based on the serial and concurrent extensions as follows [13]:

- The root is labeled with $\emptyset$.
- Each node n of the tree is labeled with a state. $Label(n)$ is the corresponding label of the node n.
- Each node n of the tree corresponds to an episode. $Pattern(n)$ is the corresponding episode of the node n.
- If $Pattern(n) = \alpha$, the corresponding episode of each child of n is either a serial extension or a concurrent extension of α.
- The left sibling is less than the right sibling.

Figure 6 shows a part of the pattern tree constructed on RS [11]. Note that $\forall i, j, 1 \leq i \leq N, 1 \leq j \leq M, (R_i, S_j) \in RS$. Here, the lexicographic order is defined on RS as $(R_1, S_1) < \cdots < (R_1, S_M) < (R_2, S_1) < \cdots < (R_2, S_M) < \cdots < (R_N, S_1) < \cdots < (R_N, S_M)$. The root of the tree is null. All the patterns in the tree are generated only by the serial extension or the concurrent extension. For example the episode $((R_1, S_1)(R_2, S_1))$ is generated from the concurrent extension of (R_1, S_1) with (R_2, S_1) and the episode $((R_1, S_1) \rightarrow (R_N, S_M))$ is generated from the serial extension of (R_1, S_1) with (R_N, S_M).

Mining frequent episodes might lead to extracting a huge number of patterns. To improve mining efficiency and avoid information loss, a compressed set of episodes, called **closed episodes**, is extracted [43].

Definition 15 The episode β is a sub-episode of the episode α (or α is a super-episode of β), $\beta \sqsubseteq \alpha$, if all the states of β and the partial order between them exist in α.

Definition 16 Given the episode $\alpha = G'_1 \rightarrow G'_2 \rightarrow \cdots \rightarrow G'_k$ and $1 \leq i \leq k$, $Prefix(\alpha, i)$= $G''_1 \rightarrow G''_2 \rightarrow \cdots \rightarrow G''_i$ and $Postfix(\alpha, i) = G''_i \rightarrow G''_{i+1} \rightarrow \cdots \rightarrow G''_k$, where $G''_i \subseteq G'_i$.

Definition 17 **The episode** α **is closed under gap constraints** δ **and** Δ iff there is no other episode β whose prefix or suffix is α and $freq(\alpha) = freq(\beta)$ [12].

Example 7 Consider the two episodes $\alpha = (Memory, Low) \rightarrow (Disk, High)$ and $\beta = (CPU, High)(Memory, Low) \rightarrow (Disk, High)$. It is clear that $\alpha \sqsubseteq \beta$ and $\alpha = Suffix(\beta, 1)$. So if $freq(\alpha) = freq(\beta)$, then the episode α is not closed.

Based on the definitions of the serial extension and the concurrent extension, if nodes n' and n'' are the serial and concurrent extensions of the node n respectively, then we have:

$$\underbrace{Pattern(n')}_{\alpha} = \underbrace{Pattern(n)}_{\beta} \oplus \underbrace{Label(n')}_{x} \tag{3.5}$$

$$\underbrace{Pattern(n'')}_{\gamma} = \underbrace{Pattern(n)}_{\beta} \odot \underbrace{Label(n'')}_{y} \tag{3.6}$$

So without scanning the stream, a maximal non-overlapped set of minimal occurrences of α and γ can be determined by using the join of $LO(\beta)$ with the occurrences of x and y respectively [12].

Firstly, the pattern mining engine extracts frequent episodes by the complete traverse of the pattern tree in a depth-first way. For this purpose, all the frequent 1-node episodes are extracted. Then, the pattern tree is traversed in a depth-first manner from each of the frequent 1-node episodes. When the serial and concurrent extensions of the episode are constructed, it is checked whether any of the super patterns has the same frequency as the episode's or not; if not, the episode is added to the list of candidate closed episodes. After extracting the candidate closed episodes, a post-processing step is performed on them using a hashing procedure with frequency as the key. Finally, a set of all the closed frequent episodes are extracted. Note that to avoid enlarging the pattern tree, we could limit the number of CNGs of episodes. We define $Level$ as the maximum number of CNGs of episodes.

4 Improving space complexity: computing *NO* frequency based on redundant occurrences

This section focuses on improvements to space complexity of the pattern mining engine. In Sect. 4.1, redundant LOs are introduced. The improved representation of the stream is introduced to identify the redundant occurrences in Sect. 4.2. In Sect. 4.3, algorithms are proposed to compute the NO frequency under gap constraints.

4.1 Redundant *LO*

In this section, the redundant LOs are defined and it is proved that removing these occurrences does not affect the frequency of episodes. Therefore, the redundant occurrences could be removed without any information loss, which improves memory consumption.

Definition 18 Given the episode $\alpha = G'_1 \rightarrow G'_2 \rightarrow \cdots \rightarrow G'_k$ and the occurrence $O = ([t^i_1, t^i_2]^k_{i=1}) \in LO(\alpha)$, the occurrence O is a redundant occurrence iff three conditions below are satisfied:

1. There exists another occurrence $Q = ([w^i_1, w^i_2]^k_{i=1}) \in LO(\alpha)$ such that $w^1_1 = t^1_1$ and $w^k_2 < t^k_1$
2. There is no sub-interval of $[t^k_2 + \delta, t^k_2 + \Delta]$ such that only O covers it.
3. There exists no event $e = (r, s, st, et)$ such that $|t^k_2 - st| \leq \epsilon$ and $|t^k_1 - st| \leq \epsilon$.

In Sect. 3.3, we explained how the pattern tree is constructed based on the serial and concurrent extensions. If for the occurrence O there is an event $e = (r, s, st, et)$ such that $|t^k_2 - st| \leq \epsilon$ and $|t^k_1 - st| \leq \epsilon$, then an episode β can be extended from α by the concurrent extension ($\beta = \alpha \odot (r, s)$). Therefore, the occurrence O is not redundant and it should not be removed.

Example 8 Given the episode $\alpha = G'_1 \rightarrow G'_2, \epsilon = 1, \delta = 12$ and $\Delta = 23$, Fig. 7 shows the occurrences of $G'_i, i = 1, 2$. $A_1 = [80, 81]$ and $A_2 = [92, 93]$ are the occurrences of G'_1 and $B_1 = [100, 101]$, $B_2 = [103, 104]$ and $B_3 = [106, 106]$ are the occurrences of G'_2. There are three LOs: $O_1 = (A_1, B_1)$, $O_2 = (A_1, B_2)$ and $O_3 = (A_2, B_3)$. Note that the occurrence O_2 satisfies the first two conditions of Definition 18: the occurrence O_1 satisfies the first condition and as Fig. 8 shows $VI([103, 104], 3)$ is covered completely by the valid intervals of O_1 and O_3. Therefore if there exists no event $e = (v, s, st, et)$ such that $|104 - st| \leq \epsilon$ and $|103 - st| \leq \epsilon$, then O_2 is redundant.

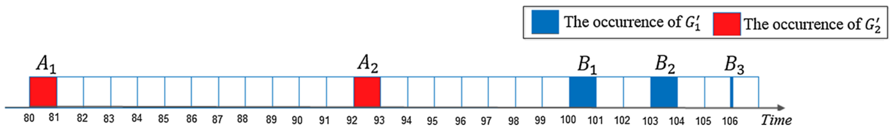


Fig. 7 The occurrences of $G'_i, i = 1, 2$ of the episode $\alpha = G'_1 \rightarrow G'_2$

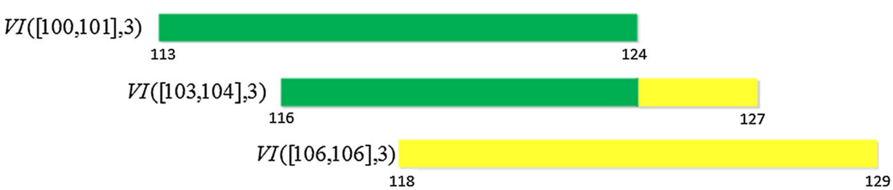


Fig. 8 The valid intervals of the occurrences O_1, O_2 and O_3 in Example 8

Lemma 2 *Given the episode α, if β and γ are the serial and concurrent extensions of α, removing redundant occurrences from $LOList(\alpha)$ does not affect $freq(\alpha)$, $freq(\beta)$ and $freq(\gamma)$.*

Lemma 3 *Given the episode $\alpha = G'_1 \rightarrow G'_2 \rightarrow \cdots \rightarrow G'_k$ and the occurrences $A = ([a^i_1, a^i_2]^k_{i=1})$, $B = ([b^i_1, b^i_2]^k_{i=1})$ and $C = ([c^i_1, c^i_2]^k_{i=1})$, where $A, B, C \in LO(\alpha)$ and A and C are LOs immediately before and after B respectively, if $a^1_1 \neq b^1_1$ and $[b^k_1, b^k_2]$ is not covered by $[a^k_1, a^k_2]$ and $[c^k_1, c^k_2]$, then all of the LOs before A start before B and $[b^k_1, b^k_2]$ is covered by none of the LOs before A and after C.*

Lemma 4 *If $\epsilon \geq \frac{\delta}{4}$ and $\Delta \in [\delta, 2\delta)$, then there is no redundant LO.*

4.2 Improved representation of the stream based on pointers (*PROSPER*)

According to Lemma 2, all the LOs of the episode don't include useful information. So removing these LOs could improve memory consumption of $LOList$ of episodes. To identify redundant LOs, according to the third condition of Definition 18, LOs should not extend concurrently. For this purpose, the occurrence list of all the states of RS should be checked, which might be time-consuming. To expedite the identification of concurrent events and the removal of redundant LOs, we propose $PROSPER$, which is the improved representation of the stream based on pointers. $PROSPER$ is based on the vertical representation of the stream. It connects the concurrent events by using pointers.

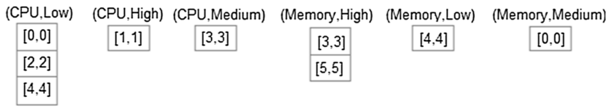
In the vertical representation of the stream [44], each $(r, s) \in RS$ is associated with a list whose entries include the starting intervals of that (r, s). In $PROSPER$, each entry of the list is augmented with two pointers to the entry, which are called $Next$ and $Previous$. The concurrent events are connected by the pointers. The corresponding list of (r, s) in $PROSPER$ is called $LOListRS(r, s)$.

Definition 19 $LOListRS(r, s)$ includes a 4-tuple $([v, v'], Next, Previous)$ for each occurrence of $(r, s) \in RS$, where $[v, v']$ is the starting interval of the occurrence and $Next$ and $Previous$ are the pointers that connect the concurrent events. $LOListRS(r, s)[i]$ is the i-th member of $LOListRS(r, s)$. (Note that for each occurrence of (r, s), $v = v'$.)

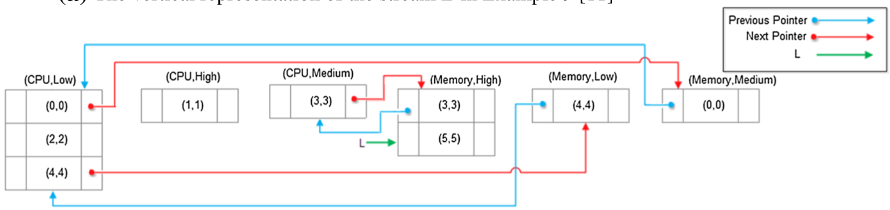
Example 9 Consider the stream E:

$$E = \langle (CPU, Low, 0, 1), (Memory, Medium, 0, 3), (CPU, High, 1, 2), (CPU, Low, 2, 3), (CPU, Medium, 3, 4), (Memory, High, 3, 4), (CPU, Low, 4, 6), (Memory, Low, 4, 5), (Memory, High, 5, 6)\rangle$$

Figure 9 shows the vertical representation and $PROSPER$ of the stream E for $\epsilon = 0$. As Fig. 9b shows, the concurrent events could be identified by using the pointers easily. Note that the pointer L always points to the last event of the stream.



(a) The vertical representation of the stream E in Example 9 [11]



(b) The representation of the stream E in Example 9 in the form of $PROSPER$

Fig. 9 The representation of the stream E in Example 9 in the forms of the vertical representation and $PROSPER$

4.3 *NO* frequency under gap constraints

To compute the NO frequency of episodes, their $LOList$ should be extracted firstly [11]. In this section, based on $PROSPER$, we modify the algorithms SSMakeLOList and SCMakeLOList presented in [11] to extract the non-redundant LOs of episodes.

4.3.1 Extracting the non-redundant *LO*s of episodes using the serial extension

The algorithm $SMakeLOList$ (Algorithm 1) is proposed to extract the non-redundant LOs of episodes using the serial extension. The algorithm receives $LOList(\alpha)$ (see Definition 12) and $LOListRS(r, s)$ (see Definition 19) that α is an episode and $(r, s) \in RS$, and computes $LOList(\beta = \alpha \oplus (r, s))$ without scanning the stream. Note that $LOListRS(r, s)$ is the occurrence list of (r, s) in $PROSPER$. The counters i, z and j traverse the $LOList$s of α and β and $LOListRS(r, s)$ respectively. Lines 2 to 23 consider for each LO of α which LOs of (r, s) could create a non-redundant LO for β. Lines 4 to 6 check whether the i-th LO of α could be the latest prefix occurrence for the j-th occurrence of (r, s) or not. If not, this LO of α could not be the latest prefix occurrence for the next occurrences of (r, s). So the next LOs of α are considered (line 22). For the new LOs of α, we start from the occurrences of (r, s) that there is no latest prefix occurrence for them. In lines 4 to 5, if an LO of α is the latest prefix occurrence for an LO of (r, s), the corresponding LO of β is generated and inserted in $LOList(\beta)$. In lines 7 to 17, the LOs immediately before and after each LO of β are considered whether that LO is redundant based on Definition 18. In line 12, the function $CExtending$ (see Algorithm 7 in "Appendix B") considers whether LO could extend concurrently or not. According to Lemma 3, if the conditions of Definition 18 are not satisfied for the next and previous LOs, then the conditions would not be satisfied by the other LOs. In line 13, if an LO is redundant, it is removed and the counter z is updated. Note that in line 10, if $b_3 + \delta \leq b_1 + \Delta$, then $[b_2 + \delta, b_2 + \Delta]$ is covered because we have $b_1 < b_2 < b_3$. So if $b_1 + \delta < b_2 + \delta < b_3 + \delta \leq b_1 + \Delta < b_2 + \Delta < b_3 + \Delta$, then $VI([a_2, b_2], k+1)$ is completely

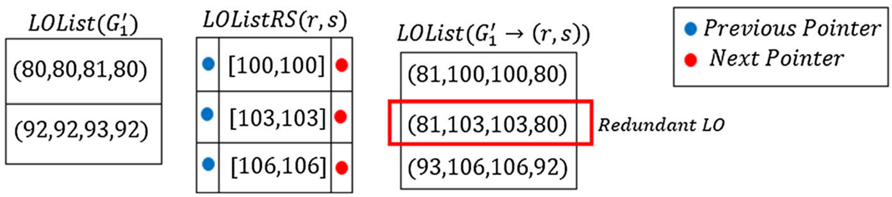


Fig. 10 The serial extension of G_1' with (r, s) and extracting $LOList(G_1' \rightarrow (r, s))$ by the algorithm SMakeLOList in Example 10

covered by $VI([a_1, b_1], k+1)$ and $VI([a_3, b_3], k+1)$, where $|CNG_\alpha| = k$. Time complexity of the algorithm is discussed in Lemma 9 in "Appendix A".

Algorithm 1 SMakeLOList

Input: $\epsilon, \delta, \Delta, LOList(\alpha), LOListRS(r, s)$ % α is an episode; $(r, s) \in RS$; $LoListRS(r, s)[i] = ([v_i, v_i], Next, Previous)$;$LoList(\alpha)[i] = (x_i, t_i, t_i', t_i^\alpha)$
Output: $LOList(\beta)$
1: $i \leftarrow 1$; $j \leftarrow 1$; $z \leftarrow 1$;
2: **while** $((i \leq |LOList(\alpha)|)$ *and* $(j \leq |LOListRS(r, s)|))$ **do**
3: **while** $((j \leq |LOListRS(r, s)|)$ *and* $(i = |LOList(\alpha)|$ *or* $(i < |LOList(\alpha)|$ *and* $t_{i+1}' + \delta > v_j))$ *and* $(v_j \leq t_i' + \Delta))$ **do**
4: **if** $(t_i' + \delta \leq v_j \leq t_i' + \Delta)$ **then**
5: $Add((t_i', v_j, v_j, t_i^\alpha), LOList(\beta))$; % add $(t_i', v_j, v_j, t_i^\alpha)$ into $LOList(\beta)$
6: $j++$; $z++$;
7: **if** $(z > 3)$ **then**
8: $[(x_1, b_1, b_1, c_1), (x_2, b_2, b_2, c_2), (x_3, b_3, b_3, c_3)] \leftarrow LOList(\beta)[z-3 : z-1]$;
9: **if** $(c_1 = c_2)$ **then**
10: **if** $(b_3 + \delta \leq b_1 + \Delta)$ **then**
11: $Index \leftarrow FindIndex(b_2, b_2, LOListRS(r, s))$; % Find the $Index$ of an entry of $LOListRS(r, s)$ whose start time is in $[b_2, b_2]$
12: **if** $(!CExtending(r, s, LOListRS(r, s)[Index], b_2, b_2, \epsilon))$ **then**
13: $LOList(\beta)[z-1] \leftarrow LOList(\beta)[z]$; $z--$;
14: **end if**
15: **end if**
16: **end if**
17: **end if**
18: **else if** $(v_j < t_i' + \delta)$ **then**
19: $j++$;
20: **end if**
21: **end while**
22: $i++$;
23: **end while**
24: **return** $LOList(\beta)$;

Theorem 1 *Given the episode α and $(r, s) \in RS$, the algorithm $SMakeLoList$ only finds all the non-redundant LOs of $\beta = \alpha \oplus (r, s)$.*

Example 10 Consider Example 8. Figure 10 shows $LOList(G_1')$, $LOListRS(r, s)$ and $LOList(\beta = G_1' \rightarrow (r, s))$ extracted by Algorithm 1. Since all of the pointers of $LOListRS(r, s)$ are *null*, according to lines 10 to 20 of the algorithm, the second element of $LOList(\beta)$ is redundant. So it is removed.

4.3.2 Extracting the non-redundant *LO*s of episodes using the concurrent extension

The algorithm $CMakeLOList$ (Algorithm 2) is proposed to extract the non-redundant $LOList$ of episodes using the concurrent extension. The algorithm receives $LOList(\alpha)$ and $LOListRS(r, s)$ that α is an episode and $(r, s) \in RS$, and computes $LOList(\beta = \alpha \odot (r, s))$ without scanning the stream.

Algorithm 2 CMakeLOList

Input: $\epsilon, \delta, \Delta, LOList(\alpha), LOListRS(r, s)$ % α is an episode;$(r, s) \in RS$ $LoListRS(r, s)[i] = (v_i, v_i, Next, Previous)$;$LoList(\alpha)[i] = (x_i, t_i, t'_i, t^{\alpha}_i)$
Output: $LOList(\beta)$
1: $i \leftarrow 1; j \leftarrow 1; z \leftarrow 1;$
2: **while** $(i \leq |LOList(\alpha)|$ *and* $j \leq |LOListRS(r, s))$ **do**
3:   **while** $(j \leq |LOListRS(r, s)|$ *and* $LOListRS(r, s)[j].Next = null$ *and* $LOListRS(r, s)[j].Previous = null)$ **do**
4:     $j++;$
5:   **end while**
6:   **if** $(j \leq |LOListRS(r, s)|)$ **then**
7:     **if** $(|v_j - t_i| \leq \epsilon$ *and* $|v_j - t'_i| \leq \epsilon$ *and* $x_i + \delta \leq \min(t_i, v_j) \leq x_i + \Delta)$ **then**
8:       $Add((x_i, \min(t_i, v_j), \max(t'_i, v_j), t^{\alpha}_i), LOList(\beta));$ % add $(x_i, \min(t_i, v_j), \max(t'_i, v_j),$ %$t^{\alpha}_i)$ into $LOList(\beta)$
9:       $i++; j++; z++;$
10:       **if** $(z > 3)$ **then**
11:         $[(x_1, a_1, b_1, c_1), (x_2, a_2, b_2, c_2), (x_3, a_3, b_3, c_3)] \leftarrow LOList(\beta)[z-3 : z-1];$
12:         **if** $(c_1 = c_2)$ **then**
13:           **if** $(b_3 + \delta \leq b_1 + \Delta)$ **then**
14:             $Index \leftarrow FindIndex(a_2, b_2, LOListRS(r, s));$ % Find the $Index$ of an entry of $LOListRS(r, s)$ whose start time is in $[a_2, b_2]$
15:             **if** $(!CExtending(r, s, LOListRS(r, s)[Index], a_2, b_2, \epsilon))$ **then**
16:               $LOList(\beta)[z-1] \leftarrow LOList(\beta)[z]; z--;$
17:             **end if**
18:           **end if**
19:         **end if**
20:       **end if**
21:     **else if** $v_j > t'_i$ **then**
22:       $i++;$
23:     **else**
24:       $j++;$
25:     **end if**
26:   **end if**
27: **end while**
28: **return** $LOList(\beta);$

The counters i, z and j traverse the $LOList(\alpha)$, $LOList(\beta)$ and $LOListRS(r, s)$ respectively. In lines 3 to 5, the first element of $LOListRS(r, s)$ that is concurrent with at least one event is found. There are three cases for $LOList(\alpha)$ and $LOListRS(r, s)$: 1) In lines 7 to 20, if the corresponding entries of $LOList(\alpha)[i]$ and $LOListRS(r, s)[j]$ could generate an LO of $\beta = \alpha \odot (r, s)$, it is inserted in $LOList(\beta)$ and three counters i, j and z increase by +1. In lines 10 and 20, if an LO is redundant, it is removed and the counter z is updated. (2) In lines 21 to 22, if $LOListRS(r, s)[j]$ occurs after $LOList(\alpha)[i]$, then $LOList(\alpha)[i]$ should not be considered for the members after $LOListRS(r, s)[j]$. So the next LO of α is checked. (3) In lines 23 and 24, if $LOListRS(r, s)[j]$ occurs before $LOList(\alpha)[i]$,

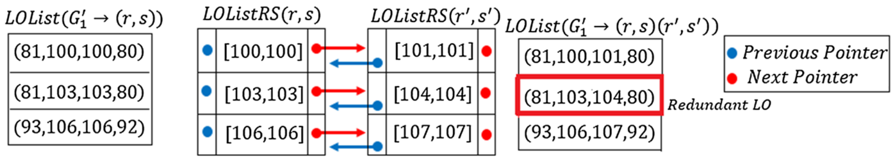


Fig. 11 The concurrent extension of α with (r, s) and extracting $LOList(\beta)$ by the algorithm CMakeLOList in Example 11

the next occurrences of (r, s) are considered for $LOList(\alpha)[i]$. Time complexity of the algorithm $CMakeLOList$ is discussed in Lemma 10 in A.

Theorem 2 *Given the episode $\alpha = G'_1 \rightarrow G'_2 \rightarrow \cdots \rightarrow G'_k$ and $G = (r, s) \in RS$, the algorithm $CMakeLOList$ only finds all the non-redundant LOs of $\beta = \alpha \odot G$.*

Example 11 Given the episode $\alpha = G' \rightarrow (r, s)$, $(r', s') \in RS$, $(r, s) < (r', s')$, $\epsilon = 1$, $\delta = 12$ and $\Delta = 23$, Fig. 11 shows $LOList(\alpha)$, $LOListRS(r, s)$, $LOListRS(r', s')$ and $LOList(\beta)$ extracted by Algorithm 2. According to the algorithm, the second element of $LOList(\beta)$ is removed because it is redundant.

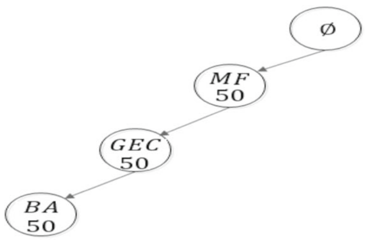
After extracting $LOList$ of episodes, their NO frequency could be computed by calling the function $ComputeFreq$ presented in [11] easily. $ComputeFreq$ scans $LOList$ of the episode and counts the number of the non-overlapped LOs.

5 Improving time complexity: a new approach for mining the closed episodes

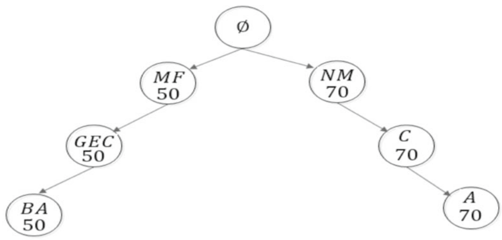
As it was mentioned, the common approach to extract closed episodes under gap constraints is based on the hash table [13]. It is a two-step approach. In the first step, candidate closed episodes are extracted. In the next step, they are considered and closed episodes are determined by using a hashing procedure with frequency as the key [12,13]. As it will be shown in the evaluation results, the number of closed episodes is usually much fewer than candidate closed episodes'. So extracting closed episodes from among a huge number of candidate closed episodes is very time-consuming. For this purpose, we introduce a new data structure, called $CPBT$, to store closed episodes and present depth-first search algorithms based on $CPBT$ to extract closed episodes directly. In this section, $CPBT$ is introduced firstly. Then, the algorithms for mining closed frequent episodes are presented.

5.1 The data structure *CPBT*

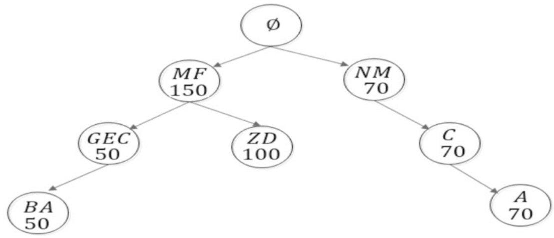
The data structure $CPBT$ is introduced to store closed episodes compactly. The root of $CPBT$ is labeled with $\varnothing$. The episode is traversed in the backward direction and inserted in $CPBT$ in a way that each node is corresponding to one CNG of the episode. So the episodes whose postfixes are the same share the same nodes. To avoid losing the frequency of the episodes, each node maintains the sum of the frequency of the episodes that share that node. Each $Node\ n$ of $CPBT$ includes:



(a) *CPBT* after inserting the episode α in Example 12



(b) *CPBT* after inserting the episodes α and β in Example 12



(c) *CPBT* after inserting the episodes α, β and γ in Example 12

Fig. 12 The step-by-step construction of $CPBT$ while inserting the episodes of Example 12

- $label$ Given the episode $\alpha = G'_1 \rightarrow \cdots \rightarrow G'_k$, if the node n is corresponding to G'_i, $1 \leq i \leq n$, then $label(n)$ is the inverse of G'_i.
- $freq$: It is the sum of the frequency of the episodes that share n.
- $children$ There is a node as a child of n for each episode that shares n and n is not corresponding to the first CNG of the episode.

Example 12 Given $RS_1 = \{A, B, C, D, E, F, G, M, N, Z\}$, where $RS_1 \subseteq RS$, assume the three episodes α, β and γ are extracted as follows:

$$\alpha = AB \rightarrow CEG \rightarrow FM \qquad \beta = A \rightarrow C \rightarrow MN \quad \gamma = DZ \rightarrow FM$$
$$freq(\alpha) = 50 \qquad freq(\beta) = 70 \quad freq(\gamma) = 100$$

As Fig. 12a shows, the episode α is inserted in the backward direction in $CPBT$. Each node of $CPBT$ is corresponding to one CNG of α. Note that each node includes the frequency of α. Figure 12b shows $CPBT$ after inserting the episode β. Since the episodes α and β have no equal postfix, the episode β is inserted as a new branch of the root. Since the postfixes of the episodes γ and α are equal, so the node labeled MF is shared between them. Note that the frequency of this node is the sum of the frequency of α and γ.

Algorithm 3 AllClosedFreqEpisodes

Global Variables: $CPBT, PathList$;
Input: $\epsilon, \delta, \Delta, Level, \theta$ % parameters and thresholds, $0 \leq \theta \leq 1$
Output: $ClosedSet$ % $ClosedSet$ is a set of all the closed frequent episodes

```
1: P ← CreateList(RS);   % Create a list of x ∈ RS, where |LOListRS(x)| > 0
2: Sort P based on the Order defined on RS;
3: for each (p ∈ P) do
4:     FindClosedFreqEpisode(ϵ, δ, Δ, Level, θ, p, ConvertToLOList(LOListRS(p)));   %
   ConvertToLOList(LOListRS(p)) converts the LOListRS(p) to the form of LOList
5: end for
6: ClosedSet ← ExtractClosedEpisodesFromCBBT();
7: return ClosedSet
```

5.2 Mining closed frequent episodes

Firstly, we introduce two new supersets of closed episodes, called Forward Closed and Backward Closed. Then based on them and $CPBT$, we propose algorithms that extract closed frequent episodes directly.

Definition 20 Given $\alpha = G'_1 \rightarrow \cdots \rightarrow G'_k$, if there is no episode β such that $\alpha = Prefix(\beta, k)$ and $freq(\alpha) = freq(\beta)$, then α is **Forward Closed (FC)**.

Definition 21 Given $\alpha = G'_1 \rightarrow \cdots \rightarrow G'_k$, if there is no episode β such that $|CNG_\beta| = u$, $\alpha = Suffix(\beta, u - k + 1)$ and $freq(\alpha) = freq(\beta)$, then α is **Backward Closed** (BC).

If an episode is FC and BC then it is a closed episode. Therefore, we extract the FC episodes and insert them in $CPBT$ firstly. From among the FC episodes, the episodes that are not BC are absorbed by their super-episodes. Thus, $CPBT$ only includes closed frequent episodes. Note that for the two FC episodes α and β, checking the CNG occurrences of the episodes is redundant because we could check whether α is a suffix of β and identify their frequency by using $CPBT$ simply. As it will be shown, closed episodes and their frequency could be extracted from $CPBT$ easily.

Algorithm 3 extracts closed episodes by the complete traverse of the pattern tree in a depth-first way. At first, in line 1, all the 1-node episodes (denoted by P) are extracted. Then, in lines 3 and 4, the pattern tree is traversed in a depth-first manner from each of the 1-node episodes using the recursive calls of the algorithm $FindClosedFreqEpisode$ (see Algorithm 8 in "Appendix B"). Note that $LOListRS(p)$ is the corresponding list of p in $PROSPER$. The algorithm $FindClosedFreqEpisode$ identifies the FC episodes and inserts them in $CPBT$ by calling the function $InsertCPBT$. Finally, after extracting closed episodes, Algorithm 3 calls the function $ExtractClosedEpisodesFromCBBT$ to traverse $CPBT$ and extract closed episodes from it. Note that episodes are represented in the form of SAVE [11] to expedite the episode extraction. In the next section, we comprehensively explain how to insert the FC episodes in CPBT.

5.3 Insert in *CPBT*

When the FC episode α is inserted in $CPBT$, two cases could occur: (1) In $CPBT$, there is another episode β whose suffix is α and $freq(\alpha) = freq(\beta)$. It means that $\beta < \alpha$ because β has been inserted in $CPBT$ sooner than α. So α is not BC and should not be inserted in $CPBT$. (2) After inserting α, another episode β whose suffix is α and $freq(\alpha) = freq(\beta)$ might be inserted. It means that $\alpha < \beta$. So α should be removed from $CPBT$.

Definition 22 If the episodes α and β are FC, α is a suffix of β and $freq(\alpha) = freq(\beta)$, it is said that β **absorbs** α.

The procedure for calling functions to insert an FC episode in $CPBT$ is shown in Fig. 13. According to the figure, the episode is converted to a branch by calling the function $CreateBranch$. In the next step, the function $SearchInTree$ is called. This function calls the function $EpisodeAbsorbByTree$. It considers whether $CPBT$ absorbs the episode or not. If not, the function $TreeAbsorbByEpisode$ is called. This function finds the branches that are absorbed by the episode. Then, these branches are updated by calling the function $UpdateBranch$. Finally, the function $InsertInTree$ is called to insert the episode in the right place.

The algorithm $InsertInCPBT$ inserts the FC episode α in $CPBT$. As Algorithm 4 shows this function receives the FC episode α and its frequency. The pointer R points to the root of $CPBT$ at first. In line 2, the corresponding branch of the episode α, α', is created by calling the function $CreateBranch$ (Algorithm 9). In line 3, the branch α' is searched in $CPBT$ by calling the function $SearchInTree$ (Algorithm 5). If α' is absorbed by a branch of $CPBT$, this function returns -1. Otherwise the branches of $CPBT$ that are absorbed by α' are removed and the function returns 0. Finally, the branch α' is inserted in $CPBT$ in lines 4 to 6 by calling the function $InsertInTree$ (Algorithm 6). So while inserting the episode α in $CPBT$, it should be considered whether α absorbs the other episodes or is absorbed by another episode. In the following section, the functions called by the function $InsertInCPBT$ are presented in detail.

(1) **Function** $CreateBranch$**:** The algorithm $CreateBranch$ (Algorithm 9 in "Appendix B") receives the FC episode α and its frequency, converts them into a branch of $CPBT$ and returns a pointer to this branch. Time complexity of the algorithm is discussed in Lemma 11 in "Appendix A".

Algorithm 4 InsertInCPBT

Input: α, $freq_\alpha$ % α is an episode and $freq_\alpha$ is its frequency
Output: % **The algorithm inserts** α **in** $CPBT$;
1: $R \leftarrow CPBT$;
2: $\alpha' \leftarrow CreateBranch(\alpha, freq_\alpha)$;
3: $v \leftarrow SearchInTree(\alpha', R, |CNG_\alpha|)$;
4: **if** $(v = 0)$ **then**
5: $InsertInTree(\alpha', R, 1, |CNG_\alpha|)$
6: **end if**

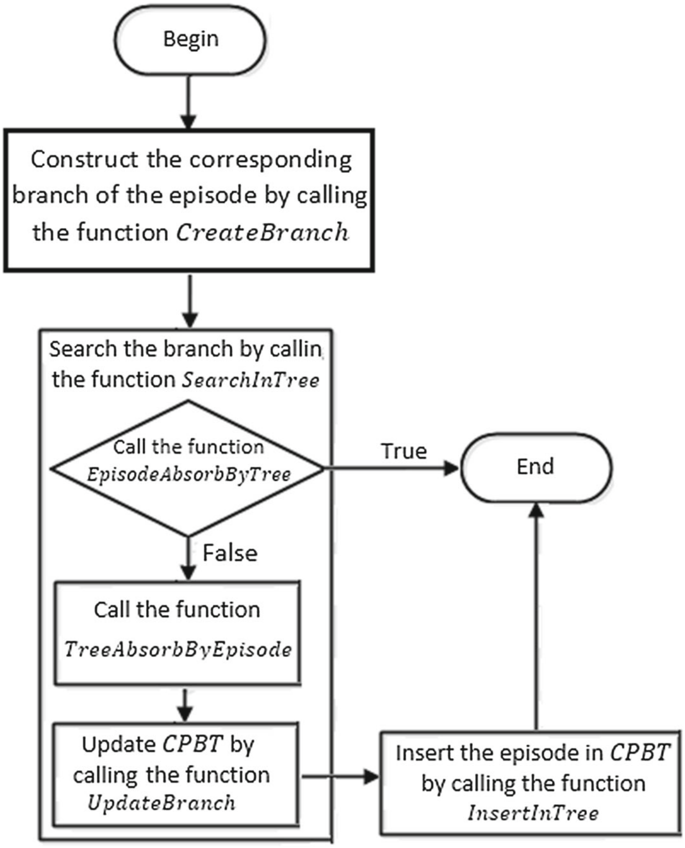


Fig. 13 The flowchart of the function $InsertInCPBT$ to insert the FC episodes in $CPBT$

Definition 23 Given the node n of $CPBT$, $Episode(n)$ is the corresponding episode of a branch that starts from the root and ends in the node n.

(2) **Function** $SearchInTree$: This function (Algorithm 5) searches an episode in $CPBT$ and checks whether the episode is absorbed by a branch of the tree or not. If it is not absorbed, the function checks whether the episode absorbs branches of $CPBT$ or not. As Algorithm 5 shows, by calling the function $EpisodeAbsorbByTree$ (Algorithm 10 in "Appendix B") in lines 2 to 4, it is checked whether the branch α' is absorbed by a branch of $CPBT$ or not. If α' is absorbed, the value 0 is returned, which shows α' should not be inserted in $CPBT$. If α' is not absorbed, it should be checked whether α' could absorb branches of $CPBT$ or not. If it could, before the α' is inserted, these branches should be removed from $CPBT$. The stack $Path$ defined in line 1, stores the path of branches that should be removed. The function $TreeAbsorbByEpisode$ (Algorithm 11 in "Appendix B") in line 6 finds these $Path$s and inserts them into the global variable $PathList$. Finally, in lines 7 to 9, the function $UpdateBranch$ (Algorithm 12 in "Appendix B") is called to update the corresponding branches of the $Path$s in $PathList$.

Algorithm 5 SearchInTree

```
Input: α′, R, |CNG_α|        % α′ is the corresponding branch of the episode α, R is a node of CPBT
Output: -1: if α′ is absorbed by another episodes; Otherwise 0
Path: an empty stack;        % Path is a stack to store the corresponding path of the episodes in CPBT
if (EpisodeAbsorbByTree(α′, R, 1, |CNG_α|)) then
    return -1;
end if
PathList ← ∅;        % PathList is a global variable and is a list of Paths.
TreeAbsorbByEpisode(α′, R, 1, Path, |CNG_α|);
for (j = 1 to |PathList|) do
    UpdateBranch(R, α′.freq, PathList[j]);
end for
return 0;
```

Algorithm 6 InsertInTree

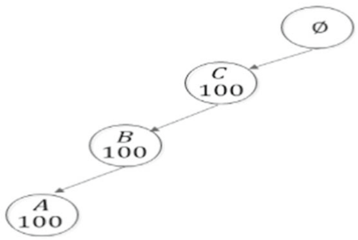
```
Input: α′, R, i, |CNG_α|        % α′ is the corresponding branch of the episode α, R is a node of CPBT
Output:        % The function inserts the branch α′ into the subtree of the node R in CPBT
1: flag ← false;
2: for each (x ∈ R.children) do
3:     if (α′.label = x.label) then
4:         x.freq = x.freq + α′.freq;
5:         if (i < |CNG_α|) then
6:             α′ ← α′.children[1];
7:             InsertInTree(α′, x, i + 1, |CNG_α|);
8:         end if
9:         flag ←True;
10:        break;
11:    end if
12: end for
13: if (!flag) then
14:     add α′ to R.children;
15: end if
```

(3) **Function** $InsertInTree$**:** This function (Algorithm 6) inserts the corresponding branch of the episode α, α', in the subtree of the node R in $CPBT$. As Algorithm 6 shows, in lines 2 to 12, a child of R whose label is equal to the label of α' is found. Note that there exists just one such node because the labels of nodes are unique. Since this node is shared with the branch α', so the frequency of α' is added to the node's in line 4. In lines 5 to 8, the following nodes of the branch α' are traversed and this process is repeated. While inserting α', if no node whose label is equal to the label of α' is found, the value of $flag$ remains False. So in lines 13 to 15, α' is added to the children of R as a new child.

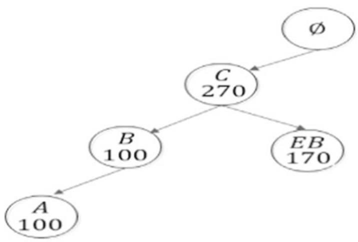
Theorem 3 *The algorithm $AllClosedFreqEpisodes$ only finds all the closed episodes.*

Example 13 Given $RS_1 = \{A, B, C, E, F, G\}$, where $RS_1 \subseteq RS$, assume the FC episodes α_i, $i = 1, 2, 3, 4, 5$ are identified as follows:

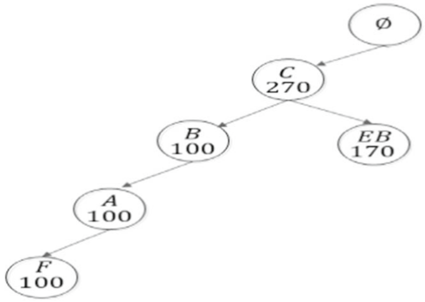
$$\alpha_1 = A \to B \to C,\ freq(\alpha_1) = 100 \quad \alpha_2 = BE \to C,\ freq(\alpha_2) = 170$$
$$\alpha_3 = B \to C,\ freq(\alpha_3) = 170 \quad \alpha_4 = F \to A \to B \to C,\ freq(\alpha_4) = 100$$



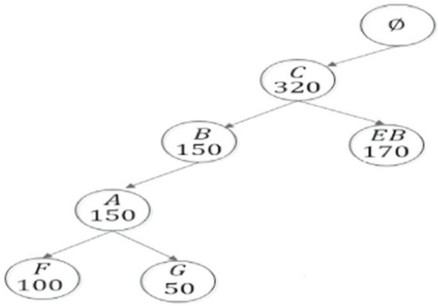
(a) *CPBT* after inserting α_1



(b) *CPBT* after inserting α_2



(c) *CPBT* after inserting α_4



(d) *CPBT* after inserting α_5

Fig. 14 Inserting the FC episodes of Example 13 in $CPBT$

$$\alpha_5 = G \rightarrow A \rightarrow B \rightarrow C,\ freq(\alpha_5) = 50$$

We define the lexicographic order on RS as $A < B < C < E < F < G$. Therefore, based on the lexicographic tree of episodes, we have $\alpha_1 < \alpha_2 < \alpha_3 < \alpha_4 < \alpha_5$ [11]. It is clear that the function $InsertInCPBT$ is called for episodes in ascending order:

- α_1: Since $CPBT$ has no branch, α_1 is inserted in it. Figure 14a shows $CPBT$ after inserting α_1.
- α_2: As Fig. 14b shows, the episode α_2 is also inserted in $CPBT$.
- α_3: When the function $InsertInCPBT$ is called for α_3, the function $SearchInTree$ returns -1 because the function $EpisodeAbsorbByTree$ detects that α_3 is absorbed by α_2. Therefore, α_3 is not inserted in $CPBT$.
- α_4: Since none of the branches of $CPBT$ absorbs α_4, the function $TreeAbsorbByEpisode$ is called for α_4. It detects that α_1 is absorbed by α_4. As Fig. 14c shows, α_1 is replaced with α_4.
- α_5: α_5 is absorbed by no episode. Furthermore, it absorbs no episode. Therefore, it is inserted in $CPBT$ as Fig. 14d shows.

The function $ExtractClosedEpisodesFromCPBT$ (see Algorithm 3 in "Appendix B") extracts the closed episodes stored in $CPBT$ and provides fast access to them.

5.4 Analysis of time complexity

In general, the running time of an algorithm is roughly proportional to how many times some basic operation is done [45]. To analyze time complexity, we consider the comparison of the FC/candidate closed episodes as the basic operation. In the hashing approach, all the candidates with the same frequency are hashed to the same bucket in the hash table. If there are v distinct frequency values, then there are v buckets such that $|bucket_i| = l_i, 1 \leq i \leq v$ and $\sum_{i=1}^{v} l_i = |CandidateClosedEpisodes| = |FC\ Episodes|$. Among the candidate patterns which are hashed to the same bucket, those patterns for which a super-pattern with the same frequency is found, are discarded. So the number of comparisons is $\sum_{i=1}^{v} |bucket_i|^2 \leq |CandidateClosedEpisodes|^2$. It means that time complexity of the hashing approach is $O(|CandidateClosedEpisodes|^2)$. Since the maximum number of CNGs of episodes is $Level$, the height of $CPBT$ is also $Level$. Therefore, in our approach, the number of comparisons is $O(BranF \times Level)$, where $BranF$ is the branching factor of $CPBT$. **In the worst case**, the branching factor of $CPBT$ is $MaxBranF = \sum_{|CNG|=1}^{N} \binom{N}{|CNG|} M^{|CNG|}$, where $|ResourceType| = N$ and $|Status| = M$. As we will see in Sect. 6.2, we should choose the small values such as 6 for $Level$. In addition, M and N are not large (in this paper, we set $M = N = 4$). In general, $BranF$ depends on the extracted FC episodes and in practice $BranF \ll MaxBranF$. As we will see in Sect. 6, if $|CandidateClosedEpisodes|$ is small, the hashing approach is a good choice. Otherwise, our approach could identify closed episodes much faster than the hashing approach.

6 Evaluation

In [2,11], we evaluate POSITING and RELENTING on both the real and synthetic workloads comprehensively and investigate the impact of different parameters on them. According to the main concepts introduced in Sect. 3.1, there are some parameters for the pattern mining engine: δ, Δ, ϵ, μ, $Level$, θ. The parameters setting for the evaluation of the proposed approach is as follows:

- Δ and δ: As we mentioned in Sect. 3.1, δ and Δ are internal gaps, which determine the starting interval of CNGs. The values of δ depend on the time spent on booting VMs. In [2,11], the valid interval of Δ is $[\delta, \delta + \epsilon]$. To provide a more general approach for mining closed episodes in different fields and conduct more comprehensive experiments, we extend the valid interval of Δ as follows:
    - Given the episode $\alpha = G'_1 \rightarrow G'_2 \rightarrow \cdots \rightarrow G'_k$ and the occurrence $O = ([t_j, t'_j]_{j=1}^{k}) \in LOList(\alpha)$ such that $k > 2$ and $1 \leq i < k - 1$, if there is overlap between $VI([t_i, t'_i], i + 1)$ and $VI([t_{i+1}, t'_{i+1}], i + 2)$, then the two serial episodes $\beta = G'_1 \rightarrow \cdots \rightarrow G'_i \rightarrow G'_{i+1}$ and $\gamma = G'_1 \rightarrow \cdots \rightarrow G'_i \rightarrow G'_{i+2}$ could be extracted. Based on these episodes, different status could be predicted for the next slots. So Δ should be selected in a way that the precise prediction is possible and extracting redundant episodes is avoided. For this purpose, as (6.1) implies Δ should be in the interval of $[\delta, 2\delta)$:

$$\left.\begin{array}{r} t'_{i+1} \geq t'_i + \delta \\ t'_{i+1} + \delta > t'_i + \Delta \end{array}\right\} \rightarrow \Delta < t'_{i+1} - t'_i + \delta \rightarrow \Delta < 2\delta \rightarrow \Delta \in [\delta, 2\delta) \tag{6.1}$$

- ϵ: According to Definition 1, the span of the events should be greater than ϵ. ϵ should be determined based on the length of sampling intervals. Since the real workloads [46,47] are coarse-grained and the synthetic workloads are also generated in a similar way to them, we set $\epsilon = 0$ in all the experiments.
- μ: The events whose span is large, are decomposed into two events based on the decomposition unit μ. For smooth workloads, the small values of μ might lead to generating many events, which could increase the duration of the episode extraction. On the other hand, the large values of μ might lead to the inability to extract all the hidden useful episodes. We evaluate the impact of μ on the pattern mining engine for both the real and synthetic workloads.
- θ: It is a threshold value that is used to extract frequent episodes. It is clear that the small values of θ might lead to identifying a huge number of episodes, which might be very time-consuming. On the other hand, the large values of θ could lead to losing some useful episodes for prediction. We evaluate the impact of θ on the pattern mining engine for both the real and synthetic workloads.
- $Level$: To avoid enlarging the pattern tree, $Level$ limits the length of episodes. As $Level$ increases, the height of the pattern tree, the number of episodes and the time consumed to identify them increase. So we investigate the impact of $Level$ on the pattern mining engine for both the real and synthetic workloads.

Since the focus of the paper is on the pattern mining engine, we evaluate the efficiency of the proposed approach to extract closed episodes. For this purpose, the effect of two important parameters θ and μ is considered on the pattern mining engine for both the synthetic and real workloads.

6.1 Workloads

The data set GWA-T-12[2] Bitbrains contains the performance metrics of 1750 VMs from a distributed data center from Bitbrains, which is a service provider that specializes in managed hosting and business computation for enterprises. The workload traces are corresponding to requested and actually used resources in a distributed data center servicing business-critical workloads. The data set focuses on four key types of resources, which can become bottlenecks for business-critical applications: CPU, disk I/O, memory and network I/O. For each VM, the performance metrics are sampled every 5 min. The traces include data for 1750 nodes, with over 5000 cores and 20 TB of memory, and operationally include over 5 million CPU hours in 4 operational months [48].

Since the workloads of the data set GWA-T-12 Bitbrains are more dynamic than the other public workloads [48], in a similar way to [11], we use these workloads for evaluation. Furthermore, we use the synthetic workloads generated in [2,11]. Table 1

[2] These traces can be accessed at http://gwa.ewi.tudelft.nl/datasets/Bitbrains.

Table 1 The types of the synthetic workloads and their embedded episodes [11]

Embedded episodes	Type of the synthetic workload	Parameters of the episode
$\alpha : (Memory, Low)(Disk, Verylow) \rightarrow (CPU, Low)(Network, High)$	$SWT1$	$\epsilon = 0$
$\beta : (CPU, Low)(Network, Low) \rightarrow (Memory, High)(Disk, Medium) \rightarrow (CPU, High), (Network, Medium)$	$SWT2$	$\epsilon = 0$
$\alpha : (Memory, Low)(Disk, Verylow) \rightarrow (CPU, Low)(Network, High)$	$SWT3$	$\epsilon = 0$
$\beta : (CPU, Low)(Network, Low) \rightarrow (Memory, High)(Disk, Medium) \rightarrow (CPU, High), (Network, Medium)$		

shows the types of the generated synthetic workloads and their corresponding embedded episodes. Since the time it takes to instantiate a new VM instance is about 5–15 min [49] and VMs are sampled every 5 min, so three values 1, 2 and 3 should be evaluated for δ.

6.2 Impact of *Level*

Since both the methods extend the pattern tree the same, we only report the impact of $Level$ on the hashing approach. For this purpose, we set $\theta = 0.1$, $\mu = 3$ and $\delta = \Delta$ and investigate the impact of $Level$ on both the real and synthetic workloads.

Impact of $Level$ on the real workload: One VM, called VM_1, is selected randomly from GWA-T-12 to evaluate the impact of $Level$ on the number of episodes and the processing time to identify them. Table 2 shows the impact of $Level$ on VM_1 for $\delta = \Delta = 1, 2, 3$. According to the table, as $Level$ increases, the number of episodes and the time consumed to identify them increase. Although there is no significant change in the number of episodes for $Level > 6$, the processing time increases strongly. So, according to the bolded row, there is a trade-off between the processing time ($Time$) and the number of episodes for $Level = 6$.

Impact of $Level$ on the synthetic workload: As Table 3 shows, For each workload type, one trace is generated by the workload generator. The distinct values of δ are selected for each workload type randomly. Table 4 shows the impact of $Level$ on $Trace_i, i = A, B, C$. According to the table, as $Level$ increases, the number of episodes and the time consumed to identify them increase. In a similar way to the real workloads for $Level > 6$, although there is no significant change in the number of episodes, the processing time increases strongly. According to the bolded rows, for all the traces, there is a trade-off between the processing time and the number of episodes for $Level = 6$. So in all of the experiments, we set $Level = 6$.

Table 2 The impact of the parameter $Level$ on the pattern tree for $VM_1(\mu = 3, \theta = 0.1)$

$\delta = \Delta$	$Level$	$\|Episodes\|$	$\|CandidateClosedEpisodes\|$	$\|ClosedEpisodes\|$	$Time(s)$
1	2	299	255	140	1.77
	4	1556	877	222	8.98
	6	**3826**	**1774**	**247**	**26.1**
	8	15,833	6962	255	85.74
2	2	231	187	131	1.77
	4	1420	813	239	8.33
	6	**4372**	**2080**	**261**	**27.6**
	8	10,071	4000	264	73.38
3	2	295	252	135	1.93
	4	4163	2786	323	20.39
	6	**40290**	**21174**	**435**	**210.4**
	8	344,975	155,787	458	3029.9

Table 3 The traces generated from the different types of the synthetic workload

Name	Type of synthetic workload	$\delta = \Delta$
$Trac_A$	$SWT1$	3
$Trac_B$	$SWT2$	1
$Trac_C$	$SWT3$	2

Table 4 The impact of the parameter $Level$ on the pattern tree for $Trace_i, i = A, B, C$ $(\mu = 3, \theta = 0.1)$

Name of Trace	$Level$	$\|Episodes\|$	$\|CandidateClosedEpisodes\|$	$\|ClosedEpisodes\|$	$Time(s)$
$Trace_A$	2	666	529	97	8
	4	60,729	47,077	306	540.7
	6	**169511**	**127195**	**442**	**6716.5**
	8	180,449	135,217	450	10204.3
$Trace_B$	2	447	316	60	6.8
	4	10,788	8170	147	170
	6	**743,678**	**473,002**	**292**	**15,667**
	8	1,238,648	706,872	318	27453.6
$Trace_C$	2	963	688	261	6
	4	8049	5600	780	59.8
	6	**120,647**	**69,262**	**1134**	**650.4**
	8	406,889	222,227	1247	3489.6

6.3 Evaluation results

To the best of our knowledge, the hashing approach is the only approach that is used in different literature to identify closed patterns among frequent patterns [12,13,44].

Therefore, to evaluate the performance and efficiency of the approach proposed in this paper, the approach is compared to the hashing approach [13]. For evaluation, there are four parameters δ, Δ, θ and μ, which should be investigated. As it was mentioned the valid values of δ are 1, 2 and 3 and Δ is in the interval of $[\delta, 2\delta)$. The parameter θ is in the interval of [0, 1]. Since $\mu \geq \max(2\epsilon + 1, \epsilon + 2)$ [11] and $\epsilon = 0$, then we have $\mu \geq 2$. If μ is bound to 10 and evaluated in the steps of 1 and θ is evaluated in the steps of 0.1, then there are $6 \times 9 \times 10 = 540$ distinct combinations of the parameters for evaluation. Due to space limitation, we select some combinations of the parameters and investigate the impact of the parameters on the pattern extraction of some VMs. The valid values of the parameters δ and Δ could be divided into the three groups $\delta = \Delta$ and $\delta < \Delta$ with spans of 1 and 2 time slots. So we select ($\delta = 1$, $\Delta = 1$) from the group $\delta = \Delta$, ($\delta = 2$, $\Delta = 3$) from the group $\delta < \Delta$ with the span of 1 and ($\delta = 3$, $\Delta = 5$) from the group $\delta < \Delta$ with the span of 2. The impact of $\Delta - \delta$ on episode mining is investigated on the real workloads. Since the main goal of cloud is to satisfy SLA and avoid wasting resources [3], the occurrence time of the future events should be determined precisely. So the main focus of evaluation is on the group $\delta = \Delta$. The values of δ and Δ are selected from this group randomly for the synthetic workloads. All of the experiments run on a machine with an Intel Core 2 Duo 2.53 GHz processor and 4GB of RAM.

6.3.1 Experimental results of the real workload

In addition to VM_1, We select two other VMs of GWA-T-12 randomly, called VM_2 and VM_3. Each VM is evaluated for distinct values of δ and Δ: $VM_1(\delta = 1, \Delta = 1)$, $VM_2(\delta = 2, \Delta = 3)$ and $VM_3(\delta = 3, \Delta = 5)$. For each VM, the impact of the parameters μ and θ on the number of patterns and the processing time is investigated. Note that since the same sets of closed episodes are extracted from each VM by using both of the methods, the number of closed episodes is only reported for the hashing approach.

Impact of θ: To consider the impact of θ, the values of μ and $Level$ are set to 3 and 6 respectively. For different values of θ, Table 5 shows the number of episodes, candidate closed and closed episodes, the extraction time of candidate closed episodes ($Time_E$) and the processing time of candidate closed episodes ($Time_P$) in seconds for the hashing approach. The symbol ∞ indicates that the hashing approach could not complete the extraction of candidate closed episodes due to insufficient memory. In this case, the number of closed episodes is reported by using the proposed approach. As the table shows, the small values of θ increase the number of extracted episodes (and closed episodes) and the time consumed to identify them. According to the table, as the span of $\Delta - \delta$ increases, the number of candidate and closed episodes increases abruptly. For example, if $\theta = 0.1$, for VM_1 with $\Delta - \delta = 0$, the number of closed episodes is 247, for VM_2 with $\Delta - \delta = 1$, the number of closed episodes is 22166 and for VM_3 with $\Delta - \delta = 2$, the hashing approach could not extract candidate closed episodes due to insufficient memory. On the other hand, as the table shows for VM_2 with $\theta = 0.1$ and VM_3 with $\theta = 0.3$, for a large number of candidate closed episodes, processing candidate closed episodes consumes more time in comparison

Table 5 The impact of the parameter θ on extracting closed episodes from $VM_i, i = 1, 2, 3$ using the hashing approach ($\mu = 3$)

Name of VM	θ	$\|Episodes\|$	$\|CandidateClosedEpisodes\|$	$\|ClosedEpisodes\|$	$Time_E(s)$	$Time_P(s)$
VM_1	0.1	3826	1774	247	25.99	0.17
	0.2	1214	577	133	15.05	0.01
	0.3	203	125	80	7.6	0.008
	0.4	101	78	56	7.13	0.008
	0.5	50	40	35	2.93	0.008
	0.6	28	23	22	2.63	0.004
	0.7	23	20	19	2.61	0.003
	0.8	20	17	16	2.62	0.003
	0.9	11	10	10	0.9	0.004
	1	11	11	10	0.88	0.004
VM_2	0.1	341,629	198,652	22,166	1109.27	3855.49
	0.2	123,055	68,502	10,053	428.78	339.92
	0.3	94,059	50,978	5843	300.22	181.06
	0.4	42,679	21,904	2078	114.63	41.44
	0.5	40,172	20,368	1380	94.41	33.42
	0.6	36,722	18,439	805	79.52	34.02
	0.7	30,964	15,476	321	59.74	28.3
	0.8	29,536	14,770	302	61.38	29.82
	0.9	29,182	14,609	242	54.87	24.47
	1	27,072	13,479	160	52.42	22.65

Table 5 continued

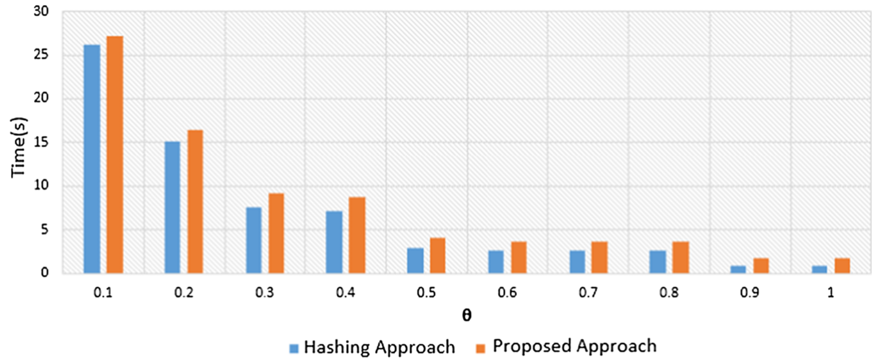
Name of VM	θ	$\lvert Episodes \rvert$	$\lvert CandidateClosedEpisodes \rvert$	$\lvert ClosedEpisodes \rvert$	$Time_E(s)$	$Time_P(s)$
VM_3	0.1	∞	∞	397,020	∞	∞
	0.2	∞	∞	∞	104578	∞
	0.3	937,947	569,369	36,351	2181.56	26562.2
	0.4	113,973	75,394	16,327	287.50	245.41
	0.5	58,099	38,012	9091	159.99	85.43
	0.6	48,138	31,090	5827	135.99	68.93
	0.7	29,170	18,619	1676	75.50	25.91
	0.8	27,848	17,730	1145	73.25	29.12
	0.9	24,945	15,945	818	74.13	29.28
	1	22,080	14,112	677	51.88	15.56

to extracting them. Furthermore, the number of closed episodes is much fewer than candidate closed episodes'. These points imply that the hashing approach is not a good choice for small values of θ or large spans of $\Delta - \delta$.

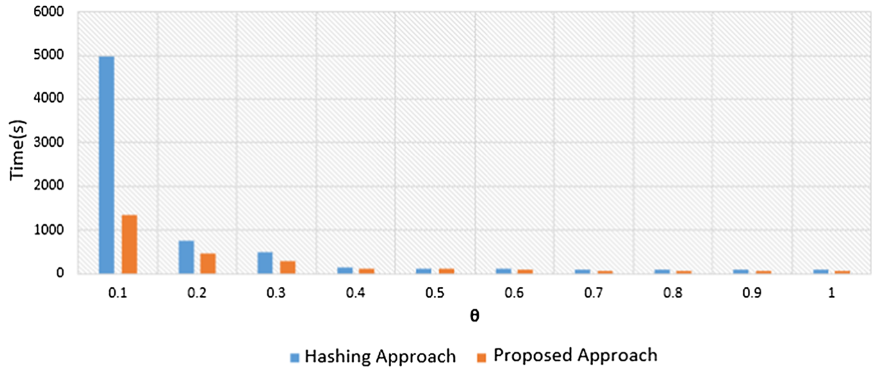
The total time consumed by the hashing approach to extract closed episodes is defined as $Time_E + Time_P$. Figure 15 shows the total time consumed to extract the closed episodes by the hashing approach and the proposed approach for different values of θ. According to the figure, as the value of θ increases, the consumed time decreases. Furthermore, for VM_1 with $\Delta - \delta = 0$ (Fig. 15a), the consumed time of the proposed approach is reasonable and similar to the hashing approach's. As the span of $\Delta - \delta$ increases for VM_2 and VM_3 (Fig. 15b, c), the number of candidate closed episodes increases and the total time consumed by the hashing approach increases abruptly. As Fig. 15 shows the proposed approach extracts the closed episodes much faster than the hashing approach for small values of θ and large spans of $\Delta - \delta$ because it extracts closed episodes directly without storing/processing all of the candidate closed episodes.

Impact of μ: To evaluate the impact of μ, we set $\theta = 0.1$ and $Level = 6$. Since $\mu \geq \max(2\epsilon + 1, \epsilon + 2)$ and $\epsilon = 0$, then we have $\mu \geq 2$. Table 6 shows the impact of μ in the interval of [2, 10] on the number of patterns and the consumed time of the hashing approach. For small values of μ such as 2 and 3, the number of candidate closed episodes increases abruptly. On the other hand, as the span of $\Delta - \delta$ increases the number of episodes increases suddenly. So for VM_2 with $\mu = 2$ and VM_3 with $\mu = 2, 3$, the hashing approach could not complete the extraction of candidate closed episodes due to insufficient memory. In these cases, the number of closed episodes is reported by using the proposed approach. It is clear that as μ increases, the span of events increases and the number of events decreases subsequently. So the number of episodes decreases as μ increases. Furthermore, for VM_2 with $\mu = 3$ and VM_3 with $\mu = 4$, processing candidate closed episodes consumes more time in comparison to extracting them due to a large number of candidate closed episodes. Therefore, since the hashing approach extracts a large number of candidate closed episodes, it could not be an appropriate choice for small values of μ or large spans of $\Delta - \delta$.

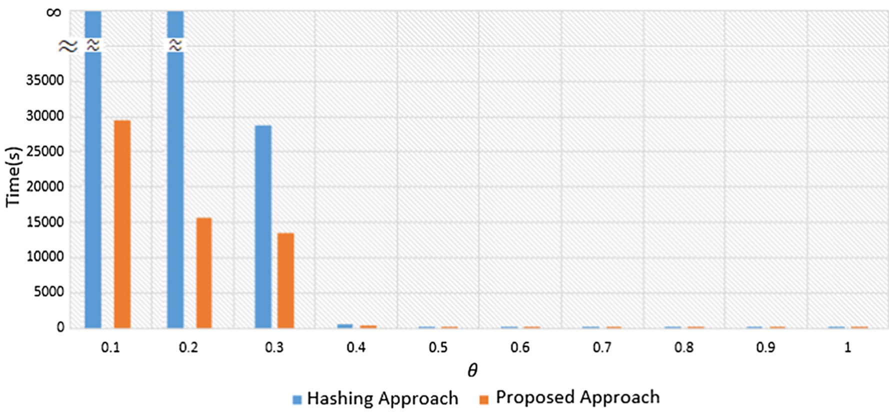
Figure 16 shows the total time consumed by the hashing approach and the proposed approach to extract the closed episodes for different values of μ and $\theta = 0.1$. According to the figure, as the value of μ increases, the consumed time decreases. Furthermore, for VM_1 with $\Delta - \delta = 0$ (Fig. 16a), the processing time of the hashing approach is reasonable. As the span of $\Delta - \delta$ increases for VM_2 and VM_3 (Fig. 16b, c), the number of candidate closed episodes increases and the total time consumed by the hashing approach increases abruptly. As Fig. 16 shows the proposed approach extracts closed episodes much faster than the hashing approach for small values of μ and large spans of $\Delta - \delta$. For example, for VM_3, the time consumed by the hashing approach for $\mu = 4$ is nearly equal to the time consumed by the proposed approach for $\mu = 2$. All these points show that extracting closed episodes without storing/processing all of the candidate closed episodes improves the mining efficiency significantly.



(a) The total time consumed to extract closed episodes from VM_1



(b) The total time consumed to extract closed episodes from VM_2



(c) The total time consumed to extract closed episodes from VM_3

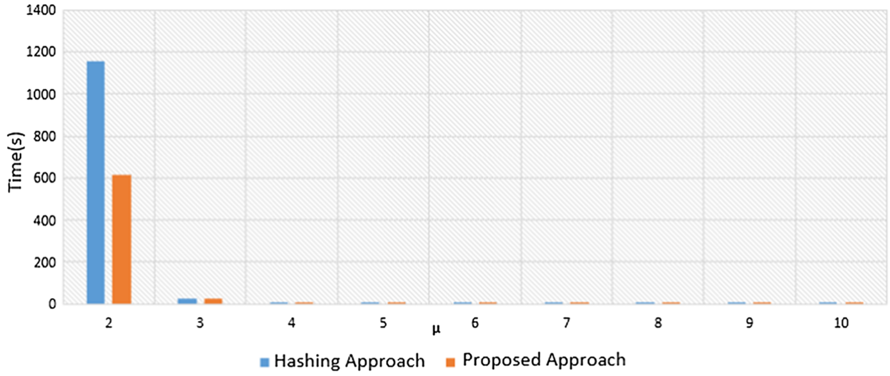
Fig. 15 The total time consumed by the hashing approach and the proposed approach to extract closed episodes from VM_i, $i = 1, 2, 3$ for different values of θ and $\mu = 3$

Table 6 The impact of the parameter μ on extracting closed episodes from $VM_i, i = 1, 2, 3$ using the hashing approach ($\theta = 0.1$)

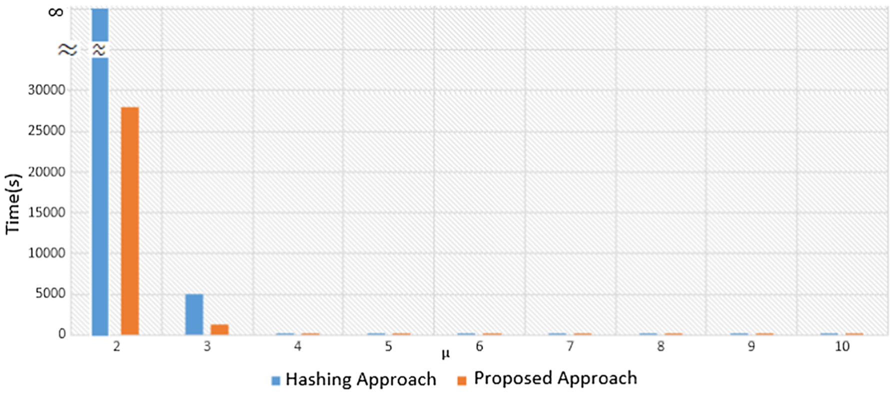
Name of VM	μ	$\|Episodes\|$	$\|CandidateClosedEpisodes\|$	$\|ClosedEpisodes\|$	$Time_E(s)$	$Time_P(s)$
VM_1	2	144136	69658	1200	649.95	507.49
	3	3826	1774	247	25.99	0.17
	4	1624	823	176	9.19	0.03
	5	1097	514	129	5.7	0.02
	6	673	334	103	4.11	0.01
	7	662	314	106	3.95	0.01
	8	773	402	85	4.07	0.02
	9	356	196	79	2.28	0.01
	10	548	286	85	2.81	0.01
VM_2	2	∞	∞	310780	∞	∞
	3	341,629	198,652	22,166	1109.27	3855.49
	4	12,116	6205	853	57.15	1.86
	5	14,204	7195	819	73.60	2.14
	6	1760	816	300	9.25	0.04

Table 6 continued

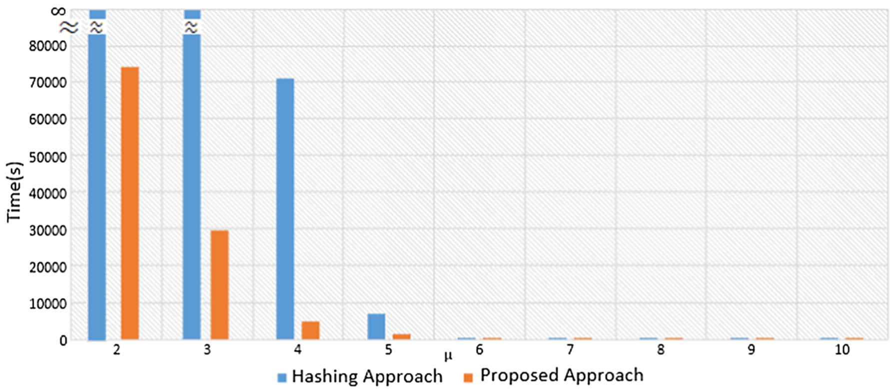
Name of VM	μ	$\|Episodes\|$	$\|CandidateClosedEpisodes\|$	$\|ClosedEpisodes\|$	$Time_E(s)$	$Time_P(s)$
	7	426	260	111	3.03	0.01
	8	513	290	110	4.29	0.01
	9	374	232	106	2.74	0.01
	10	329	195	96	2.35	0.01
VM_3	2	∞	∞	2305884	∞	∞
	3	∞	∞	397020	∞	∞
	4	1,528,657	877,433	81,440	3916.55	70508.1
	5	412,733	239,247	15,208	1173.31	5629.63
	6	70,410	35,116	3247	192.94	132.04
	7	33,063	16,402	1727	129.32	21.34
	8	34,873	17,737	1689	124.64	14.15
	9	18,179	10,144	1114	84.53	4.42
	10	12,348	6907	837	55.82	2.06



(a) The total time consumed to extract closed episodes from VM_1



(b) The total time consumed to extract closed episodes from VM_2



(c) The total time consumed to extract closed episodes from VM_3

Fig. 16 The total time consumed by the hashing approach and the proposed approach to extract closed episodes from $VM_i, i = 1, 2, 3$ for different values of μ and $\theta = 0.1$

6.3.2 Experimental results of the synthetic workload

In this section, the impact of the two parameters θ and μ on the number of closed episodes extracted from the synthetic workloads and the time consumed by the methods to identify them is considered. We use the traces generated in Table 3 and compare the time consumed by the hashing approach with the proposed approach's.

Impact of θ: To consider the impact of θ, the value of μ is set to 3. For different values of θ, Table 7 shows the number of episodes, candidate closed and closed episodes, $Time_E$ and $Time_P$ in seconds for the hashing approach. According to the table, the small values of θ increase the number of extracted episodes (and closed episodes) and the time consumed to extract them. As the table shows for $Trace_B$ with $\theta = 0.1$, for a large number of candidate closed episodes, processing candidate closed episodes consumes more time in comparison to extracting them. So the time consumed by the hashing approach mainly depends on the number of candidate closed episodes. For example, $Trace_C$ with $\theta = 0.1$ needs the minimum time for the episode extraction due to the minimum number of candidate closed episodes. Furthermore, for all the traces, closed episodes are a small fraction of candidate closed episodes. These points imply that the hashing approach is not an appropriate choice for small values of θ because it generates a large number of candidate closed episodes.

Figure 17 shows the total time consumed by the hashing approach and the proposed approach to extract closed episodes for different values of θ. According to the figure, as the value of θ increases, the consumed time decreases for both the methods. Since a large number of candidate closed episodes are extracted for $Trace_A$ and $Trace_B$ with small values of θ, the proposed approach improves the consumed time for them significantly (Fig. 17a, b). As it is observed, for $Trace_B$ with a larger number of candidate closed episodes, the effect of the proposed approach on the consumed time is more prominent. On the contrary, since a smaller set of candidate closed episodes is extracted by the hashing approach for $Trace_C$, the consumed time of the hashing approach is less than the proposed approach's (Fig. 17c). However, the time consumed by both the methods is similar and comparable.

Impact of μ: To evaluate the impact of μ, we set $\theta = 0.1$. Table 8 shows the impact of μ in the interval of [2, 10] on the number of patterns and the consumed time of the hashing approach. Unlike the real workloads, there is no clear behavior of the impact of μ on the traces. The increase of μ does not show clear behavior on the training phase of $Trace_A$. For example, the number of candidate closed episodes for $\mu = 10$ is more than the number of candidate closed episodes for $\mu = 4$. On the other hand, for $\mu \geq 3$, there is no change in the number of candidate closed episodes of $Trace_B$. For $\mu \geq 8$, there is no significant change in the episode extraction of $Trace_C$. These results imply that the increase of μ does not affect the span of events of dynamic workloads. As the table shows the time consumed by the hashing approach mainly depends on the number of episodes and candidate closed episodes.

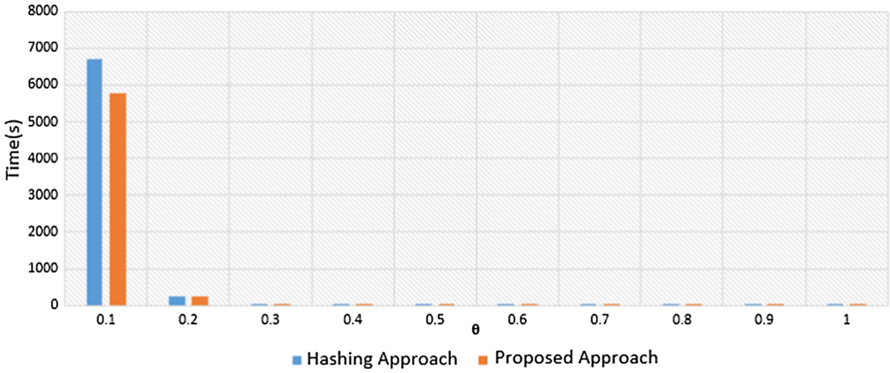
Figure 18 shows the total time consumed by the hashing approach and the proposed approach to extract the closed episodes for different values of μ and $\theta = 0.1$. According to the figure, there is no clear behavior of the impact of μ on the consumed time of both the methods. However, as the figure shows, since there are a large number of

Table 7 The impact of the parameter θ on extracting closed episodes from $Trace_i, i = A, B, C$ using the hashing approach ($\mu = 3$)

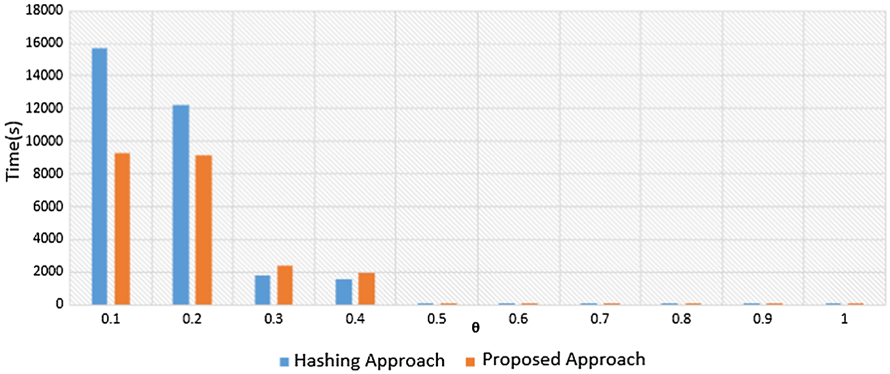
Name of Trace	θ	$\lvert Episodes \rvert$	$\lvert CandidateClosedEpisodes \rvert$	$\lvert ClosedEpisodes \rvert$	$Time_E(s)$	$Time_P(s)$
$Trace_A$	0.1	16,9511	127,195	42	6492.06	224.45
	0.2	4623	3836	158	234.25	0.28
	0.3	514	417	79	45.15	0.01
	0.4	391	321	70	25.79	0.01
	0.5	49	44	24	5.84	0.01
	0.6	40	36	21	5.39	0.01
	0.7	40	36	21	5.39	0.01
	0.8	40	36	21	5.32	0.01
	0.9	22	18	12	3.75	0.01
	1	9	8	7	1.53	0.004
$Trace_B$	0.1	743,676	473,002	298	7611.41	8055.44
	0.2	681,603	435,283	279	7004.45	5228.45
	0.3	44,972	28,290	119	1768.79	10.84
	0.4	38,526	23,500	89	1567.31	10.07
	0.5	196	167	25	15.53	0.01
	0.6	69	60	14	7.53	0.01
	0.7	60	54	10	6.28	0.01
	0.8	60	54	10	6.23	0.01
	0.9	60	54	10	6.31	0.01
	1	58	52	8	6.09	0.01

Table 7 continued

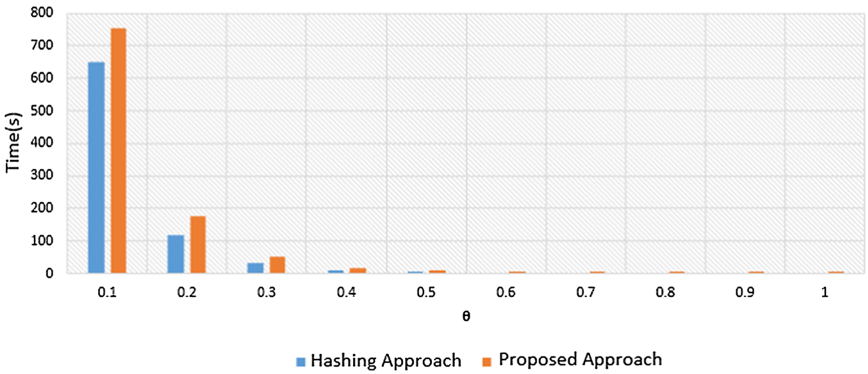
Name of Trace	θ	$\lvert Episodes \rvert$	$\lvert CandidateClosedEpisodes \rvert$	$\lvert ClosedEpisodes \rvert$	$Time_E(s)$	$Time_P(s)$
$Trace_C$	0.1	120,647	69,262	1134	596.01	54.4
	0.2	8344	5040	493	118.36	0.06
	0.3	1730	1163	245	32.28	0.01
	0.4	349	286	115	9.20	0.01
	0.5	162	135	67	5.47	0.09
	0.6	84	67	46	3.37	0.01
	0.7	58	45	34	2.67	0.01
	0.8	39	32	26	2.25	0.01
	0.9	29	24	20	1.82	0.01
	1	23	19	16	1.56	0.01



(a) The total time consumed to extract closed episodes from $Trace_A$



(b) The total time consumed to extract closed episodes from $Trace_B$



(c) The total time consumed to extract closed episodes from $Trace_C$

Fig. 17 The total time consumed by the hashing approach and the proposed approach to extract closed episodes from $Trace_i, i = A, B, C$ for different values of θ and $\mu = 3$

Table 8 The impact of the parameter μ on extracting closed episodes from $Trace_i, i = A, B, C$ using the hashing approach ($\theta = 0.1$)

Name of Trace	θ	$\|Episodes\|$	$\|CandidateClosedEpisodes\|$	$\|ClosedEpisodes\|$	$Time_E(s)$	$Time_P(s)$
$Trace_A$	2	135477	109218	1782	4928.08	99.67
	3	169,511	127,195	442	6492.06	224.45
	4	98,293	77,649	1154	3353.30	47.76
	5	105,317	88,275	891	3356.84	96.36
	6	107,382	87,118	903	3755.77	109.70
	7	98,142	78,100	886	2853.61	45.66
	8	101,336	80,229	966	2923.19	44.18
	9	106,493	85,118	891	3095.08	82.43
	10	113,295	92,323	819	3421.51	91.75
$Trace_B$	2	3,248,758	1,745,792	113,563	27593.17	23875.46
	3	743,676	473,002	298	8303.51	7363.49
	4	743,676	473,002	298	8481.92	6471.31
	5	743,676	473,002	296	8454.51	6652.56
	6	743,676	473,002	292	8586.73	6748.78
	7	743,676	473,002	292	8353.43	7604.75
	8	743,676	473,002	292	8442.85	6531.87
	9	743,676	47,3002	292	8353.42	6471.31
	10	743,676	473,002	292	8442.85	6531.87

Table 8 continued

Name of Trace	θ	$\|Episodes\|$	$\|CandidateClosedEpisodes\|$	$\|ClosedEpisodes\|$	$Time_E(s)$	$Time_P(s)$
$Trace_C$	2	459,717	295,821	62,920	3020.73	734.17
	3	120,647	69,262	1134	596.01	54.4
	4	8912	5948	444	67.69	0.32
	5	18,786	11,895	499	164.77	0.41
	6	1388	1062	252	22.65	0.01
	7	1140	881	188	17.63	0.01
	8	1118	866	174	17.18	0.01
	9	1118	866	174	15.84	0.01
	10	1118	866	174	15.06	0.01

candidate closed episodes for $Trace_A$ and $Trace_B$, the proposed approach reduces the consumed time for episode mining (Fig. 18a, b). According to the results, for a larger number of candidate closed episodes, the effect of the proposed approach on the consumed time is more eminent (Fig. 18b). On the contrary, although the time consumed by both the methods is comparable for $Trace_C$, the hashing approach needs less time due to a smaller set of candidate closed episodes.

7 Conclusion and future work

The prediction of the future workload of applications is an essential step before resource provisioning in cloud. This paper improves the efficiency of the previous predictors proposed based on pattern mining. The paper proposes a general approach, which not only improves time and space complexities of the pattern mining engine of the predictors, but also can be employed in different fields of SPM. To improve space complexity, redundant LOs are identified and omitted based on the improved vertical representation of the stream. To improve time complexity, a new data structure, called $CPBT$, is introduced to store closed episodes. Based on $CPBT$, a new approach is suggested to extract closed episodes directly. The experimental results show that for small values of the frequency threshold and the decomposition unit, the proposed approach improves the efficiency of mining closed episodes significantly in comparison to the hashing approach.

In the future work, we plan to conduct more experiments to evaluate the efficiency of the proposed approach for pattern mining in different fields. Furthermore, we plan to propose an approach to select the values of the parameters according to workload changes dynamically.

Acknowledgements The GWA-T-12 Bitbrains traces are provided by Bitbrains IT Services Inc., which is a service provider that specializes in managed hosting and business computation for enterprises. We thank the GWA team and all those who have graciously provided the data for us.

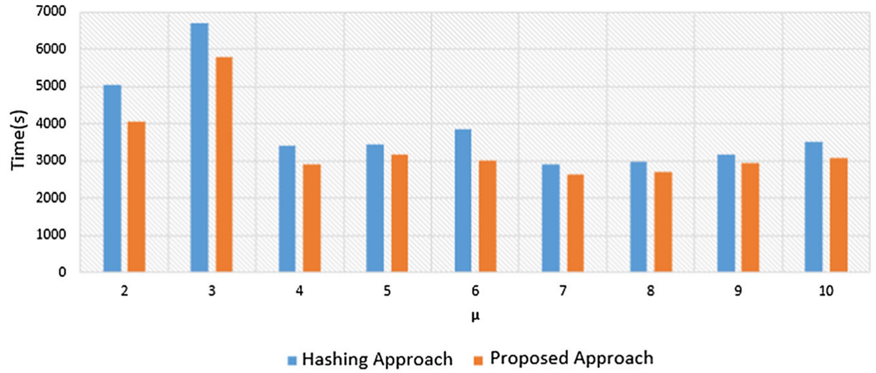
A Proofs

The proof of all of the theorems, lemmas and corollaries are presented in this appendix. Furthermore, we might present some new lemmas that are used to prove the other lemmas and theorems.

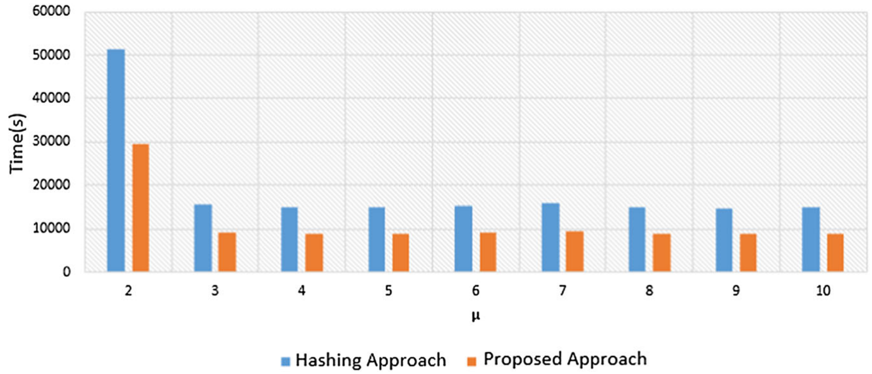
Lemma 5 *Given the episode* $\alpha = G'_1 \rightarrow \cdots \rightarrow G'_k$ *and the occurrence* $x = ([t_1^j, t_2^j]_{j=1}^k) \in LO(\alpha)$, *if there exists a valid occurrence* $y = ([w_1^j, w_2^j]_{j=1}^k)$ *such that* $t_1^1 < w_1^1$, *then* $t_1^k < w_1^k$.

Proof The proof is by induction on k: **Base case** for $k = 2$: The proof is by contradiction: Assume $w_1^2 < t_1^2$. We have:

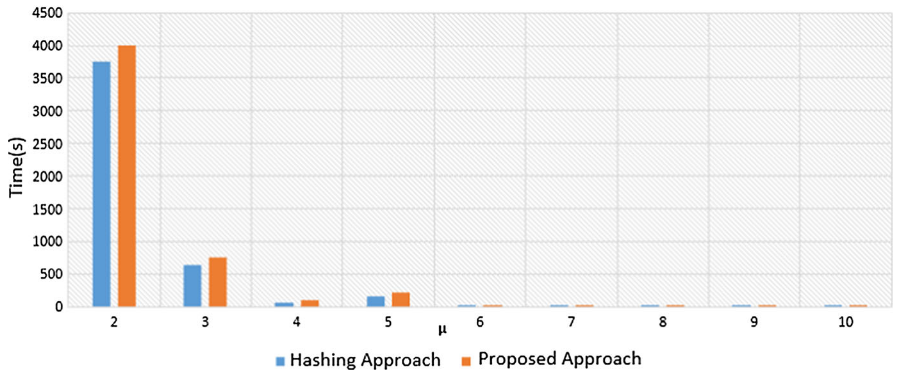
$$\left.\begin{aligned} w_2^1 + \delta < w_1^2 < w_2^1 + \Delta \\ t_2^1 + \delta < t_1^2 < t_2^1 + \Delta \\ t_2^1 < w_2^1 \end{aligned}\right\} \rightarrow w_2^1 + \delta < w_1^2 < t_1^2 < w_2^1 + \Delta \tag{A.1}$$



(a) The total time consumed to extract closed episodes from $Trace_A$



(b) The total time consumed to extract closed episodes from $Trace_B$



(c) The total time consumed to extract closed episodes from $Trace_C$

Fig. 18 The total time consumed by the hashing approach and the proposed approach to extract closed episodes from $Trace_i, i = A, B, C$ for different values of μ and $\theta = 0.1$

Therefore, x does not include LPO and it is not an LO. **Induction Step**: Assume it is true for $k = m - 1$. Now, we should prove it for $k = m$: The proof is by contradiction: Assume $w_1^m < t_1^m$. We have:

$$\left.\begin{array}{r} w_2^{m-1} + \delta < w_1^m < w_2^{m-1} + \Delta \\ t_2^{m-1} + \delta < t_1^m < t_2^{m-1} + \Delta \\ t_2^{m-1} < w_2^{m-1} \end{array}\right\} \rightarrow w_2^{m-1} + \delta < w_1^m < t_1^m < w_2^{m-1} + \Delta \quad \text{(A.2)}$$

It means that x does not include LPO. So it is not an LO, which is in contradiction to the assumption. □

Lemma 1 *Given the episode $\alpha = G'_1 \rightarrow \cdots \rightarrow G'_k$, if $MPO(\alpha)$ is a set of all the minimal prefix occurrences of α, then $LO(\alpha) \subseteq MPO(\alpha)$.*[3]

Proof The proof is by contradiction: assume there exists at least one LO $x = ([t_1^j, t_2^j]_{j=1}^k)$ such that $x \notin MPO(\alpha)$. Since $x \notin MPO(\alpha)$, so there should exist a valid occurrence $y = ([w_1^j, w_2^j]_{j=1}^k)$ where $w_1^1 > t_1^1$ and $w_1^k \leq t_1^k$. According to Lemma 5, for each valid occurrence $y = ([w_1^j, w_2^j]_{j=1}^k)$ that $t_1^1 < w_1^1$, we should have $t_1^k < w_1^k$. So x is not an LO, which is in contradiction to the assumption. □

Lemma 2 *Given the episode α, if β and γ are the serial and concurrent extensions of α, removing redundant occurrences from $LOList(\alpha)$ does not affect $freq(\alpha)$, $freq(\beta)$ and $freq(\gamma)$.*

Proof We define $OSet_M^N(\alpha) = \{O_1, \ldots, O_L\}$ as a set of all the non-overlapped minimal occurrences that O_1 is the first minimal occurrence of α and O_{i+1}, $1 \leq i < L$, is the first non-overlapped minimal occurrence after O_i. In [11], we proved that $OSet_M^N(\alpha)$ is a maximal non-overlapped set of the minimal occurrences of α in the stream and $freq(\alpha) = |OSet_M^N(\alpha)|$. The proof of the lemma includes three cases:

- Impact of removing redundant LOs on $freq(\alpha)$: according to the first condition of Definition 18, there is overlap between the two occurrences O and Q. So at most one of them could be in $OSet_M^N(\alpha)$. On the other hand, O is not a minimal occurrence. So $O \notin OSet_M^N(\alpha)$ and removing it does not affect $freq(\alpha)$.
- Impact of removing redundant LOs on $freq(\beta)$: If there exists a sub-interval of $[t_2^k + \delta, t_2^k + \Delta]$ such that the occurrence O covers it exclusively or O is a minimal occurrence, then O might be a non-overlapped occurrence of α and form a non-overlapped occurrence for β. Therefore, removing O might lead to losing a non-overlapped occurrence of β. According to the first condition of Definition 18, each non-overlapped occurrence of β whose LPO is O could be formed by using Q. On the other hand, there exists no sub-interval of $[t_2^k + \delta, t_2^k + \Delta]$ such that the occurrence O covers it exclusively. Therefore, removing O does not affect $freq(\beta)$.

[3] The proof of lemmas and theorems could be found in "Appendix A".

– Impact of removing redundant LOs on $freq(\gamma)$: If there exists the event $e = (v, s, st, et)$ such that $|t_2^k - st| < \epsilon$ and $|t_1^k - st| < \epsilon$, then removing O affects the frequency of $\gamma = \alpha \odot (r, s)$. So if there is no such event, removing O does not affect $freq(\gamma)$.

Therefore, if the three conditions are satisfied together, removing O does not affect $freq(\alpha)$, $freq(\beta)$ and $freq(\gamma)$. □

Lemma 6 *Given the episode α such that $|CNG_\alpha| = k$ and $\mu \geq \max(2\epsilon + 1, \epsilon + 2)$, the successive starting intervals of G_i, $1 \leq i \leq k$, have no overlap.*

Proof The proof is by contradiction: suppose there are two starting intervals $[t_1, t_2]$ and $[t_1', t_2']$ of G_i such that there is overlap between them: $t_1 \leq t_1' \leq t_2 < t_2'$. According to the definition of the episode occurrence in [11], $t_2 - t_1 \leq \epsilon$ and $t_2' - t_1' \leq \epsilon$. $\forall e = (r, s, st, et) \in E$ *that* $t_2 < st \leq t_2'$, then there should exist the other event $e' = (r' = r, s' = s, st', et')$ such that $t_1 \leq st' \leq t_2$. Since $et' \leq st$, if $t_1' \leq st' \leq t_2$ then $\Delta e' < \epsilon$. If $t_1 \leq st' < t_1'$ and $et' = st$, we have $\Delta e' = \mu < 2\epsilon$, which is in contradiction to $\mu > 2\epsilon$. If $t_1 \leq st' < t_1'$ and $et' < st$, there should exist the other event $e'' = (r, s'' \neq s, st'', et'')$ such that $et' \leq st'' < et'' \leq st'$. Since $\Delta e' > \epsilon$, we have $et' > t_2$ and $\Delta e'' < \epsilon$. These show that the successive starting intervals of G_i have no overlap. □

Lemma 7 *Given the episode α such that $|CNG_\alpha| = k$, $1 \leq i \leq k$, $\mu \geq \max(2\epsilon + 1, \epsilon + 2)$ and the two occurrences $O, O' \in OSet(\alpha)$, if $[u, u']$ is the starting interval of G_i in O, the following starting interval of G_i in O' is $[w, w']$ that $w > 2\epsilon + u$.*

Proof According to the definition of the occurrence O in [11], $\exists A_j^i \in G_i, 1 \leq i \leq k, j \in \{1, \ldots, l_i\}$ that $g_\alpha(A_j^i) = (r, s)$, $h(A_j^i) = a$ and $e_a = (r, s, st = u, et)$. Since $\Delta e_a > \epsilon$, we have $et > \epsilon + u$. According to Lemma 6, $w > u'$. For the occurrence O', $h'(A_j^i) = b$ such that $e_b' = (r' = r, s' = s, st' = v, et')$, $w \leq v \leq w'$, $\Delta e_b' > \epsilon$ and $et' > v + \epsilon$. If $st' > et$, there should exist the other event $e_m = (r, s_m \neq s, st_m, et_m)$ such that $st_m = et$. Since $\Delta e_a > \epsilon$ and $\Delta e_m > \epsilon$, then $w > 2\epsilon + u$. If $st' = et$, we have $\Delta e_a = \mu$. So $w - u = \mu > 2\epsilon$. Note that the condition $\mu > 2\epsilon$ is reasonable because the minimum span of events is $\epsilon + 1$ [11]. So the decomposition unit of events, μ, could be twice more than the minimum span. □

Lemma 8 *Given the episode $\alpha = G_1' \to \cdots \to G_k'$ and the two occurrences $O = ([t_1^i, t_2^i]_{i=1}^k)$ and $Q = ([w_1^i, w_2^i]_{i=1}^k)$, where $O, Q \in LO(\alpha)$, if $\exists j, 1 \leq j \leq k - 1$, such that for $r = 1, 2, \ldots, j - 1$: $t_1^r = w_1^r$, $t_2^r = w_2^r$ and $t_1^j < w_1^j$, then $t_1^k < w_1^k$ and $t_2^k < w_2^k$.*

Proof The proof is by induction on k: **Base case** for $k = 2$: we have j=1 and according to Lemmas 6 and 7 , $w_1^1 > t_2^1$. If $t_1^2 \in [t_2^1 + \delta, \min(t_2^1 + \Delta, w_2^1 + \delta - 1)]$, then we have $O \in LO(\alpha)$. If $w_1^2 \in [w_2^1 + \delta, w_2^1 + \Delta]$, then $Q \in LO(\alpha)$. So we have $w_1^2 > t_1^2$ and according to Lemmas 6 and 7 , $w_2^2 > t_2^2$. **Induction Step:** Assume it is true for $k = m - 1$. It should be proved for $k = m$. Since the lemma is correct for $k = m - 1$, so if $t_1^j < w_1^j$, $1 \leq j \leq m - 2$, then $t_2^{m-1} < w_2^{m-1}$. The starting interval of G_m' in O, t_1^m, should be in the interval of $[t_2^{m-1}+\delta, \min(t_2^{m-1}+\Delta, w_2^{m-1}+\delta-1)]$ because if $t_1^m < t_2^{m-1}+\delta$, then O is not a valid occurrence and if $t_1^m > \min(t_2^{m-1}+\Delta, w_2^{m-1}+\delta-1)$, then O is not a valid occurrence or since $w_2^{m-1} > t_2^{m-1}$ and $t_1^1 \leq w_1^j$, O cannot include the starting interval of G'_m (because O does not include an LPO).

The starting interval of G'_m in Q, w_1^m, should also be in the interval of $[w_2^{m-1}+\delta, w_2^{m-1}+\Delta]$ because gap constraints are satisfied. Since $t_1^1 \leq w_1^1$ and $w_2^{m-1} > t_2^{m-1}$, so Q could include the starting interval of G'_m because it includes an LPO. So we have:

$$\left.\begin{array}{r} w_1^m \geq w_2^{m-1}+\delta \\ t_1^m < w_2^{m-1}+\delta \end{array}\right\} \rightarrow t_1^m < w_1^m$$

On the other hand, according to Lemmas 6 and 7 , two occurrences of G'_m have no overlap. So we have $w_2^m \geq w_1^m > t_2^m$. For $j = m-1$, in a similar way to the previous case, if $t_1^m \in [t_2^{m-1}+\delta, \min(t_2^{m-1}+\Delta, w_2^{m-1}+\delta-1)]$, then O is an LO. If $w_1^m \in [w_2^{m-1}+\delta, w_2^{m-1}+\Delta]$, then Q is an LO. So we have $w_1^m > t_1^m$ and according to Lemmas 6 and 7, $w_2^m \geq w_1^m > t_2^m \geq t_1^m$. □

Lemma 3 *Given the episode* $\alpha = G'_1 \rightarrow G'_2 \rightarrow \cdots \rightarrow G'_k$ *and the occurrences* $A = ([a_1^i, a_2^i]_{i=1}^k)$, $B = ([b_1^i, b_2^i]_{i=1}^k)$ *and* $C = ([c_1^i, c_2^i]_{i=1}^k)$, *where* $A, B, C \in LO(\alpha)$ *and* A *and* C *are* LO*s immediately before and after* B *respectively, if* $a_1^1 \neq b_1^1$ *and* $[b_1^k, b_2^k]$ *is not covered by* $[a_1^k, a_2^k]$ *and* $[c_1^k, c_2^k]$, *then all of the* LO*s before* A *start before* B *and* $[b_1^k, b_2^k]$ *is covered by none of the* LO*s before* A *and after* C.

Proof According to Lemmas 6, 7 and 8 , the starting intervals of all the LOs before A are equal to $[a_1^1, a_2^1]$ or before $[a_1^1, a_2^1]$. If $a_1^1 \neq b_1^1$, it means that $a_1^1 < b_1^1$. It is clear that the starting intervals of all the LOs before A don't coincide with b_1^1. If $VI([b_1^k, b_2^k], k+1)$ is not covered by $VI([a_1^k, a_2^k], k+1)$ and $VI([c_1^k, c_2^k], k+1)$, the starting intervals of all the LOs before A are before $[a_1^k, a_2^k]$ and the starting intervals of all the LOs after C are after $[c_1^k, c_2^k]$. So VIs of these intervals don't cover $VI([b_1^k, b_2^k], k+1)$. □

Lemma 4 *If* $\epsilon \geq \frac{\delta}{4}$ *and* $\Delta \in [\delta, 2\delta)$, *then there is no redundant* LO.

Proof According to the second condition of Definition 18 and Lemma 3, $[t_2^k+\delta, t_2^k+\Delta]$ of O (the redundant LO) should be covered by the LOs immediately before and after O. Assume O_1 and O_2 are the LOs immediately before and after O respectively. We define $[b_1, b'_1]$, $[b, b']$ and $[b_2, b'_2]$ as the starting intervals of the last group of the episode in O_1, O and O_2 respectively. It is clear that $b'_1 < b' < b'_2$. If $b'_2+\delta \leq b'_1+\Delta$, then we have $b'_1+\delta < b'+\delta < b'_2+\delta \leq b'_1+\Delta < b'+\Delta < b'_2+\Delta$. It means that $[b'+\delta, b'+\Delta]$ is covered completely. Since $b'_2+\delta \leq b'_1+\Delta$, we have $b'_2-b'_1 \leq \Delta-\delta$. According to the upper bound of Δ $(\delta \leq \Delta < 2\delta)$, $b'_2-b'_1 < \delta$. On the other hand, we have $b > b_1+2\epsilon$ and $b_2 > b+2\epsilon$ according to Lemma 7. So we have $b'_2-b'_1 > 4\epsilon$. It means that $4\epsilon < b'_2-b'_1 < \delta$. Therefore, if $\epsilon \geq \frac{\delta}{4}$, the second condition of Definition 18 is not satisfied and there is no redundant LO. □

Theorem 1 *Given the episode* α *and* $(r, s) \in RS$*, the algorithm* $SMakeLoList$ *only finds all the non-redundant* LOs *of* $\beta = \alpha \oplus (r, s)$.

Proof To prove this theorem, we focus on the span of LOs in $LOList$ of episodes. The proof includes three parts: (1) Occurrences extracted by the algorithm are an LO. Given $\alpha = G'_1 \rightarrow G'_2 \rightarrow \cdots \rightarrow G'_{k-1}$ and $G' = (r, s) \in RS$, we have $\beta = G'_1 \rightarrow G'_2 \rightarrow \cdots G'_{k-1} \rightarrow G'$. The proof is by contradiction: there is at least one extracted occurrence of β that is not the latest occurrence. For this occurrence, assume there are the corresponding occurrences O_α and O_G of α and G with spans $[r, r']$ and $[x, x']$ respectively. There are two cases: (a) The gap constraints have not been satisfied, which is impossible due to line 11 of the algorithm. (b) There is the other LO Q_α of the episode α with the span $[u, u']$ for the episode α that satisfies the gap constraints for $[x, x']$ and $u > r$, $u' > r'$. Otherwise, $[r, r']$ and $[x, x']$ could form an LO for β. So, we have:

$$\left.\begin{aligned} r' + \delta \le x \le r' + \Delta \\ u \ge r, u' > r' \\ u' + \delta \le x \le u' + \Delta \end{aligned}\right\} \rightarrow r' + \delta < u' + \delta \le x \le r' + \Delta < u' + \Delta \qquad \text{(A.3)}$$

Since $[r, r']$ is before $[u, u']$, $\forall [f, f'] \in LOList(\alpha)$ that $r \le f \le u$, we have:

$$r' < f' < u' \text{ and } r' + \delta < f' + \delta < u' + f \le x \le r' + \Delta < f' + \Delta < u' + \Delta \qquad \text{(A.4)}$$

Since line 7 of the algorithm is satisfied for O_α, so $[r, r']$ could not be the latest prefix occurrence. 2) The extracted LOs are non-redundant. According to Definition 18, a redundant LO satisfies three conditions. In lines 15 to 27, these conditions are checked. According to Lemma 3, it is sufficient that the immediately next and previous LOs are investigated. The first condition of Definition 18 is considered in line 19. The second and third conditions are also investigated in lines 20 and 21 respectively. So if all the conditions are satisfied, $LOList(\beta)[z - 1]$ is redundant and is removed in line 22. 3) It should be proved that "all" of the non-redundant LOs are found. The proof is by contradiction: suppose there is at least a non-redundant LO O_β of β, composed of O_α and O_G with spans $[r, r']$ and $[x, x']$ respectively, which is not extracted. Since this LO is non-redundant, then the conditions of lines 19 to 22 are not satisfied. Since $LOListRS(G)$ is complete, there are two cases for O_α: a) $[r, r'] \in LOList(\alpha)$: it is checked in the first while loop. If $[x, x']$ is not checked for $[r, r']$, it means that the other latest prefix occurrence has been found for it previously. So $[r, r']$ could not be the latest prefix occurrence. When $[x, x']$ is checked for $[r, r']$, if $[u, u']$ is after $[r, r']$ in $LOList(\alpha)$, then $u > r$. So we have $u' > r'$ according to Lemma 8. Since $[r, x']$ is not redundant, it means that $VI([x, x'], k + 1)$ is not covered or $[x, x']$ extends concurrently or there is not an LO of β with the span $[r, y']$ such that $y' < x'$. So if $[r, x']$ is not redundant, then $[u, x']$ is not redundant. Thus, the non-redundant LO with the span $[u, x']$ is formed. Therefore, $[r, r']$ could not be an LPO and $[r, x']$ is not an LO, which is in contradiction to the assumption. (b) If $[r, r'] \notin LOList(\alpha)$, so there is the latest occurrence with the span $[u, r']$ such that $u > r$. Since r' satisfies the gap constraints with x, the latest occurrence with the span $[u, r']$ also satisfies the gap constraints with $[x, x']$. So there is another valid occurrence with the span $[u, x']$ that $\exists j, 1 \le j \le k-2$ that the starting interval of G'_j in the span $[u, r']$ is greater than $[r, r']$. So the occurrence O_β is not an LO. Therefore if $[r, r'] \in LOList(\alpha)$, all the non-redundant LOs whose LPO is $[r, r']$, are extracted. □

Lemma 9 *Given the episode* α, $(r, s) \in RS$, $|ResourceType| = r$, $|LOList(\alpha)| = q$ *and* $|LOListRS(r, s)| = p$, *time complexity of the algorithm SMakeLoList is* $O(p(r + \frac{\delta}{\epsilon}) + q)$ *in the worst case and* $O(p)$ *and* $O(q)$ *in the best cases.*

Proof Generally, time complexity of the algorithm $SMakeLOList$ is $O(k\% \times p \times r + q - f)$ where $0 \le k \le 100$ and $0 \le f \le q$. It means that when $k\%$ of $LOListRS(r, s)$ have been traversed by the f elements of $LOList(\alpha)$, a member of $LOListRS(r, s)$ is met that should be compared with the remaining $q - f$ elements of $LOList(\alpha)$. In the worst case, the first element of $LOList(\alpha)$ connects to all the $p - 1$ elements of $LOListRS(r, s)$ and the last element of $LOListRS(r, s)$ connects to no element of $LOListRS(r, s)$. Furthermore, for all the extracted occurrences, the functions $CExtending$ and $FindIndex$ are called. Time complexity of $CExtending$ is $O(r)$. For the redundant LOs we have $b_3 + \delta \le b_1 + \Delta$. In addition, in [11], we proved that $\Delta - \delta < \delta$. So we have $b_1 < b_2 < b_3 < b_1 + \delta$. On the other hand, in [11], we proved that if $[x_1, x_1]$ and $[y_1, y_1]$ are two consecutive starting intervals of (r, s), then $y_1 - x_1 > \epsilon$. So time complexity of $FindIndex$ is $O(\frac{\delta}{\epsilon})$. Therefore, time complexity is $O(p(r + \frac{\delta}{\epsilon}) + q)$. In the best case, the first element of $LOList(\alpha)$ connects to all the members of $LOListRS(r, s)$ or the first element of $LOListRS(r, s)$ connects to no element of $LOList(\alpha)$. In these cases, the functions $CExtending$ and $FindIndex$ are not also called. Therefore, time complexity is $O(p)$ and $O(q)$ respectively. □

Theorem 2 *Given the episode* $\alpha = G'_1 \to G'_2 \to \cdots \to G'_k$ *and* $G = (r, s) \in RS$, *the algorithm CMakeLOList only finds all the non-redundant* LO*s of* $\beta = \alpha \odot G$.

Proof The proof of the theorem includes three parts: (1) The occurrences extracted by the algorithm are an LO. According to the definition of the concurrent extension, we have $\beta = G'_1 \to G'_2 \to \cdots \to (G'_k \cup G = G')$. Since $LOList(\beta)$ is constructed based on $LOList(\alpha)$, so all the occurrences extracted by the algorithm satisfy the definition of LO. (2) The extracted LOs are non-redundant. According to Definition 18, a redundant LO satisfies three conditions. In lines 14 to 26, these conditions are checked. According to Lemma 3, it is sufficient that the immediately next and previous LOs are investigated. The first condition of Definition 18 is considered in line 18. The second and third conditions are also investigated in lines 19 and 21 respectively. So if all the conditions are satisfied, $LOList(\beta)[z-1]$ is redundant and is removed in line 22. (3) It should be proved that "all" of the non-redundant LOs are found. The proof is by contradiction: there is at least a non-redundant LO A of β that is not extracted. According to the previous parts, all the extracted LOs are non-redundant. Then it means that the algorithm does not recognize the occurrence A as an LO. Since each LO of β includes one LO of α and one LO of G, then it means that $LOList(\alpha)$ or $LOListRS(G)$ is not complete or the algorithm could not find the occurrence A of β. Since $LOList(\alpha)$ and $LOListRS(G)$ are complete and line 9 of the algorithm checks the concurrent extensions of α with G, A and all the possible LOs of β are extracted. Therefore, A is extracted by the algorithm, which is in contradiction to the assumption. □

Lemma 10 *Given the episode α, $G = (r, s) \in RS$, $|LOList(\alpha)| = q$ and $|LOListRS(r, s)| = p$ and $|ResourceType| = r$, time complexity of the algorithm $CMakeLOList$ is $O(p(r + \frac{\delta}{\epsilon}) + q)$ or $O(p + q(r + \frac{\delta}{\epsilon}))$ in the worst cases and $O(\min(p, q))$ in the best case.*

Proof In the best case, the corresponding element of $LOList(\alpha)[i]$ matches the corresponding element of $LOListRS(G)[j]$ and both the counters i and j increase repeatedly. Furthermore, all the LOs are non-redundant and the functions $CExtending$ and $FindIndex$ are called for none of the identified LOs. So, time complexity is $O(\min(p, q))$. In the worst case, each element of $LOList(\alpha)$ is checked with the t elements of $LOListRS(G)$ where $1 \leq t \leq p$, then an LO is identified for each element of $LOList(\alpha)$ for which the functions $CExtending$ and $FindIndex$ are called in a similar way to Lemma 9. So time complexity is $O(q(r + \frac{\delta}{\epsilon}) + p)$. In the other case, each element of $LOListRS(G)$ is checked with the w elements of $LOList(\alpha)$ where $1 \leq w \leq q$, then an LO is identified for each element of $LOListRS(G)$ for which the functions $CExtending$ and $FindIndex$ are called. So time complexity is $O(p(r + \frac{\delta}{\epsilon}) + q)$. □

Lemma 11 *Given $|ResourceType| = r$, time complexity of the algorithm $CreateBranch$ (Algorithm 9) for the episode α is $O(r|CNG_\alpha|)$.*

Proof Algorithm 9 has a for loop that processes each CNG of α in each repeat. Since episodes are represented in the form of $SAVE$ [11], we have $O(|RArray_\alpha[j]|) = r$ where $1 \leq j \leq |CNG_\alpha|$. Thus, time complexity of reversing each CNG is $O(r)$. So, time complexity of the algorithm is $O(r|CNG_\alpha|)$. □

Lemma 12 *Given the episodes α and β and the threshold $\theta \in \mathbb{R}_{\geq 0}$, if $\beta \sqsubseteq \alpha$ and $freq(\alpha) \geq \theta$, then $\theta \leq freq(\alpha) \leq freq(\beta)$ (the anti-monotonic constraint).*

Proof Since $\beta \sqsubseteq \alpha$, each occurrence of the episode α includes an occurrence of β. So $\forall O_i \in OSet_M^N(\alpha), \exists O_i' \subseteq O_i$ *that* $O_i' \in OSet_M^N(\beta)$ (see Lemma 2). Therefore, $|OSet_M^N(\alpha)| \leq |OSet_M^N(\beta)|$ *or* $freq(\alpha) \leq freq(\beta)$. □

Lemma 13 *The function $InsertInCPBT$ is only called for FC episodes.*

Proof According to lines 31 to 33 of Algorithm 8, when $Flag$ is true, the function $InsertInCPBT$ is called for the episode α. At first, $Flag$ is True. When there exists a serial extension or a concurrent extension of α whose frequency is equal to α's, $Flag$ is set to False. So according to Definition 20, α is not FC. Therefore, if there are no such episodes, according to Lemma 12, the frequency of episodes extended from α is less than α's. Therefore, α is FC. It means that the value of $Flag$ remains True and the function $InsertInCPBT$ is called for it. □

Lemma 14 *Given the episode α, if the function $FindClosedFreqEpisode$ is not called for α, then α is not a closed frequent episode.*

Proof The function $FindClosedFreqEpisode$ is called by the function $AllClosedFreqEpisodes$ and itself. The function $AllClosedFreqEpisodes$ calls it for $P = \{(r, s)|\ |LOListRS(r, s)| > 0, \forall(r, s) \in RS\}$. The function $AllClosedFreqEpisodes$ does not call the function $FindClosedFreqEpisode$ for the members of RS that are not frequent. If an episode is not frequent, then it could not also be closed frequent. The function $FindClosedFreqEpisode$ is recursively called for the serial and concurrent extensions whose frequency is larger than the threshold value c (see lines 13 and 27 of Algorithm 8). So if this function is not called for an episode α, it means that $freq(\alpha)$ is less than c and it is not a frequent episode. So it could not also be a closed frequent episode. □

Theorem 3 *The algorithm $AllClosedFreqEpisodes$ only finds all the closed episodes.*

Proof The proof includes two parts: (1) all the extracted episodes are closed. The proof is by contradiction: assume there is an episode α such that $|CNG_\alpha| = k$ and it is not closed. It means that at least one of the scenarios below occurs:

- There is at least one episode β_1 such that $\alpha = Prefix(\beta_1, k)$ and $freq(\alpha) = freq(\beta_1)$: Since $\alpha < \beta_1$, then α is processed sooner. While processing α, all the serial and concurrent extensions of α are generated. If the frequency of one of them is equal to α's, $Flag$ is set to False in Algorithm 8. So α is not inserted in $CPBT$. It means that such an episode cannot be found in $CPBT$.
- There is at least one episode β_2 such that $|CNG_\beta| = n, \alpha = Suffix(\beta_2, n-k+1)$ and $freq(\alpha) = freq(\beta_2)$. There are two cases: a) $\alpha < \beta_2$: In this case, α is inserted in $CPBT$ sooner. Since α is FC and $\alpha = Suffix(\beta_2, n - k + 1)$, β_2 is also FC and according to Lemma 13, it is inserted in $CPBT$. While inserting β_2, the algorithm $SearchInTree$ detects that α is absorbed by β_2. Therefore $CPBT$ is updated and α is removed from it. (b) $\beta_2 < \alpha$: Since β_2 is inserted in $CPBT$ sooner, the function $SearchInTree$ returns -1 and α is not inserted in $CPBT$. Therefore, α does not exist in $CPBT$. It means that all the episodes stored in $CPBT$ are closed.

(2) All of the closed episodes are found. The proof is by contradiction: there exists at least one closed episode α which has not been found. There are two cases: (a) α has not been inserted in $CPBT$. Since α is a closed episode, then it is FC. So when the function $FindClosedFreqEpisode$ is called for α, it is inserted in $CPBT$. Therefore if the function $InsertInCPBT$ has not been called for α, it means that $FindClosedFreqEpisode$ has not been called for it. Therefore α cannot be a closed frequent episode according to Lemma 14. So the function $InsertInCPBT$ is called for α. (2) α has been removed from $CPBT$. It means that there is another FC episode β whose suffix is α and $freq(\alpha) = freq(\beta)$. So, according to the definition of closed episodes, α is not a closed episode, which is in contradiction to the assumption. Therefore, all the closed episodes are extracted. □

B Algorithms

In this appendix, we present some functions in a canonical form and explain them in detail.

B.1 Function *CExtending*

This function considers whether the third condition of Definition 18 is satisfied for an occurrence of the episode. As Algorithm 7 shows, the function receives (r, s), which is the last member of the last CNG in the episode, the corresponding entry of $LOListRS(r, s)$ in the occurrence and the starting interval of the last CNG in the occurrence. It considers whether a concurrent event with (r, s) exists or not. The state of the concurrent event should be greater than (r, s) based on the order defined on RS. In lines 1 to 3, if the pointers $Next$ and $Previous$ are null, it means that there is no concurrent event for it. So it cannot extend concurrently. In lines 4 to 12, the list linked to the pointer $Next$ is considered to find the concurrent event. In lines 13 to 21, the list linked to the pointer $Previous$ is considered in a similar way to the pointer $Next$'s.

Algorithm 7 CExtending

Input: $r, s, ([a, a], Next, Previous), Min, Max, \epsilon$ % $([a, a], Next, Previous)$ is the corresponding entry of $LOListRS(r, s)$ in the occurrence;$[Min, Max]$ is the starting interval of the last CNG in the occurrence. Note that for the serial extension, we have $a = Min = Max$

Output: True/False % It considers whether a concurrent event with (r, s) exists or not.

1: **if** $(Next = null$ and $Previous = null)$ **then**
2:     **return False**;
3: **end if**
4: **if** $(Next \neq null)$ **then**
5:     $P \leftarrow Next$;
6:     **while** $(|P.a - Min| \leq \epsilon$ *and* $|P.a - Max| \leq \epsilon)$ **do**
7:         **if** $(P.(r, s) > (r, s))$ **then**
8:             **return True**;
9:         **end if**
10:         $P \leftarrow P.Next$;
11:     **end while**
12: **end if**
13: **if** $(Previous \neq null)$ **then**
14:     $P \leftarrow Previous$;
15:     **while** $(|P.a - Min| \leq \epsilon$ *and* $|P.a - Max| \leq \epsilon)$ **do**
16:         **if** $(P.(r, s) > (r, s))$ **then**
17:             **return True**;
18:         **end if**
19:         $P \leftarrow P.Previous$;
20:     **end while**
21: **end if**
22: **return False**;

B.2 Function *FindClosedFreqEpisode*

The algorithm $FindClosedFreqEpisode$ (Algorithm 8) receives the parameters δ, Δ, ϵ, the thresholds θ and $Level$, the episode α and $LOList(\alpha)$ and forms the concurrent and serial extensions of α (lines 6 and 20) as the episode β and computes $LOList(\beta)$ by calling the functions $CMakeLOList$ and $SMakeLOList$ (lines 7 and 21). Then the NO frequency of β is computed by calling the function $ComputeFreq$ (lines 8 and 22). If $freq(\beta)$ is above the threshold c (computed based on θ [11]), the tree is traversed further down by calling $FindClosedFreqEpisode$ in lines 13 and 27 recursively with β and $LOList(\beta)$ as its parameters. When the serial and concurrent extensions of α are constructed, it is checked (in lines 9 and 23) whether any of the super patterns β formed from α has the same frequency as α's or not; if not, it means that α is FC. So the function $InsertCPBT$ is called to insert the FC episode α in $CPBT$ (line 32).

Algorithm 8 FindClosedFreqEpisode

Input: $\epsilon, \delta, \Delta, Level, \theta, \alpha, LOList(\alpha)$; % The threshold c is computed based on θ [11].
Output: % **Identifying the FC episodes and inserting them in $CPBT$**
1: $Flag \leftarrow True$;
2: **if** $(|\alpha.RArray| \leq Level)$ **then**
3: $F_\alpha \leftarrow ComputeFreq(LOList(\alpha))$;
4: $Q \leftarrow CExt(\alpha)$; % $CExt(\alpha)$ is a set of all the valid states for the concurrent extensions of α [11]
5: **for each** $(q \in Q)$ **do**
6: $\beta \leftarrow \alpha \odot q$;
7: $L \leftarrow CMakeLOList(\epsilon, \delta, \Delta, LOList(\alpha), LOListRS(q))$;
8: $F \leftarrow ComputeFreq(L)$;
9: **if** $(F = F_\alpha)$ **then**
10: $Flag \leftarrow False$;
11: **end if**
12: **if** $(F > c)$ **then**
13: $FindClosedFreqEpisode(\epsilon, \delta, \Delta, Level, \theta, \beta, L)$;
14: **end if**
15: **end for**
16: **end if**
17: **if** $(|\alpha.RArray| < Level)$ **then**
18: $Q \leftarrow SExt(\alpha)$; % $SExt(\alpha)$ is a set of all the valid states for the serial extensions of α [11]
19: **for each** $(q \in Q)$ **do**
20: $\beta \leftarrow \alpha \oplus q$;
21: $L \leftarrow SMakeLOList(\epsilon, \delta, \Delta, LOList(\alpha), LOListRS(q))$;
22: $F \leftarrow ComputeFreq(L)$;
23: **if** $(F = F_\alpha)$ **then**
24: $Flag \leftarrow False$;
25: **end if**
26: **if** $(F > c)$ **then**
27: $FindClosedFreqEpisode(\epsilon, \delta, \Delta, Level, \theta, \beta, L)$;
28: **end if**
29: **end for**
30: **end if**
31: **if** $(Flag)$ **then**
32: $InsertInCPBT(\alpha, F_\alpha)$
33: **end if**

B.3 Function *CreateBranch*

The algorithm $CreateBranch$ receives the FC episode α and its frequency, converts them into a branch of $CPBT$ and returns a pointer to this branch. In lines 3 to 16, in the backward direction, CNGs of α are processed. In lines 4 and 5, the order of members of CNG is reversed. In lines 6 to 15, for each CNG, a $Node$ is created and added to the end of the branch. Finally, L_1, which is a pointer to the first of the branch, is returned in line 17.

Algorithm 9 CreateBranch

Input: α, f_α % α is an episode and f_α is its frequency
Output: The corresponding branch of α
1: $L_1, L_2, n : *Node$
2: $k \leftarrow |CNG_\alpha|$;
3: **for** ($j = k$ down to 1) **do**
4: $G' \leftarrow$ Reverse($RArray_\alpha[j].GList$);
5: $G' \leftarrow G' + RArray_\alpha[j].x$;
6: $n \leftarrow$ Create a new $Node$;
7: $n.label \rightarrow G'$;
8: $n.freq \leftarrow f_\alpha$;
9: **if** ($L_2 \neq null$) **then**
10: $L_2.children[1] \leftarrow n$;
11: $L_2 \leftarrow L_2.children[1]$;
12: **else**
13: $L_2 \leftarrow n$;
14: $L_1 \leftarrow n$;
15: **end if**
16: **end for**
17: **return** L_1;

B.4 Function *EpisodeAbsorbByTree*

Algorithm 10 checks whether $CPBT$ could absorb the episode α or not. Finding at least one branch that absorbs α is sufficient to omit it. In line 1, the function finds the children of the node R whose label includes $\alpha'.label$ and frequency is greater than $\alpha'.freq$. Note that α' is the pointer to the first node of the corresponding branch of the episode α. In lines 2 to 4, if there are no such children, α' cannot be absorbed by $CPBT$ and the function returns False. In lines 5 to 12, if the last node of α' is being checked, it should be considered whether there exists a member of $SubSetChildren$ that satisfies the condition of equality of the frequency (see function $CheckFreq$ in B.7). If such an episode is found, it means that α could be absorbed by $CPBT$. So α is not a closed episode and the function returns True. In line 11, if there exists no such episode, the function returns False. In lines 12 to 19, the middle nodes of the branch α' are checked whether there exists a super-episode that absorbs α. As soon as such a super-episode is found, the function returns True in line 15.

Algorithm 10 EpisodeAbsorbByTree

Input: $\alpha', R, i, |CNG_\alpha|$ % α' is the corresponding branch of the episode α, R is a node of $CPBT$
Output: True/False
1: $SubSetChildren \leftarrow \{x \in R.children | \alpha'.label \subseteq x.label\ and\ \alpha'.freq \leq x.frq\}$;
2: **if** $SubSetChildren = \varnothing$ **then**
3: **return** False;

```
end if
if (i = |CNG_α|) then
    for each (x ∈ SubSetChildren) do
        if (CheckFreq(α', x)) then
            return True;
        end if
    end for
    return False;
else
    for each (x ∈ SubSetChildren) do
        if (EpisodeAbsorbByTree(α'.Children[1], x, i + 1, |CNG_α|)) then
            return True;
        end if
    end for
    return False;
end if
```

B.5 Function *TreeAbsorbByEpisode*

Algorithm 11 finds all the branches of $CPBT$ that are absorbed by the episode α (α' is the corresponding branch of α). The path of these branches is completed in $Path$. The completed paths are added to $PathList$. Finally, $PathList$ includes all the paths whose corresponding episodes should be removed from $CPBT$. In line 1, the children of the node R whose label is a subset of $\alpha'.label$ and frequency is greater than or equal to $\alpha'.freq$ are found. All the found children are considered in lines 2 to 13. In lines 5 to 7, if an episode is found that α absorbs it, the corresponding $Path$ of the episode is added to $PathList$ and $Path$ is updated. In lines 8 to 9, the middle nodes of α' are checked to find the episodes that could be absorbed by α. In lines 10 and 11, since the last node of α' is met, $Path$ is updated.

Algorithm 11 TreeAbsorbByEpisode

Input: α', R, i, $Path$, $|CNG_\alpha|$ % α' is the corresponding branch of the episode α, R is a node of $CPBT$

Output: **% The function finds the corresponding $Path$s of the tree that are absorbed by α' and inserts $Path$s in $PathList$**

```
1: SubSetChildren ← {x ∈ R.children | x.label ⊆ α'.label and α'.freq ≤ x.frq};
2: for each (x ∈ SubSetChildren ) do
3:     Path.push(x);
4:     CNF ← ComputeNodeFreq(x);
5:     if (CNF = α'.freq) then
6:         PathList.add(Path);
7:         Path.pop();
8:     else if (i < |CNG_α|) then
9:         TreeAbsorbByEpisode(α'.children[1], x, i + 1, Path, |CNG_α|);
10:    else if (i = |CNG_α|) then
11:        Path.pop();
12:    end if
13: end for
14: if (Path is not empty) then
15:     Path.pop();
16: end if
```

B.6 Function *UpdateBranch*

After the non-closed episodes of the tree are recognized by Algorithm 11, the corresponding branches of them should be updated. The function $UpdateBranch$ (Algorithm 12) updates these branches based on the frequency of the episodes. The function receives $Path$ and $freq$ of a non-closed episode and the node R that the search starts from it towards down. In lines 2 to 3, the function starts the search of $Path$ from the node R and decreases the frequency of the node corresponding to $Path[1]$ by $freq$. If the frequency of the node is 0, it means that the frequency of its corresponding episode and all of its super-episodes is 0. So in lines 4 and 5, that node and its subtree are removed. Otherwise, this procedure is repeated for the remaining entries of $Path$.

Algorithm 12 UpdateBranch

Input: R, $freq$, $Path$ % R is a node of $CPBT$, $Path$ is a path of $CPBT$ that should be updated.
Output: % **The function updates $CPBT$ based on $Path$ and $freq$**
1: **if** ($Path$ is not empty) **then**
2: $n \leftarrow \{x \in R.children | x.label = Path[1]\}$; % n only includes one node
3: $n.freq = n.freq - freq$;
4: **if** ($n.freq = 0$) **then**
5: remove n from $R.children$;
6: **end if**
7: **else**
8: delete $Path[1]$;
9: $UpdateBranch(n, freq, path)$;
10: **end if**

B.7 Functions *CheckFreq* and *ComputeNodeFreq*

The function $CheckFreq$ (Algorithm 13) is proposed to consider whether there is an episode in the sub-tree of the node n of $CPBT$ whose frequency is equal to the frequency of the episode α. This function receives α' (the corresponding branch of α) and the node n of $CPBT$. It returns True if such an episode exists in $CPBT$. In lines 1 and 2, it is checked whether $freq(Episode(n))$ is equal to $freq(\alpha)$ or not. If not, the children of n are traversed by calling $CheckFreq$ recursively in lines 4 to 8. As soon as a child with the frequency $freq$ is found, the search is stopped and True is returned. The function $ComputeNodeFreq$ (Algorithm 14) computes the frequency of $Episode(n)$. For this purpose, in lines 2 to 4, the frequency of the node n decreases by the sum of the frequency of its children. It is clear that the function $ComputeNodeFreq(n)$ computes $freq(Episode(n))$. If $ComputeNodeFreq(n) > 0$, then $Episode(n)$ has occurred in the stream.

B.8 Function *ExtractClosedEpisodeFromCPBT*

Algorithm 15 shows the function $ExtractClosedEpisodeFromCPBT$. The main loop of Algorithm 15 traverses $CPBT$ until there is no node except the root of $CPBT$. In line 3, the traverse starts from the most left child. In lines 6 to 11, the most left branch of $CPBT$ is found. The corresponding episode of this branch is stored in the episode α. Since episodes have been inserted in the backward direction in $CPBT$, α

Algorithm 13 CheckFreq

Input: α',n % α' is the corresponding branch of the episode α and n is a node of $CPBT$
Output: True/False
if $(\alpha'.freq = ComputeNodeFreq(n))$ **then**
    return True;
end if
for each $(n' \in n.children)$ **do**
    if $(CheckFreq(\alpha', n'))$ **then**
        return True;
    end if
end for
return False;

Algorithm 14 ComputeNodeFreq

Input: n % n is a node of $CPBT$
Output: The frequency of an episode that starts from n and ends in the root
$sum \leftarrow n.freq$;
for each $(x \in n.children)$ **do**
    $sum \leftarrow sum - x.freq$;
end for
return sum;

is added to $ClosedSet$ in reverse order in line 12. Furthermore, the branch should be updated. In lines 13 to 25, the frequency of all the nodes of the branch decreases by α's. In lines 15 to 18, if the frequency of a node is 0, then that node and its subtree are removed. Finally, in line 27, the algorithm returns $ClosedSet$, which includes all the closed frequent episodes.

Algorithm 15 ExtractClosedEpisodeFromCPBT

Output: $ClosedSet$ % A set of all the closed frequent episodes stored in $CPBT$;
1: define α as an empty episode in the form of $SAVE$
2: **while** $(|CPBT.children| > 0)$ **do**
3:     $R \leftarrow CPBT.children[1]$;
4:     $U \leftarrow R$;
5:     $PatternU \leftarrow CPBT$;
6:     **while** $(|R.children| > 0)$ **do**
7:         $G \leftarrow$ the reverse of $R.children[1]$;
8:         add a new entry to $RArray$ of α;
9:         $RArray_{\alpha}.Last().x \leftarrow G[1]$;
10:         $RArray_{\alpha}.Last().GList \leftarrow G[2..|G|]$;
11:     **end while**
12:     add $RArray_{\alpha}$ in reverse order to $ClosedSet$;
13:     **while** (1) **do**
14:         $U.freq \leftarrow U.freq - R.freq$;
15:         **if** $(U.freq = 0)$ **then**
16:             remove node U from $ParentU$;
17:             break;
18:         **end if**
19:         **if** $(|U.children| > 0)$ **then**
20:             $parentU \leftarrow U$;
21:             $U \leftarrow U.children[1]$;
22:         **else**
23:             break;
24:         **end if**

end while
end while
return *ClosedSet*

References

1. Petcu D, Vzquez-Poletti JL (2012) European research activities in cloud computing. Cambridge Scholars Publishing, Cambridge
2. Amiri M, Mohammad-Khanli L, Mirandola R (2018) An online learning model based on episode mining for workload prediction in cloud. Future Gener Comput Syst 87:83
3. Amiri M, Mohammad-Khanli L (2017) Survey on prediction models of applications for resources provisioning in cloud. J Netw Comput Appl 82:93–113
4. Jiang Y, Perng C-S, Li T, Chang RN (2013) Cloud analytics for capacity planning and instant VM provisioning. IEEE Trans Netw Serv Manag 10(3):312–325
5. Cetinski K, Juric MB (2015) AME-WPC: advanced model for efficient workload prediction in the cloud. J Netw Comput Appl 55:191–201
6. Amiri M, Feizi-Derakhshi MR, Mohammad-Khanli L (2017) IDS fitted Q improvement using fuzzy approach for resource provisioning in cloud. J Intell Fuzzy Syst 32(1):229–240
7. Altevogt P, Denzel W, Kiss T (2016) Cloud modeling and simulation. Wiley-IEEE Press, London
8. Yang J, Liu C, Shang Y, Cheng B, Mao Z, Liu C, Niu L, Chen J (2014) A cost-aware auto-scaling approach using the workload prediction in service clouds. Inf Syst Front 16(1):7–18
9. Shi P, Wang H, Yin G, Fengshun L, Wang T (2012) Prediction-based federated management of multi-scale resources in cloud. Adv Inf Sci Serv Sci 4(6):324–334
10. Matsunaga A, Fortes JAB (2010) On the use of machine learning to predict the time and resources consumed by applications. In: Proceedings of the 2010 10th IEEE/ACM international conference on cluster, cloud and grid computing, Melbourne, Victoria, Australia, pp 495–504. IEEE Computer Society
11. Amiri M, Mohammad-Khanli L, Mirandola R (2018) A sequential pattern mining model for application workload prediction in cloud environment. J Netw Comput Appl 105:21–62
12. Achar A, Ibrahim A, Sastry PS (2013) Pattern-growth based frequent serial episode discovery. Data Knowl Eng 87:91–108
13. Yan X, Han J, Afshar R (2003) CloSpan: mining—closed sequential patterns in large datasets. In: Proceedings of the 2003 SIAM international conference on data mining, San Francisco, CA, USA, pp 166–177
14. Fahed L, Brun A, Boyer A (2014) Episode rules mining algorithm for distant event prediction. Technical Report hal-01062542, HAL
15. Huang P, Liu CJ, Yang X, Xiao L, Chen J (2014) Wireless spectrum occupancy prediction based on partial periodic pattern mining. IEEE Trans Parallel Distrib Syst 25(7):1925–1934
16. Li K, Fu Y (2014) Prediction of human activity by discovering temporal sequence patterns. IEEE Trans Pattern Anal Mach Intell 36(8):1644–1657
17. Wright AP, Wright AT, McCoy AB, Sittig DF (2015) The use of sequential pattern mining to predict next prescribed medications. J Biomed Inf 53:73–80
18. Gan W, Lin JCW, Fournier-Viger P, Chao HC, Yu PS (2018) A survey of parallel sequential pattern mining. CoRR, arXiv:1805.10515
19. Dinh D-T, Le B, Fournier-Viger P, Huynh V-N (2018) An efficient algorithm for mining periodic high-utility sequential patterns. Appl Intell 48(12):4694–4714
20. Martin F, Méger N, Galichet S, Becourt N (2012) Forecasting failures in a data stream context application to vacuum pumping system prognosis. Trans Mach Learn Data Min 5(2):87–116
21. D'Andreagiovanni M, Baiardi F, Lipilini J, Ruggieri S, Tonelli F (2019) Sequential pattern mining for ict risk assessment and management. J Log Algebraic Methods Program 102:1–16
22. Van T, Yoshitaka A, Le B (2018) Mining web access patterns with super-pattern constraint. Appl Intell 48(11):3902–3914
23. Mannila H, Toivonen H, Verkamo AI (1997) Discovery of frequent episodes in event sequences. Data Min Knowl Discov 1(3):259–289
24. Rathore S, Goyal V (2015) Top-K high utility episode mining in complex event sequence. PhD thesis

25. Höppner F (2001) Discovery of temporal patterns. Learning rules about the qualitative behaviour of time series. In: Proceedings of the 5th European conference on principles of data mining and knowledge discovery, PKDD '01. Springer, London, pp 192–203
26. Papapetrou P, Kollios G, Sclaroff S, Gunopulos D (Nov 2005) Discovering frequent arrangements of temporal intervals. In: Fifth IEEE international conference on data mining (ICDM'05), Houston, TX, USA. IEEE
27. Batal I, Cooper GF, Fradkin D, Harrison J Jr, Moerchen F, Hauskrecht M (2016) An efficient pattern mining approach for event detection in multivariate temporal data. Knowl Inf Syst 46(1):115–150
28. Winarko E, Roddick JF (2007) ARMADA: an algorithm for discovering richer relative temporal association rules from interval-based data. Data Knowl Eng 63(1):76–90 **(Data Warehouse and Knowledge Discovery, DAWAK'05)**
29. Papadopoulos S, Drosou A, Tzovaras D (2016) Fast frequent episode mining based on finite-state machines. In: Abdelrahman OH, Gelenbe E, Gorbil G, Lent R (eds) Information sciences and systems 2015. Springer International Publishing, Cham, pp 199–208
30. Lin M-Y, Lee S-Y (2002) Fast discovery of sequential patterns by memory indexing. Springer, Berlin, pp 150–160
31. Moskovitch R, Shahar Y (2009) Medical temporal-knowledge discovery via temporal abstraction. AMIA Annu Symp Proc 2009:452–456
32. Moskovitch R, Walsh C, Wang F, Hripcsak G, Tatonetti N (Nov 2015) Outcomes prediction via time intervals related patterns. In: 2015 IEEE international conference on data mining, pp 919–924
33. Sacchi L, Larizza C, Combi C, Bellazzi R (2007) Data mining with temporal abstractions: learning rules from time series. Data Min Knowl Discov 15(2):217–247
34. Allen JF (1984) Towards a general theory of action and time. Artif Intell 23(2):123–154
35. Patel D, Hsu W, Lee ML (2008) Mining relationships among interval-based events for classification. In: Proceedings of the 2008 ACM SIGMOD international conference on management of data, SIGMOD '08. ACM, New York, NY, USA, pp 393–404
36. Batal I, Fradkin D, Harrison J, Moerchen F, Hauskrecht M (2012) Mining recent temporal patterns for event detection in multivariate time series data. In: Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining, KDD '12. ACM, Beijing, China, pp 280–288
37. Ghosh S, Li J, Cao L, Ramamohanarao K (2017) Septic shock prediction for ICU patients via coupled HMM walking on sequential contrast patterns. J Biomed Inf 66:19–31
38. Laxman S, Sastry P, Unnikrishnan K (2007) Discovering frequent generalized episodes when events persist for different durations. IEEE Trans Knowl Data Eng 19(9):1188–1201
39. Tatti N, Cule B (2010) Mining closed strict episodes. In: Proceedings of the 2010 IEEE international conference on data mining, ICDM '10. IEEE Computer Society, Washington, DC, USA, pp 501–510
40. Wu S-Y, Chen Y-L (2007) Mining nonambiguous temporal patterns for interval-based events. IEEE Trans Knowl Data Eng 19(6):742–758
41. Laxman S, Sastry PS, Unnikrishnan KP (2005) Discovering frequent episodes and learning hidden markov models: a formal connection. IEEE Trans Knowl Data Eng 17(11):1505–1517
42. Hwang K, Bai X, Shi M, Li Y, Chen WG, Wu Y (2016) Cloud performance modeling and benchmark evaluation of elastic scaling strategies. IEEE Trans Parallel Distrib Syst 27(1):130–143
43. Tatti N, Cule B (2012) Mining closed strict episodes. Data Min Knowl Discov 25(1):34–66
44. Zaki MJ (2001) Spade: an efficient algorithm for mining frequent sequences. Mach Learn 42(1–2):31–60
45. Neapolitan RE, Neapolitan R, Naimipour K (2010) Foundations of algorithms. Jones & Bartlett Learning, Burlington
46. Alam M, Shakil KA, Sethi S (2016) Analysis and clustering of workload in google cluster trace based on resource usage. In: 2016 IEEE international conference on computational science and engineering (CSE) and IEEE international conference on embedded and ubiquitous computing (EUC) and 15th international symposium on distributed computing and applications for business engineering (DCABES), pp 740–747. IEEE
47. Alexandru I, Hui L, Mathieu J, Shanny A, Catalin D, Lex W, Epema Dick HJ (2008) The grid workloads archive. Future Gener Comput Syst 24(7):672–686
48. Shen S, van Beek V, Iosup A (2015) Statistical characterization of business-critical workloads hosted in cloud datacenters. In: 2015 15th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid), pp 465–474. IEEE

49. Li A, Yang X, Kandula S, Zhang M (2010) Cloudcmp: comparing public cloud providers. In: Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, pp 1–14. ACM