Data Min Knowl Disc (2012) 25:67–108
DOI 10.1007/s10618-011-0233-y

Discovering injective episodes with general partial orders

Avinash Achar · Srivatsan Laxman · Raajay Viswanathan · P. S. Sastry

Received: 31 August 2010 / Accepted: 3 August 2011 / Published online: 28 August 2011

Abstract Frequent episode discovery is a popular framework for temporal pattern discovery in event streams. An episode is a partially ordered set of nodes with each node associated with an event type. Currently algorithms exist for episode discovery only when the associated partial order is total order (serial episode) or trivial (parallel episode). In this paper, we propose efficient algorithms for discovering frequent episodes with unrestricted partial orders when the associated event-types are unique. These algorithms can be easily specialized to discover only serial or parallel episodes. Also, the algorithms are flexible enough to be specialized for mining in the space of certain interesting subclasses of partial orders. We point out that frequency alone is not a sufficient measure of interestingness in the context of partial order mining. We propose a new interestingness measure for episodes with unrestricted partial orders which, when used along with frequency, results in an efficient scheme of data mining. Simulations are presented to demonstrate the effectiveness of our algorithms.

Responsible editor: Eamonn Keogh.

A. Achar (✉) · P. S. Sastry
Indian Institute of Science, Bangalore, India
e-mail: achar.avinash@gmail.com

P. S. Sastry
e-mail: sastry@ee.iisc.ernet.in

S. Laxman · R. Viswanathan
Microsoft Research, Bangalore, India
e-mail: slaxman@microsoft.com

R. Viswanathan
e-mail: raajay.v@gmail.com

Keywords Episode mining · General partial order · Non-overlapped count · Bidirectional evidence

1 Introduction

Frequent episode discovery (Mannila et al. 1997) is a popular framework for discovering temporal patterns in symbolic time series data, with applications in several domains like manufacturing (Laxman 2006; Unnikrishnan et al. 2009), telecommunication (Mannila et al. 1997; Hätönen et al. 1996), WWW (Laxman et al. 2008), biology (Bouqata et al. 2006; Patnaik et al. 2008), finance (Nag and Fu 2003), intrusion detection (Luo and Bridges 2000; Wang et al. 2008), text mining (Iwanuma et al. 2004) etc. The data in this framework is a single long time-ordered stream of events and each temporal pattern (called an episode) is essentially a small, partially ordered collection of nodes, with each node associated with a symbol (called event-type). The partial order in the episode constrains the time-order in which events should appear in the data, in order for the events to constitute an occurrence of the episode. The task is to unearth all episodes whose frequency in the data exceeds a user-defined threshold.

Currently, there are algorithms in the literature for discovering frequent episodes if we restrict our attention to only *serial* episodes (where the partial order is a total order) or only *parallel* episodes (where the partial order is trivial) (Mannila et al. 1997; Casas-Garriga 2003; Laxman et al. 2005; Laxman 2006; Patnaik et al. 2008). However, no algorithms are available for the case of episodes with unrestricted partial orders. In this paper we present a new data mining method to discover episodes with unrestricted partial orders from event streams. We begin with a brief overview of the frequent episodes framework (Mannila et al. 1997). Then, in the rest of this section we explain the contributions of this paper in the context of current episodes literature.

1.1 Episodes in event streams

In the frequent episode framework (Mannila et al. 1997), the data, referred to as an *event sequence*, is denoted by $\mathbf{D} = \langle (E_1, t_1), (E_2, t_2), \ldots (E_n, t_n) \rangle$, where n is the number of events in the data stream. In each tuple (E_i, t_i), E_i denotes the event-type and t_i the time of occurrence of the event. The E_i, take values from a finite set, $\mathcal{E}$. The sequence is ordered so that, $t_i \leq t_{i+1}$ for $i = 1, 2, \ldots$.

The event-types denote some information regarding nature of each event and they are application-specific. For example, event stream could represent a sequence of user actions in a browsing session (Laxman et al. 2008) and then event-types denote some relevant information about type of buttons clicked by the user. Another example could be an event stream of fault alarms in an assembly line in a manufacturing plant (Unnikrishnan et al. 2009) and the event-types represent some codes that characterize each such fault-reporting event. The objective is to analyze such sequential data streams to unearth interesting temporal patterns that are useful in the context of applications. In the above two applications, we may be interested in temporal patterns that may enable us to predict user behavior in a browsing session or to diagnose the root-cause for some fault alarm that is currently seen.

The temporal patterns that we may wish to represent and discover are called *episodes* which we formally define below after explaining some relevant mathematical notations. Given any set V, a *relation* R over V (which is a subset of $V \times V$) is said to be a *strict partial order* if it is *irreflexive* (i.e., for all $v \in V$, $(v, v) \notin R$), *asymmetric* (i.e., $(v_1, v_2) \in R$ implies that $(v_2, v_1) \notin R$, for all distinct $v_1, v_2 \in V$) and *transitive* (i.e., $\forall v_1, v_2, v_3 \in V$, $(v_1, v_2) \in R$ and $(v_2, v_3) \in R$ implies that $(v_1, v_3) \in R$). We note that when R is a strict partial order, there can be v_1, v_2 such that neither (v_1, v_2) nor (v_2, v_1) belong to R. If R is a strict partial order, and, in addition, it is true that given any pair of distinct elements $v_1, v_2 \in V$, either $(v_1, v_2) \in R$ or $(v_2, v_1) \in R$, then, we say R is a *total order*. In the rest of the paper we use the term partial order to denote a strict partial order. (In the standard mathematical terminology, the difference is that a partial order is a relation that is asymmetric, transitive and *reflexive*. Thus, on a set of sets, the relation 'subset of' is a partial order while 'strict subset of' is a strict partial order.) We write $v_1 R v_2$ to denote $(v_1, v_2) \in R$. We also note that if R is empty (i.e., a null set) then R is (vacuously) a partial order.

Definition 1 (Mannila et al. 1997) An N-node episode α, is a tuple, $(V_\alpha, <_\alpha, g_\alpha)$, where $V_\alpha = \{v_1, v_2, \ldots, v_N\}$ is a collection of nodes, $<_\alpha$ is a (strict) partial order on V_α and $g_\alpha : V_\alpha \to \mathcal{E}$ is a map that associates each node with an event-type from $\mathcal{E}$.

In other words, an episode is a multiset with a partial order on it. When $<_\alpha$ is a total order, α is referred to as a *serial episode* and when $<_\alpha$ is empty, α is referred to as a *parallel episode*. In general, since $<_\alpha$ is defined to be a partial order, episodes can be neither serial nor parallel. An example of a three-node episode could be $\alpha = (V_\alpha, <_\alpha, g_\alpha)$, where $v_1 <_\alpha v_2$ and $v_1 <_\alpha v_3$, and $g_\alpha(v_1) = B$, $g_\alpha(v_2) = A$, $g_\alpha(v_3) = C$. As a simple graphical notation, we represent this episode as $(B \to (A\,C))$ because it captures the essence of the temporal pattern represented by this episode, namely, B is followed by A and C in any order. Similarly, a parallel episode with event-types A and B is denoted as (AB). Figure 1 shows some examples of episodes.

Definition 2 (Mannila et al. 1997) Given a data stream, $\langle (E_1, t_1), \ldots, (E_n, t_n) \rangle$ and an episode $\alpha = (V_\alpha, <_\alpha, g_\alpha)$, an occurrence of α in the data stream is a map h: $V_\alpha \to \{1, \ldots, n\}$ such that $g_\alpha(v) = E_{h(v)} \forall v \in V_\alpha$, and $\forall v, w \in V_\alpha$, $v <_\alpha w$, implies $t_{h(v)} < t_{h(w)}$.

We can represent an occurrence of the episode by the events which constitute the occurrence. Figure 2 gives a data stream indicating some occurrences of $(B \to (A\,C))$, with each occurrence marked with the same color. In this event sequence, the set of events $\langle (B, 3), (A, 3), (C, 8) \rangle$, for instance, does not represent an occurrence as A doesn't occur strictly after B.

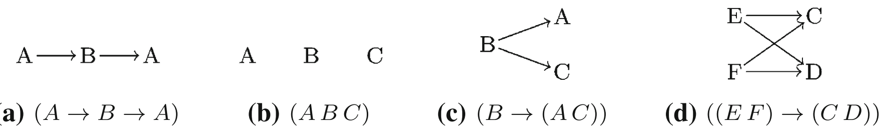


Fig. 1 Example episodes: **a** a serial episode, **b** a parallel episode, **c**, **d** general episodes

$\langle$(B,1), (A,2), $(F, 2)$, (B,3), (C,4), $(G, 5)$, (C,5), (A,5), (B,6), (C,8), (A,9)$\rangle$.

Fig. 2 Occurrences of $(B \rightarrow (A\,C))$

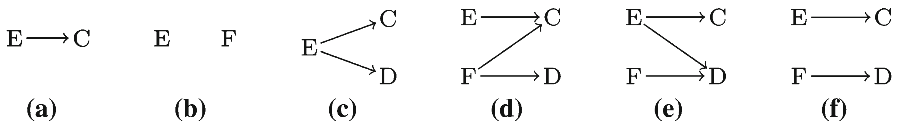


Fig. 3 Example subepisodes of $((E\ F) \rightarrow (C\ D))$ (Fig. 1d)

$\langle$(B,1), (A,2), $(F, 2), (B, 3)$, (C,4), $(G, 5), (C, 5), (A, 5)$, (B,6), (C,8), (A,9)$\rangle$.

Fig. 4 Maximal set of non-overlapped occurrences of $(B \rightarrow (A\,C))$

Definition 3 (*Mannila et al. 1997*) Episode $\beta = (V_\beta, <_\beta, g_\beta)$ is said to be a *subepisode* of $\alpha = (V_\alpha, <_\alpha, g_\alpha)$ (denoted $\beta \preceq \alpha$) if there exists a $1-1$ map $f_{\beta\alpha}\colon V_\beta \rightarrow V_\alpha$ such that (i) $g_\beta(v) = g_\alpha(f_{\beta\alpha}(v)) \forall v \in V_\beta$, and (ii) $\forall v, w \in V_\beta$ with $v <_\beta w$, we have $f_{\beta\alpha}(v) <_\alpha f_{\beta\alpha}(w)$ in V_α.

Thus, $(B \rightarrow A)$, $(B \rightarrow C)$ and (AC) are two-node subepisodes of $(B \rightarrow (AC))$ while (BAC) is a three-node subepisode of it. Figure 3 illustrates a number of subepisodes of $((E\ F) \rightarrow (C\ D))$. The importance of the notion of subepisode is that if $\beta \preceq \alpha$, then every occurrence of α contains an occurrence of β (Mannila et al. 1997). We say β is a strict subepisode of α if $\beta \preceq \alpha$ and $\alpha \neq \beta$.

Given an event sequence, the data mining task is to discover all frequent episodes, i.e., episodes whose frequencies exceed a given threshold. Frequency is some measure of how often an episode occurs. The frequency of episodes can be defined in many ways (Mannila et al. 1997; Laxman et al. 2005; Iwanuma et al. 2004). In this paper, we consider the non-overlapped frequency (Laxman et al. 2005).

Definition 4 (*Laxman et al. 2005*) Consider a data stream (event sequence), **D**, and an N-node episode, α. Two occurrences h_1 and h_2 of α are said to be non-overlapped in **D** if either $\max_i t_{h_1(v_i)} < \min_j t_{h_2(v_j)}$ or $\max_i t_{h_2(v_i)} < \min_j t_{h_1(v_j)}$. A set of occurrences is said to be non-overlapped if every pair of occurrences in the set is non-overlapped. A set H, of non-overlapped occurrences of α in $\mathbb{D}$ is *maximal* if $|H| \geq |H'|$, where H' is any other set of non-overlapped occurrences of α in $\mathbb{D}$. The *non-overlapped frequency* of α in $\mathbb{D}$ (denoted as f_{no}) is defined as the cardinality of a maximal set of non-overlapped occurrences of α in $\mathbb{D}$.

Two occurrences are non-overlapped if no event of one occurrence appears in between events of the other. The notion of a maximal set of non-overlapped occurrences is needed since there can be many sets of non-overlapped occurrences of an episode with different cardinality. Figure 4 shows a maximal set of non-overlapped occurrences of $(B \rightarrow (A\,C))$ in the data sequence of Fig. 2. Thus the frequency of $(B \rightarrow (A\,C)$ is 2. Another maximal set of non-overlapped occurrences for example would be $\{\langle (B, 3), (C, 5), (A, 5)\rangle, \langle (B, 6), (C, 8), (A, 9)\rangle\}$ in the same data stream. In the rest of the paper, whenever we refer to frequency of an episode, (unless otherwise mentioned) we mean non-overlapped frequency.

1.2 Discovering episodes with unrestricted partial orders

The temporal data mining framework of frequent episodes is introduced in Mannila et al. (1997) where an episode is defined as a collection of event-types along with a partial order over them (cf. Definition 1). However, in Mannila et al. (1997) as well as in all subsequent work in this area, the algorithms proposed for frequent episode discovery are capable of handling only serial or parallel episodes.[1] In Mannila et al. (1997), two notions of frequency of episodes, namely, windows-based frequency and minimal occurrences based frequency were proposed and algorithms to mine for serial or parallel episodes were presented under these two frequencies. Further refinements for the windows-based frequency were proposed in Casas-Garriga (2003) and Iwanuma et al. (2004). The notion of nonoverlapped occurrences based frequency was introduced in Laxman et al. (2005) where an algorithm for discovering frequent serial episodes was also presented. A very efficient algorithm for discovering serial episodes under the nonoverlapped occurrences based frequency was proposed in Laxman et al. (2007a). There has also been work on assessing the statistical significance of the discovered patterns (Laxman et al. 2005; Tatti 2009; Sastry and Unnikrishnan 2010). While all the algorithms are for mining serial/parallel episodes only, there are other generalizations of the episodes framework that have been addressed. For example, a generalization where different events have different time durations and the episode structure allows for specifying patterns that also depend on time durations of events, is presented in Laxman et al. (2007b). However, that paper also deals with only serial episodes of this generalized kind. Thus, though the framework of frequent episodes in event streams is more than 10 years old now, the problem of discovering frequent episodes with unrestricted partial orders is still largely an open problem. This is the problem that is addressed in this paper.

Frequent episodes essentially capture sets of event-types that repeat often in a specified order in the data and hence are generally useful as patterns that capture some dependencies or causative influences which are important in the process generating the data. For example, when the data is the sequence of user actions in a browsing session (Laxman et al. 2008), frequent episodes can be useful for deriving rules for predicting a future action (e.g., changing the search engine) of the user. In the application of analyzing fault alarms in a manufacturing plant (Laxman 2006; Unnikrishnan et al. 2009), a frequent episode ending in an event-type, say, A, can be useful for analyzing the root causes for the fault type A.

In this context, restricting our attention to only serial or only parallel episodes may severely restrict our ability to represent and discover all the useful dependencies in the data. For example, if we discover only serial episodes in the application of analyzing fault report logs in a manufacturing process (Laxman 2006), then we can only discover

[1] However, much after the initial submission of this manuscript there has appeared a recent paper (Tatti and Cule 2010) which also addresses the problem of mining general partial order episodes from a single long event stream. It mines based on the windows based frequency as opposed to the non-overlapped frequency we use here (unlike the windows based frequency, this is an occurrences based measure). It mines based on frequency alone as an interestingness measure as opposed to our approach here where we consider both frequency and bidirectional evidence (a measure we introduce in this paper for general partial order episodes).

dependencies in the form a single chain of events each causing the other. However, many useful dependencies can be in the form of graphs where, for example, two faults A and B, occurring in any order (but within some time) may cause a fault C. Similarly, the needed dependency structure for predicting a user action in a browsing session may be a graph. To be able to discover such dependencies also, we need methods that can mine for episodes with unrestricted partial orders.

As another example, consider the problem of analyzing multi-neuronal spike train data (Brown et al. 2004). Here, the data consists of a series of spikes (or action potentials) recorded simultaneously from a number of potentially interacting neurons in a neural tissue. Such data would be a mixture of spontaneous stochastic spiking activity of the individual neurons (or that of neurons weakly interacting with each other) as well as some coordinated spiking by groups of neurons that have strong interactions among them. Unearthing the patterns of strong interactions will help in inferring functional connectivities in the neural tissue and this is a problem of current interest in computational neuroscience (Diekman et al. 2009). Frequent episode methods based on discovery of serial or parallel episodes has been explored in this application (Patnaik et al. 2008). However, most microcircuits or functional connectivity patterns would be in the form of graphs and hence one needs methods that can discover episodes with general partial orders. In this paper, we present some simulation results to illustrate the effectiveness of our algorithms for analyzing multi-neuronal spike train data.

While developing a priori-based methods for mining unrestricted partial orders, there are three important issues to be considered. First is that of a counting algorithm that can track and count occurrences of episodes with general partial orders. In this paper we present an algorithm for counting nonoverlapped occurrences of episodes with unrestricted partial orders that can be viewed as an extension of the algorithm for serial episodes as in Laxman et al. (2005). The second issue is that of an efficient scheme for candidate generation. When considering serial/parallel episodes, given two n-node episodes, generating $(n + 1)$-node episodes is relatively simple because there is only one place in the partial order graph where a new event-type can be attached. This is not true for general partial order graphs. In this paper, we present a novel algorithm for candidate generation for which it is proved that all episodes that would be frequent would be generated and that no episode would be generated twice. (In our method, we assume that the event-types in an episode are all distinct. We call such episodes injective episodes which are precisely defined in Sect. 2.) An interesting feature of our candidate generation algorithm is that we can easily specialize the method to focus the search on many interesting subclasses of partial orders including only serial episodes or only parallel episodes. The third issue that becomes important while mining for unrestricted partial orders (irrespective of the discovery method) is that frequency alone is insufficient as an indicator of the interestingness of an episode. We argue here that we need another measure, in addition to frequency, to properly assess the support for a general partial order episode in the data stream. Towards this end, in this paper we propose a novel notion of 'interestingness' for episodes with unrestricted partial orders which we call *bidirectional evidence*. We show that this bidirectional evidence, when used along with frequency, results in an efficient method for discovering episodes with unrestricted partial orders.

The main contribution of the paper is a new method for discovering interesting episodes with unrestricted partial orders from event streams. To validate our method, we present simulation results on synthetically generated data streams. The main reason for using synthetic data is that, we would then know the ground truth. By generating data streams where we embed specific partial orders, we demonstrate the effectiveness of our method in unearthing the hidden dependencies in the form of unrestricted partial orders, present in the data. We show that the algorithm scales well with data length and number of embedded patterns in the data. We also briefly discuss one application area, namely, analyzing multi-neuronal spike train data which is described earlier.

The rest of this paper is organized as follows. In Sect. 2, we define injective episodes. The candidate generation step is described in Sect. 3. Section 4 describes our new interestingness measure. Finite state automata (FSA) for tracking occurrences of injective episodes and algorithms using these automata for counting frequencies of episodes with unrestricted partial orders are described in Sect. 5. At the end of that section, we describe how the bidirectional evidence can be computed along with frequency counting and substantiate that in most reasonable scenarios, an additional threshold based on bidirectional evidence at each level filters the uninteresting episodes while retaining the interesting ones. We present simulation results in Sect. 6. We discuss the related work in the sequential patterns literature and conclude in Sect. 7.

2 Injective episodes

In this paper, we consider a sub-class of episodes called *injective episodes*. An episode, $\alpha = (V_\alpha, <_\alpha, g_\alpha)$ is said to be injective if g_α is an injective map, that is, the labels or the associated event-types in the episode are unique. For example, $(B \rightarrow (AC))$ is an injective episode, while $(B \rightarrow (AC) \rightarrow B)$ is not. Thus, an injective episode, is simply a subset of event-types (out of the alphabet, $\mathcal{E}$) with a partial order defined over it. This subset, which we will denote by X^α, is same as the range of g_α. The partial order that $<_\alpha$ induces over X^α is denoted by R^α. From now on, unless otherwise stated, when we say episode we mean an injective episode.

Although (X^α, R^α) is a simpler notation, sometimes, e.g., when referring to episode occurrences, $(V_\alpha, <_\alpha, g_\alpha)$ notation comes in handy. However, there can be multiple $(V_\alpha, <_\alpha, g_\alpha)$ representations for the same underlying pattern under Definition 1. Consider, for example, 2 three-node episodes, $\alpha_1 = (V_1, <_{\alpha_1}, g_{\alpha_1})$ (Fig. 5a) and $\alpha_2 = (V_2, <_{\alpha_2}, g_{\alpha_2})$ (Fig. 5b), defined as: (i) $V_1 = \{v_1, v_2, v_3\}$ with $v_1 <_{\alpha_1} v_2$, $v_1 <_{\alpha_1} v_3$ and $g_{\alpha_1}(v_1) = B$, $g_{\alpha_1}(v_2) = A$, $g_{\alpha_1}(v_3) = C$, and (ii) $V_2 = \{v_1, v_2, v_3\}$ with $v_2 <_{\alpha_2} v_1$, $v_2 <_{\alpha_2} v_3$ and $g_{\alpha_2}(v_1) = A$, $g_{\alpha_2}(v_2) = B$ and $g_{\alpha_2}(v_3) = C$. Both α_1 and α_2 represent the same pattern namely $(B \rightarrow (A\,C)$, and they are indistinguishable

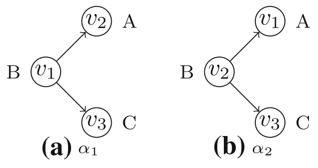
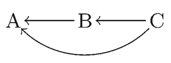
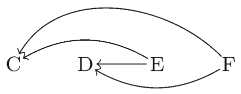
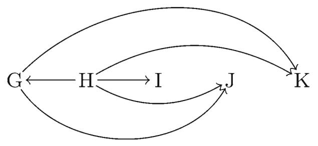


Fig. 5 Two distinct representations of the same pattern $(B \rightarrow (AC))$

Table 1 Some example episodes

$V_\alpha = \{v_1, v_2, v_3\}$. $g_\alpha(v_1) = A, g_\alpha(v_2) = B$, $g_\alpha(v_3) = C$. $<_\alpha = \{(v_2, v_1), (v_3, v_1)(v_3, v_2)\}$.	$V_\alpha = \{v_1, v_2, v_3, v_4\}$. $g_\alpha(v_1) = C, g_\alpha(v_2) = D$, $g_\alpha(v_1) = E, g_\alpha(v_2) = F$. $<_\alpha = \{(v_3, v_1), (v_3, v_2) (v_4, v_1), (v_4, v_2)\}$.	$V_\alpha = \{v_1, v_2, v_3, v_4, v_5\}$. $g_\alpha(v_1) = G, g_\alpha(v_2) = H, g_\alpha(v_3) = I, g_\alpha(v_4) = J, g_\alpha(v_5) = K$. $<_\alpha = \{(v_1, v_4), (v_1, v_5), (v_2, v_1) (v_2, v_3), (v_2, v_4), (v_2, v_5)\}$.
$(C \to B \to A)$	$((E\,F) \to (C\,D))$	$(H \to ((G \to (J\,K))\,I))$
$\boxed{A}\boxed{B}\boxed{C}$	$\boxed{C}\boxed{D}\boxed{E}\boxed{F}$	$\boxed{G}\boxed{H}\boxed{I}\boxed{J}\boxed{K}$
$\begin{pmatrix} 0&0&0 \\ 1&0&0 \\ 1&1&0 \end{pmatrix}$	$\begin{pmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 1&1&0&0 \\ 1&1&0&0 \end{pmatrix}$	$\begin{pmatrix} 0&0&0&1&1 \\ 1&0&1&1&1 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{pmatrix}$
		

First row gives the $(V_\alpha, <_\alpha, g_\alpha)$ notation. Second row gives the graphical notation. Third row gives the array of event-types, $\alpha.g$, which is the X^α set ordered as per the lexicographic ordering on $\mathcal{E}$. Fourth row shows the corresponding adjacency matrix, $\alpha.e$. This is a representation of the relation R^α. Last row shows the partially ordered set (X^α, R^α) graphically with event-types placed in order as in $\alpha.g$

based on their occurrences, no matter what the given data sequence is (there is no such ambiguity in the (X^α, R^α) representation). To obtain a unique $(V_\alpha, <_\alpha, g_\alpha)$ representation for α, we assume a lexicographic order over the alphabet, $\mathcal{E}$, and ensure that $(g_\alpha(v_1), \ldots, g_\alpha(v_N))$ is ordered as per this ordering. Note that this lexicographic order on $\mathcal{E}$ is not related in anyway to the actual partial order, $<_\alpha$. Referring to the earlier example involving α_1 and α_2, we will use α_2 to denote the pattern $(B \to (AC))$.[2]

Each ℓ-node episode in our algorithms is represented by an ℓ-element array of event-types, $\alpha.g$, and an $\ell \times \ell$ matrix, $\alpha.e$, containing the adjacency matrix of the partial order. For an episode $\alpha = (V_\alpha, <_\alpha, g_\alpha)$, $\alpha.g[i] = g_\alpha(v_i)$ and $\alpha.e[i][j] = 1$ iff $v_i <_\alpha v_j$. Since we work with representations where $(g_\alpha(v_1), \ldots, g_\alpha(v_N))$ is ordered as per this lexicographic ordering on $\mathcal{E}$, the array $\alpha.g$ would contain the elements of X^α sorted as per this lexicographic ordering. We refer to $\alpha.g[1]$ as the first node of α, $\alpha.g[\ell]$ as the last node of α and so on. Note that this notion of ith node of an episode has no relationship whatsoever with the associated partial order R^α.

If α and β are injective episodes, and if $\beta \preceq \alpha$ (cf. Definition 3), then the associated partial order sets are related as follows: $X_\beta \subseteq X_\alpha$ and $R_\beta \subseteq R_\alpha$. Some examples of injective episodes, illustrating different notations, is given in Table 1.

[2] In all our examples, the event-types are alphabets with the usual lexicographic ordering.

The data mining task is to extract all episodes whose frequency (i.e., the number of non-overlapped occurrences) exceeds a user-defined threshold. Like most current algorithms for frequent serial/parallel episode discovery (Mannila et al. 1997; Laxman et al. 2005), we use an A priori-style level-wise procedure for mining. Each level has two steps: *candidate generation* and *frequency counting*.

In our method of discovering frequent episodes with unrestricted partial orders, the reason for restricting our attention to injective episodes is mainly to ensure an unambiguous representation for all patterns of interest. Such an unambiguous representation is needed for the efficient candidate generation method proposed here. We explain this in greater detail in Remark 2 towards the end of Sect. 3.

3 Candidate generation

In this section, we describe the candidate generation algorithm for injective episodes with unrestricted partial orders. Under the frequency measure (based on non-overlapped occurrences) we know that no episode can be more frequent than any of its subepisodes. The candidate generation method uses this property to construct the set, $\mathcal{C}_{\ell+1}$, of $(\ell+1)$-node candidate episodes given $\mathcal{F}_\ell$, the set of frequent episodes of size ℓ.

3.1 Steps in candidate generation

Each $(\ell+1)$-node candidate in $\mathcal{C}_{\ell+1}$ is generated by combining two suitable ℓ-node frequent episodes (out of $\mathcal{F}_\ell$). The method of constructing $\mathcal{C}_{\ell+1}$ has three main steps:

1. Picking suitable pairs of episodes from $\mathcal{F}_\ell$.
2. Combining each such pair to generate up to three episodes of size $\ell+1$ which we call *potential* candidates.
3. Finally constructing $\mathcal{C}_{\ell+1}$ by retaining only those potential candidates for which each of their ℓ-node subepisodes are in $\mathcal{F}_\ell$.

We explain each of these steps below.

3.1.1 Pairs of episodes that can be combined

For any episode α, let $X^\alpha = \{x_1^\alpha, \ldots, x_\ell^\alpha\}$ denote the ℓ distinct event-types in α, indexed in lexicographic order. Two episodes α_1 and α_2 can be combined if the following hold:

1. $x_i^{\alpha_1} = x_i^{\alpha_2},\ i = 1, \ldots, (\ell-1), x_\ell^{\alpha_1} \neq x_\ell^{\alpha_2}$.
2. $R^{\alpha_1}|_{(X^{\alpha_1}\setminus\{x_\ell^{\alpha_1}\})} = R^{\alpha_2}|_{(X^{\alpha_2}\setminus\{x_\ell^{\alpha_2}\})}$; that is, the restriction of R^{α_1} to the first $(\ell-1)$ nodes of α_1 is identical to the restriction of R^{α_2} to the first $(\ell-1)$ nodes of α_2. This means, $(x_i, x_j) \in R^{\alpha_1}$ if and only if $(x_i, x_j) \in R^{\alpha_2}$ for $i, j = 1, \ldots, (\ell-1)$.

To ensure that the same pair of episodes are not picked up two times, we follow the convention that α_1 *and* α_2 *are such that* $x_\ell^{\alpha_1} < x_\ell^{\alpha_2}$ *under the lexicographic ordering.*

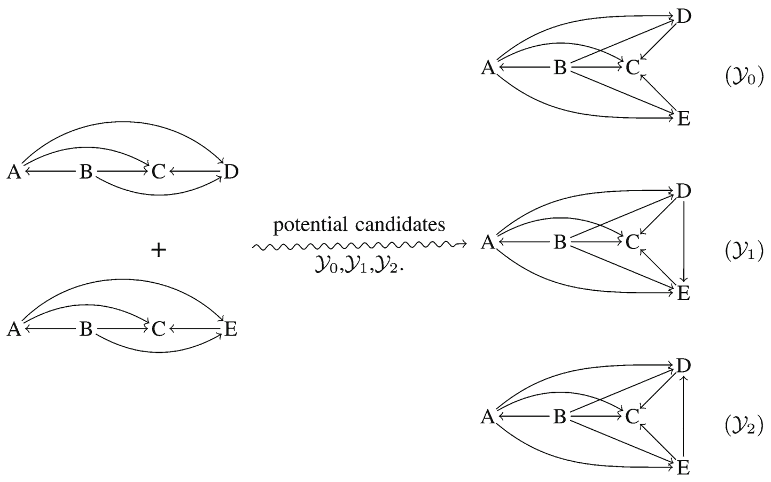


Fig. 6 An example where all combinations are potential candidates

As an example, consider the episodes $(C \to A \to B)$ and $((AB) \to D)$ which satisfy the first condition in the above para. These would not be combined since different subepisodes, namely $(A \to B)$ and (AB), are obtained on dropping their last nodes (which are C and D respectively). On the other hand, episodes $(B \to A \to D \to C)$ and $(B \to A \to E \to C)$, whose last nodes are D and E respectively, would be combined since the same subepisode, namely $(B \to A \to C)$ is obtained by dropping their last nodes. These two episodes are shown in the left part of Fig. 6 where the episodes are listed so that the event-types are in lexicographic ordering and all the edges of the partial order are shown (the rest of the figure is explained after we explain our second step).

3.1.2 Finding potential candidates

Given α_1 and α_2, satisfying the two conditions mentioned earlier, we first construct three possible episodes of size $(\ell + 1)$, $(X^{\mathcal{Y}_i}, R^{\mathcal{Y}_i})$, $i = 0, 1, 2$, as:

$$X^{\mathcal{Y}_0} = X^{\alpha_1} \cup X^{\alpha_2}, \quad R^{\mathcal{Y}_0} = R^{\alpha_1} \cup R^{\alpha_2} \tag{1}$$

$$X^{\mathcal{Y}_1} = X^{\alpha_1} \cup X^{\alpha_2}, \quad R^{\mathcal{Y}_1} = R^{\mathcal{Y}_0} \cup \left\{\left(x_\ell^{\alpha_1}, x_\ell^{\alpha_2}\right)\right\} \tag{2}$$

$$X^{\mathcal{Y}_2} = X^{\alpha_1} \cup X^{\alpha_2}, \quad R^{\mathcal{Y}_2} = R^{\mathcal{Y}_0} \cup \left\{\left(x_\ell^{\alpha_2}, x_\ell^{\alpha_1}\right)\right\} \tag{3}$$

For each of $j = 0, 1, 2$, if $R^{\mathcal{Y}_j}$ is a valid partial order over $X^{\mathcal{Y}_j}$, then the $(\ell + 1)$-node (injective) episode, $(X^{\mathcal{Y}_j}, R^{\mathcal{Y}_j})$ is considered as a *potential* candidate. To verify the same, we need to check antisymmetry and transitivity of each $R^{\mathcal{Y}_j}$. One can show that each $R^{\mathcal{Y}_j}$ always satisfies antisymmetry because α_1 and α_2 share the same $(\ell - 1)$ subepisode on dropping their last nodes. Since $(R^{\alpha_1} \cup R^{\alpha_2}) \subseteq R^{\mathcal{Y}_j}$ and since $(X^{\alpha_1}, R^{\alpha_1})$ and $(X^{\alpha_2}, R^{\alpha_2})$ are known to be transitively closed, we need to perform the

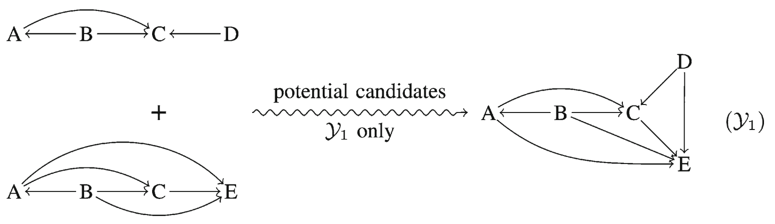


Fig. 7 An example where only $\mathcal{Y}_1$ is a potential candidate

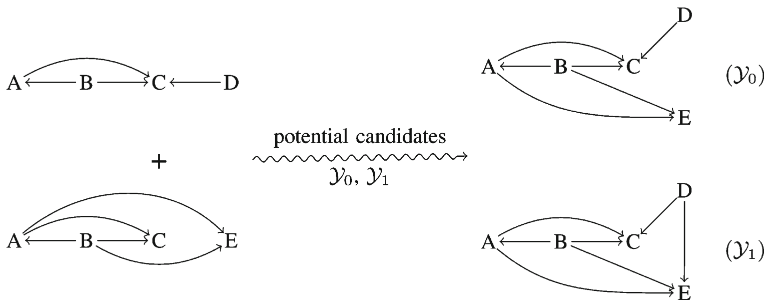


Fig. 8 An example where only $\mathcal{Y}_0$ and $\mathcal{Y}_1$ are potential candidates

transitivity check only for all size-3 subsets of $X^{\mathcal{Y}_j}$ that are of the form $\{x_\ell^{\alpha_1}, x_\ell^{\alpha_2}, x_i^{\alpha_1}\colon 1 \leq i \leq (\ell - 1)\}$. Hence, the transitivity check is $\mathcal{O}(\ell)$.

The above method of combining suitable pairs of episodes to generate potential candidates is illustrated in Fig. 6. In this example it so turns out that all the three combinations are valid partial orders and hence are potential candidates. These are shown on the right side of the figure. However, not all pairs of episodes can be combined in this manner to obtain three potential candidates. Figures 7 and 8 show examples where only one or two potential candidates are obtained.

We can intuitively think of $\mathcal{Y}_0$ as parallel episode building block and $\mathcal{Y}_1$ and $\mathcal{Y}_2$ as serial episode building blocks. For example, consider the case where we use only the $\mathcal{Y}_0$ combination at all levels. At level 1, all episodes are such that the partial order is null. Hence, if we use only the $\mathcal{Y}_0$ option then at level 2 also all partial orders generated would be empty. Thus, the resulting partial orders continue to be empty at all levels and we would be generating only parallel episodes if we use only the $\mathcal{Y}_0$ combination at all levels. Similarly, if we use only the *valid* $\mathcal{Y}_1$ and $\mathcal{Y}_2$ options at each level, then all the potential candidates would be only serial episodes. For example, the serial episodes $(A \to B)$ and $(C \to A)$ can be combined only as a $\mathcal{Y}_2$ combination yielding $(C \to A \to B)$. It is easy to see that their $\mathcal{Y}_1$ combination is not a valid partial order. However, using all three combinations at each level, allows us to generate all relevant partial orders as proved later on in this section.

3.1.3 Forming the final candidate episodes

The final step of candidate generation is to decide which of the potential candidates are actual candidates and hence can be placed in $\mathcal{C}_{\ell+1}$. For this, we need to check whether *all* the ℓ-node subepisodes of a potential $(\ell + 1)$-node candidate are frequent, that is, they are in $\mathcal{F}_\ell$. The number of such sub-episodes can in general be large. For example, consider a three-node episode $(X^\alpha = \{A, B, C\}, R^\alpha = \{(A, B), (A, C)\})$. Under our graphical notation, this is the episode $(A \to (B\,C))$. Its two-node sub-episodes are the serial episodes $(A \to B)$ and $(A \to C)$, and parallel episodes $(A\,B)$, $(A\,C)$ and $(B\,C)$. So in general, an $(\ell + 1)$-node injective episode has more than $(\ell + 1)$ number of ℓ-node subepisodes. Let us consider those ℓ-node sub-episodes of (X^α, R^α) which are obtained by restricting R^α to an ℓ-element subset of X^α. We can have $(\ell + 1)$ such subepisodes. In this example, $(A \to B)$, $(A \to C)$ and $(B\,C)$ are the 3 two-node subepisodes of α obtained by restricting R^α to all the possible 2-element subsets of X^α. Note that the remaining two-node subepisodes of $(A \to (B\,C))$, namely $(A\,B)$ and $(A\,C)$, are subepisodes of one or the other of these 3 two-node subepisodes. We call subepisodes of α obtained like this by restricting R^α to a strict subset of X^α as maximal subepisodes of α.[3] It is easy to see that if the maximal ℓ-node subepisodes of a potential $(\ell + 1)$-node candidate are frequent, then all its ℓ-node subepisodes would also be frequent. Hence, while deciding which potential candidates are actual candidates, it is enough to check if all the ℓ-node maximal subepisodes are frequent.

Remark 1 The candidate generation scheme described above is such that an ℓ-node episode is generated as a final candidate if and only if all its $(\ell - 1)$-node subepisodes are frequent. In the context of general partial order episodes, it is worth noting that an ℓ-node episode can have subepisodes of the same size. Hence, it might also be better (from an efficiency point of view) to exploit this part of the subepisode structure as well, while generating an ℓ-node candidate episode. However, there are two issues with this approach, one of which would be clear after one understands bidirectional evidence, a new measure of interestingness for general partial order episodes introduced in this paper. Hence, we discuss this issue and explain the reasons for the strategy of candidate generation that we chose, in Sect. 7.

3.2 Implementation of the candidate generation algorithm

For a given frequent episode α_1, we now describe how one can efficiently search for all other combinable frequent episodes of the same size. Let $\mathcal{F}_\ell[i]$ denote the ith episode in the collection, $\mathcal{F}_\ell$, of ℓ-node frequent episodes. At level 1 (i.e., $\ell = 1$), we ensure that $\mathcal{F}_1$ is ordered according to the lexicographic ordering on the set of event-types $\mathcal{E}$. Suppose $\mathcal{F}_1$ consists of the frequent episodes A, C and E, then we have $\mathcal{F}_1[1] = A$, $\mathcal{F}_1[2] = C$ and $\mathcal{F}_1[3] = E$. All the 3 one-node episodes share the same subepisode, namely, ϕ, on dropping their last event. As per the candidate generation algorithm, any 2 one-node episodes are combined to form a parallel episode and two serial episodes. For example A and C are combined to form (AC), $(A \to C)$ and $(C \to A)$. Accordingly here, episode A is combined with C and E to form 6 candidates in $\mathcal{C}_2$. Similarly, C is combined with E to add three more candidates to

[3] We define maximal subepisodes as follows. Let $\alpha = (X^\alpha, R^\alpha)$ be an ℓ-node episode. Let $\beta = (X^\beta, R^\beta)$ where X^β is a k-element subset of X^α and R^β is the restriction of R^α to X^β, $k < \ell$. Then, β is called a k-node *maximal* subepisode of α.

$\mathcal{C}_2$. Note that the first six candidates share the same one-node subepisode, namely, A on dropping their last event. Also, the next three candidates share the same one-node subepisode, C, on dropping their last event. The candidate generation procedure adopted at each level here, is such that the episodes which share the same subepisode on dropping their last events appear consecutively in the generated list of candidates, at each level. We refer to such a set of episodes as a *block*. In addition, we maintain the episodes in each block so that they are ordered lexicographically with respect to the array of event-types. Since, the block information aids us to efficiently decide the kind of episodes to combine, at each level right from level one, we store the block information. At level 1, all nodes belong to a single block. For a given $\alpha_1 \in \mathcal{F}_\ell$, the set of all valid episodes, α_2, (satisfying the conditions explained before) with which α_1 can be combined, are all those episodes placed below α_1 (except the ones which share the same set of event types with α_1) in the same block. All candidate episodes obtained by combining a given α_1 with all permissible episodes below it in the same block of $\mathcal{F}_\ell$, will give rise to a block of episodes in $C_{\ell+1}$, each of them having α_1 as their common ℓ-node sub-episode on dropping their last nodes. Hence, the block information of $C_{\ell+1}$ can be naturally obtained during its construction itself. Even though the episodes within each block are sorted in lexicographic order of their respective arrays of event-types, we point out that the full $\mathcal{F}_\ell$ doesn't obey the lexicographic ordering based on the arrays of event-types. For example, the episodes $((AB) \rightarrow C))$ and $(A \rightarrow (BC))$ both have the same array of event-types, but would appear in different blocks because different subepisodes are obtained by dropping their last nodes. Thus, for example, an episode like $((AB) \rightarrow D)$ appears in the same block as $((AB) \rightarrow C)$, while $(A \rightarrow (BC))$, since it belongs to a different block, may appear later in $\mathcal{F}_\ell$.

The pseudocode for the candidate generation procedure, `GenerateCandidates()`, is listed in Algorithm 1. The input to Algorithm 1 is a collection, $\mathcal{F}_\ell$, of ℓ-node frequent episodes (where, $\mathcal{F}_\ell[i]$ is used to denote the ith episode in the collection). The episodes in $\mathcal{F}_\ell$ are organized in blocks, and episodes within each block appear in lexicographic order with respect to the array of event-types. We use an array $\mathcal{F}_\ell.blockstart$ to store the block information of every episode. $\mathcal{F}_\ell.blockstart[i]$ will hold a value k such that $\mathcal{F}_\ell[k]$ is the first element of the block to which $\mathcal{F}_\ell[i]$ belongs to. The output of the algorithm is the collection, $\mathcal{C}_{\ell+1}$, of candidate episodes of size $(\ell + 1)$. Initially, $\mathcal{C}_{\ell+1}$ is empty and, if $\ell = 1$, all (one-node) episodes are assigned to the same block (lines 1–3, Algorithm 1). The main loop is over the episodes in $\mathcal{F}_\ell$ (starting on line 4, Algorithm 1). The algorithm tries to combine each episode, $\mathcal{F}_\ell[i]$, with episodes in the same block as $\mathcal{F}_\ell[i]$ that come after it (line 6, Algorithm 1). In the notation used earlier to describe the procedure, we can think of $\mathcal{F}_\ell[i]$ as α_1 and $\mathcal{F}_\ell[j]$ as α_2. If $\mathcal{F}_\ell[i]$ and $\mathcal{F}_\ell[j]$ have identical event-types, we do not combine them (line 7, Algorithm 1). The `GetPotentialCandidates()` function takes $\mathcal{F}_\ell[i]$ and $\mathcal{F}_\ell[j]$ as input and returns the set, $\mathcal{P}$, of *potential* candidates corresponding to them (called in line 8, Algorithm 1). This function first generates the three candidates by combining $\mathcal{F}_\ell[i]$ and $\mathcal{F}_\ell[j]$ as described in Eqs. 1–3. For each of the three possibilities, it then does a transitive closure check to ascertain their validity as partial orders. As explained before, one only needs to do a

Algorithm 1: Generate candidates ($\mathcal{F}_\ell$)

```
   Input: Sorted array, $\mathcal{F}_\ell$, of frequent episodes of size $\ell$
   Output: Sorted array, $\mathcal{C}_{\ell+1}$, of candidates of size $(l+1)$
1  Initialize $\mathcal{C}_{\ell+1} \leftarrow \phi$ and $k \leftarrow 0$;
2  if $\ell = 1$ then
3      for $h \leftarrow 1$ to $|\mathcal{F}_\ell|$ do $\mathcal{F}_\ell[h].blockstart \leftarrow 1$;
4  for $i \leftarrow 1$ to $|\mathcal{F}_\ell|$ do
5      $currentblockstart \leftarrow k+1$;
6      for ($j \leftarrow i+1$; $\mathcal{F}_\ell[j].blockstart = \mathcal{F}_\ell[i].blockstart$; $j \leftarrow j+1$) do
7          if $\mathcal{F}_\ell[i].g[\ell] \neq \mathcal{F}_\ell[j].g[\ell]$ then
8              $\mathcal{P} \leftarrow$ GetPotentialCandidates($\mathcal{F}_\ell[i], \mathcal{F}_\ell[j]$);
9              foreach $\alpha \in \mathcal{P}$ do
10                 $flg \leftarrow$ TRUE ;
11                 for ($r \leftarrow 1$; $r < l$ and $flg$ =TRUE; $r \leftarrow r+1$) do
12                     for $x \leftarrow 1$ to $r-1$ do
13                         Set $\beta.g[x] = \alpha.g[x]$;
14                         for $z \leftarrow 1$ to $r-1$ do $\beta.e[x][z] \leftarrow \alpha.e[x][z]$;
15                         for $z \leftarrow r$ to $\ell$ do $\beta.e[x][z] \leftarrow \alpha.e[x][z+1]$;
16                     for $x \leftarrow r$ to $\ell$ do
17                         $\beta.g[x] \leftarrow \alpha.g[x+1]$;
18                         for $z \leftarrow 1$ to $r-1$ do $\beta.e[x][z] \leftarrow \alpha.e[x+1][z]$;
19                         for $z \leftarrow r$ to $\ell$ do $\beta.e[x][z] \leftarrow \alpha.e[x+1][z+1]$;
20                     if $\beta \notin \mathcal{F}_\ell$ then $flg \leftarrow$ FALSE ;
21                 if $flg$ = TRUE then
22                     $k \leftarrow k+1$;
23                     Add $\alpha$ to $\mathcal{C}_{\ell+1}$;
24                     $\mathcal{C}_{\ell+1}[k].blockstart \leftarrow currentblockstart$;
25 return $\mathcal{C}_{\ell+1}$
```

transitivity check on size-3 subsets of the form $\{x_\ell^{\alpha_1}, x_\ell^{\alpha_2}, x_i^{\alpha_1} \colon 1 \leq i \leq (l-1)\}$ separately on the three possible combinations of $\mathcal{Y}_0$, $\mathcal{Y}_1$ and $\mathcal{Y}_2$.[4] For each potential candidate, $\alpha \in \mathcal{P}$, we construct its ℓ-node (maximal) subepisodes (denoted as β in the pseudocode) by dropping one node at a time from α (lines 12–19, Algorithm 1). Note that there is no need to check the case of dropping the last and last-but-one nodes of α, since they would result in the subepisodes $\mathcal{F}_\ell[i]$ and $\mathcal{F}_\ell[j]$, which are already known to be frequent. If all ℓ-node maximal subepisodes of α were found to be frequent, then α is added to $\mathcal{C}_{\ell+1}$, and its block information suitably updated (lines 20–24, Algorithm 1).

To sum up, the candidate generation goes as follows. For each episode $\alpha \in \mathcal{F}_\ell$, we look for episodes with which it can be combined by looking below α and within the same block as α. For every suitable pair, we do the $\mathcal{Y}_0$, $\mathcal{Y}_1$, $\mathcal{Y}_2$ combinations and by checking which are valid partial orders, decide which are potential candidates. For each potential candidate we check to see if all its ℓ-node maximal subepisodes are in $\mathcal{F}_\ell$ and thus decide whether it will go into $\mathcal{C}_{\ell+1}$.

[4] Actually we can save time in the transitivity check further. As explained in Achar et al. (2009), we need to generate only the $\mathcal{Y}_0$ combination and perform some special checks on its nodes to decide the valid partial orders to be generated in $\mathcal{P}$.

3.3 Correctness of candidate generation

In this section, we show that: (i) a given partial order episode is generated only once in the algorithm, and (ii) every frequent episode is generated by our candidate generation algorithm.

Theorem 1 *The candidate generation algorithm doesn't generate any duplicate discrete structures.*

Proof It is easy to see from Eqs. 1–3 that two partial orders generated from a given pair (α_1, α_2) of ℓ-node episodes are all different. Hence we need to consider whether the same candidate is generated from two different pairs of episodes.

Suppose exactly same candidate is generated from different pairs (α_1, α_2) and (α'_1, α'_2). For simplicity, consider the case when both these candidates come up as '$\mathcal{Y}_0$ combination'. That is, the two generated episodes, $(X^{\mathcal{Y}_0}, R^{\mathcal{Y}_0}) = (X^{\alpha_1} \cup X^{\alpha_2}, R^{\alpha_1} \cup R^{\alpha_2})$ and $(X'^{\mathcal{Y}_0}, R'^{\mathcal{Y}_0}) = (X^{\alpha'_1} \cup X^{\alpha'_2}, R^{\alpha'_1} \cup R^{\alpha'_2})$ are the same episode. Since the candidates are same, (i)$X^{\mathcal{Y}_0} = X'^{\mathcal{Y}_0}$ and (ii)$R^{\mathcal{Y}_0} = R'^{\mathcal{Y}_0}$. Recall from the conditions for forming candidates that $X^{\alpha_1} \cup X^{\alpha_2} = \{x_1^{\alpha_1}, \ldots x_\ell^{\alpha_1}, x_\ell^{\alpha_2}\}$, where the elements are indexed as per the lexicographic ordering on $\mathcal{E}$. An analogous thing holds for $X'^{\mathcal{Y}_0}$. Since $X^{\mathcal{Y}_0}$ and $X'^{\mathcal{Y}_0}$ are same, their ith elements must also match. This means $x_i^{\alpha_1} = x_i^{\alpha'_1}$ for $i = 1, \ldots \ell$ and $x_\ell^{\alpha_2} = x_\ell^{\alpha'_2}$. Thus $X^{\alpha_1} = X^{\alpha'_1}$. Also from the conditions of generating candidates we have $x_i^{\alpha_1} = x_i^{\alpha_2}$ and $x_i^{\alpha'_1} = x_i^{\alpha'_2}$ for $i = 1, \ldots (\ell - 1)$. This together with $X^{\alpha_1} = X^{\alpha'_1}$ and $x_l^{\alpha_2} = x_\ell^{\alpha'_2}$, gives $X^{\alpha_2} = X^{\alpha'_2}$. Thus $X^{\mathcal{Y}_0} = X'^{\mathcal{Y}_0} \implies X^{\alpha_1} = X^{\alpha'_1}$ and $X^{\alpha_2} = X^{\alpha'_2}$. Since the pairs (α_1, α_2) and (α'_1, α'_2) are to be distinct, we need to have either $(R^{\alpha_2} \neq R^{\alpha'_2})$ or $(R^{\alpha_1} \neq R^{\alpha'_1})$. Without loss of generality assume, $(R^{\alpha_1} \neq R^{\alpha'_1})$. Since $X^{\alpha_1} = X^{\alpha'_1}$, again without loss of generality assume there is an edge (x, y) in R^{α_1} (and hence is in $R^{\mathcal{Y}_0}$) that is absent in $R^{\alpha'_1}$. Since $R^{\mathcal{Y}_0} = R'^{\mathcal{Y}_0}$, we must have the edge (x, y) in $R^{\alpha'_2}$. By the conditions for combining episodes, the restriction of $R^{\alpha'_1}$ to the first $(\ell - 1)$ nodes of $X^{\alpha'_1}$ is identical to the restriction of $R^{\alpha'_2}$ to the first $(\ell - 1)$ nodes of $X^{\alpha'_2}$. Hence, $R'^{\mathcal{Y}_0}$ can be viewed as the disjoint union of $R^{\alpha'_1}$ and E^2, where E^2 is the set of all edges in $R^{\alpha'_2}$ involving $x_l^{\alpha'_2}$. Now, the edge (x, y) cannot belong to E^2 as neither x nor y can be $x_l^{\alpha'_2}$. (This is because $(x, y) \in R^{\alpha_1}$ and hence, $x, y \in X^{\alpha_1} = X^{\alpha'_1}$; but $x_\ell^{\alpha'_2}$ is not in $X^{\alpha'_1}$.) Therefore this edge (x, y) cannot appear in $R'^{\mathcal{Y}_0}$. Thus we must have $R^{\alpha_1} = R^{\alpha_2}$ and $R^{\alpha'_1} = R^{\alpha'_2}$. This means, the pairs (α_1, α_2) and (α'_1, α'_2) that we started with, cannot be distinct.

Using similar arguments, we can show that no $\mathcal{Y}_r$ combination can be equal to any $\mathcal{Y}_s$ combination of two distinct pairs of episodes. This completes the proof that every candidate partial order is uniquely generated. Thus we can see that our algorithm does not generate any candidate twice. □

Theorem 2 *Every frequent episode would belong to the set of candidates generated.*

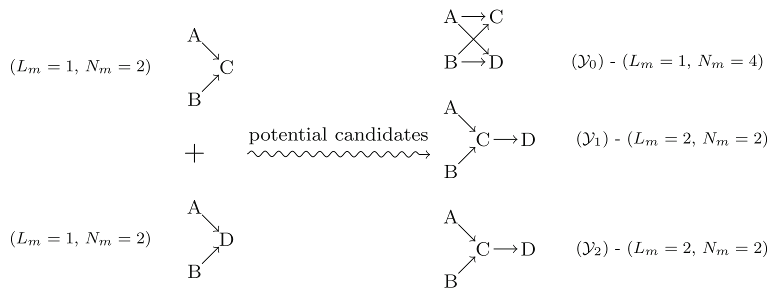
Proof We prove this by induction on the size of the episode. At level one, the set of candidates contain all the one node episodes and hence contains all the frequent one node episodes. Now suppose at level ℓ, all frequent episodes of size ℓ are indeed generated as candidates. If an $(\ell+1)$-node episode $\alpha = (X, R)$ is frequent, then all its subepisodes are frequent. The maximal ℓ-node subepisodes $(X\backslash\{x_{\ell+1}\}, R|_{X\backslash\{x_{\ell+1}\}})$ and $(X\backslash\{x_\ell\}, R|_{X\backslash\{x_\ell\}})$ in particular, are also frequent and hence generated at level ℓ (as per the induction hypothesis). The important point to note is that the $(\ell-1)$-node subepisodes obtained by dropping the last event-types of these two episodes are the same. Hence, the candidate generation method would combine these two frequent episodes. *The crucial idea behind the candidate generation algorithm is that* (X, R) *can only be either a* $\mathcal{Y}_0, \mathcal{Y}_1$ *or* $\mathcal{Y}_2$ *combination of its two maximal subepisodes obtained by dropping the last and last but-one event-types.* Since (X, R) is also a valid partial order, it would either be a $\mathcal{Y}_0, \mathcal{Y}_1$ or $\mathcal{Y}_2$ combination of these two frequent subepisodes, satisfying anti-symmetry and transitivity. Hence (X, R) would be a potential candidate. Further, since all its ℓ-node maximal subepisodes are frequent, they would all be generated at level ℓ by induction hypothesis. Hence (X, R) would be finally output in the set of candidates generated by our method. □

3.4 Candidate generation with structural constraints

The candidate generation scheme is very flexible. As explained earlier, we can easily specialize it so that we generate only parallel episodes (by using only the $\mathcal{Y}_0$ combination at all levels) or only serial episodes (by using only the potential $\mathcal{Y}_1$ and $\mathcal{Y}_2$ combinations at all levels).

We can also specialize the algorithm for some other general classes of partial order episodes. For example, consider a class of partial order episodes in which, for every episode in the class, all its maximal subepisodes also lie in the same class. We refer to such a class as satisfying a *maximal subepisode* property. Mining partial orders restricted to a class satisfying this property is a natural choice given our candidate generation method. The method generates an $(\ell+1)$-node episode as a candidate if and only if all ℓ-node maximal subepisodes are found to be frequent. Hence, to generate candidates restricted to any such class, we simply perform an additional check and retain only those candidates that belong to the class. Such a procedure is guaranteed to extract all frequent partial order episodes belonging to any such class. The classes of serial episodes and parallel episodes are the simplest classes satisfying such a property. (Note that for generating serial or parallel episodes, one need not perform this check; instead a more efficient approach can be adopted, as described earlier.)

We discuss a few interesting classes of partial orders satisfying the maximal subepisode property. The first of them is the set of all partial orders, where length (i.e., no. of edges) of any maximal path of each partial order (denoted as L_m) is bounded above by a user-defined threshold, denoted as L_{th}. Consider the episode $\alpha = (A \rightarrow (F(B \rightarrow (C\ D) \rightarrow E)))$. It has three maximal paths namely $A \rightarrow B \rightarrow C \rightarrow E, A \rightarrow B \rightarrow D \rightarrow E$ and $A \rightarrow F$ and the length of its largest maximal path (L_m) is 3. For $L_{th} = 0$, we get the set of all parallel episodes because any N-node parallel episode has N-maximal paths each of length 0, and every non-parallel episode has at least one maximal path of length 1. In general, for $L_{th} = k$, the corresponding class of partial orders contains all parallel episodes, serial episodes of size less than or equal to $(k+1)$ and many more partial orders all of whose maximal paths have length less than $(k+1)$.



$(L_m = 1,\ N_m = 2)$
$(L_m = 1,\ N_m = 2)$
potential candidates
$(\mathcal{Y}_0)$ - $(L_m = 1,\ N_m = 4)$
$(\mathcal{Y}_1)$ - $(L_m = 2,\ N_m = 2)$
$(\mathcal{Y}_2)$ - $(L_m = 2,\ N_m = 2)$

Fig. 9 Illustration of how structural constraints specified by thresholds L_{th} and N_{th} eliminate potential candidates (episodes displayed in the transitively reduced form)

It is easy to see that for any partial order belonging to such a class, all its subepisodes too belong to the same class. As k is increased, this class expands into the space of all partial orders from the parallel episode end. Another class of partial orders of interest could be one, where the number of maximal paths in each partial order (denoted as N_m) is bounded above by a threshold, denoted by N_{th}. For any partial order belonging to this class, all its maximal subepisodes are guaranteed to belong to the same class. When $N_{th} = 1$, the class obtained is exactly equal to the set of serial episodes. For example, consider a serial episode $(A \to B \to C)$. All its maximal sub-episodes are serial episodes. However, its non-maximal subepisodes like $(A\ B)$ do not belong to the set of serial episodes. We could also work on a class of partial orders characterized by the thresholds L_{th} and N_{th} simultaneously, as such a class would also satisfy the maximal subepisode property. Using such structural constraints can make the episode discovery process more focused and efficient depending on the application. Please note that the L_m and N_m values are tagged to a specific partial order, where as, L_{th} and N_{th} are the respective thresholds on these quantities used while discovering episodes.

As and when a potential candidate is generated, we calculate and check whether its L_m or N_m value satisfy the corresponds thresholds. We can use dynamic programming to calculate length of longest maximal path or number of maximal paths on the transitively reduced graph of each generated candidate partial order. As an example of how these thresholds L_{th} and N_{th} can prune the generated potential candidates, consider the case of Fig. 9. If we work with $L_{th} = 1$, then the $\mathcal{Y}_1$ and $\mathcal{Y}_2$ combinations would be dropped. On the other hand if we work with $N_{th} = 3$, then the $\mathcal{Y}_0$ combination would be pruned.

Remark 2 The candidate generation algorithm presented here makes use of the unambiguous representation of an injective episode in terms of our (X^α, R^α) notation. If we allow repeated event-types in the episode then X^α has to be a multi-set. This can, in principle, be handled. However, now even the (X^α, R^α) notation becomes ambiguous. Recall that the $(V_\alpha, <_\alpha, g_\alpha)$ notation for an episode is ambiguous and we overcame this by choosing a representation such that $(g_\alpha(v_1), \ldots g_\alpha(v_N))$ is ordered as per the lexicographic ordering on $\mathcal{E}$ (this corresponds to our (X^α, R^α) notation). When we include all possible partial orders even this lexicographic ordering cannot make the (X^α, R^α) notation unambiguous. For example, consider the pattern $((A \to B)(C \to B))$. This is a parallel combination of two serial episodes. An occurrence of this is constituted by four events, two of which are of type B and the other two being of types A and C, such that A occurs before one of the B's and C occurs before the other B with no restriction on the relative times of occurrence of A and C. We can represent this as either of the two episodes β_1 and β_2 given by $\beta_1 = (V, <_1, g)$ and $\beta_2 = (V, <_2, g)$ with $V = \{v_1, v_2, v_3, v_4\}$, $g(v_1) = A$, $g(v_2) = g(v_3) = B$, $g(v_4) = C$ and $<_1, <_2$ represented by the two adjacency matrices given below.

$$<_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, <_2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

Since $(g(v_1), \ldots, g(v_4))$ are ordered as per the lexicographic ordering in both β_1 and β_2 and since the two adjacency matrices are different, these two episodes would be treated as different by our candidate generation algorithm. However, in any event stream data, these two episodes would have exactly the same set of occurrences. This means we can no longer guarantee that our candidate generation method will not generate duplicate episodes. This is the reason we restricted ourselves to injective episodes (see Achar (2010) for a more detailed discussion of the need for assuming injective episodes and for some ways of partially relaxing this assumption).

4 Selection of interesting partial order episodes

The frequent episode mining method would ultimately output all frequent episodes of up to some size. However, as we see in this section, frequency alone is not a sufficient indicator of interestingness (or support in the data) in case of episodes with general partial orders.

Suppose we have a data stream that actually contains many occurrences of the partial order episode $((AB) \to C)$. If the dependencies in the data stream are well captured by this partial order, then it is quite reasonable to suppose that, in occurrences of $((A\ B) \to C)$, A follows B roughly as often as it precedes B because this partial order does not constrain the order of occurrences of A and B. On mining from such a stream with reasonable thresholds, in addition to $((AB) \to C)$, we would also have all its three-node subepisodes, namely, $((A \to C)B)$, $((B \to C)A)$ and the parallel episode $(A\ B\ C)$, reported frequent (because frequency of subepisodes is at least as much as that of the episode). Since $((AB) \to C)$ well-captures the dependencies in the data, we can suppose that it is mostly the occurrences of $((AB) \to C)$ which contribute to the frequency of these subepisodes. Now the fact that we have seen A following B roughly as often as A preceding B and that we have rarely seen C not following both A and B should mean that the partial order $((AB) \to C)$ is a better representation of the dependencies in data as compared to episodes such as $((A \to C)B)$, $((B \to C)A)$ and $(A\ B\ C)$ in spite of all these being frequent. Hence, frequency alone is not a sufficient indicator of 'interestingness' for unrestricted partial order episodes.

An important point to note here is that when we consider episodes with unrestricted partial orders, an episode of size ℓ can have subepisodes which are also of size ℓ (as in the above example). It is easy to see that such subepisodes of size ℓ are its non-maximal subepisodes. As the size ℓ of the episode increases, in general, the number of such non-maximal subepisodes of size ℓ can increase exponentially with ℓ. For example, the four-node episode $(A \rightarrow B \rightarrow C \rightarrow D)$ has four-node non-maximal subepisodes like $(A(B \rightarrow C \rightarrow D))$, $(B(A \rightarrow C \rightarrow D))$, $(C(A \rightarrow B \rightarrow D))$, $(D(A \rightarrow B \rightarrow C))$, $(AB(C \rightarrow D))$, $(AB) \rightarrow C \rightarrow D$, $(ABC) \rightarrow D$, $A \rightarrow (BC) \rightarrow D$ etc. Such a situation does not arise if the mining process is restricted to either serial or parallel episodes only. For example, there is no four-node serial episode that is a subepisode of $(A \rightarrow B \rightarrow C \rightarrow D)$ and there is no four-node parallel episode that is a subepisode of $(A\ B\ C\ D)$.

4.1 Bidirectional evidence

We can state the intuition gained from the above example in general terms as follows. Given a set of event-types, choosing a partial order over them involves two choices: which pairs of event-types to constrain regarding their relative order of occurrence and which pairs of event-types to leave unconstrained. By definition, an occurrence of an episode captures the partial order information only of pairs of event-types which are constrained. Since any frequency measure counts some subset of the set of all occurrences, we can say that frequency is an indicator of the evidence in data in support of a partial order, only in so far as the pairs of events constrained by the partial order are concerned. It says nothing on whether there is also support in the data for leaving the other pairs of event-types unconstrained. Thus, we could say that for a partial order episode (X^α, R^α) to be interesting in addition to being frequent, we should also demand that in the set of occurrences of the episode, any two event-types, $i, j \in X^\alpha$, such that i and j are not related under R^α, should occur in either order ‘sufficiently often’.

Referring to our previous example, let $\alpha = ((A\ B) \rightarrow C)$ and let $\beta = ((A \rightarrow C)B)$. As explained earlier, in our data stream both would be frequent and since β is a subepisode of α it may have slightly higher frequency. In both these episodes the pair of event-types A and B are left unconstrained. In β the pair of event-types B and C are also left unconstrained. In the set of occurrences of β, if we see that most of the time B precedes C (as would be the case with our example data) then this should indicate that β is uninteresting (in the sense that it does not properly capture the dependencies in the data) even though it is frequent. Note that we are not saying α should be preferred over β because both are three-node and β is a subepisode of α. We are saying that there is evidence in the data to support not constraining A and B but there is no evidence in the data for not constraining B and C, and hence, α should be interesting, but not β.

When we mine in the space of only serial episodes or only parallel episodes, the above issue does not arise. In these cases, in any episode all pairs of event-types are treated the same way: in serial episodes every pair of event-types is constrained as regards their order in any occurrence of the episode while in parallel episodes every pair of event-types is left unconstrained. However, when we want to mine for episodes with general partial orders, an episode structure represents a choice as to which pair of event-types to constrain and which to leave unconstrained and hence there is need for looking for support in the data for both. As explained above, frequency captures the support for the decision of constraining some pairs of event-types. In this section we introduce another measure of interestingness, called *bidirectional evidence*, that tries to quantify the support in the data for a partial order episode as regards the pairs of event-types left unconstrained by the partial order.

Given an episode α let $\mathcal{G}^\alpha = \{(i, j) : i, j \in X^\alpha, i \neq j, (i, j), (j, i) \notin R^\alpha\}$. Let f^α denote the number of occurrences (i.e., frequency) of α counted by our algorithm and let f^α_{ij} denote the number of these occurrences where i precedes j. Let $p^\alpha_{ij} = f^\alpha_{ij}/f^\alpha$. Note that $p^\alpha_{ji} = 1 - p^\alpha_{ij}$, $\forall\, i, j \in \mathcal{G}^\alpha$.

Based on our earlier discussion, it is clear that we want p^α_{ij} to be close to p^α_{ji} for all $i, j \in \mathcal{G}^\alpha$. There are many functions, symmetric in (i, j), to obtain such a figure of merit (e.g., $p^\alpha_{ij}(1 - p^\alpha_{ij})$). In this paper we chose the entropy of distribution given by $(p^\alpha_{ij},\ 1 - p^\alpha_{ij})$ for this purpose. This is because entropy is a standard measure to quantify the deviation of a distribution from the uniform distribution. Let

$$H^\alpha_{ij} = -p^\alpha_{ij} \log(p^\alpha_{ij}) - (1 - p^\alpha_{ij}) \log(1 - p^\alpha_{ij}) \tag{4}$$

The *bidirectional evidence* of an episode α, denoted by $H(\alpha)$, is defined as follows.

$$H(\alpha) = \min_{(i,j)\in\mathcal{G}^\alpha} H^\alpha_{ij} \tag{5}$$

If $\mathcal{G}^\alpha$ is empty (which will be the case for serial episodes) then, by convention, we take $H(\alpha) = 1$.

Essentially, if $H(\alpha)$ is above some threshold, then there is sufficient evidence that all pairs of event-types in α that are not constrained by the partial order R^α appear in either order sufficiently often.

We use $H(\alpha)$ as an additional interestingness measure for α. We say that an episode α is interesting if (i) the frequency is above a threshold, and (ii) $H(\alpha)$ is above a threshold. In Sect. 6, we show through extensive simulation that the bidirectional evidence is very useful in making process of discovering general partial order episodes effective and efficient.

5 Counting algorithm

In this section, we present an algorithm for frequency counting. We first explain the construction of FSA to track occurrences of a general partial order episode followed by how to use such automata to compute the frequency. We also indicate how one can calculate bidirectional evidence of an episode while computing its frequency.

5.1 Finite state automaton for tracking occurrences

All the current algorithms for tracking serial episodes use (implicitly or explicitly) FSA. To track occurrences of serial episode $(A \to B)$, we should, while scanning data, first look for a A and after seeing it start looking for a B and so on. FSA are well suited for this purpose. In a similar manner FSA can be used for recognizing occurrences of episodes with unrestricted partial orders also. Such an automaton would start waiting for a given event-type $E \in X^\alpha$ immediately after having seen all its parents in R^α. This strategy of computation can be formalized using an FSA described below for a general injective episode α.

Definition 5 FSA $\mathcal{A}_\alpha$, used to track occurrences of α in the data stream is defined as follows. Each state, i, in $\mathcal{A}_\alpha$, is represented by a pair of subsets of X^α, namely $(\mathcal{Q}_i^\alpha, \mathcal{W}_i^\alpha)$. $\mathcal{Q}_i^\alpha$ contains all event-types accepted by the time FSA came to state i and $\mathcal{W}_i^\alpha$ contains the event-types that can be accepted by FSA in state i. The initial state, namely, *state 0*, is associated with the subsets pair, $(\mathcal{Q}_0^\alpha, \mathcal{W}_0^\alpha)$, where $\mathcal{Q}_0^\alpha = \phi$ and $\mathcal{W}_0^\alpha$ is the collection of *least* elements in X^α with respect to R^α. Let i be the current state of $\mathcal{A}_\alpha$ and let the next event in the data be of type, $E \in \mathcal{E}$. $\mathcal{A}_\alpha$ remains in *state i* if $E \in (X^\alpha \setminus \mathcal{W}_i^\alpha)$. If $E \in \mathcal{W}_i^\alpha$, then $\mathcal{A}_\alpha$ accepts E and transits into state j, with:

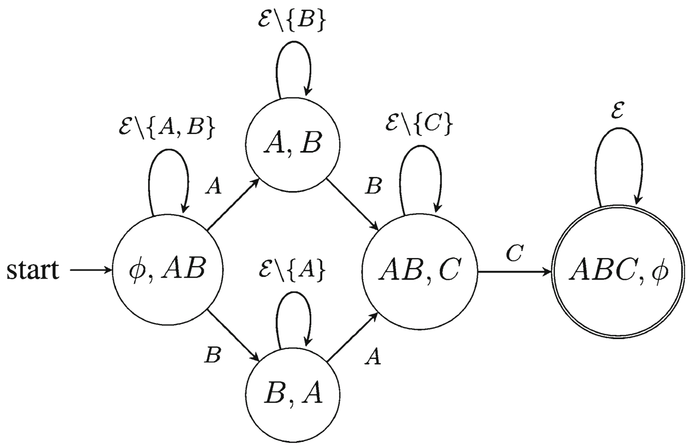
$$\mathcal{Q}_j^\alpha = \mathcal{Q}_i^\alpha \cup \{E\} \tag{6}$$

$$\mathcal{W}_j^\alpha = \{E' \in (X^\alpha \setminus \mathcal{Q}_j^\alpha) : \pi_\alpha(E') \subseteq \mathcal{Q}_j^\alpha\} \tag{7}$$

where $\pi_\alpha(E)$ is the subset of elements in X^α that are less than E (with respect to R^α). When $\mathcal{Q}_j^\alpha = X^\alpha$, (and $\mathcal{W}_j^\alpha = \phi$), j is the *final state* of $\mathcal{A}_\alpha$.

For example, consider episode $\alpha = ((AB) \to C)$. Here, $X^\alpha = \{A, B, C\}$ and $R^\alpha = \{(A, C), (B, C)\}$. The FSA used to track occurrences of this episode is shown in Fig. 10. Initially, the automaton has not accepted any events and is waiting for either of A and B, i.e., $\mathcal{Q}_0^\alpha = \phi$ and $\mathcal{W}_0^\alpha = \{A, B\}$. If we see a B first, the automaton transits to *state 2* with $\mathcal{Q}_2^\alpha = \{B\}$, $\mathcal{W}_2^\alpha = \{A\}$; on the other hand, if we first encounter an A, then it would transit into *state 1*, where $\mathcal{Q}_1^\alpha = \{A\}$, $\mathcal{W}_1^\alpha = \{B\}$. Once both A and B appear in the data, the automaton will transit, either from *state 1* or *state 2*, into *state 3*, where $\mathcal{Q}_3^\alpha = \{A, B\}$, $\mathcal{W}_3^\alpha = \{C\}$. Finally, if the automaton now encounters a C in the data stream, it will transit to the final state, *state 4*, and thus recognizes a full occurrence of the episode, $((AB) \to C)$.

It may be noted that not all possible tuples of $(\mathcal{Q}, \mathcal{W})$, where $\mathcal{Q} \subseteq X_\alpha$, $\mathcal{W} \subseteq X_\alpha$, constitute valid states of the automaton. For example, in Fig. 10, there can be no valid



start
ϕ, AB
A, B
B, A
AB, C
ABC, ϕ
$\mathcal{E}\backslash\{A,B\}$
$\mathcal{E}\backslash\{B\}$
$\mathcal{E}\backslash\{A\}$
$\mathcal{E}\backslash\{C\}$
$\mathcal{E}$
A
B
B
A
C

Fig. 10 Automaton for tracking occurrences of the episode $((A\ B) \to C)$

state corresponding to $\mathcal{Q} = \{A, C\}$ since C could not have been accepted without B being accepted before it (see Achar et al. (2009) for more discussion). We note here that, in view of (7) and (6), one of $\mathcal{Q}_j^{\alpha}$, $\mathcal{W}_j^{\alpha}$ will determine the other and hence one of them can completely specify the state. However, we use this redundant representation for the states of the automata for convenience.

5.2 Counting non-overlapped occurrences

We now explain how the frequency counting step computes, through one pass over the data, frequencies (non-overlapped sense) of all candidate episodes of size ℓ and returns the set of frequent ℓ-node episodes. The algorithm that we present here will also take care of what is known as the *expiry time* constraint. Under this, we count only those occurrences whose time span (which is the largest difference between the times of occurrence of any two events constituting the occurrence) is less than a user specified threshold T_X. In many applications, such a constraint is useful because episode occurrences constituted by widely separated events may not really indicate any underlying dependencies. Also, such a constraint can make mining more efficient by reducing the number of interesting patterns at each level.

The overall structure of the algorithm is as follows. We first initialize, for each candidate episode, the corresponding automaton in its start state. Then we traverse the data stream and for each event in the data stream, look at all the automata that can make a state transition on this event-type and effect the state transitions. In addition, whenever an automaton moves out of its start state (that is, when it accepts its first event-type), we initialize another automaton for that episode in the start state and this automaton also makes transitions as and when relevant event-types appear in the data stream. Whenever, because of state transition by an automaton, two automata of an episode come to the same state we retain the automaton that moved out of its start state most recently (and retire the older automaton). This strategy helps us to track the innermost occurrence in a set of overlapped occurrences and is useful because we want to find maximum number of occurrences that satisfy the expiry time constraint. When an automaton reaches its final state, we check whether the occurrence tracked by it satisfies the expiry time constraint. If so, we increment the frequency by one and retire all automata of this episode except for one in the start state, so that we can start tracking the next non-overlapped occurrence. If the occurrence tracked by the automaton that reached the final state does not satisfy the expiry time constraint, we retire only the automaton that reached the final state (and do not increment frequency), so that, through the other existing automata of this episode, we can keep looking for occurrences that satisfy the expiry time constraint.

We first illustrate the algorithm with an example data stream and then indicate the idea of its proof of correctness. For both these purposes, we need the concept of an earliest transiting (ET) occurrence of a general partial order episode (Achar 2010). For ease of illustration, we now stick to data streams with one event per time tick. While defining ET occurrences, we denote an occurrence h by the times of occurrence of the events constituting the occurrence, with the times arranged in an increasing order, that is, $h = [t_1^h, \ldots t_\ell^h]$, where $t_i^h < t_{i+1}^h, i = 1 \ldots (\ell - 1)$. We denote by E_i^h the event-type of the event at t_i^h (within the occurrence h).

Definition 6 An occurrence h of a general injective episode α in a data stream $\mathbf{D}$ is said to be an ET occurrence if (i) E_1^h is a least element of X^α and (ii) for $i = 2, 3, \ldots \ell$, t_i^h is the time of the first occurrence of the event-type E_i^h after the occurrence of all event-types of $\pi_\alpha(E_i^h)$ in the portion of the occurrence h seen so far.

Consider the data stream in Fig. 11 (for now ignore the occurrences shown in color). There are six ET occurrences of $((A\ B) \to C)$ which are: $h_1^e = [1\,4\,6]$, $h_2^e = [4\,7\,8]$, $h_3^e = [5\,7\,8]$, $h_4^e = [7\,10\,13]$, $h_5^e = [10\,11\,13]$ and $h_6^e = [11\,12\,13]$. Here we are using the vector of times representation for specifying the occurrences. Thus, for example, h_1^e is the occurrence consisting of events $\langle (B, 1), (A, 4), (C, 6)\rangle$. The occurrence $[5\,7\,9]$ is not an ET occurrence in the data sequence as $(C, 9)$ is not the earliest C after seeing its set of parents $\{A, B\}$ at time tick 5 and 7. Similarly, $[1\,5\,6]$ is also not an ET occurrence. The occurrences of $((A\ B) \to C)$ indicated in Fig. 11, namely, h_3^e and h_5^e, are the ones tracked by our algorithm for an expiry constraint $T_X = 4$.

For an episode α, the associated FSA (as per Definition 5) basically track ET occurrences of α in the data stream. In our algorithm (described earlier in this section), the first automaton initially in the start state would exactly track h_1^e. Similarly the second automata initialized in the start state after the first one moves out of its start state would now track h_2^e. Thus our algorithm searches in the space of ET occurrences. However, not every automaton of α would reach the final state because when two automata come into the same state the older one is removed. If we follow the algorithm as described earlier, the automaton corresponding to h_1^e is the first one that reaches final state. But since h_1^e violates expiry of 4 we retire only this automata and continue. The next automaton which reaches final state would track h_3^e. (Note that the automaton tracking h_3^e, after accepting event $(A, 5)$, would reach the same state that was occupied by the automaton tracking h_2^e. Hence the automaton tracking h_2^e would be retired.) Since the occurrence h_3^e satisfies expiry constraint of 4, we increment the frequency and discard all automata except the one in start state. Continuing like this, one can see that the next automaton that reaches final state tracks h_6^e. These two are the occurrences highlighted in Fig. 11.

We now discuss the idea of the proof of correctness for the counting scheme. Consider a set of occurrences $\mathcal{H}_n = \{h_1, h_2, \ldots h_f\}$ chosen in a greedy fashion as follows. h_1 is the first ET occurrences satisfying T_X. h_2 is the first ET occurrence non-overlapped with h_1 and satisfying T_X and so on. There is no ET occurrence satisfying T_X beyond $t_{h_f(N)}$. For example, in the data stream of Fig. 11, $h_1 = h_2^e$ and $h_2 = h_f = h_5^e$. We have indicated these occurrences separately in Fig. 12. Because of the greedy strategy, it is intuitively clear that $\mathcal{H}_n$ is a maximal set of non-overlapped occurrences satisfying T_X and it can be formally shown to be so (Achar 2010). The important property to note from Figs. 11 and 12 is that the set of ET occurrences tracked by our algorithm (say $\{h_1', h_2' \ldots\}$), even though different from $\mathcal{H}_n$, is such that h_i' is the last ET occurrence that ends with h_i. For a formal proof of this, refer

$\langle (B,1), (A,4), (A,5), (C,6), (B,7), (C,8), (C,9), (A,10), (B,11), (A,12), (C,13) \rangle.$

Fig. 11 A maximal set of non-overlapped occurrences of $((A\ B) \to C)$ with $T_X = 4$ tracked by the algorithm

$\langle (B,1), (A,4), (A,5), (C,6), (B,7), (C,8), (C,9), (A,10), (B,11), (A,12), (C,13) \rangle$.

Fig. 12 A maximal set of non-overlapped occurrences of $((A\ B) \rightarrow C)$ with $T_X = 4$ chosen in a greedy fashion

to Achar (2010). Because of this, the maximality of the set of occurrences tracked by our algorithm follows.

The algorithm for counting non-overlapped occurrences of general injective episodes with an expiry-time constraint is given as a pseudocode in Algorithm 2. As explained in Sect. 2, the event-types associated with an ℓ-node episode, α, are stored in the array $\alpha.g$, i.e., $\alpha.g[i] = g_\alpha(v_i)$, $i = 1, \ldots, \ell$. We store the partial order, $<_\alpha$, associated with the episode as a binary adjacency matrix, $\alpha.e$. The notation is: $\alpha.e[i][j] = 1$ iff $v_i <_\alpha v_j$.

The main data structure is an array of lists, $waits()$, indexed by the set of event-types. The elements of each list store the relevant information about all the automata that are waiting for a particular event-type. The entries in the list are of the form $(\alpha, \mathbf{q}, \mathbf{w}, j)$ where α is a candidate episode, $(\mathbf{q}, \mathbf{w})$ is one of the possible states of the automaton associated with α (cf. Definition 5) and j is an integer. Recall that each state of the automaton is specified by a pair of subsets, $(\mathcal{Q}^\alpha, \mathcal{W}^\alpha)$, of the set of event-types X^α of α. In our representation, $\mathbf{q}$ and $\mathbf{w}$ are $|X^\alpha|$-length binary vectors encoding the two sets $(\mathcal{Q}^\alpha, \mathcal{W}^\alpha)$. For an event-type E, if $(\alpha, \mathbf{q}, \mathbf{w}, j) \in waits(E)$, it denotes that an automaton of the episode α (with $\alpha.g[j] = E$) is currently in state $(\mathbf{q}, \mathbf{w})$ and is waiting for an event-type E to make a state transition. Consider the episode $\beta = (A\ B) \rightarrow (C\ D)$ with $X^\beta = \{A, B, C, D\}$. Suppose an automaton corresponding to this episode has already accepted an A and B and is waiting for a C or D. We would have $(\beta, \mathbf{q}, \mathbf{w}, 3) \in waits(C)$ and $(\beta, \mathbf{q}, \mathbf{w}, 4) \in waits(D)$ where $\mathbf{q} = [1\,1\,0\,0]$ and $\mathbf{w} = [0\,0\,1\,1]$.

In addition to the $\alpha.g$ and $\alpha.e$ arrays, we also have three other quantities associated with each episode, namely, $\alpha.freq$, $\alpha.init$ and $\alpha.\mathbf{w}_{start}$. The frequency of an episode is stored in $\alpha.freq$. The list $\alpha.init$ keeps track of the times at which the currently active automata of α made their transitions out of the start state. Each entry in the list is a pair, $(\mathbf{q}, t)$, which indicates that an automaton that made its first state transition at time t is currently in a state with the set of accepted events represented by $\mathbf{q}$. Since an automaton in the start state is yet to make its first transition, it has no corresponding entry in $\alpha.init$. $\alpha.\mathbf{w}_{start}$ is a binary vector of length $|X^\alpha|$ which encodes $\mathcal{W}_0^\alpha$, the set of all least elements of X^α (that is, the set of all event-types that an automaton for α would be ready to accept in its start state).

Algorithm 2 works as follows. We start by initializing all $waits$ lists. For each candidate episode, we put one automaton (in its start state) in the list $waits(E)$ for all $E \in \mathcal{W}_0^\alpha$. The main computation consists of traversing the data stream, and, for each event, (E_k, t_k), effecting appropriate changes in the automata waiting for E_k. If $(\alpha, \mathbf{q}_{cur}, \mathbf{w}_{cur}, j) \in waits(E_k)$, we have to find the next state $(\mathbf{q}_{nxt}, \mathbf{w}_{nxt})$ of this automaton and make appropriate changes in all lists to reflect this state transition. The $\mathbf{q}_{nxt}$ is easily computed by adding E_k to $\mathbf{q}_{cur}$. If $\mathbf{q}_{cur} = \mathbf{0}$ (the start state), it means that this automaton is moving out of its start state at the current time t_k and hence we add $(\mathbf{q}_{nxt}, t_k)$ to $\alpha.init$ list. We also need to initialize another automaton for α and this done by adding α to bag. (After processing all automata in $waits(E_k)$, we initialize fresh automata for all episodes in bag.) If $\mathbf{q}_{nxt} \neq \mathbf{1}$ (the final state), we update the entry for this automaton in $\alpha.init$ list. Also, if there is already an automaton in state $\mathbf{q}_{nxt}$ in $\alpha.init$, we drop the older automaton. Then, we compute $\mathbf{w}_{nxt}$ using $\mathbf{q}_{nxt}$ and $\alpha.e$ (cf. Eqn. 7 or more efficiently as in lines 21–23 of Algorithm 2). Next, we make changes to appropriate waits lists so that the automaton is in $waits(E)$ only for those E indicated by $\mathcal{W}_{nxt}$. Note that we effect all these changes only if $\mathbf{q}_{nxt} \neq \mathbf{1}$. If $\mathbf{q}_{nxt} = \mathbf{1}$ then this automaton has reached its final state. Also, this automaton would have made its first state transition at t_{cur} if entry in $\alpha.init$ reads $(\mathbf{q}_{cur}, t_{cur})$. Hence if $t_k - t_{cur} < T_X$, the occurrence tracked by this automaton satisfies the expiry time constraint and hence we increment frequency and retire all currently active automata for this episode.

Except for the features that the automaton structure here is more complicated and that an active automaton can simultaneously be in multiple $waits()$ lists, the overall structure of this algorithm is similar to the serial episode mining algorithm in Laxman et al. (2005).

In the algorithm described above we implicitly assumed that different events in the data stream have distinct time stamps. This is because, in the data pass loop an automaton can accept E_{k+1} after accepting E_k in the previous pass through the loop. With small changes to Algorithm 2 we can use the same general approach to track occurrences even when multiple events in the data have the same time-stamp. The main change needed is in the state transition step of Algorithm 2. Let us denote the set of event-types occurring at time t by $\mathcal{S}$. In general, an automaton waiting for $\mathcal{W}^{\alpha}_{cur}$ just before time t, should now accept all events in $\mathcal{S} \cap \mathcal{W}^{\alpha}_{cur}$ and we will have $\mathcal{Q}^{\alpha}_{nxt} = \mathcal{Q}^{\alpha}_{cur} \cup (\mathcal{S} \cap \mathcal{W}^{\alpha}_{cur})$. $\mathcal{W}^{\alpha}_{nxt}$ can be computed from $\mathcal{Q}^{\alpha}_{nxt}$ as in Eq. 7. We could do the same thing by processing one event after another as in Algorithm 2, but this would need some additional book-keeping. Consider an automaton for the episode $(B\,C) \rightarrow D$ waiting in its start state $(\mathcal{Q}^{\alpha}_{cur}, \mathcal{W}^{\alpha}_{cur}) = (\phi, \{B, C\})$. Suppose we have the event-types B, C and D occurring together at time t. We should accept both B and C but not D, even though we add (α, [1 1 0], [0 0 1], 3) to $waits(D)$ after seeing $(B, t)\&(C, t)$. This potential transition cannot be active at time t. Such potential transitions, newly added to $waits()$ (at the current time) must be initially inactive, till all event-types at the current time are processed. After performing the state transitions pertaining to all event-types at the current time instant, the rest of the steps are same as in Algorithm 2. Since we increment frequency only after processing all events at a given time instant, during state transition, we also need to remember which automata (if any) reached their final states.

5.2.1 Space and time complexity of the counting algorithm

The number of automata that may be active (at any time) for each episode is central to the space and time complexities of the counting algorithm. The number of automata currently active for a given ℓ-node episode, α, is one more than the number of elements in the $\alpha.init$ list. Suppose there are m entries in $\alpha.init$ list, namely, $(\mathbf{q}_1, t_{i_1}), \ldots, (\mathbf{q}_m, t_{i_m})$, with $t_{i_1} < t_{i_2} < \cdots < t_{i_m}$. (Recall our notation: $\mathbf{q}_j$ represents

Algorithm 2: Count frequency expiry time ($\mathcal{C}_l$, $\mathbb{D}$, γ, $\mathcal{E}$, T_X)

Input: Set $\mathcal{C}_l$ of candidate episodes, event stream $\mathbb{D} = \langle(E_1, t_1), \ldots, (E_n, t_n)\rangle$, frequency threshold γ, set $\mathcal{E}$ of event-types (alphabet), Expiry Time, T_X

Output: Set $\mathcal{F}$ of frequent episodes out of $\mathcal{C}_l$

```
1  /* Initialization */
2  $\mathcal{F}_l \leftarrow \phi$ and $bag \leftarrow \phi$ ;
3  foreach event-type $E \in \mathcal{E}$ do $waits[E] \leftarrow \phi$;
4  foreach $\alpha \in \mathcal{C}_l$ do
5      $\alpha.freq \leftarrow 0$ and $\alpha.\mathbf{w}_{start} \leftarrow \mathbf{0}$;
6      forall $i$ such that $i$ has no parents in $\alpha.e$ do $\alpha.\mathbf{w}_{start}[i] \leftarrow 1$;
7      forall $i$ such that $\alpha.\mathbf{w}_{start}[i] = 1$ do Add $(\alpha, \mathbf{0}, \alpha.\mathbf{w}_{start}, i)$ to $waits[\alpha.g[i]]$;
8      /* $\mathbf{0}$ is a vector of all zeros */
9  /* Database pass */
10 for $k \leftarrow 1$ to $n$ do
11     foreach $(\alpha, \mathbf{q}_{cur}, \mathbf{w}_{cur}, j) \in waits[E_k]$ do
12         /* $n$ is the number of events and $E_k$ is the currently processed event-type in the event stream
           */
13         /* Transit the current automaton to the next state */
14         $\mathbf{q}_{nxt} \leftarrow \mathbf{q}_{cur}$ and $\mathbf{q}_{nxt}[j] \leftarrow 1$;
15         if $\mathbf{q}_{cur} = \mathbf{0}$ then Add $(\mathbf{q}_{nxt}, t_k)$ to $\alpha.init$ and Add $\alpha$ to $bag$;
16         /* $t_k$ - time associated with the current event in event stream */
17         else
18             if $\mathbf{q}_{nxt} \neq \mathbf{1}$ then Update $(\mathbf{q}_{cur}, t_{cur})$ in $\alpha.init$ to $(\mathbf{q}_{nxt}, t_{cur})$;
19             /* $\mathbf{1}$ is a vector of all ones and $t_{cur}$ is the first state transition time of the current
               automaton*/
20         if $\mathbf{q}_{nxt} \neq \mathbf{1}$ then
21             $\mathbf{w}_{nxt} \leftarrow \mathbf{w}_{cur}$, $\mathbf{w}_{nxt}[j] \leftarrow 0$ and $\mathbf{w}_{temp} \leftarrow \mathbf{w}_{nxt}$ ;
22             for each child $i$ of $j$ in $\alpha.e$ do
23                 if for each parent $m$ of $i$, $\mathbf{q}_{nxt}[m] = 1$ then $\mathbf{w}_{nxt}[i] \leftarrow 1$;
24             for $i \leftarrow 1$ to $|\alpha|$ do
25                 if $\mathbf{w}_{temp}[i] = 1$ then Replace $(\alpha, \mathbf{q}_{cur}, \mathbf{w}_{cur}, i)$ from $waits[\alpha.g[i]]$ to
                   $(\alpha, \mathbf{q}_{nxt}, \mathbf{w}_{nxt}, i)$;
26                 if $(\mathbf{w}_{temp}[i] = 0$ and $\mathbf{w}_{nxt}[i] = 1)$ then Add $(\alpha, \mathbf{q}_{nxt}, \mathbf{w}_{nxt}, i)$ to $waits[\alpha.g[i]]$;
27         Remove $(\alpha, \mathbf{q}_{cur}, \mathbf{w}_{cur}, j)$ from $waits[\alpha.g[j]]$;
28         /* Removing an older automaton if any in the next state */
29         if $((\mathbf{q}_{nxt}, t') \in \alpha.init$ and $t' < t_{cur})$ then
30             /* $t'$ is the first state transition time of an older automaton existing in state $\mathbf{q}_{nxt}$ */
31             Remove $(\mathbf{q}_{nxt}, t')$ from $\alpha.init$ and all $waits[]$ entries corresponding to this older
               automaton;
32         /* Increment the frequency */
33         if $(\mathbf{q}_{nxt} = \mathbf{1}$ and $(t_k - t_{cur}) \leq T_X)$ then
34             $\alpha.freq \leftarrow \alpha.freq + 1$ and Empty $\alpha.init$ list;
35             Remove all $waits[]$ entries corresponding to $\alpha$ and Add $\alpha$ to $bag$;
36     /* Add automata initialized in the start state */
37     foreach $\alpha \in bag$ do
38         forall $i$ such that $\alpha.\mathbf{w}_{start}[i] = 1$ do Add $(\alpha, \mathbf{0}, \alpha.\mathbf{w}_{start}, i)$ to $waits[\alpha.g[i]]$;
39         /* $\mathbf{0}$ is a vector of all zeros */
40     Empty $bag$;
41 foreach $\alpha \in \mathcal{C}_l$ do if $\alpha.freq > \gamma$ then Add $\alpha$ to $\mathcal{F}_l$;
42 return $\mathcal{F}_l$
```

the set of accepted event-types for the jth active automaton which moved out of start state at time t_{i_j}.) Consider k, ℓ such that $1 \leq k < \ell \leq m$. The events in the data stream that effected transitions in the ℓth automaton (i.e., automaton which moved out of start state at t_{i_ℓ}) would have also been *seen* by the kth automaton. If the kth automaton has not already accepted previous events with the same event-types, it will do so now on seeing the events which affect the transitions of the ℓth automaton. Hence, $\mathbf{q}_\ell \subsetneq \mathbf{q}_k$ for any $1 \leq k < \ell \leq m$. Since $|X^\alpha| = \ell$, there are at most ℓ (distinct) telescoping subsets of X^α, and so, we must have $m \leq \ell$.

The time required for initialization in our algorithm is $\mathcal{O}(|\mathcal{E}| + |\mathcal{C}_\ell|\ell^2)$. This is because, there are $|\mathcal{E}|waits()$ lists to initialize and it takes $\mathcal{O}(\ell^2)$ time to find the least elements for each of the $|\mathcal{C}_\ell|$ episodes. For each of the n events in the data, the corresponding $waits()$ list contains no more than $\ell|\mathcal{C}_\ell|$ elements as there can exist at most ℓ-automata per episode. The updates corresponding to each of these entries takes $\mathcal{O}(\ell^2)$ time to find the new elements to be added to the $waits()$ lists. (Note that we take $\mathcal{O}(\ell^2)$ time because we are explicitly computing the next state using the adjacency matrix $\alpha.e$ rather than store the state transition matrix of the automaton. This is done to save space. Otherwise, computing next state would be $\mathcal{O}(1)$ time.) Thus, the worst-case time complexity of the data pass is $\mathcal{O}(n\ell^3|\mathcal{C}_\ell|)$.

For each automaton, we store its state information in the binary ℓ-vectors $\mathbf{q}$ and $\mathbf{w}$. To be able to make $|\mathcal{W}|$ transitions from a given state, we maintain $|\mathcal{W}|$ elements in various $waits()$ lists with each element ready to accept one of the event-types in $\mathcal{W}$. In the pseudocode (for ease of explanation), we kept the state information of $\mathbf{q}$ and $\mathbf{w}$ in each of these $|\mathcal{W}|$ elements. We can instead do it more efficiently by storing the state information of $\mathbf{q}$ and $\mathbf{w}$ just once in the $\alpha.init$ list along with the first state transition time and keeping a pointer to this element in the various $waits()$ lists elements. Hence, we only require $\mathcal{O}(\ell)$ space per automaton. Since there are $\ell|C_\ell|$ such automata in the worst case, the contribution of the various automaton to the space complexity is $\mathcal{O}(\ell^2|C_\ell|)$. Also since each episode needs $\mathcal{O}(\ell^2)$ space to store its adjacency matrix the overall worst-case space complexity is $\mathcal{O}(\ell^2|C|)$.

5.3 Computing $H(\alpha)$ and mining with an additional $H(\alpha)$ threshold

We now explain how bidirectional evidence ($H(\alpha)$) can be computed during our frequency counting process. For each candidate episode α, the matrix $\alpha.H$ is initialized to 0 just before counting. For each automaton that is initialized, we initialize a separate $\ell \times \ell$ matrix stored with the automaton. Whenever an automaton makes state transitions on an event-type j, for all i such that event-type i is already seen, we increment the (i, j) entry in this matrix. The matrix associated with an automaton that reaches its final state, is added to $\alpha.H$ and results in increment of relevant entries. Thus, at the end of the counting, $\alpha.H$ gives the f_{ij}^α information.

One way to use a threshold on bidirectional evidence is to apply it as only a post processing filter. This way, we will only report all frequent episodes that also meet the minimum $H(\alpha)$ criterion. While this may reduce the number of frequent episodes in the final output, there will still be a combinatorial explosion in the number of frequent/candidate episodes as the algorithm traverses from lower to higher size episodes.

For example, on embedding a serial episode of size 8 (based on the data generation described in Sect. 6.1), the discovery algorithm (with a level-wise frequency threshold only) generated in excess of 100,000 candidates at level 6. In order to avoid this, we filter frequent episodes using a threshold on $H(\alpha)$ at each level in the algorithm. Note that, unlike a threshold on frequency, a threshold on $H(\alpha)$ may not meet anti-monotonicity criteria in general, since $H(\alpha)$ is computed using the specific set of occurrences tracked by the counting algorithm. However, if an episode α has a bidirectional evidence $H(\alpha) = e$ in a given set of occurrences, then any maximal subepisode of α (obtained by the restriction of R^{α} onto a subset of X^{α}) will also have a bidirectional evidence of at least e in the same set of occurrences (by Eq. 5). If occurrences of the subepisodes of α predominantly coincide with the corresponding partial occurrences of α, then bidirectional evidence of all its maximal subepisodes will be at least that of α. Since our candidate generation is based on the existence of all maximal subepisodes at the lower levels, the level-wise procedure would not eliminate interesting patterns with high $H(\alpha)$. Further, the bidirectional evidence of the non-maximal subepisodes of α will be low (often close to zero). The reason for this is as follows. In any non-maximal subepisode γ (of α) we can find a pair of nodes i and j, such that, while there is an edge between them in α, there is no edge between them in γ. Now, if most occurrences of γ coincide with the corresponding partial occurrences of α, then i precedes j in almost all occurrences of γ and hence $H(\gamma)$ will be small (by Eq. 5). Hence almost all non-maximal subepisodes of α, though frequent, will have negligible bidirectional evidence. These non-maximal subepisodes, if not weeded out, would otherwise contribute to the generation of an exponential no. of patterns at various levels. This huge reduction in candidates across all levels results in efficient mining when $H(\alpha)$ threshold is incorporated level-wise. In the results section, we show through simulation that mining with a level-wise threshold on H_{α} is effective in practice.

6 Simulation results

In this section we present experimental results obtained on synthetic data as well as on multi-neuronal spike train data. We show that our algorithm for mining unrestricted partial orders that uses the new notion of bidirectional evidence is capable of efficiently discovering all the ‘correct’ partial order patterns embedded in synthetic data (while reporting only a small number of spurious or noisy patterns as frequent). Next we show the performance of our algorithms on two classes of neuroscience data sets—(i) data from a mathematical model of multiple spiking neurons (Sastry and Unnikrishnan 2010), and (ii) real neuronal activity recorded from dissociated cortical cultures (Wagenaar et al. 2006).[5]

6.1 Synthetic data generation

Input to the data generator is a set of episodes that we want to embed in the synthetic data. For each episode in the set, we generate an *episode event-stream* which contains

[5] We are thankful to Professor Steve Potter at Georgia Tech for providing us this data.

just non-overlapped occurrences of the episode (and no other events). Thus, an episode event-stream for $(A \rightarrow (BC))$ would, e.g., look like $\langle (A, t_1), (B, t_2), (C, t_3), (A, t_4), (C, t_5), (B, t_6), \ldots \rangle$. Separately, we generate a noise stream $\langle (X_1, \tau_1), (X_2, \tau_2), \ldots \rangle$ where X_i's take values from the entire alphabet of event-types. All the episode event-streams and the noise stream are merged to generate the final data stream (by stringing together all events in all the streams in a time-ordered fashion). The data generation process has three important user-specified parameters: η (span parameter), p (inter-occurrence parameter) and ρ (noise parameter), whose roles are explained below.

To generate an episode event-stream, we generate several occurrences of the episode successively. For each occurrence, we randomly choose one of its *serial extensions*[6] and this fixes the sequence of event-types that will appear in the occurrence being embedded. The time difference $(t_{i+1} - t_i)$ between successive events in an occurrence is generated according to a geometric distribution with parameter $\eta (0 < \eta \leq 1)$. The time between end of an occurrence and the start of the next is also distributed geometrically with (a different) parameter $p(0 < p \leq 1)$. Thus, η determines the expected span of an occurrence and p controls the expected time between consecutive occurrences.

We generate the noise stream as follows. For each event-type in the alphabet we generate a separate sequence of its occurrences with inter-event times distributed geometrically. For all *noise* event-types, namely event-types that are not in any of the embedded episodes, the geometric parameter is $\rho (0 < \rho \leq 1)$ and for all other event-types this parameter is set to $\rho/5$. This way, we introduce some random occurrences of the event-types associated with the embedded partial orders. All these streams are merged to form a single noise stream. Noise stream is generated in this way so that there may be multiple events (constituting noise) at the same time instant. We note here that the value of ρ does not indicate any percentage of noise. For example, with $\rho = 0.05$ we expect each noise event-type to appear once every 20 time-ticks and if there are 40 noise event-types, then (on the average) there would be two noise events at every time tick. Thus, even small values of ρ can insert substantial levels of noise in the data. We chose such a method for synthetic data generation as it allows us to control the expected spans/frequency of embedded episodes independently of the level of noise.

While presenting our results, in all our tables, we give the values of different parameters in the table caption. In addition to ρ, p and η, the other parameters are as follows: M denotes the total number of event-types or the cardinality of $\mathcal{E}$, T_X is the expiry-time threshold, f_{th} and H_{th} are the thresholds on frequency and bidirectional evidence.

6.2 Effectiveness of partial order mining

We generated a data stream of about 50,000 events (using an alphabet of 60 event-types) with 10,000 time-ticks or distinct event-times, by embedding 2 six-node partial orders, $\alpha_1 = (A \rightarrow (B\,C) \rightarrow (D\,E) \rightarrow F)$ and $\alpha_2 = (G \rightarrow ((H \rightarrow (J\,K))(I \rightarrow L))$. (As is evident, the data stream has more than one event-type per time tick, specifically

[6] A serial extension of a partially ordered set (X^α, R^α) is a totally ordered set (X^α, R') such that $R^\alpha \subseteq R'$.

Table 2 Results obtained in three cases: (A) frequency threshold (f_{th}) only, (B) bidirectional evidence threshold (H_{th}) as a post filter, (C) both f_{th} and H_{th} level-wise

Level	(A)		(B)		(C)	
	#Cand	#Freq	#Cand	#Freq	#Cand	#Freq
1	60	60	60	60	60	60
2	5310	565	5310	565	5310	565
3	3810	435	3810	331	3810	331
4	1358	760	1358	129	623	125
5	1861	1855	1861	37	36	32
6	2993	2993	2993	6	6	6
7	0	0	0	0	0	0
Run-time	134 s		142 s		52 s	

Patterns: α_1 and α_2, $\eta = 0.7$, $\rho = 0.055$, $p = 0.07$, $M = 60$, $f_{th} = 350$, $T_X = 15$, $H_{th} = 0.4$

has five event-types on an average per time-tick.) Other data generation parameters are given in the caption of Table 2. These were chosen such that the one-node frequencies of all event-types were almost same and many of two-node episodes involving noise event-types are frequent. Table 2 shows the results obtained with our mining algorithm.[7] We show the number of candidates (#Cand) and the number of frequent episodes (#Freq) at different levels. (Recall that at level k, the algorithm finds all frequent episodes of size k.) The table shows the results for three cases: (A) when we apply (at each level) only a frequency threshold, f_{th}, to determine the output, (B) when we apply f_{th} as usual, but also use a threshold, H_{th}, on bi-directional evidence, $H(\alpha)$, to post-filter the output, and (C) when we apply f_{th} as usual and use H_{th} *within each level* (during the level-wise procedure) to filter episodes propagating to the next level.

The two embedded patterns are reported as frequent in all the three cases. (To keep the terminology simple, we generally refer to the output, even in cases that use H_{th}, as 'frequent episodes'.) However, with only a frequency threshold (case A), a lot of uninteresting patterns (like the subepisodes of the embedded patterns) are also reported frequent. When we use an $H(\alpha)$ threshold for post-filtering the output (case B), the number of candidates remains unaffected, but the numbers of frequent episodes (at different levels) reduce considerably. This shows the utility of bi-directional evidence in reporting only the interesting partial orders. However, the run-time actually increases marginally because of the overhead of post-filtering based on $H(\alpha)$. If we filter using the $H(\alpha)$ threshold within the level-wise procedure (case C), then the efficiency also improves considerably, as can be seen from the reduction both in numbers of candidates as well as in run-times. In terms of output quality, we observe that the same episodes are output in both case B and case C at level 6.

Table 3 provides details of the kinds of episodes obtained at different levels (starting with level 4) under the three cases described earlier. Columns #Cand and #Freq

[7] The source codes have all been written in C++. The experiments have been run on a 2 GHz Pentium PC under a Linux operating system.

Table 3 Details of frequent episodes obtained with (A) frequency threshold (f_{th}) only, (B) bidirectional evidence threshold (H_{th}) as a post filter, (C) both f_{th} and H_{th} level-wise

Level	Case	#Cand	#Freq	Subepisodes				#Others
				#Max		#Non-max		
				α_1	α_2	α_1	α_2	
	(A)	1358	760	15	15	411	142	117
4	(B)	1358	129	15	15	15	14	70
	(C)	623	125	15	15	15	13	67
	(A)	1861	1855	6	6	1268	228	347
5	(B)	1861	37	6	6	6	1	18
	(C)	36	32	6	6	6	1	13
	(A)	2993	2993	1	1	2385	174	432
6	(B)	2993	6	1	1	1	0	3
	(C)	6	6	1	1	1	0	3

Patterns: α_1 and α_2, $\eta = 0.7$, $\rho = 0.055$, $p = 0.07$, $M = 60$, $f_{th} = 350$, $T_X = 15$, $H_{th} = 0.4$

indicate the number of candidates and frequent episodes obtained at each level. The remaining columns group the frequent episodes output (at each level) into different categories. The four columns under *Subepisodes* category indicate the number of frequent episodes output which are subepisodes of the embedded patterns. These are categorized under maximal (#Max) and non-maximal (#Non-max) subepisodes of α_1 and α_2. (Frequent episodes corresponding to the embedded patterns themselves will come under the maximal subepisode category.) The final column (#others) shows the number of frequent episodes output that are not subepisodes of either of the embedded patterns. The *Case* column refers to the three cases A, B and C defined earlier—which indicate whether (and how) bi-directional evidence was used in the pattern mining algorithm.

From Table 3 we see that using only a frequency threshold (Case A) leads to a total of 2,993 episodes of size 6 being reported as frequent. Of these, 2,559 ($= 2,385 + 174$) are non-maximal subepisodes of α_1 and α_2. When we use bi-directional evidence (cases B and C), only six episodes of size 6 are reported in the output: the two embedded patterns, one non-maximal subepisode of α_1 and three super-episodes of α_2. Thus, when we use only a threshold on frequency (case A), many of the frequent episodes are the non-maximal subepisodes of embedded patterns which can never be eliminated based on their frequencies. This is the issue of an explosive number of frequent but uninteresting patterns being thrown up by frequent partial order mining that we pointed out in Sect. 4. It was also observed that the remaining frequent episodes (other than the super-episodes of α_2) generated in the others category for case A were neither subepisodes nor superepisodes of the embedded patterns. They involved the same event types as α_2 and contain a pair of event-types i, j unrelated in them, but related in α_2. Bidirectional evidence is effective in eliminating not only non-maximal subepisodes but also a significant number of such spurious episodes (others category) (see discussion in the second part of Sect. 5.3). The results in Table 3 show that using a level-wise threshold on $H(\alpha)$ provides substantial improvement in efficiency while not missing any important patterns present in the data. In all the subsequent experiments, we use a level-wise threshold on $H(\alpha)$ in addition to the frequency threshold.

6.2.1 Robustness with respect to parameters

Next we study the performance of our algorithms when patterns are embedded with different strengths in the data (in varying levels of noise). Toward this end, we generated nine data sets, each containing occurrences of the six-node patterns α_1 and α_2 defined earlier. The data sets are generated by varying the parameters ρ, η and p as given in Table 4. The span parameter η is varied across the X sets; in the Y sets the inter occurrence parameter p is varied; and in the Z sets, we vary the noise parameter ρ. As p increases, the expected inter-occurrence time reduces and hence the extent of overlap between occurrences of α_1 and α_2 also increases. This makes the mining task harder (the data denser) because, spurious patterns consisting of a part of α_1 and part of α_2 can also become frequent.

Table 5 gives the ranks (based on frequency) of α_1 and α_2 obtained for a range of expiry-time thresholds. The results show that our algorithms consistently report the two embedded partial orders in the top three patterns. In case of very small expiry times ($T_X = 9$) we do not find the embedded partial orders in sets X and Y since

Table 4 Synthetic data sets

	X1	X2	X3	Y1	Y2	Y3	Z1	Z2	Z3
η	0.4	0.6	0.8	0.6	0.6	0.6	0.8	0.8	0.8
p	0.03	0.03	0.03	0.03	0.05	0.07	0.07	0.07	0.07
ρ	0.03	0.03	0.03	0.05	0.05	0.05	0.03	0.05	0.07

Patterns: α_1 and α_2, $M = 60$

Table 5 Frequency ranks of the embedded patterns α_1, α_2 (Rank) and no. of frequent patterns (#Fre) at level 6 as T_X is varied, on Table 4 data sets at $H_{th} = 0.4$

$T_X \rightarrow$	9		11		13		15		17	
	Rank	#Fre	Rank	#Fre	Rank	#Fre	Rank	#Fre	Rank	#Fre
X1	–, –	0	–, –	0	–, –	0	–, –	0	2, 3	3
X2	–, –	0	2, 1	2	2, 1	5	2, 1	5	2, 1	5
X3	2, 1	4	2, 1	4	2, 3	6	2, 3	6	2, 3	6
Y1	–, –	0	3, 1	3	3, 1	4	3, 1	6	5, 3	13
Y2	–, –	0	2, –	2	2, 3	3	2, 3	4	2, 3	4
Y3	–, –	0	3, 2	3	3, 1	6	3, 2	6	3, 1	39
Z1	2, 1	5	2, 1	5	3, 1	6	3, 1	6	3, 1	10
Z2	1, 2	4	2, 3	6	2, 3	6	2, 3	8	2, 3	42
Z3	2, 3	4	2, 3	6	4, 5	8	4, 5	12	4, 5	135

Dash indicates the pattern(s) were not found in the output

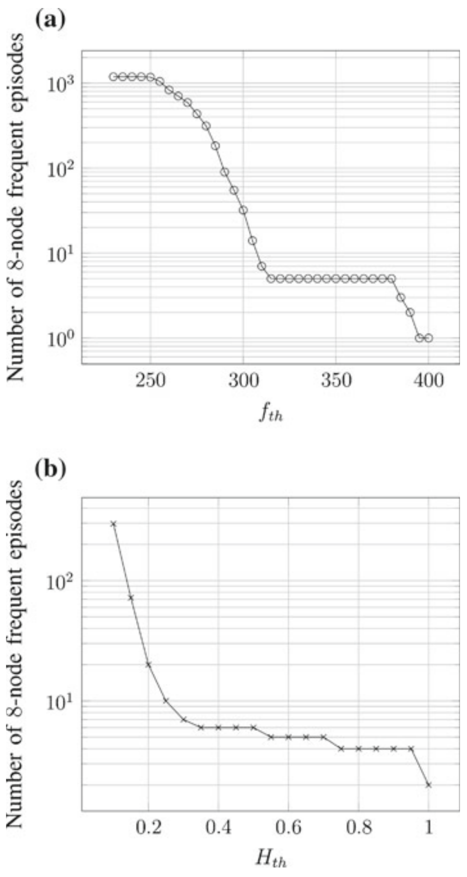


Fig. 13 Variation in number of frequent episodes as a function of frequency and bi-directional evidence thresholds. Number of embedded episodes = 5 (a, c, e, f, h from Fig. 14), $\rho = 0.055$, $p = 0.07$, $\eta = 0.7$, $M = 100$, $T_X = 15$: **a** effect of H_{th} (fixing $H_{th} = 0.75$), **b** effect of f_{th} (fixing $f_{th} = 360$)

the expected spans in these sets exceeds T_X. (In Z sets, the higher η parameter keeps the expected span to less than T_X.) Of the nine data sets, the patterns are weakest in X1 (since both η and p are lowest); our algorithms do not find the embedded patterns except at a large expiry time of $T_X = 17$. These experiments show how T_X can be used as a tool to guide the pattern discovery process. Small expiry-times can be used when we are looking for strongly correlated patterns with small spans and vice-versa. Table 5 also reports the number of frequent patterns output. (We set the frequency threshold f_{th} as 80% of the expected frequency of patterns in the data.) We can see that, mostly, the number of spurious patterns reported is small (less than 4); this number increases with T_X, as also with increasing noise level ρ and also as p increases.

In the next experiment (Figs. 13a, b) we plot the number of frequent patterns reported as a function of frequency threshold f_{th} and bi-directional evidence threshold H_{th}. For these experiments, we used a more difficult data set with 5 eight-node partial orders (taken from Fig. 14). At low thresholds the numbers of patterns reported is high, but the numbers fall sharply as the thresholds increase. We observe that there is a reasonably wide range of thresholds for both frequency and bidirectional evidence within which the number of eight-node frequent patterns output is fairly constant. Any threshold in the ‘flat’ region of these curves results in a small number of output patterns containing the embedded episodes. This shows that our algorithm under reasonable values of thresholds, will unearth only the patterns present in the data.

6.3 Flexibility in candidate generation

As described in Sect. 3.4, the same algorithm (with minor modifications in the candidate generation) can be used to mine either serial episodes, parallel episodes or any sub-class of partial orders satisfying the maximal-subepisode property. To illustrate this, we generated a data stream of about 50,000 events where, in addition to the episodes α_1 and α_2 defined in Sect. 6.2, we embedded two more serial episodes and two more parallel episodes. We ran our algorithm on this data in the serial episode, parallel episode and the general modes. When run in the serial episode mode and the parallel episode mode, we recovered the two serial and the two parallel episodes respectively. In the general mode, all six embedded partial orders (along with two other episodes which were superepisodes of the embedded partial orders) were output.

Next, we generated synthetic data by embedding all the 8 partial orders of Fig. 14. Recall that L_{th} is the threshold on the length of the largest maximal path and N_{th} is the bound on number of maximal paths. We present results obtained by mining in this data under different values for L_{th} and N_{th} (Table 6). The column titled ‘*Satisfying (Fig. 14)*’ refers to the partial orders in Fig. 14 which satisfy the L_{th} and N_{th} threshold constraints in the corresponding row. We get all the embedded patterns that satisfy the L_{th}, N_{th} constraint as frequent episodes along with a few extra episodes (as seen under the column titled #Freq). From the table we see that at lower values on either L_{th} OR N_{th}, the algorithm runs faster. At higher thresholds, the run-times were almost the same as those for mining all partial orders. This is because most of the computational

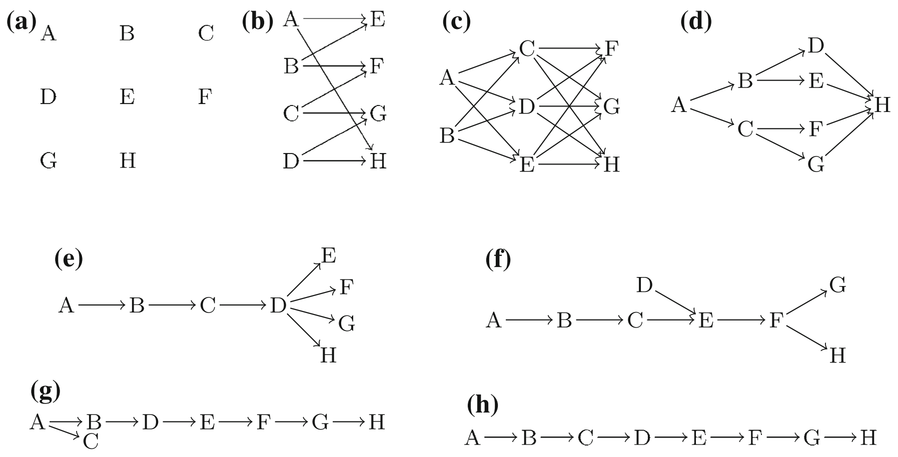


Fig. 14 Partial order episodes used for embedding in the data streams: **a** $L_m = 0, N_m = 8$ (parallel episode), **b** $L_m = 1, N_m = 8$, **c** $L_m = 2, N_m = 18$, **d** $L_m = 3, N_m = 4$, **e** $L_m = 4, N_m = 4$, **f** $L_m = 5, N_m = 4$, **g** $L_m = 6, N_m = 2$, **h** $L_m = 7, N_m = 1$ (serial episode)

Table 6 Results obtained when mining with various values for L_{th} and N_{th} simultaneously

$\rho = 0.045$, $p = 0.055$, $\eta = 0.7$, $M = 100$, $f_{th} = 300$, $H_{th} = 0.35$, $T_X = 15$

L_{th}	N_{th}	Satisfying (Fig. 14)	#Freq	Run-time
0	10	a	1	6 m 29 s
2	10	a, b	3	9 m 48 s
3	3	None	0	9 m 27 s
5	4	d, e, f	5	9 m 45 s
6	2	g	2	2 m 57 s
7	1	h	1	53 s
7	6	d–h	10	9 m 55 s
7	18	a–h	13	10 m 0 s

Table 7 Run-time as noise level is increased by varying ρ

ρ	Noise level (L_{ns})	Run-time	Avg. #Freq	Avg. #FN	Avg. #FP
0.005	0.43	3 s	2	0	0
0.02	0.75	6 s	2	0	0
0.03	0.82	30 s	2	0	0
0.045	0.87	1 m 45 s	2.2	0	0.2
0.05	0.885	6 m 1 s	2	0	0

Patterns embedded: c and f from Fig. 14; $p = 0.055$, $\eta = 0.7$, $M = 100$, $f_{th} = 300$, $T_X = 15$, $H_{th} = 0.35$

Table 8 Run-time as the data length is increased

T	Data length (n)	Run-time	Avg. #Freq	Avg. #FN	Avg. #FP
5,000	22,500	52 s	2.2	0	0.2
10,000	45,000	1 m 45 s	2.2	0	0.2
15,000	67,500	2 m 36 s	2	0	0
20,000	90,000	3 m 25 s	2.1	0	0.1

$f_{th}/T = 0.03$, $\rho = 0.045$, rest same as Table 7

burden is due to large number of candidates at levels 2 and 3, and the candidates at these lower levels are not reduced if the thresholds L_{th} and N_{th} are high.

6.4 Scaling properties of the algorithm

The algorithm scales well with number of embedded patterns, data length and noise level. In Tables 7, 8 and 9 the data is generated with different eight-node episodes embedded from Fig. 14. The run-times given are average values obtained over ten different runs. In these tables, the column titled Avg. #Freq gives the number of frequent episodes at level 8 averaged over the 10 runs. Column titled Avg. #FN denotes the number of embedded patterns missed by the algorithm, averaged over ten trials. Avg. #FP denotes the average (over 10 runs) number of false positives, i.e., the number of non-embedded frequent patterns at level 8. Table 7 describes increase in run-times

Table 9 Run-time as the number of embedded patterns is increased

N_{emb}	Patterns (Fig. 14) $[a\ b\ c\ d\ e\ f\ g\ h]$	Run-time	Avg. #Freq	Avg. #FN	Avg. #FP
0	[0 0 0 0 0 0 0 0]	8 m 42 s	0	0	0
2	[0 0 1 0 0 1 0 0]	10 m 27 s	2.2	0	0.2
5	[1 0 1 0 1 1 0 1]	12 m 5 s	6.2	0	1.2
8	[1 1 1 1 1 1 1 1]	14 m 17 s	13.4	0	5.4
10	[1 1 2 1 1 2 1 1]	17 m 31 s	18.7	0	8.7
12	[2 1 2 1 2 2 1 1]	17 m 52 s	19.6	0	7.6
15	[2 1 2 2 2 2 2 2]	18 m 17 s	29.7	0	14.7
18	[3 2 3 2 2 2 2 2]	19 m 5 s	34.4	0.1	16.5
20	[3 2 3 2 3 3 2 2]	18 m 33 s	34.6	0	14.6

$\rho = 0.04$, $p = 0.05$, $\eta = 0.7$, $M = 200$, $f_{th} = 300$, $T_X = 15$, $H_{th} = 0.3$

with noise level L_{ns}, which is the ratio of the number of noise events to the total number of events in the data. Similarly, Table 8 describes the run-time variations with data length. We observe that the run-times increase almost linearly with data length. As the data length is increased, the ratio of f_{th}/T is kept constant, where T denotes the number of time ticks up to which we carry out the simulation. Table 9 shows the run-time variations with the number of embedded partial orders (N_{emb}). We use one or more patterns from the set of eight patterns given in Fig. 14. The patterns embedded in a given experiment is represented as a vector of eight components, where each component corresponds to a particular structure in Fig. 14 and the value of the component corresponds to the number of patterns of that structure. For example, if this vector is [2 1 0 0 0 0 0 0], then it means 2 eight-node episodes of the structure in Fig. 14 (basically parallel episodes) and 1 eight-node episode of the structure in Fig. 14 are used for embedding and each of these 3 embedded episodes have no event-types in common. We observe that the algorithm scales reasonably well with the density of the embedded patterns. The increase in run-times with the number of embedded patterns is because of increased number of candidates. We observe that the false negatives are almost negligible. We also infer from the Avg. #FP (false positives) column that there is no blow-up in the number of non-embedded patterns reported, even though it roughly increases with the number of embedded patterns.

6.5 Application of partial order mining to multi-neuronal spike train data

In this section we illustrate the utility of partial order mining for multi-neuronal spike train data analysis. As explained in Sect. 1.2, the data consists of a time-series of spikes (or so called action potentials) recorded simultaneously from a group of potentially interacting neurons. This contains spikes due to the spontaneous activities of individual neurons as well as spikes due to coordinated action by a group of neurons that are functionally interconnected. The data mining approach based on serial and parallel episodes has been explored for this problem (see Patnaik et al. (2008) and Sastry and Unnikrishnan (2010) for details). Here we show utility of our partial order

Table 10 Unearthing patterns α and β from multi-neuronal spike train data: performance of the algorithm for different values of E_s

E_s	Data length	f_β (f_α)	f_{th}	#Freq	Rank of $\alpha(\beta)$	Run-time
0.4	17,500	45 (79)	40	5	2 (1)	292 s
0.6	20,000	117 (173)	100	2	2 (1)	30 s
0.75	22,500	187 (298)	150	5	2 (1)	25 s

$M = 26$, $T_X = 10$, $H_{th} = 0.7$

mining method for directly inferring the graph patterns. We use data generated using a neuronal-spike simulator described in Sastry and Unnikrishnan (2010). In this simulator, the spiking of each neuron is modeled as an inhomogeneous Poisson process whose rate changes with time due to the inputs (spikes) received from other neurons. The background firing rate of the simulator models the spontaneous activity of individual neurons and is chosen to be about 5 Hz (which means a given neuron fires randomly every 200 ms). We keep random connections of small strength among all neurons and we can embed patterns by adding specific connections with large strength. The strength of a connection of, say, $A \rightarrow B$, is specified in terms of conditional probability of B firing after a specific delay in response to a spike from A. We refer to such a strength parameter as E_s. We generate spike train data from a network of 26 neurons using this simulator and show that our algorithms are effective in unearthing the connectivity pattern. Since, in this application, effect of one neuron on another is felt within a delay time, an expiry time constraint for an episode is very natural.

In our first experiment, we generate a data stream with 2 six-node patterns $\alpha = (A \rightarrow (BC) \rightarrow (DE) \rightarrow F)$ and $\beta = (G \rightarrow (HIJ) \rightarrow (KL))$ embedded, with each synaptic delay of 3 ms, at different values of E_s. The stream had a timespan of 100 s. On such a data we see if our partial order mining algorithm is able to infer the connectivity pattern. Table 10 shows the results obtained for different levels of strength of connectivity, E_s. The table shows that our algorithm is able to discover the underlying connectivity correctly.

In our second experiment we embed a large 11-node pattern $\phi = (A \rightarrow (BCDE) \rightarrow (FG) \rightarrow (HIJK)$, with each synaptic delay as 3 ms. The stream here also had a timespan of 100 s with about 32,500 event-types. We analyze the effect of expiry-time, T_X, on the run-times of the algorithm (cf. Table 11). It was observed that as long as the T_X is greater than the span of occurrence, the algorithm is able to discover the pattern. We also noted that the run-times of the algorithm decreases as the expiry time becomes tight. This is because for high T_X more (random) patterns become frequent resulting in an increase in number of candidates.

We ran our algorithms on in-vitro recordings from neural cultures, obtained using a multi-electrode array setup (Wagenaar et al. 2006). Neuron cells used in the culture had a synaptic time-delay of about 100 ms. In our experiments, we chose a $T_X = 500$ ms, in a bid to unearth all interesting neuronal correlations up to size 6. The algorithms were run on a data slice of about 100 s, with $M = 56$ participating neurons and a data length of about 10,000 event-types, for a variety of frequency and $H(\alpha)$ thresholds. We report the no. of serial, parallel and *general* (neither serial nor parallel) episodes at various levels and thresholds in Tables 12 and 13. As is evident from the Table 12, the

Table 11 Run-times of the algorithm for different expiry time thresholds

Pattern embedded: ϕ, $E_s = 0.9$, $f_{th} = 300$, $H_{th} = 0.9$

T_X	Run-time	Pattern found
25 ms	28 min 30 s	Yes
20 ms	11 min 35 s	Yes
15 ms	6 min 47 s	Yes
10 ms	2 min 32 s	Yes
8 ms	1 min 53 s	No

Table 12 Number of serial (Ser), parallel (Par) and general (neither serial nor parallel) episodes for different frequency thresholds, $H_{th} = 0.5$, $T_X = 0.5s$, $M = 56$

Level	$f_{th} = 250$			$f_{th} = 300$			$f_{th} = 350$		
	Ser	Par	General	Ser	Par	General	Ser	Par	General
3	65	47	250	30	20	112	6	10	65
4	25	30	926	0	8	269	0	5	70
5	0	10	1020	0	1	94	0	1	0
6	0	1	55	0	0	0	0	0	0

Table 13 Number of serial (Ser), parallel (Par) and general (neither serial nor parallel) episodes for different bidirectional evidence thresholds, $f_{th} = 250$, $T_X = 0.5s$, $M = 56$

Level	$H_{th} = 0$			$H_{th} = 0.3$		$H_{th} = 0.5$		$H_{th} = 0.8$	
	Ser	Par	General	Par	General	Par	General	Par	General
3	65	48	286	48	269	47	250	11	88
4	25	33	1378	33	1143	30	926	2	151
5	0	11	2459	11	1615	10	1020	0	15
6	0	1	84	1	68	1	55	0	0

algorithm, in addition to the serial and parallel episodes, unearths a large number of general partial order episodes having a $H(\alpha)$ greater than 0.5. This illustrates the utility of algorithms that can mine for general partial order episodes. Since a serial episode has a $H(\alpha) = 1$, variation in H_{th} doesn't affect the serial episode output for a fixed f_{th} and T_X. So in Table 13, we record only one column for serial episodes. One can also observe how the number of general partial order patterns reported, drastically reduces with increase in $H(\alpha)$ threshold. A drop in the number of interesting parallel episodes is also seen with H_{th}. At level 5, for $H_{th} = 0.8$, we only find episodes which are neither serial nor parallel. In the set of general episodes, we observed interesting partial order structures like $(A1\ B1) \rightarrow (C1\ D1\ E1)$, $(A2 \rightarrow (B2\ C2\ D2) \rightarrow E3)$. Also the number of patterns reported do not grow exponentially with the size of the episode. With this, we demonstrate that our a priori-based general partial order discovery algorithms tackles the combinatorial explosion of the pattern space and simultaneously unearths interesting patterns. An interpretation of the unearthed episodes in the multi-neuronal context is beyond the scope of this paper.

7 Discussions and conclusions

In this paper we presented a method for discovering frequent episodes with unrestricted partial orders. Episode discovery from event streams is a very useful data mining technique though all the currently available methods can discover only serial or parallel episodes. Here, we restricted our attention to injective episodes where event-types do not repeat within an episode. However we place no restrictions on the partial order. We presented a novel candidate generation algorithm for episodes with unrestricted partial orders. The candidate generation algorithm presented here is very flexible and can be used to focus the discovery process on many interesting subclasses of partial orders. In particular, our method can be easily specialized to mine only serial or only parallel episodes. Thus, the algorithm presented here can be used as a single method to discover serial episodes or parallel episodes or episodes with general partial orders. We presented a FSA based algorithm for counting non-overlapped occurrences of such injective episodes. The method is efficient and can take care of expiry-time constraints. Another important contribution of this paper is a new measure of interestingness for partial order episodes, namely, bidirectional evidence. We showed that frequency alone is not a sufficient indicator of interestingness when one considers mining episodes with general partial orders. Our bidirectional evidence is very useful in discovering the most appropriate partial orders from the data stream. We also believe that this new notion of interestingness is equally relevant for partial order mining in the sequential pattern context. The effectiveness of the data mining method is demonstrated through extensive simulations.

As noted in Remark 1, our candidate generation exploits the necessary condition that, if an ℓ-node episode is frequent then all its $(\ell - 1)$-node subepisodes are also frequent. It doesn't exploit the part of subepisode lattice structure where an ℓ-node partial order episode can have ℓ-node subepisodes as well. Thus an alternate strategy for candidate generation would be to generate an ℓ-node episode as a final candidate not only when all its $(\ell - 1)$-node subepisodes are frequent but also when all its ℓ-node subepisodes are also frequent. This would, of course, mean that we would generate multiple ℓ-node candidate sets and for each, we have one database pass to find frequent episodes. This approach has two problems as mentioned in Remark 1. One is with regard to the number of data passes needed (i.e., to extract frequent episodes of a given size, one would have to now traverse the data multiple times). This can affect the run-times severely when the data resides in the secondary memory. Also, as the size of episode increases, the subepisodes lattice (among candidates of a given size) would have many levels and hence we may need many passes over the data.

A second and equally important reason for the approach we adopted is that we use thresholds on both frequency and bidirectional evidence (which is a new measure we introduced in this paper) at each level. Based on our experiments, using a frequency threshold alone during discovery (and bidirectional evidence (BE) based threshold as a post-filter) turned out to be time-wise very inefficient because of the combinatorially explosive number of non-maximal subepisode candidates generated at the higher levels. *We note that these candidates would be generated even if we exploit the subepisode structure at the same level to generate candidates*. Even though the BE measure is not strictly anti-monotonic, we roughly argued how level-wise BE threshold was effective in not only retaining the embedded interesting patterns, but also pruning lots of uninteresting non-maximal subepisodes of the embedded pattern right from the lower levels (see the discussion in Sect. 5.3 and the simulation results presented in Sect. 6.2). The embedded pattern having a high enough frequency and BE was recovered by the algorithm mainly because its maximal subepisodes also had a high BE and frequency. The non maximal subepisodes on account of having a low BE were pruned right from the lower levels. Hence thresholds based on both frequency and BE are needed for discovery in our method. Given this, in case we change the candidate generation exploiting the subepisode structure at the same level, we would actually miss out on interesting patterns. To illustrate this, suppose we have a data stream with just non-overlapped occurrences of some four-node serial episode α. Most of its four-node subepisodes (all of which are non-maximal) even though are frequent would have a low BE. As per the modified candidate generation scheme, with both frequency and BE thresholds, none of α's four-node subepisodes would be generated as interesting (frequent and satisfying BE threshold) and hence the modified scheme would never generate α as a candidate. With all these considerations, we feel the candidate generation scheme employed here (where by an ℓ-node episode is a candidate when all its $(\ell - 1)$-node maximal subepisodes are frequent) is a good strategy even though it may be counting the frequency of some of the episodes unnecessarily.

Apart from the episodes idea, another important framework of data mining for ordered data (event) sequences is that of the sequential patterns approach (Agrawal and Srikant 1995; Casas-Garriga 2005; Pei et al. 2006; Mannila and Meek 2000). Here the data consists of a large number of event sequences (where each sequence is typically short) and we are interested in patterns (episodes) that occur *at least once* in a large fraction of all the sequences. There have been methods to mine for episodes with unrestricted partial orders in the context of sequential patterns. Mannila and Meek (2000) tries to learn a partial order which can best explain a set of sequences, where each sequence is such that a given event-type occurs at most once in the data sequence. The learning problem is posed as a maximum-likelihood estimation problem, which equivalently boils down to a search over the space of all partial orders. For computational feasibility, the search is carried out over the space of series-parallel partial orders only. Casas-Garriga (2005) proposes a two-step method to extract all frequent closed partial orders from a set of sequences. The first step involves discovering all closed patterns with a total order along with the identification of the sequences in which they occur. For this, one can use any of the existing algorithms for closed sequential patterns (with a total order) like BIDE (Wang and Han 2004). The second step involves grouping together patterns which occur in the same sequences. From each such group, one constructs a partial order in a manner that each serial pattern in the group exactly corresponds to a maximal path in the constructed partial order and vice versa. Further, it is shown that each such partial order is a frequent closed partial order. Pei et al. (2006) proposes a more efficient algorithm for the same problem, but on data where each sequence has no repeated event-types. This additional assumption helps them view the frequent closed partial order mining problem as an equivalent closed itemset mining problem by representing each sequence by its transitively closed total order graph and viewing each edge in this graph as an item. The final algorithm is an efficient version of this equivalent closed itemset mining solution, by trying to mine transitively reduced frequent partial orders directly.

In contrast to the sequential patterns approach, in the frequent episodes framework, the data consists of a single long stream of events and we are interested in the number of times an episode occurs. Consequently, the computational techniques that work well in the sequential patterns context will not be suitable for discovering frequent episodes from an event stream. The sequential patterns approach is more suitable when the data itself naturally comes as many different sequences. (For example, given the individual sequence information of a number of proteins we may want to find motifs that occur in a large fraction of them.) In data where frequent episodes framework is suitable (e.g., the neuronal spike train data considered here), it is not possible to a priori cut the sequence into many small pieces so that it is enough to find partial orders that occur at least once in each piece. We do not know beforehand where occurrences are. Also, within our framework, while different occurrences of a single frequent episode are non-overlapped with each other, the occurrences of different frequent episodes can be (and mostly will be) overlapped and hence one way of cutting the long sequence into pieces may not work for all episodes. On the other hand, if we have data as a number of short sequences (where most sequences will not have multiple occurrences of relevant patterns) then we can string all these together into a single long data stream and can still discover partial order episodes which would be frequent under the sequential patterns approach also by suitable choice of frequency threshold. Thus, the problem we consider in this paper, namely discovering of interesting episodes with unrestricted partial orders from a single long event stream, is more general and the currently available techniques are not adequate for tackling it.

In this paper we have considered the case of only injective episodes. Even though in principle, the counting ideas presented here can be extended to general episodes, our candidate generation algorithm will not work without the assumption of injective episodes as explained in Remark 2. Extending the ideas presented here to the class of all partial order episodes is an important direction in which the work reported here can be extended. In whole of frequent pattern mining, the choice of frequency thresholds is typically user-defined or arbitrary. Assessing interestingness of patterns based on sound statistical methods is a popular approach in pattern discovery. Another potential future direction based on this work is to statistically assess significance of general partial order patterns in event streams. We will address these in our future work.

References

Achar A (2010) Discovering frequent episodes with general partial orders. PhD thesis, Department of Electrical Engineering, Indian Institute of Science, Bangalore

Achar A, Laxman S, Raajay V, Sastry PS (2009) Discovering general partial orders from event streams. Technical report. arXiv:0902.1227v2 [cs.AI]. http://arxiv.org

Agrawal R, Srikant R (1995) Mining sequential patterns. In: Proceedings of the 11th international conference on data engineering, Taipei, Taiwan. IEEE Computer Society, Washington, DC

Bouqata B, Caraothers CD, Szymanski BK, Zaki MJ (2006) Vogue: a novel variable order-gap state machine for modeling sequences. In: Proceedings of the 10th European conference on principles and practice of knowledge discovery in databases, vol 4213. Springer-Verlag, Berlin, Heidelberg, pp 42–54

Brown E, Kass K, Mitra P (2004) Multiple neuronal spike train data analysis: state of art and future challenges. Nat Neurosci 7:456–461

Casas-Garriga G (2003) Discovering unbounded episodes in sequential data. In Proceedings of the 7th European conference on principles and practice of knowledge discovery in databases (PKDD'03). Cavtat-Dubvrovnik, Croatia, pp 83–94

Casas-Garriga G (2005) Summarizing sequential data with closed partial orders. In: Proceedings of 2005 SIAM international conference on data mining (SDM'05)

Diekman C, Sastry PS, Unnikrishnan KP (2009) Statistical significance of sequential firing patterns in multi-neuronal spike trains. J Neurosci Methods 182:279–284

Hätönen K, Klemettinen M, Mannila H, Ronkainen P, Toivonen H (1996) Knowledge discovery from telecommunication network alarm databases. In: Proceedings of the twelfth international conference on data engineering (ICDE '96). IEEE Computer Society, Washington, DC, pp 115–122

Iwanuma K, Takano Y, Nabeshima H (2004) On anti-monotone frequency measures for extracting sequential patterns from a single very-long sequence. In: Proceedings of the 2004 IEEE conference on cybernetics and intelligent systems, vol 1, pp 213–217

Laxman S (2006) Discovering frequent episodes: fast algorithms, connections with HMMs and generalizations. PhD thesis, Department of Electrical Engineering, Indian Institute of Science, Bangalore

Laxman S, Sastry PS, Unnikrishnan KP (2005) Discovering frequent episodes and learning Hidden Markov models: a formal connection. IEEE Trans Knowl Data Eng 17:1505–1517

Laxman S, Sastry PS, Unnikrishnan KP (2007a) A fast algorithm for finding frequent episodes in event streams. In: Proceedings of the 13th ACM SIGKDD international conference on knowledge discovery and data mining (KDD'07). San Jose, CA, 12–15 Aug, pp 410–419

Laxman S, Sastry PS, Unnikrishnan KP (2007b) Discovering frequent generalized episodes when events persist for different durations. IEEE Trans Knowl Data Eng 19:1188–1201

Laxman S, Tankasali V, White RW (2008) Stream prediction using a generative model based on frequent episodes in event sequences. In: Proceedings of the 14th ACM SIGKDD international conference on knowledge discovery and data mining (KDD'09), pp 453–461

Luo J, Bridges SM (2000) Mining fuzzy association rules and fuzzy frequent episodes for intrusion detection. Int J Intell Syst 15:687–703

Mannila H, Meek C (2000) Global partial orders from sequential data. In: Proceedings of the sixth ACM SIGKDD international conference on knowledge discovery and data mining (KDD'07). ACM, New York, pp 161–168

Mannila H, Toivonen H, Verkamo AI (1997) Discovery of frequent episodes in event sequences. Data Min Knowl Discov 1(3):259–289

Nag A, Fu AW (2003) Mining frequent episodes for relating financial events and stock trends. In: Proceedings of 7th Pacific-Asia conference on knowledge discovery and data mining (PAKDD 2003). Springer-Verlag, Berlin, pp 27–39

Patnaik D, Sastry PS, Unnikrishnan KP (2008) Inferring neuronal network connectivity from spike data: a temporal data mining approach. Sci Program 16:49–77

Pei J, Wang H, Liu J, Ke W, Wang J, Yu PS (2006) Discovering frequent closed partial orders from strings. IEEE Trans Knowl Data Eng 18:1467–1481

Sastry PS, Unnikrishnan KP (2010) Conditinal probability based significance tests for sequential patterns in multi-neuronal spike trains. Neural Comput 22(4):1025–1059

Tatti N (2009) Significance of episodes based on minimal windows. In: Proceedings of 2009 IEEE international conference on data mining

Tatti N, Cule B (2010) Mining closed strict episodes. In: Proceedings of 2010 IEEE international conference on data mining

Unnikrishnan KP, Shadid BQ, Sastry PS, Laxman S (2009) Root cause diagnostics using temporal datamining. US Patent 7509234, 24 Mar 2009

Wagenaar DA, Pine J, Potter SM (2006) An extremely rich repertoire of bursting patterns during the development of cortical cultures. BMS Neurosci

Wang J, Han J (2004) BIDE: efficient mining of frequent closed sequences. In: 20th international conference on data engineering. Boston

Wang M-F, Wu Y-C, Tsai M-F (2008) Exploiting frequent episodes in weighted suffix tree to improve intrusion detection system. In: Proceedings of the 22nd international conference on advanced information networking and applications—workshops. IEEE Computer Society, Washington, DC, pp 1246–1252