Frequent Episode Mining and Multi-Neuronal Spike Train Data Analysis

A Project Report
Submitted in partial fulfilment of the
requirements for the Degree of
Master of Engineering
in
System Science and Automation

by

V Raajay
(Under the Guidance of Prof P.S.Sastry)

Department of Electrical Engineering
Indian Institute of Science
Bangalore – 560012

June 2009

Acknowledgements

Contents

Abstract

In this report, we discuss the application of frequent episode discovery techniques in analysing multi-neuronal spike train data. Frequent episode discovery is a popular framework in temporal data mining. In the frequent episode discovery framework, the data is a sequence of events occurring at different time instants. The events are characterized by event types and are ordered according to their time of occurrence in the event sequence. An episode is defined as a collection of events types with a specified partial order. An occurrence of an episode is an ordered collection of events in the data sequence which have the same event types as in the episode and are consistent with the order as specified in the episode. There are efficient algorithms for discovering episodes with a total or null order. In this report, we study the algorithms for discovering episodes with general partial orders from event streams.

Multi neuronal data corresponds to the recording of spiking activities of neurons in a tissue. A major goal of the multi neuronal data analysis is to characterize how neurons that are part of a network interact with each other in-order to achieve higher functions. The objective of the analysis is to discover different temporal dependencies in the spikes from the neurons. Neurons in a network exhibit correlated spikings. Simultaneous firing of neurons, also called as synchronous firing, is a widely observed pattern. People have also observed what is known as " synfire " chains in multineuronal data. Here the neurons in a pattern fire in a sequence regularly. Efficient data mining algorithms for unearthing such patterns already exists in the literature. In this work, we directly look to unearth the underlying connectivity of neurons by mining for graph patterns using the frequent episode discovery frame work.

An important issue in frequent episode discovery is that of assessing statistical significance of the discovered episodes. This is particularly important in the application of spike train data analysis. The availability of statistical techniques enables us to distinguish between embedded patterns and those that occur by chance. A method to assess statistical significance of serial episodes is available. We extend the idea to the case of parallel episodes. The conventional techniques for analyzing spike train data are essentially correlation based. They are found to computationally inefficient. We

illustrate the computational advantage offered by the data mining techniques. Together with the techniques for assessing statistical significance, we project the frequent episode discovery as a useful tool for multi-neuronal data analysis.

Chapter 1

Introduction

1.1 Introduction

The field of datamining is mostly concerned with analysing large volumes of data to automatically unearth interesting information that is of value to the data owner [1, 2]. Interestingness may be defined as regularities or correlations in the data depending upon the nature of the application. There are several application areas of datamining which benefit from finding such patterns in data. A few of them are: finding customer buying pattern in market research [3], weather forecast using remote sensing data, motif discovery in protein and gene sequences etc. The regularities found in the data can be represented as association rules, clusters, and recurrent patterns in time series etc. The methods used to discover patterns in data need to be efficient in terms of both memory and time requirements, since in most of the applications one has to deal with large amounts of data.

The scope of the datamining techniques discussed in this report is restricted to *temporal datamining*. In temporal data mining the data is ordered (typically with respect to time) and the goal is to find patterns that characterize underlying temporal dependencies. In this thesis we discuss algorithms for finding certain class of temporal patterns in symbolic time series data. The project is mainly motivated by application of temporal datamining techniques for analysing multi-neuronal spike train data. This data consists of a time-ordered sequence of spikes recorded simultaneously from a number of neurons in a neural tissue. The goal is to find underlying connectivity patterns among the neurons.

This chapter is organized as follows. Section 1.2 gives a brief introduction to the field of temporal datamining. It also introduces the different frameworks in temporal data mining and defines the concept of *episodes* in event streams, which is the framework addressed in this project. Section 1.3

gives an overview of the problem of understanding the coordinated behavior of a group of neurons by analyzing the spike train data. The section introduces *multi neuronal spike train data* and motivates the use of temporal datamining techniques for its analysis. Section 1.4 gives an overview of the project work.

1.2 Temporal Datamining

Temporal datamining, a sub-field of datamining, is concerned with mining of large sequential or ordered data streams. Examples of such data are alarms in a telecommunication network, fault logs in manufacturing plants, genome data, multi-neuronal spike train recordings, etc. In many applications the data items maybe symbolic (i.e. non-numeric) and thus standard time series analysis techniques are often inadequate. In addition, temporal datamining differs from the classical time series analysis in the kind of information that one seeks to discover. The exact model parameters (e.g. co-efficients of an ARMA model or the weights of a recurrent neural network) are of little interest here. Discovering trends or patterns in data that are more readily interpretable by the user are of importance. Several efficient techniques have been proposed in temporal datamining for learning different types of patterns in ordered data stream [4]. There are mainly two popular frameworks in temporal datamining, namely, sequential patterns and episodes in event streams.

The sequential pattern discovery framework was introduced in [3]. The data here is viewed as a collection of sequences. A sequence is an ordered list of itemsets. An itemset in turn is a set of items. An example of such data is the credit card transaction data. It corresponds to the transaction details (log of items purchased) of different credit cards. The number of sequences in the data is equal to the number of cards that are monitored. Every sequence represents the transactions (ordered in time) using a single credit card. An itemset in a sequence corresponds items purchased at a single transaction. A sequence $\langle a_1 a_2 \ldots a_n \rangle$ is said to be contained in another sequence $\langle b_1 b_2 \ldots b_n \rangle$ if we can find integers $i_1 < i_2 < \cdots < i_n$ such that $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, \cdots, a_n \subseteq b_{i_n}$. Thus, if one sequence is contained in another then, each itemset of the first sequence is a subset of an itemset in the second sequence and the corresponding itemsets of second sequence are in the same order as the first sequence. A frequent sequential pattern is defined as a sequence of itemsets, which is contained in sufficiently many sequences of the database. Finding such sequential patterns can be of great use. For example, finding sequential patterns in credit card transaction data can help us in understanding customer buying patterns. These insights can be used for targeted marketing.

1.2.1 Frequent Episode Discovery

Another class of approaches to discovering temporal patterns in sequential data is the *frequent episode discovery* framework [5]. The data for frequent episode discovery is viewed as a single sequence of events, denoted by $\langle (E_1, t_1), (E_2, t_2), \ldots, (E_n, t_n) \rangle$, where n is the number of events in the data sequence. In each event, (E_i, t_i), E_i denotes the event type and t_i the time of occurrence of the event. The sequence is ordered with respect to time of occurrence of each event so that, $t_i \leq t_{i+1}$, for all $i = 1, 2, \ldots$. For example, the following is an event sequence containing ten events:

$$\langle (A,1), (B,3), (D,4), (C,6), (A,11), (E,14), (B,15), (D,17), (C,20\rangle, (A,21)\rangle \tag{1.1}$$

A, B, C, D and E are the five event types in (1.1). Fig.1.1 shows the same example graphically. The implicit assumption here is that all the events are essentially instantaneous.

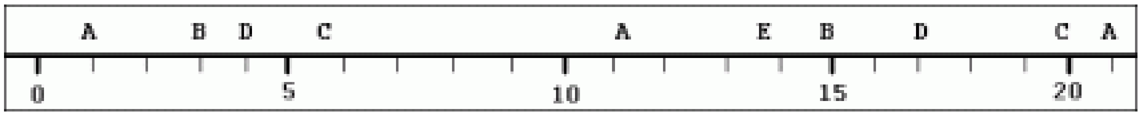


Figure 1.1: An example of event sequence

The patterns to be discovered in the event sequence are called *Episodes*. Informally, an episode is a partially ordered collection of events occurring together. These ordered collections of events may carry useful information regarding correlations among events types. Episodes can be described by directed acyclic graphs. An *episode* is said to occur in an event sequence if there are events of appropriate event types in the data sequence with a time ordering that conforms to the partial order specified by the episode. Three different types of episodes are shown in Fig.1.2. Fig.1.2 *a* shows a *serial episode*: it occurs in a data stream only if there are events of types X, Y and Z that occur in this order (though not consecutively) in the sequence. Fig.1.2 *b* shows a *parallel episode*: no constraints on the relative order of X, Y and Z are necessary. Fig.1.2 *c* shows an example of non-serial and non-parallel episode: it occurs in a sequence if events of type X and Y occur in any order followed some time later by an event of type Z.

Consider the event sequence (1.1) and a 3-node serial episode $(A \rightarrow B \rightarrow C)$. The events $\langle (A,1), (B,3), (C,6) \rangle$ constitute an occurrence of the episode $(A \rightarrow B \rightarrow C)$, while $\langle (B,15), (C,20), (A,21) \rangle$ does not. However both the sets of events are valid occurrences of the parallel episode (ABC).

An example of application of frequent episode discovery is that of analyzing fault logs in a manufacturing plant [6, 7]. In a general assembly line there will be many zones and many controllers organized in a hierarchical fashion. There will be many fault alarm reports generated by individual

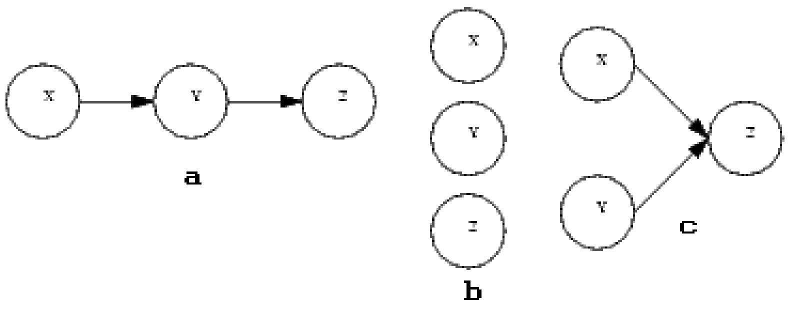


Figure 1.2: Types of episodes: a)Serial Episode, b)Parallel Episode, and c) Neither parallel nor serial Episode

controllers. In general controllers are capable of recovering from many fault situations. However when this is not possible and a manual intervention is needed to get the assembly line going again, the most recent fault alarm generated is not necessarily the actual cause for the stoppage of the assembly line. The problem is to infer the root cause. We can view the sequence of fault alarms generated as an event sequence with the event type being the fault code. Then frequent episode discovery in a proper window of the data stream can give useful information for the root cause diagnosis.

The computational problem is to find all "frequent episodes" up to a given size. An episode is said to be frequent if its frequency (some measure of the number of occurrences) exceeds an user-defined threshold. In any frequent episode discovery algorithm (or, in general, in any frequent pattern mining method) the episodes (or patterns) output by the algorithm depends on the frequency threshold used. If we use a very low threshold then many frequent episodes may correspond only to random patterns. On the other hand, if the threshold is too high, then significant temporal dependencies may not be captured by the frequent episodes discovered. Hence an important consideration is to be able to find a frequency threshold so that frequent episodes would represent *statistically significant* correlations. There has not been much literature addressing such statistical analysis if datamining methods. (See, for example, [] for a recent analysis of this kind). In this thesis we will also address, to some extent, the issue of statistical significance if the discovered episodes in the application domain of multi-neuronal spike train data analysis.

In many applications of frequent episodes method, one may need to impose certain temporal constraints on the occurrence if the episodes that are counted. One useful constraint is the so called expiry time constraint where we count only those occurrences whose span in time is below some threshold. For example, in the application of analyzing fault alarm logs, the faults that are far separated in time are not likely to be related to each other. While discussing algorithms for frequent episode discovery we will also be considering the expiry time constraint.

1.3 Analysis of Multi-Neuronal Spike Train Data

Neuroscience is concerned with the study of the structure and functioning of the nervous system. Different branches of Neuroscience focus at the different levels of organization of the nervous system, ranging from the macroscopic aspects like, learning and memory, to the microscopic level of understanding the electro-physiological and anatomical features of the cells constituting the nervous system. The current micro-level understanding of each cell (neurons) in the neuronal network is impressively detailed. What is not that well understood is how these neurons interact to form a neural circuit capable of producing different behaviours. Thus an important problem in neuroscience is to discover the interactive principles governing the organization of the neural systems. In this project we try to study the application of temporal datamining techniques to certain aspects of the problem of understanding the interactive principles governing the behavior of a group of neurons.

Neurons communicate with each other through characteristic electric pulses called action potentials or *spikes*. Hence one can study the activity of a specific neural tissue by gathering data in the form of sequences of action potentials or spikes generated by each of a group of potentially interconnected neurons. Such data is known as *multi-neuronal spike train data*.

Over the past thirty years or so, increasingly better methods are becoming available for simultaneously recording the activities of many neurons. By using techniques such as micro electrode arrays, imaging of currents, voltages, and ionic concentrations etc., spike data can be recorded simultaneously from hundreds of neurons. Vast amounts of such data is now routinely gathered from different neuronal systems. For example, in [8] the authors describe experiments where tens of cortical cultures are maintained for over five weeks and on each day the spiking activities of neurons (both with and without external stimulation) in each culture are recorded for tens of minutes. Each recording session contains data with tens of thousands of spikes. Such multi-neuronal spike train data can now be obtained *in vitro* from neuronal cultures or *in vivo* from brain slices, awake behaving animals, and even humans. Such spike train data is a mixture of the stochastic spiking activities of individual neurons as well as correlated spiking activity due to interactions or connections among neurons.

The availability of such data has resulted in development of various techniques for analysing multi-neuronal data. The computational challenge is to make reasonable inferences regarding the connectivity information or the microcircuits present in the neuronal tissue. The grand objective is to come out with a host of data processing and analysis techniques that would enable reliable inference of the underlying functional connectivity patterns which characterize the microcircuits in the neuronal systems [9].

Most of the existing techniques for analyzing spike train data essentially rely on correlations between the spike trains of different neurons. They infer the underlying functional connectivity among neurons by obtaining correlation based counts of co-occurrences of spikes from different neurons and using statistical techniques to assess the significance of these counts. A couple of such techniques are briefly discussed in Chapter 3. The techniques based on such correlation counts are computationally inefficient and suffer from combinatorial explosion when used analyze interactions involving many neurons.

One can view the spike train data as a stream of events as in the frequent episode framework. Here the event types are the name/identity of the neurons and the time of occurrences associated with the event types are the spiking times of the corresponding neurons. Frequent episodes discovery techniques can then be used to obtain temporal dependencies or patterns in the spiking times of neurons. These frequent episodes can characterize the significant connection structures among the neurons. Recently some of the frequent episode discovery techniques are shown to be efficient for unearthing functional connectivity for spike train data [10].

1.4 Overview of the Project

The objective of the project is to develop efficient temporal datamining algorithms for discovering frequent episodes of different types and to explore their utility in inferring the connectivity structure of neurons given the multi-neuronal spike train data.

As explained earlier, when multi-neuronal spike train data is viewed as an event stream the serial episodes correspond to a chain of neurons. Similarly parallel episodes (whose occurrence spans a small amount of time, as explained in Chapter 3) correspond to synchronous firing by a group of neurons. The current methods used for analysing spike train data are only aimed at finding such sequential patterns or synchronous firing patterns [11]. Currently, in temporal datamining, algorithms exist only for discovering serial and parallel episodes. Thus the current temporal datamining approaches also can discover sequential pattern of firing or synchronous firing patterns only. However, the connectivity structure among the neurons is, in general, a graph. Inferring such connectivity structure needs datamining algorithms that can discover episodes with general partial order (rather that serial or parallel episodes). In this project we developed an algorithm that can discover frequent episodes with general partial orders when we restrict the search to episodes where event types do not repeat. This new algorithm is discussed in Chapter 2

As explained earlier, an important issue in frequent episode discovery is that of assessing statistical significance of the discovered episodes. This is particularly important in the application of spike train data analysis. A method to assess statistical significance of serial episodes is available. We extend the idea to the case of parallel episodes. This is presented in Chapter 3. The chapter presents simulations to show effectiveness of assessing statistical significance of both serial and parallel episodes.

We show the effectiveness of the algorithms through simulations on synthetically generated spike trains. For this, we have built a simulator that generates spike trains for a number of neurons with embedded inter-dependencies

Chapter 2

Frequent Partial Orders

2.1 Introduction

Finding frequent episodes from event streams is an important problem in datamining. The task is to unearth all episodes whose frequency exceeds an user-defined threshold. An episode is essentially a small, partially ordered collection of nodes with each node being associated with an event type. The partial order, defines the time-ordering of nodes in the event sequence such that they constitute an occurrence of an episode. Many algorithms exist for discovering episodes from event streams. However, in all of these, separate algorithms for mining serial and parallel episodes have been proposed. Algorithms for finding frequent episodes with general partial orders have not been considered before.

In this work we consider a sub-class of episodes, called *injective* episodes. All the nodes of these episodes are mapped to distinct event types. There is no restriction on their partial order structure. We propose a single algorithm for finding such generalized episodes (includes serial and parallel episodes). Two other measures of interestingness of a partial order (apart from the frequency of occurrence) are also proposed. We use them as post-processing filters to refine the output of the algorithm. Empirical studies are done, on both synthetic and spike train data, to verify the utility of the algorithm.

This chapter is organized as follows. Section 2.2 formally defines the event sequence, episodes, frequency of episodes, etc. The counting algorithms for finding frequent episodes with and without expiry time constraint are presented in Section 1.2.1. Section 2.5 discusses in detail the candidate generation procedure. Simulation results on synthetic are presented in Section 2.7. Section 2.8 discusses the utility of partial order mining algorithm in obtaining graph patterns from Multi-Nueronal

Spike Train Data.

2.2 Episodes in event streams

The data, referred to as an *event sequence*, is denoted by $\mathbb{D} = \langle (E_1, t_1), (E_2, t_2), \ldots (E_n, t_n) \rangle$, 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 event types E_i, take values from a finite set, $\mathcal{E}$. The sequence is ordered so that, $t_i \le t_{i+1}$ for all $i = 1, 2, \ldots$. The following is an example sequence with 10 events:

$$\begin{aligned}&\langle (A,2),(B,3),(A,3),(A,7),(C,8),(B,9)\\ &(D,11),(C,12),(A,13),(B,14),(C,15)\rangle \end{aligned} \tag{2.1}$$

Definition 1 *[5] An N-node episode α, is a tuple, $(V_\alpha, <_\alpha, g_\alpha)$, where $V_\alpha = \{v_1, v_2, \ldots, v_N\}$ denotes a collection of nodes, $<_\alpha$ is a strict partial order[1] on V_α and $g_\alpha : V_\alpha \to \mathcal{E}$ is a map that associates each node in the episode with an event-type (out of the alphabet $\mathcal{E}$).*

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, episodes can be neither serial nor parallel. We denote episodes using a simple graphical notation. For example, consider a 3-node episode $\alpha = (V_\alpha, <_\alpha, g_\alpha)$, where $v_1 <_\alpha v_2$ and $v_1 <_\alpha v_3$, and with $g_\alpha(v_1) = B$, $g_\alpha(v_2) = A$ and $g_\alpha(v_3) = C$. We denote this episode as $(B \to (A\,C))$, implying that B is followed by A and C in any order.

Definition 2 *[5] 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 is a map $h : V_\alpha \to \{1, \ldots, n\}$ such that $g_\alpha(v) = E_{h(v)}$ for all $v \in V_\alpha$, and for all $v, w \in V_\alpha$ with $v <_\alpha w$ we have $t_{h(v)} < t_{h(w)}$.*

For example, $\langle (B,3),(A,7),(C,8) \rangle$ and $\langle (B,9),(C,12),(A,13) \rangle$ constitute occurrences of $(B \to (A\,C))$ in the event sequence (2.1), while $\langle (B,3),(A,3),(C,8) \rangle$ is not a valid occurrence since B does not occur *before* A.

Given any N-node episode, α, it is sometimes useful to represent an occurrence, h, of α as a vector of integers $[h(1), h(2) \ldots h(N)]$, where $h(i) < h(i+1), i = 1, \ldots, (N-1)$. For example, in sequence (2.1), the occurrence corresponding to the sub-sequence $\langle (B,3),(A,7),(C,8) \rangle$ is associated with the vector [2 4 5] (since $(B,3)$, $(A,7)$ and $(C,8)$ are the second, fourth and fifth events in (2.1) respectively).

[1] *A strict partial order is a relation which is irreflexive, asymmetric and transitive.*

Definition 3 *[5] 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 $f_{\beta\alpha} : V_\beta \to V_\alpha$ such that (i) $g_\beta(v) = g_\alpha(f_{\beta\alpha}(v))$ for all $v \in V_\beta$, and (ii) for all $v, w \in V_\beta$ with $v <_\beta w$, we have $f_{\beta\alpha}(v) <_\alpha f_{\beta\alpha}(w)$ in V_α.*

In other words, for β to be a sub-episode of α, all event-types of β must also be in α, and the order among the event-types in β must also hold in α. Thus,(AB), $(B \to A)$, $(B \to C)$ and (AC) are the 2-node subepisodes of $(B \to (AC))$. We note here that if $\beta \preceq \alpha$, then every occurrence of α contains an occurrence of β.

2.2.1 Frequency of an Episode

Given an event sequence the datamining task here is to discover all frequent episodes, i.e., those episodes whose frequencies exceed a given threshold. Frequency is some measure of how often an episode occurs in the data stream. The frequency of episodes can be defined in more than one way [5, 12]. In this paper, we consider the non-overlapped occurrences-based frequency measure for episodes [12]. Informally, two occurrences of an episode are said to be non-overlapped if no event corresponding to one occurrence appears in-between events of the other. The frequency of an episode is the size of the largest set of non-overlapped occurrences for that episode in the given data stream.

Definition 4 *[12] Consider a data stream (event sequence), $\mathbb{D}$, and an N-node episode, α. Two occurrences h_1 and h_2 of α are said to be non-overlapped in $\mathbb{D}$ if either $t_{h_1(N)} < t_{h_2(1)}$ or $t_{h_2(N)} < t_{h_1(1)}$. A set of occurrences is said to be non-overlapped if every pair of occurrences in the set is non-overlapped. The cardinality of the largest set of non-overlapped occurrences of α in $\mathbb{D}$ is referred to as the* **non-overlapped frequency** *of α in $\mathbb{D}$.*

In a data sequence,like the multi-neuronal spike train data, events widely spread out in time may not be related to each other. Chance occurrences of such spread out events should not be considered in counting the frequencies of episodes. Therefore it may be useful to have some temporal constraints on episode occurrences. One such temporal constraint is *episode expiry* constraint. This requires all the events of an episode to occur within an expiry time T_X. Span of an episode occurrence is the time from the occurrence of the first event to the last event in it. With episode expiry in place, frequency is defined as the maximum number of non-overlapped occurrences of an episode in the event stream such that the span of each occurrence is less than T_X time units.

2.2.2 Injective Episodes

In this work, we consider a sub-class of episodes called *injective episodes*. An episode, $\alpha = (V_\alpha, <_\alpha, g_\alpha)$ is said to be injective if the g_α is an injective (or 1-1) map. For example, the episode $(B \to (AC))$

is an injective episode, while $B \to (AC) \to B$ 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 is induced over X^α by $<_\alpha$ is denoted by R^α. It is often much simpler to view an injective episode, α, in terms of the *partial order set*, (X^α, R^α), that is associated with it. From now on, unless otherwise stated, when we say episode we mean an injective episode.

We will use either $(V_\alpha, <_\alpha, g_\alpha)$ or (X^α, R^α) to denote episode α, depending on the context. Although (X^α, R^α) is simpler, in some contexts, e.g., when referring to episode occurrences, the $(V_\alpha, <_\alpha, g_\alpha)$ 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, two 3-node episodes, $\alpha_1 = (V_1, <_{\alpha_1}, g_{\alpha_1})$ and $\alpha_2 = (V_2, <_{\alpha_2}, g_{\alpha_2})$, 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 $g(v_1) = B$, $g(v_2) = A$, $g(v_3) = C$, and (ii) $V_2 = \{v_1, v_2, v_3\}$ with $v_2 <_{\alpha_2} v_1$, $v_2 <_{\alpha_2} v_3$ and $g(v_1) = A$, $g(v_2) = B$ and $g(v_3) = C$. Both α_1 and α_2 represent the same pattern, and they are

indistinguishable based on their occurrences, no matter what the given data sequence is. (Notice that there is no such ambiguity in the (X^α, R^α) representation). In order 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, $\leq_\alpha$. The lexicographic ordering over $\mathcal{E}$ is only required to ensure a unique representation of injective episodes in the $(V_\alpha, <_\alpha, g_\alpha)$ notation. Referring to the earlier example involving α_1 and α_2, we will use α_2 to denote the pattern $(B \to (AC))$.

Finally, note that, 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 $R_\beta \subseteq R_\alpha$.

2.3 Frequent episode discovery

Counting frequencies of all combinatorially possible episodes is computationally very intensive and is infeasible in most applications. Hence frequent episode discovery techniques usually employ an Apriori-style [13] procedure. This is an iterative process where, in each iteration, through one pass over the data sequence, one finds frequent episodes of a given size. We start with discovering frequent 1-node episodes based on a frequency measure. The frequent 1-node episodes are then combined to obtain a set of candidate 2-node episodes. And then by counting their frequencies, we declare a subset of the candidates as frequent 2-node episodes. This process is continued till frequent episodes of all different sizes are obtained. The entire procedure is captured in *Algorithm 1* below.

However to employ such a process, the frequency count of an episode should guarantee that

Algorithm 1: Episode Discovery Algorithm

```
1 Generate an initial list of candidate episodes
2 repeat
3     Count the number of occurrences of the candidate episodes
4     Retain only those episodes whose count is greater that the frequency treshold
5     Using the list of frequent episodes, generate the next level of candidates
6 untill there are no candidate episodes remaining
7 Output all frequent episodes discovered
```

any sub-episode is at least as frequent as the episode itself. Both the non-overlapped count and non-overlapped count with expiry time constraints meet this requirement. For example, comsider the occurrence $\langle (B,3),(A,7),(C,8)\rangle$ of the episode $(B \rightarrow (A\,C))$ in the event sequence (2.1). The above occurrence also corresponds to one occurrence each of (AB) , $(B \rightarrow A)$, $(B \rightarrow C)$ and (AC). The occurrences respectively are : $(\langle (B,3),(A,7)\rangle)$,$(\langle (B,3),(A,7)\rangle)$, $(\langle (B,3),(C,8)\rangle)$ and $(\langle (A,7),(C,8)\rangle)$. From the examples it is clear that if the occurence of $(B \rightarrow (A\,C))$ satisfies the expiry time constraint then the corresponding occurrence of all the sub-episodes will also satisfy the constraint. In the following sections we present the counting and the candidate generation algorithms for mining frequent episodes with general partial orders.

2.4 Frequency Counting Algorithms

The inputs to the frequency counting algorithm(s) are a set of (candidate) episodes whose frequecies we need to count, the data sequence (in which to count the frequencies) and a frequency threshold. In case of counting occurrences under an expiry-time constraint (i.e. for *Algorithm 3* described in Sec. 2.4.3), the user also specifies an expiry-time for the episodes (i.e. the maximum time-span allowed for a valid occurrence of the episode). The counting algorithms are based on Finite State Automata.

The algorithms described in this section assume that the times of occurence of each event-type are distinct, for the ease of illustration. It is easy to incorporate the general situation of non-distint times of occurence for counting non-overlapped occurences (without any time constraints), but is a bit tedious with expiry time constraints, as the associated algorithm spawns multiple automata.

2.4.1 Finite State Automata for Partial Orders

Finite State Automata (FSA) is used to track occurrences of injective episodes under general partial orders in a manner similar to the automata-based algorithms for parallel or serial episodes [12, 14, 5]. In this section, we describe the basic construction of such automata.

We first illustrate the automaton structure through an example. Consider episode ($\alpha = (AB) \rightarrow C$). 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. 2.1. Each state, i, is associated with a pair of subsets of X^α, namely, $(\mathcal{Q}_i^\alpha, \mathcal{W}_i^\alpha)$; $\mathcal{Q}_i^\alpha \subseteq X^\alpha$ denotes the event-types that the automaton has already accepted by the time it arrives in *state i*; $\mathcal{W}_i^\alpha \subseteq X^\alpha$ denotes the event-types that the automaton in *state i* is ready to accept. Initially, the automaton is in *state 0*, has not accepted any events so far 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, we accept it and continue waiting for an A, i.e., the automaton transits to *state 2* with $\mathcal{Q}_2^\alpha = \{B\}$, $\mathcal{W}_2^\alpha = \{A\}$. At this point the automaton is not yet ready to accept a C, which happens only after both A and B are encountered (in whatever order). If, instead of encountering a B, the automaton in *state 0* first encountered an A, then it would transit into *state 1* (rather than *state 2*), where it would now wait for a B to appear (Thus, $\mathcal{Q}_1^\alpha = \{A\}$, $\mathcal{W}_1^\alpha = \{B\}$). Once both A and B appear in the data, the automaton will transit, either from *state 1* or state 2, and move into *state 3*, where it now waits for a C ($\mathcal{Q}_3^\alpha = \{A, B\}$, $\mathcal{W}_3^\alpha = \{C\}$). Finally, if the automaton now encounters a C in the data stream, it will transit to the final state, namely, *state 4* ($\mathcal{Q}_4^\alpha = \{A, B, C\}$, $\mathcal{W}_4^\alpha = \phi$) and recognize a full occurrence of the episode, $((AB) \rightarrow C)$.

In any occurence of an episode α, an event $E \in X^\alpha$ can occur only after all its parents in R^α have been seen. Hence, we initially wait for all those elements of X^α which are minimal elements of R^α. Further, we start waiting for a non-minimal element, E, of R^α immediately after all elements less than E in R^α are seen. For each $E \in X^\alpha$, we refer to the subset of elements in X^α that are less than E (with respect to R^α) as the *parents* of E in episode, α, and denote it by $\pi_\alpha(E)$. 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. 2.1, there can be no valid state corresponding to $\mathcal{Q} = \{A, C\}$ (since C could not have been accepted without B being accepted before it).

2.4.2 Counting non-overlapped occurrences

For counting the number of non-overlapped occurences of a set of serial episodes, [14] proposed an algorithm which uses one automaton per candidate episode. This algorithm can be generalized to count non-overlapped occurences of a set of injective episodes (with general partial orders) by

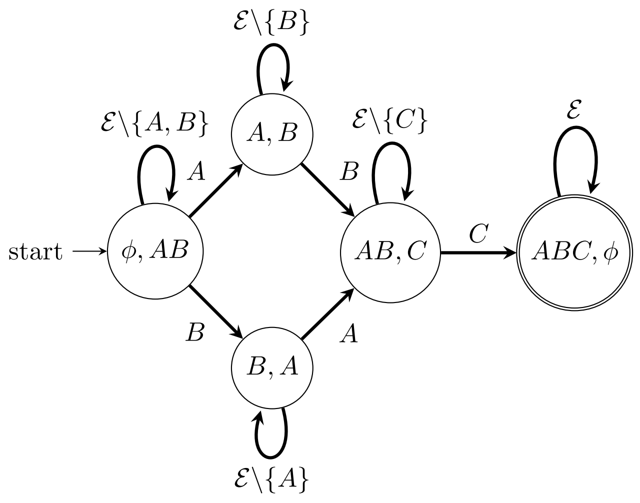

$\mathcal{E}\backslash\{B\}$
$\mathcal{E}\backslash\{A,B\}$
A,B
$\mathcal{E}\backslash\{C\}$
$\mathcal{E}$
A
B
start
ϕ,AB
AB,C
C
ABC,ϕ
B
A
B,A
$\mathcal{E}\backslash\{A\}$

Figure 2.1: Automaton for tracking occurrences of the episode $((A\,B) \rightarrow C)$.

employing the more general automata described in sub-section 2.4.1. We spawn one automaton (initially in its start state) for every candidate episode and then go down the data sequence. We allow the automata to make transitions as soon as a relevant event-type appears in the data stream to make the transition. Once an automaton reaches its final state, we increment the frequency of the corresponding episode and spawn a new automaton, which would track another occurence (if any) that is non-overlapped with the one that was just recognized. This process is continued till the end of the input data sequence. Note that, at any give time, there exists only one automata per episode.

The pseudocode for counting non-overlapped occurrences of a set of candidate injective episodes of size l is listed in *Algorithm 2*. The inputs to the algorithm are a set, $\mathcal{C}$, of candidate episodes, the event stream, $\mathbb{D}$, the alphabet, $\mathcal{E}$, and the frequency treshold, γ. The algorithm outputs the set, $\mathcal{F}_l$, of frequent episodes. The event-types associated with an l-node episode, α, are stored in the $\alpha.g[]$ array – for $i = 1, \ldots, l$, $\alpha.g[i]$ is assigned the value $g_\alpha(v_i)$. The main data structure, (as in case of serial episode counting algorithms [5, 12, 14]), is the $waits()$ list. For each event type, $E \in \mathcal{E}$, $waits(E)$ stores elements of the form (α, j), indicating that the automaton for episode α is currently waiting for an event of type E to transit into state j. The state transitions of automata are effected by making appropriate changes to the $waits()$ list. Since we are considering episodes with general partial orders, we need the partial order associated with the episode to decide which would be the next state. The main difference between the automata-based serial episode counting algorithms of [5, 12, 14] and *Algorithm 2* is that, in case of serial episodes, the order information between the event types (specified through $<_\alpha$ in the definition of α) was implicitly encoded into the $\alpha.g[]$ array, so that, for episode $(A \rightarrow B \rightarrow C)$, we would have $\alpha.g[1] = A$, $\alpha.g[2] = B$ and $\alpha.g[3] = C$. However,

Algorithm 2: CountFrequency($\mathcal{C}$, $\mathbb{D}$, γ, $\mathcal{E}$)

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

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

```
1  $\mathcal{F}_l \leftarrow \phi$;
2  foreach event type $E \in \mathcal{E}$ do $waits[E] \leftarrow \phi$;
3  foreach $\alpha \in \mathcal{C}$ do
4      $\alpha.initwaits \leftarrow \phi$, $\alpha.freq \leftarrow 0$ and $\alpha.count \leftarrow 0$;
5      for $i \leftarrow 1$ to $|\alpha|$ do
6          $j \leftarrow 1$ and $k \leftarrow 1$;  while $(j \leq |\alpha|$ and $\alpha.e[j][i] = 0)$ do $j \leftarrow j+1$ and $k \leftarrow k+1$;
7          if $k = |\alpha| + 1$ then
8              Add $(\alpha, i)$ to $waits[\alpha.g[i]]$;
9              Add $(\alpha, i)$ to $\alpha.initwaits$;
10         $\alpha.seen[i] \leftarrow$ FALSE ;
11 for $t \leftarrow 1$ to $n$ do
12     foreach $(\alpha, j) \in waits[E_t]$ do
13         $\alpha.seen[j] \leftarrow$ TRUE ;
14         $\alpha.count \leftarrow \alpha.count + 1$;
15         Remove $(\alpha, j)$ from $waits[E_t]$;
16         if $\alpha.count = |\alpha|$ then
17             $\alpha.freq \leftarrow \alpha.freq + 1$;
18             $\alpha.count \leftarrow 0$;
19             foreach $(\alpha, k) \in \alpha.initwaits$ do
20                 Add $(\alpha, k)$ to $waits[\alpha.g[k]]$;
21             for $k \leftarrow 1$ to $|\alpha|$ do $\alpha.seen[k] \leftarrow$ FALSE ;
22         else
23             for $i \leftarrow 1$ to $|\alpha|$ do
24                 if $\alpha.e[j][i] = 1$ then
25                     $flg \leftarrow$ TRUE ;
26                     for $(k \leftarrow 1$; $k \leq |\alpha|$ and $flg =$ TRUE; $k \leftarrow k+1)$ do
27                         if $\alpha.e[k][i] = 1$ and $\alpha.seen[k] =$ FALSE then
28                             $flg \leftarrow$ FALSE ;
29                     if $flg =$ TRUE then Add $(\alpha, i)$ to $waits[\alpha.g[i]]$;
30 foreach $\alpha \in \mathcal{C}$ do if $\alpha.freq > \gamma$ then Add $\alpha$ to $\mathcal{F}_l$;
31 return $\mathcal{F}_l$
```

in case of injective episodes with general partial orders, such a representation will not suffice. We store the partial order $<_\alpha$, of episode α through a binary adjacency matrix, $\alpha.e[][]$. The notation is: $\alpha.e[i][j] = 1$ if $v_i <_\alpha v_j$ for $v_i, v_j \in V_\alpha$ (or equivalently, if $(\alpha.g[i], \alpha.g[j]) \in R^\alpha$); else $\alpha.e[i][j] = 0$. Whenever an automaton for α is to be initialized, (i.e., an automaton is made to wait in startstate or state 0), the automaton waits for the *least* elements of X^α (with respect to the partial order R^α). Since we need to initialize an automaton several times as we go down the data stream, the initial *waits*() list entries are stored in $\alpha.initwaits$. The frequency of α is recorded in $\alpha.freq$ and $\alpha.count$ gives the number of state transitions the automaton has made while tracking the current occurrence (so that $\alpha.count = l$ implies that all events corresponding to all l nodes of the episode have been accepted and that the current automaton has reached its final state). Finally, the $\alpha.seen[]$ array implements $\mathcal{Q}^\alpha$, the set of event-types currently accepted by the automaton. Thus, $\alpha.seen[i] = 1$ if the automaton has so far accepted $\alpha.g[i]$ as part of the current occurrence; otherwise, $\alpha.seen[i] = 0$. *Algorithm 2*, given as pseudocode, specifies the details of the counting procedure.

Consider counting episode $\beta = (AB) \rightarrow (CD)$ in the following data stream:

$$\begin{aligned}&\langle (A,1),(B,2),(A,3)(D,4),(E,5),(C,6),(D,7),\\ &(A,8),(B,9),(B,10),(C,12),(D,14)\rangle\end{aligned} \tag{2.2}$$

Algorithm 2 tracks occurences $h_1 = \langle (A,1),(B,2),(D,4),(C,6)\rangle$ and $h_2 = \langle (A,1),(B,2),(D,4),$ $(C,6)\rangle$.

2.4.3 Counting non-overlapped occurrences with Expiry Time

Though *Algorithm 2* is efficient (it uses only one automaton per episode), it cannot implement any time constraints on occurrences of episodes. The span of an occurence is the largest difference between the times associated with any two events in the occurrence. One useful time constraint on episode occurrences is the expiry time constraint. Under this, the frequency of an episode is the maximum number of non-overlapped occurrences such that span of each occurrence is less than a user-defined threshold. (The window-width of [5] also implements a similar constraint).

Consider counting occurrences of β in sequence (2.2) with an expiry time constraint of 4. The occurence $h = \langle (B,2),\ (A,3),(D,4)(C,6)\rangle$ of β in sequence (2.2) has a span of 4 (satisfying the constraint). But *Algorithm 2* will be unable to track this occurrence. With $T_X = 4$, it would wrongly report a zero frequency for β, as the span of both the occurences tracked by it are above 4. We now describe a procedure to address this issue. Instead of spawning automata, only after the existing one reaches its final state, we spawn new automata immediately after an existing automaton accepts its first event (when it actually transits out of its start state). When counting like this, it is possible for two automata to simultaneously reach the same state. In such cases, we drop the older one and retain only the most recent automaton. This strategy tracks, in a sense, innermost occurences amongst a set of overlapping occurences that end together. For example h is the innermost occurence that ends at $(C, 6)$. For example, in (2.2), two automata reach the same state (labelled by the pair of sets $(\{A, B\}, \{C, D\})$ on seeing event $(A, 3)$. The more recent automaton among them which tracked $(B, 2)$ and $(A, 3)$ is retained and this ultimately tracks the innermost occurence h. If the span of such an innermost occurence does not exceed the expiry-time constraint, T_X, frequency is incremented and all automata for β are deleted except one in start state. This way, subsequent occurences tracked will be non-overlapped with the one just counted. On the other hand, if h failed the expiry-time constraint, we only remove the current automaton and continue looking for an innermost occurence satisfying T_X.

The pseudocode for counting non-overlapped occurrences with an expiry-time constraint is given in *Algorithm 3*. It generalizes the serial episode non-overlapped counting algorithm with expiry constraints[6]. The inputs to the algorithm are a set, $\mathcal{C}_l$, of l-node candidate episodes, the event stream, $\mathbb{D}$, the alphabet, $\mathcal{E}$, the frequency threshold, γ and T_X, the expiry-time. The algorithm outputs the set, $\mathcal{F}_l$, of frequent episodes. As earlier, the event-types associated with an l-node episode, α, are stored in the $\alpha.g[]$ array – for $i = 1, \ldots, l$, $\alpha.g[i]$ is assigned the value $g_\alpha(v_i)$. Similarly, the partial order $<_\alpha$, is stored as a binary adjacency matrix, $\alpha.e[][]$.

Unlike in *Algorithm 2*, the entries in the $waits()$ list are now tuples of the form, $(\alpha, \mathbf{q}, \mathbf{w}, j)$. This is because, we require to spawn multiple automaton for each episode α, and for every automaton $\mathcal{A}_\alpha$, we will need to keep track of its own state information. Thus, $(\alpha, \mathbf{q}, \mathbf{w}, j) \in waits(E)$ represents an automaton for α (with $\alpha.g[j] = E$) waiting in a state $(\mathcal{Q}^\alpha, \mathcal{W}^\alpha)$ where, $\mathbf{q}$ and $\mathbf{w}$ are $|X_\alpha|$-length binary vectors encoding the two sets respectively. j encodes one of the $|\mathcal{W}^\alpha|$ possible state transitions that this automaton can perform on seeing $\alpha.g[j]$. Hence, for an automaton in state $(\mathcal{Q}^\alpha, \mathcal{W}^\alpha)$, there are $|\mathcal{W}|$ tuples in the different $waits()$ lists differing only in the 4^{th} position. $\alpha.init$ list for each episode α, keeps track of the times at which the various currently active automata made their first state transition(accepts the first meaningful event type). Each entry here is a pair $(\mathbf{q}, t)$, indicating an automaton essentially initialized at time t and currently in a state with the set of accepted events represented by $\mathbf{q}$. The frequency of α is recorded in $\alpha.freq$.

Lines 3-10 initialize all the $waits()$ lists with automata waiting for the least elements of various X^αs , for every candidate $\alpha \in \mathcal{C}_l$. In the main data pass loop (lines 11-39), we process each tuple in the $waits()$ list of the current event-type, E_k. The current automaton waiting to make one of its transitions on seeing α's j^{th} event is represented as the 4-tuple $(\alpha, \mathbf{q}_{cur}, \mathbf{w}_{cur}, j)$. In line 13,

Algorithm 3: CountFrequencyExpiryTime($\mathcal{C}_l$, $\mathbb{D}$, 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 $\gamma$, set $\mathcal{E}$ of
       event types (alphabet), Expiry Time, $T_X$
Output: Set $\mathcal{F}$ of frequent episodes out of $\mathcal{C}_l$

1  $\mathcal{F}_l \leftarrow \phi$;
2  foreach event type $E \in \mathcal{E}$ do $waits[E] \leftarrow \phi$;
3  foreach $\alpha \in \mathcal{C}_l$ do
4      $\alpha.freq \leftarrow 0$, $\alpha.count \leftarrow 0$ and $\mathbf{w}_{start} \leftarrow \mathbf{0}$;
5      for $i \leftarrow 1$ to $|\alpha|$ do
6          $j \leftarrow 1$ and $k \leftarrow 1$;
7          while $(j \leq |\alpha|$ and $\alpha.e[j][i] = 0)$ do $j \leftarrow j + 1$ and $k \leftarrow k + 1$;
8          if $(k = |\alpha| + 1)$ then $\mathbf{w}_{start}[i] \leftarrow 1$;
9      for $i \leftarrow 1$ to $|\alpha|$ do
10         if $\mathbf{w}_{start}[i] = 1$ then Add $(\alpha, \mathbf{0}, \mathbf{w}_{start}, i)$ to $waits[\alpha.g[i]]$;
11 for $k \leftarrow 1$ to $n$ do
12     foreach $(\alpha, \mathbf{q}_{cur}, \mathbf{w}_{cur}, j) \in waits[E_k]$ do
13         $addwaits \leftarrow$ TRUE, $\mathbf{q}_{nxt} \leftarrow \mathbf{q}_{cur}$ and $\mathbf{q}_{nxt}[j] \leftarrow 1$;
14         if $(\mathbf{q}_{nxt}, t') \in \alpha.init$ then
15             Remove $(\mathbf{q}_{nxt}, t')$ from $\alpha.init$;
16             $addwaits \leftarrow$ FALSE;
17         if $\mathbf{q}_{cur} = \mathbf{0}$ then Add $(\mathbf{q}_{nxt}, t_k)$ to $\alpha.init$;
18         else
19             Update $(\mathbf{q}_{cur}, t_{cur})$ in $\alpha.init$ to $(\mathbf{q}_{nxt}, t_{cur})$;
20             for $i \leftarrow 1$ to $|\alpha|$ do
21                 if $\mathbf{w}_{cur}[i] = 1$ then
22                     Remove $(\alpha, \mathbf{q}_{cur}, \mathbf{w}_{cur}, i)$ from $waits[\alpha.g[i]]$
23         if $(\mathbf{q}_{nxt} = \mathbf{1}$ and $(t_k - t_{cur}) \leq T_X)$ then
24             $\alpha.freq \leftarrow \alpha.freq + 1$;
25             Empty $\alpha.init$ list;
26             for $i \leftarrow 1$ to $|\alpha|$ do
27                 foreach $(\alpha, \mathbf{q}, \mathbf{w}, i) \in waits[\alpha.g[i]]$ do
28                     if $\mathbf{q} \neq \mathbf{0}$ then Remove $(\alpha, \mathbf{q}, \mathbf{w}, i)$ from $waits[\alpha.g[i]]$;
29         if $(\mathbf{q}_{nxt} \neq \mathbf{1}$ and $addwaits =$ TRUE$)$ then
30             $\mathbf{w}_{nxt} \leftarrow \mathbf{w}_{cur}$ and $\mathbf{w}_{nxt}[j] \leftarrow 0$;
31             for $i \leftarrow 1$ to $|\alpha|$ do
32                 if $\alpha.e[j][i] = 1$ then
33                     $flg \leftarrow$ TRUE ;
34                     for $(k' \leftarrow 1; k' \leq |\alpha|$ and $flg =$ TRUE$; k' \leftarrow k' + 1)$ do
35                         if $\alpha.e[k'][i] = 1$ and $\mathbf{q}_{nxt}[k'] = 0$ then $flg \leftarrow$ FALSE ;
36                     if $flg =$ TRUE then $\mathbf{w}_{nxt}[i] \leftarrow 1$;
37             for $i \leftarrow 1$ to $|\alpha|$ do
38                 if $\mathbf{w}_{nxt}[i] = 1$ then
39                     Add $(\alpha, \mathbf{q}_{nxt}, \mathbf{w}_{nxt}, i)$ to $waits[\alpha.g[i]]$;
40 foreach $\alpha \in \mathcal{C}_l$ do if $\alpha.freq > \gamma$ then Add $\alpha$ to $\mathcal{F}_l$;
41 return $\mathcal{F}_l$
```

the next state's $\mathcal{Q}$ is computed. Immediately after transition, Lines 14-15 indicate the removal of an older automaton in the next state, if one exists. Also, if there exists an automaton in the next state, there already exist associated entries in the various waits lists corresponding to this state. Hence, under this condition we dont have to add these various entries to the appropriate lists. This information is stored in the flag variable *addwaits* which is set to FALSE here. If the current state is the start state, there is no entry corresponding to it in the $\alpha.init$ list, as it has not made its first state transition before. On seeing E_k, it performs its first state transition at the current time t_k. Hence, we add $(\mathbf{q}_{nxt}, t_k)$ as per line 17. Since an existing automata has moved out of its start state, we need to initialize a new automaton. To do this, we dont remove the various *waits*() lists entries corresponding to the start state, when $\mathbf{q}_{cur} = \mathbf{0}$. If the current automaton is not in the start state, we update the $(\mathbf{q}_{cur}, t_{cur})$ to $(\mathbf{q}_{nxt}, t_{cur})$ as $\alpha.init$ list keeps track of all the existing automata. In addition, lines 20-22 perform the removal of all *waits*() lists entries corresponding to the current state. If there didn't exist an automaton in the next state, we would need to explicitly add associated entries into different *waits*() lists. This is performed in lines 29-39 whenever *addwaits* is TRUE and the next state is not the final state. One computes $\mathbf{w}^{\alpha}_{nxt}$ in lines 30-35, based on the structure of the episode.

For those i for which $\mathbf{w}_{nxt}[i] = 1$, a tuple $(\alpha, \mathbf{q}_{nxt}, \mathbf{w}_{nxt}, i)$ is added to the appropriate list $waits(\alpha.g[i])$. Finally, if next state is final state and if the span of the occurence just tracked is less than T_X, we do the following. We increment the frequency, retire entries of all the automata except the one in the start state. Further the $\alpha.init$ list needs to be emptied. All these are performed in lines 23-28.

Space and time complexity

The number of automata that may be active (at the same time) for each episode is central to the space and time complexities of the *Algorithm 3.* The number of automata currently active for a given episode, α, is one more than the number of elements in the $\alpha.init$ list. We now show that there can be atmost l entries in the $\alpha.init$ list of *Algorithm 3.* Consider m entries in $\alpha.init$, namely, $(\mathbf{q}_1, t_{i_1}), \ldots, (\mathbf{q}_m, t_{i_m})$, such that $t_{i_1} < t_{i_2} < \cdots < t_{i_m}$. Let $\{\mathcal{Q}^{\alpha}_1, \ldots, \mathcal{Q}^{\alpha}_m\}$ represent the corresponding sets of accepted event-types for these active automata. Consider k, l such that $1 \leq k < l \leq m$. The events in the data stream that effected transitions in the l^{th}(initialized) automaton would have also been *seen* by the k^{th}(initialized) automaton. If the k^{th} automaton has not already accepted previous events with the same event-types, it will do so now on seeing these events. Hence, $\mathcal{Q}^{\alpha}_l \subsetneq \mathcal{Q}^{\alpha}_k$ for any $1 \leq k < l \leq m$. Since $\mathcal{Q}^{\alpha} \subseteq X^{\alpha}$ and $|X^{\alpha}| = l$, there are at most l (distinct) telescoping subsets of X^{α}, and so, we must have $m \leq l$.

The time required for initialization in *Algorithm 3* is $\mathcal{O}(|\mathcal{E}| + |\mathcal{C}_l|l^2)$. This is because, there are $|\mathcal{E}|$ $waits()$ lists to initialize and it takes $\mathcal{O}(l^2)$ time to find the least elements for each of the $|\mathcal{C}_l|$ episodes. For each of the n events in the stream, the corresponding $waits()$ list contains no more than $l|\mathcal{C}_l|$ elements as there can exist atmost l-automata per episode. The updates corresponding to each of these entries takes $\mathcal{O}(l^2)$ time to find the new elements to be added to the $waits()$ lists. Thus, the time complexity of the data pass is $\mathcal{O}(nl^3|\mathcal{C}_l|)$. 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, there exist $|\mathcal{W}|$ elements in various $waits()$ lists(with the same entry in the first 3 fields, but with a different j entry(4^{th} field)). Hence, for each automata we require $\mathcal{O}(l^2)$ space to store the state and its possible transitions. Since there are l such automata, the space complexity is $\mathcal{O}(l^3|C|)$.

2.5 Candidate Generation

In Sec. 2.4, we described the frequency counting algorithms for injective episodes (with general partial order constraints). In this section, we describe the candidate generation algorithm for partial order episodes. The input to the candidate generation algorithm at level $(l + 1)$, is the set, $\mathcal{F}_l$, of frequent episodes of size l. Under the non-overlapped frequency measure, we know that no episode can be more frequent than any of its subepisodes. The candidate generation step exploits this property, to construct the set, $\mathcal{C}_{l+1}$, of $(l + 1)$-node candidate episodes, given the set, $\mathcal{F}_l$, of l-node frequent episodes (which was returned by the algorithm at the previous level).

The top level algorithm for candidate generation of partial order episodes resembles the approach for parallel episodes described in [5]. Each $(l + 1)$-node candidate is generated by combining two suitable l-node frequent episodes (out of $\mathcal{F}_l$) in the following way: For every pair of l-node frequent episodes, $\alpha, \beta \in \mathcal{F}_l$, such that exactly the same $(l - 1)$-node subepisode is obtained when their respective last nodes are dropped, a *potential* $(l+1)$-node candidate, $\mathcal{Y}$, is constructed by appending the last node of β to the last node of α. If all l-node subepisodes of γ are frequent (i.e. if they can all be found in $\mathcal{F}_l$) γ is declared a candidate episode and is added to the output set, $\mathcal{C}_{l+1}$.

For ease of access and manipulation, the episodes in $\mathcal{F}_l$ are organized into *blocks*. Two episodes belong to the same block if the subepisodes obtained by dropping their respective last nodes are identical. Thus, the episodes $(A \to (BC))$ and $(A \to BD)$ would belong to the same block, while $((AB) \to D)$ would belong to a different block (since different subepisodes, namely $(A \to B)$ and (AB), are obtained on dropping the last nodes of $(A \to (BC))$ and $((AB) \to D)$). The candidate generation algorithm for parallel episodes [5] also employs a similar block structure, where each (parallel) episode is represented by a lexicographically sorted array of event-types and the set, $\mathcal{F}_l$, of frequent l-node (parallel) episodes, is maintained as a lexicographically sorted list of (parallel) episodes. For our case of general episodes also, each episode is associated with a lexicographically sorted array of event-types. (Recall from definiton of injective episodes that for episode $\alpha = (V_\alpha, \leq_\alpha, g_\alpha)$, with $V_\alpha = \{v_1, \ldots, v_N\}$, our notation is such that the sequence $(g_\alpha(v_1), \ldots, g_\alpha(v_N))$ obeys the lexicographic ordering on the alphabet $\mathcal{E}$). In *Algorithm 4*, we use the array, $\alpha.g[i] = g_\alpha(v_i),\ i = 1, \ldots, N$, to access the event-types associated with α. In addition to this array, we need to explicitly store the partial order information associated with each episode. This is done through the $\alpha.e[][]$ binary matrix, where $\alpha.e[i][j] = 1$ if $v_i \leq_\alpha v_j$, and $\alpha.e[i][j] = 0$ otherwise. Thus, there can now be several entries in $\mathcal{F}_l$ associated with the same array of event-types (i.e. with same $\alpha.g[]$ arrays), but which are representing different episodes, because they differ in their associated partial orders (i.e. they have different $\alpha.e[][]$ matrices). Within each block, episodes are sorted in lexicographic order of their respective arrays of event-types. Note that, unlike the parallel episodes case, here, the full $\mathcal{F}_l$ may not obey any lexicographic ordering. For example, the episodes $((AB) \rightarrow C))$ and $(A \rightarrow (BC))$ would both be represented by the same array of event-types, but would appear in different blocks (with, for example an episode like $((AB) \rightarrow D)$ appearing in the same block of $((AB) \rightarrow C)$, but ahead of $((AB) \rightarrow C)$ which, since it belongs to a different block, may appear later in $\mathcal{F}_l$).

Consider two frequent l-node episodes, α_1 and α_2, out of the same block of $\mathcal{F}_l$. The associated partial order set for α_1 is denoted by $(X^{\alpha_1}, R^{\alpha_1})$. Let $X^{\alpha_1} = \{x_1^{\alpha_1}, \ldots, x_l^{\alpha_1}\}$ denote the l distinct event-types in α_1, indexed in lexicographic order. In other words, we have $x_i^{\alpha_1} = g_{\alpha_1}(v_i),\ i = 1, \ldots, l$. Similarly, for the partial order set, $(X^{\alpha_2}, R^{\alpha_2})$, associated with α_2, we use $X^{\alpha_2} = \{x_1^{\alpha_2}, \ldots, x_l^{\alpha_2}\}$ to represent the event-types in α_2 (indexed in lexicographic order). Since α_1 and α_2 belong to the same block, the subepisodes obtained by dropping the last nodes of α_1 and α_2 must be identical, and this means the following two must hold: (i) $x_i^{\alpha_1} = x_i^{\alpha_2},\ i = 1, \ldots, (l-1)$, and (ii) $R^{\alpha_1}|_{(X^{\alpha_1} \setminus \{x_l^{\alpha_1}\})} = R^{\alpha_2}|_{(X^{\alpha_2} \setminus \{x_l^{\alpha_2}\})}$ i.e. the restriction of R^{α_1} to the first l nodes of α_1 is identical to the restriction of R^{α_2} to the first l nodes of α_2. It is possible that for α_1 and α_2 in the same block of $\mathcal{F}_l$, we can also have $x_l^{\alpha_1} = x_l^{\alpha_2}$, in which case, α_1 and α_2 differ only in the partial order constraints associated with their last nodes (e.g. if episodes (ABC) and $((AB) \rightarrow C)$ are both frequent, they would both belong to the same block in $\mathcal{F}_3$). Such episodes cannot be combined to generate next-size candidate(s), since, to get an $(l+1)$-node injective candidate episode, we need $(l+1)$ distinct event-types (but altogether, there are only l distinct event-types in α_1 and α_2 put-together). Thus, our candidate generation algorithm skips over episodes (in the same block) if they have exactly the same sets of event-types (i.e. if $X^{\alpha_1} = X^{\alpha_2}$).

The case of α_1 and α_2 in the same block, but with different event-types associated with their

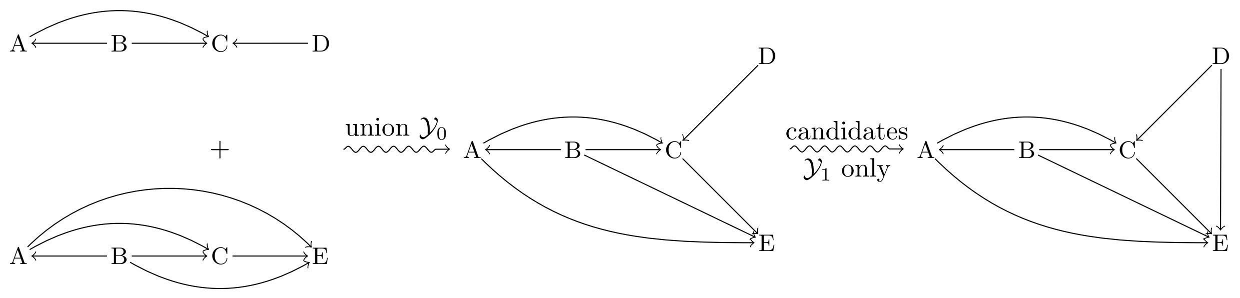


Figure 2.2: Edges (D, C) and (C, E) prevent $\mathcal{Y}_0$ and $\mathcal{Y}_2$ from coming up as candidates

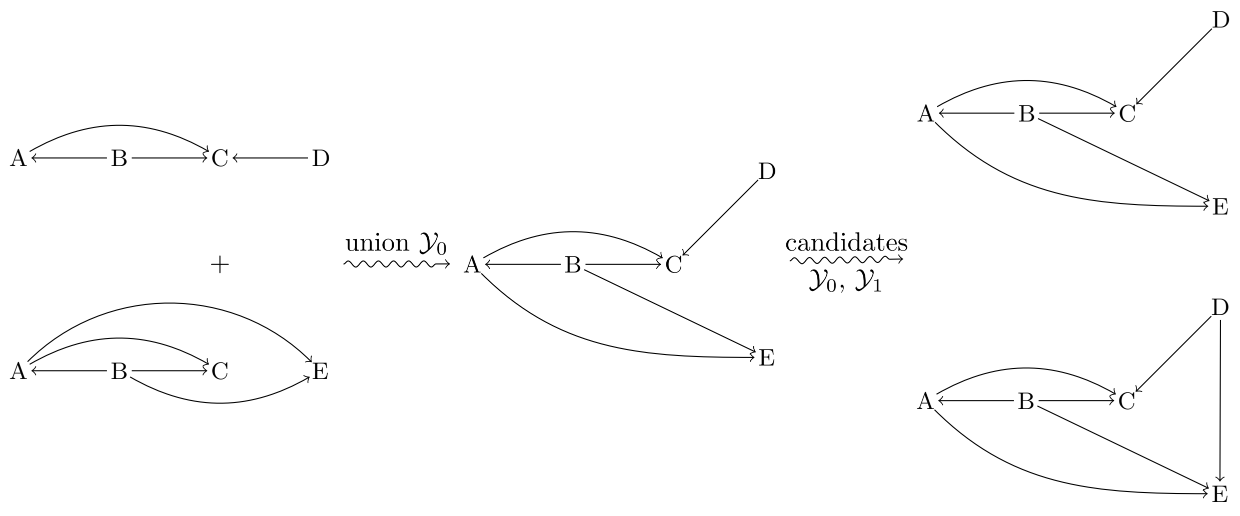


Figure 2.3: All nodes A, B and C prevent $\mathcal{Y}_2$ from coming up.

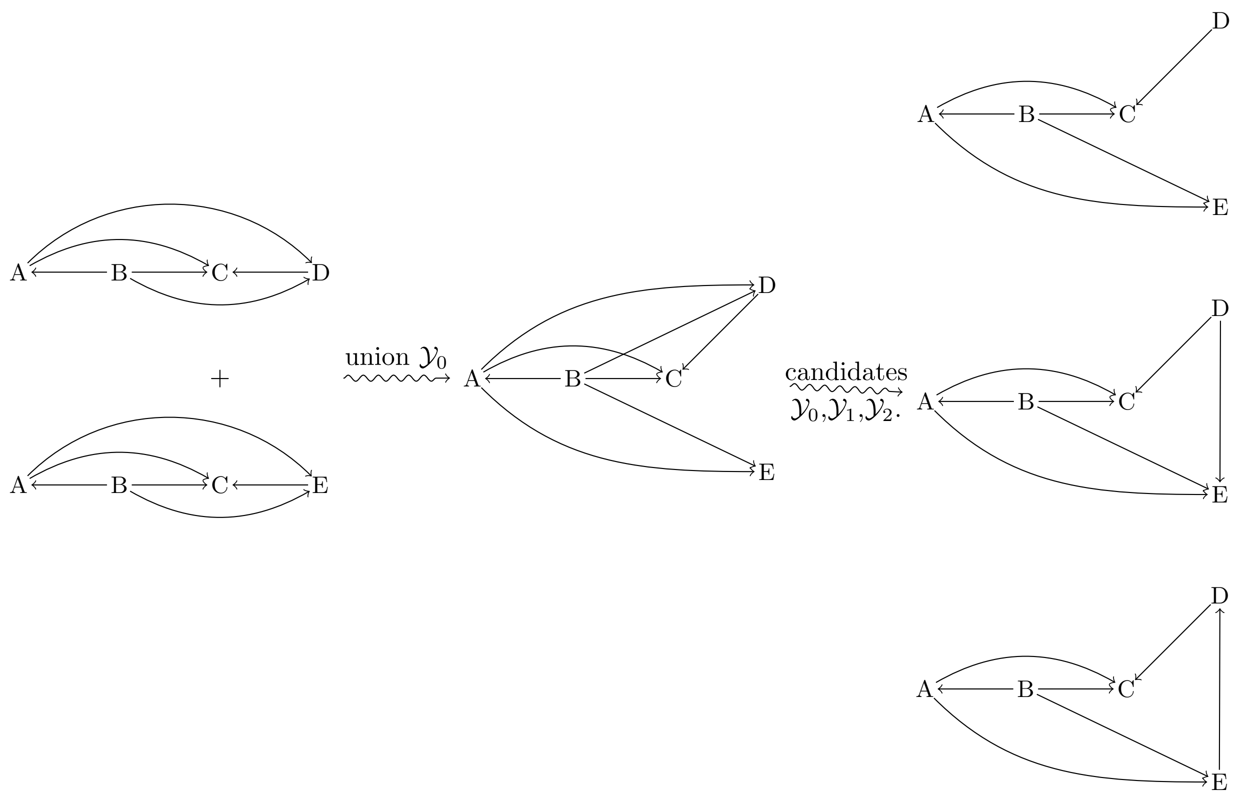


Figure 2.4: All combinations $\mathcal{Y}_0$, $\mathcal{Y}_1$ and $\mathcal{Y}_2$ come up.

last nodes, can potentially lead to a new candidate. We first illustrate the procedure through some examples. Consider the α_1 and α_2 of Fig. 2.4. We construct $\mathcal{Y}_0$ as a simple union of α_1 and α_2, i.e. we set $X^{\mathcal{Y}_0} = X^{\alpha_1} \cup X^{\alpha_2}$ and $R^{\mathcal{Y}_0} = R^{\alpha_1} \cup R^{\alpha_2}$ (so every element in the union $R^{\mathcal{Y}_0}$ belongs to either R^{α_1} or R^{α_2} or both). As it turns out, in this example, $R^{\mathcal{Y}_0}$ is a valid partial order over $X^{\mathcal{Y}_0}$ (satisfying both anti-symmetry as well as transitive closure) and hence, $\mathcal{Y}_0$ is a valid injective episode (and a potential 5-node candidate). There is no edge in $\mathcal{Y}_0$ between the last two nodes (i.e. the nodes corresponding to event-types D and E respectively). By adding an edge from D to E we get another valid partial order with the relation $R^{\mathcal{Y}_0} \cup \{(D, E)\}$, and this corresponds to a second injective candidate, $\mathcal{Y}_1$, that we can construct using the α_1 and α_2 of Fig. 2.4. Similarly, $R^{\mathcal{Y}_0} \cup \{(E, D)\}$ corresponds to a valid partial order and this gives us a third potential candidate from the same α_1 and α_2. But not all pairs, α_1 and α_2, of episodes can be combined in this manner to construct three different potential candidates. For example, for the α_1 and α_2 of Fig. 2.2, $\mathcal{Y}_1$ is the only potential candidate. While $(X^{\mathcal{Y}_1}, R^{\mathcal{Y}_1})$ obeys transitive closure, $(X^{\mathcal{Y}_0}, R^{\mathcal{Y}_0})$ is not transitively closed because (D, C) and (C, E) belong to $R^{\mathcal{Y}_0}$, but (D, E) does not. For the same reason $(X^{\mathcal{Y}_2}, R^{\mathcal{Y}_2})$ is not transitively closed either. In the example of Fig. 2.3, $\mathcal{Y}_0$ and $\mathcal{Y}_1$ are potential candidates (but $\mathcal{Y}_2$ is not a valid potential candidate because (B, E) and (E, D) are in $R^{\mathcal{Y}_2}$, while (B, D) is not).

Thus, the general strategy for combining episodes, α_1 and α_2, in the same block (but with different event-types associated with their last nodes) is as follows. Assume, without loss of generality, that α_1 precedes α_2 in the lexicographic ordering of their block in $\mathcal{F}_l$ (i.e. $x_l^{\alpha_1}$ precedes $x_l^{\alpha_2}$ as per the lexicographic ordering on $\mathcal{E}$). We will now attempt to construct an $(l + 1)$-node candidate from α_1 and α_2, by appending the last node of α_2 to the last node of α_1. There are three possibilities to consider for combining α_1 and α_2 (and these possibilities differ only in respect of the edge between the last and last-but-one nodes of the $(l + 1)$-node candidate):

$$X^{\mathcal{Y}_0} = X^{\alpha_1} \cup X^{\alpha_2} = \{x_1^{\alpha_1}, \ldots, x_l^{\alpha_1}, x_l^{\alpha_2}\} \quad , \quad R^{\mathcal{Y}_0} = R^{\alpha_1} \cup R^{\alpha_2} \tag{2.3}$$

$$X^{\mathcal{Y}_1} = X^{\alpha_1} \cup X^{\alpha_2} = \{x_1^{\alpha_1}, \ldots, x_l^{\alpha_1}, x_l^{\alpha_2}\} \quad , \quad R^{\mathcal{Y}_1} = R^{\alpha_1} \cup R^{\alpha_2} \cup \{(x_l^{\alpha_1}, x_l^{\alpha_2})\} \tag{2.4}$$

$$X^{\mathcal{Y}_2} = X^{\alpha_1} \cup X^{\alpha_2} = \{x_1^{\alpha_1}, \ldots, x_l^{\alpha_1}, x_l^{\alpha_2}\} \quad , \quad R^{\mathcal{Y}_0} = R^{\alpha_1} \cup R^{\alpha_2} \cup \{(x_l^{\alpha_2}, x_l^{\alpha_1})\} \tag{2.5}$$

In each case, if $R^{\mathcal{Y}_j}$ is a valid partial order over $X^{\mathcal{Y}_j}$, then the $(l+1)$-node (injective) episode,$\mathcal{Y}_j$, (with $(X^{\mathcal{Y}_j}, R^{\mathcal{Y}_j})$ as the associated partial order set) is considered as a *potential* candidate. (Since R^{α_1} and R^{α_2} are already valid partial orders over X^{α_1} and X^{α_2} respectively, $R^{\mathcal{Y}_j}$ is a valid partial order over $X^{\mathcal{Y}_j}$ if $(X^{\mathcal{Y}_j}, R^{\mathcal{Y}_j})$ is closed under transitivity). To check for transitive closure of $(X^{\mathcal{Y}_j}, R^{\mathcal{Y}_j})$ we would need to ensure that for every triple $z_1, z_2, z_3 \in X^{\mathcal{Y}_j}$, if $(z_1, z_2) \in R^{\mathcal{Y}_j}$ and $(z_2, z_3) \in R^{\mathcal{Y}_j}$, then we must have $(z_1, z_3) \in R^{\mathcal{Y}_j}$. However, since $R^{\mathcal{Y}_j} \subseteq (R^{\alpha_1} \cup R^{\alpha_2})$ and since $(X^{\alpha_1}, R^{\alpha_1})$ and $(X^{\alpha_2}, R^{\alpha_2})$ are already known to be transitively closed, we perform the transitivity closure check only for all size-3 subsets of $X^{\mathcal{Y}_j}$ that are of the form $\{x_l^{\alpha_1}, x_l^{\alpha_2}, x_i^{\alpha_1} \ : \ 1 \leq i \leq (l-1)\}$. Finally, if the l-node subepisodes of $\mathcal{Y}_j$, each obtained by dropping one of the $(l+1)$ nodes of $\mathcal{Y}_j$, can be found in $\mathcal{F}_l$ then $\mathcal{Y}_j$ is added to the final candidate list, $\mathcal{C}_{l+1}$, that is output by the algorithm.

The pseudocode for the candidate generation procedure, `GenerateCandidates()`, is listed in *Algorithm 4*. The input to *Algorithm 4* is a collection, $\mathcal{F}_l$, of l-node frequent episodes (where, $\mathcal{F}_l[i]$ is used to denote the i^{th} episode in the collection). The episodes in $\mathcal{F}_l$ are organized in blocks, and episodes within each block appear in lexicographic order. The output of the algorithm is the collection, $\mathcal{C}_{l+1}$, of candidate episodes of size $(l+1)$. Initially, $\mathcal{C}_{l+1}$ is empty and, if $l = 1$, all (1-node) episodes are assigned to the same block (lines 1-3, *Algorithm 4*). The main loop is over the episodes in $\mathcal{F}_l$ (starting on line 4, *Algorithm 4*). The algorithm tries to combine each episode, $\mathcal{F}_l[i]$, with episodes in the same block as $\mathcal{F}_l[i]$, but that come after it (line 6, *Algorithm 4*). In the notation used earlier to describe the procedure, we can think of $\mathcal{F}_l[i]$ as α_1 and $\mathcal{F}_l[j]$ as α_2. If $\mathcal{F}_l[i]$ and $\mathcal{F}_l[j]$ have identical event-types, we donot combine them (line 7, *Algorithm 4*). The `GetPotentialCandidates()` function, takes $\mathcal{F}_l[i]$ and $\mathcal{F}_l[j]$ as input and returns the set, $\mathcal{P}$, of *potential* candidates corresponding to them (line 8, *Algorithm 4*). (The pseudocode of the `GetPotentialCandidates()` function is listed in *Algorithm 5* and will described in the next paragraph). Now, $\mathcal{P}$ may contain either one, two or three potential candidates that were formed by combining $\mathcal{F}_l[i]$ and $\mathcal{F}_l[j]$. For each potential candidate, $\alpha \in \mathcal{P}$, we construct its l-node subepisodes (denoted β in the pseudocode) by dropping one node at-a-time from the potential candidate α (lines 13-19, *Algorithm 4*). Note that there is no need to check the case of dropping the last and last-but-one nodes of α, since they would construct the subepisodes $\mathcal{F}_l[i]$ and $\mathcal{F}_l[j]$, which are already known to be frequent. If all l-node subepisodes of α were found to be frequent, then α is added to $\mathcal{C}_{l+1}$, and the block information suitably updated (lines 20-24, *Algorithm 4*).

Finally, we describe the pseudocode of the `GetPotentialCandidates()` function listed in *Algorithm 5*. The input to *Algorithm 5* is a pair of episodes, α_1 and α_2, both of size l, and both appearing in the same block of the set, $\mathcal{F}_l$, of frequent l-node episodes. Recall that α_1 and α_2 are identical in their first $(l-1)$ nodes (in respect of both the associated event-types as well as the partial order among these event-types). The output of *Algorithm 5* is the set, $\mathcal{P}$, of potential candidates that can be constructed from α_1 and α_2. Initially, $\mathcal{P}$ is an empty set (line 1, *Algorithm 5*). The first step in the algorithm is to construct $\mathcal{Y}_0$ as a simple union of α_1 and α_2 (lines 2-9, *Algorithm 5*). This is done by assigning all the distinct event-types in α_1 and α_2 to the array, $\mathcal{Y}_0.g[]$ (lines 2-3, 5, *Algorithm 5*). Similarly, the union of partial order relations is stored in the binary matrix, $\mathcal{Y}_0.e[][]$ (lines 2, 4, 6-9,

Algorithm 4: GenerateCandidates($\mathcal{F}_l$)

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

```
Algorithm 5: GetPotentialCandidates($\alpha_1$, $\alpha_2$)
  Input: Patterns, $\alpha_1$ and $\alpha_2$, both of size $l$
  Output: $\mathcal{P}$, candidate possibilities from $\alpha_1$ and $\alpha_2$
1  Initialize $\mathcal{P} \leftarrow \phi$;
2  for $x \leftarrow 1$ to $l$ do
3      $\mathcal{Y}_0.g[x] \leftarrow \alpha_1.g[x]$;
4      for $y \leftarrow 1$ to $l$ do $\mathcal{Y}_0.e[x][y] \leftarrow \alpha_1.e[x][y]$;
5  $\mathcal{Y}_0.g[l+1] \leftarrow \alpha_2.g[l]$;
6  for $x \leftarrow 1$ to $l$ do
7      $\mathcal{Y}_0.e[x][l+1] \leftarrow \alpha_2.e[x][l]$;
8      $\mathcal{Y}_0.e[l+1][x] \leftarrow \alpha_2.e[l][x]$;
9  $\mathcal{Y}_0.e[l+1][l+1] \leftarrow 0$;
10 Copy $\mathcal{Y}_1 \leftarrow \mathcal{Y}_0$ and set $\mathcal{Y}_1.e[l][l+1] \leftarrow 1$;
11 Copy $\mathcal{Y}_2 \leftarrow \mathcal{Y}_0$ and set $\mathcal{Y}_2.e[l][l+1] \leftarrow 1$;
12 for ($j \leftarrow 0$; $j \leq 2$; $j \leftarrow j+1$) do
13     Initialize $flg \leftarrow 1$;
14     for ($i \leftarrow 1$; $i \leq l-1$ ; $i \leftarrow i+1$) do
15         if ($\mathcal{Y}_j.e[i][l] = 1$ and $\mathcal{Y}_j.e[l][l+1] = 1$ and $\mathcal{Y}_j.e[i][l+1] = 0$) then $flg \leftarrow 0$;
16         if ($\mathcal{Y}_j.e[i][l+1] = 1$ and $\mathcal{Y}_j.e[l+1][l] = 1$ and $\mathcal{Y}_j.e[i][l] = 0$) then $flg \leftarrow 0$;
17         if ($\mathcal{Y}_j.e[l][i] = 1$ and $\mathcal{Y}_j.e[i][l+1] = 1$ and $\mathcal{Y}_j.e[l][l+1] = 0$) then $flg \leftarrow 0$;
18         if ($\mathcal{Y}_j.e[l+1][i] = 1$ and $\mathcal{Y}_j.e[i][l] = 1$ and $\mathcal{Y}_j.e[l+1][l] = 0$) then $flg \leftarrow 0$;
19         if ($\mathcal{Y}_j.e[l][l+1] = 1$ and $\mathcal{Y}_j.e[l+1][i] = 1$ and $\mathcal{Y}_j.e[l][i] = 0$) then $flg \leftarrow 0$;
20         if ($\mathcal{Y}_j.e[l+1][l] = 1$ and $\mathcal{Y}_j.e[l][i] = 1$ and $\mathcal{Y}_j.e[l+1][i] = 0$) then $flg \leftarrow 0$;
21     if $flg = 1$ then Add $\mathcal{Y}_j$ to $\mathcal{P}$;
22 return $\mathcal{P}$;
```

Algorithm 5). Next $\mathcal{Y}_1$ and $\mathcal{Y}_2$ are constructed by adding an edge from the last-but-one node to the last node, and vice-versa (lines 10-11, *Algorithm 5*). After $\mathcal{Y}_0$, $\mathcal{Y}_1$ and $\mathcal{Y}_2$ are constructed, the task is to verify if each of them is transitively closed (lines 12-20, *Algorithm 5*). As mentioned earlier, this can be done efficiently, since we only need to check for transitive closure of edges involving the last and last-but-one node, along with any one other node. Every $\mathcal{Y}_j$ that passes the transitivity closure check is a potential candidates and it is added to the set, $\mathcal{P}$, that is returned by the algorithm (lines 21-22, *Algorithm 5*).

2.6 Selection of Interesting Partial Orders

The frequent episode mining method would ultimately output all frequent episodes of upto some size, say, l. However, as we see in this section, frequency alone may not be a sufficient indicator of interestingness in case of episodes with general partial orders.

Consider an l-node episode, $\alpha = (X^\alpha, R^\alpha)$. (That is $|X^\alpha| = l$). If α is frequent then all episodes $\alpha' = (X^{\alpha'}, R^{\alpha'})$ with $X^{\alpha'} = X^\alpha$ and $R^{\alpha'} \subset R^\alpha$) would also be frequent l-node episodes because every occurrence of α would constitute an occurrence of α'. Note that any such α' is a subepisode of α and hence, obviously, frequency of α' is at least as much as that of α. But the point to note is that when we consider episodes with general partial orders, an episode of size l can have subepisodes which are also of size l. Such a situation does not arise if the mining process is restricted to either serial or parallel episodes only. For example there is no 4-node serial episode that is a subepisode of $A \to B \to C \to D$. However, when considering general partial orders, given a $\alpha = (X^\alpha, R^\alpha)$ there can be, in general, exponentially many episodes $\alpha' = (X^{\alpha'}, R^{\alpha'})$ with $X^{\alpha'} = X^\alpha$ and $R^{\alpha'} \subset R^\alpha$). For example, if an N-node serial episode is frequent, then there are atleast 2^N partial orders which are also reported frequent. Essentially, drop a subset of event types of X^α out of the chain and place them in parallel with the remaining portion of the chain. This pulling out can be done in 2^N ways. For example, in the serial episode $A \to B \to C \to D \to E$, Suppose we pull out the events B and D from the chain. Then what we get is the episode $((A \to C \to E)(B\,D))$. Thus, there is an inherent combinatorial explosion in frequent episodes of a given size when we are considering general partial orders and, hence, frequency alone may not be a sufficient indicator of 'interestingness'.

2.6.1 Specificity-based filtering

One way to tackle this is to use a notion similar to that of maximal frequent patterns that has been used in other datamining contexts such as item sets or sequential patterns.

Definition 5 *An ℓ-node episode $\alpha' = (X^{\alpha'}, R^{\alpha'})$ is said to be* less specific *than ℓ-node episode*

$\alpha = (X^\alpha, R^\alpha)$ *if* $X^{\alpha'} = X^\alpha$ *and* $R^{\alpha'} \subset R^\alpha$). *Given a set of ℓ-node episodes, an episode is a most specific episode if it is not less specific than any other episode in the set. (Note that, in general, there can be many most specific episodes in a given set of episodes).*

Now, after the mining process, we can output only the most specific episodes. We call this the specificity-based filter. This prunes out many partial orders (episodes) which are presumed uninteresting because a more specific partial order (episode) is frequent. Also, given this reduced set of frequent episodes, we can generate all the episodes which are less specific and hence the full set of all frequent episodes.

2.6.2 Filtering based on bidirectional evidence

The specificity-based filter is not wholly satisfactory though it reduces the number of frequent episodes (of a given size) that are output. Suppose the data actually contains the partial order (episode) $(AB) \to C$. Suppose there are 200 occurrences of this episode of which 110 are occurrences of $A \to B \to C$ while 90 are those of $B \to A \to C$. Depending on the frequency threshold, suppose one or both of these serial episodes are also frequent. Then the specificity filter would output the serial episode(s) and suppress $(AB) \to C$. The parallel episode (ABC), being less specific, would also be frequent (and would also be supressed by the specificity filter). Suppose that in most of the occurrences of (ABC) (counted by the algorithm) C followed A and B. Now the fact that we have seen A following B roughly as often as A preceeding 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 the serial episode or the parallel episode. The specificity filter would select the serial episode as the interesting one while ranking based on frequency alone would give higher weightage to the parallel episode (because it would most probably have a few more occurrences). Thus, in addition to frequency, it would be nice to evaluate interestingness of partial orders based on whether there is evidence in the data for not constraining some of the event types. We now develop a heuristic measure for such notion of interestingness. For this we need the notion of serial extensions of a partial order.

A serial extension of a partially ordered set (X^α, R^α) is a totally ordered set (X^α, R') such that $R^\alpha \subseteq R'$. A totally ordered set (X^α, R') is said to be compatible with $\alpha = (X^\alpha, R^\alpha)$ if it is a serial extension of α.

We can say that a specific partial order pattern is interesting (or has enough evidence in data) if each of its serial extensions contribute substantially to its frequency. Actually, we do not need all serial extensions to contribute. It is enough if the partial order pattern is the most specific one with which all the contributing serial extensions are compatible with. For example, consider the episode, $((A \rightarrow B)(C\,D))$. Suppose $(A \rightarrow B \rightarrow C \rightarrow D)$ and $(D \rightarrow C \rightarrow A \rightarrow B)$ are the only serial extensions which contribute to its frequency. One can verify that $(A \rightarrow B)(C\,D))$ is the most specific partial order whose set of serial extensions(five of them) contains the two participating serial extensions. However, while counting the frequency of a partial order, it is computationally difficult to keep track of the relative contributions of each of its (relevant) serial extensions. (Note that when a partial order is a candidate episode for counting it is not necessary that each (or even any) of its serial extensions are candidates).

A simpler way to capture interestingness of a partial order episode is to demand that in the occurrences of the episode (as counted by the algorithm) 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'.

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 total number of occurrences (i.e., frequency) of α and let f^α_{ij} denote the number of these occurrences where i precedes j. Let $p^\alpha_{ij} = f^\alpha_{ij}/f^\alpha$.

It is easy to see that if all (relevant) serial extensions of α contribute non-zero amounts towards frequency of α then we would have $f^\alpha_{ij}, p^\alpha_{ij} > 0$ for all $(i,j) \in \mathcal{G}^\alpha$ and conversely. By relevant we mean that α is the most specific partial order with which all its participating serial extensions are compatible with. To rate the interestingness of the partial order episode α we define a measure that tries to capture the relative magnitudes of p^α_{ij} and p^α_{ji}. Let

$$H^\alpha_{ij} = -p^\alpha_{ij} log(p^\alpha_{ij}) \;-\; (1 - p^\alpha_{ij}) log(1 - p^\alpha_{ij}) \tag{2.6}$$

Since, in each occurrence either i preceeds j or j preceeds i, we have $p^\alpha_{ij} = 1 - p^\alpha_{ji}$ and hence H_{ij} is symmetric in i, j.

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

$$H(\alpha) = \min_{(i,j) \in \mathcal{G}^\alpha} H^\alpha_{ij} \tag{2.7}$$

We use $H(\alpha)$ as an additional interestingness measure for α. 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. Now, we say that an episode α is interesting if (i). the frequency is above a threshold, and (ii). $H(\alpha)$ is above a threshold. We can use $H(\alpha)$ as post-processing filter on the output generated at the largest size of frequent episodes. We call this the post-processing filter based on bidirectional evidence.

It is also possible to use a threshold on $H(\alpha)$ at each size (or level) in our a priori style level-wise counting procedure. This can substantially contribute towards the efficiency of mining for general partial orders. However, unlike in the case of frequency threshold, it is not quite clear whether $H(\alpha)$ also posseses the so called anti-monotonicity property. The main difficulty is that $H(\alpha)$ is tied to a specific set of occurrences counted by the algorithm. However, if an episode α has a bidirectional evidence $H(\alpha) = e$, in a given set of occurences, then one can see that any subepisode of α (obtained by the restriction of R^α onto a subset of X^α) also has a bidirectional evidence of atleast e in the same set of occurences. We show through empirical studies in the next section that a threshold on $H(\alpha)$ is quite effective.

We end this section by explaining how $H(\alpha)$ can be computed during our frequency counting process. For each episode, we maintain an $l \times l$ matrix $\alpha.H$ whose $(i,j)^{th}$ element would contain f_{ij}^{α} by the end of counting. $\alpha.H$ matrix is initialized to 0 just before counting. For each automata that is initialized, we initialize a separate $l \times l$ matrix of zeros stored with the automaton. which is different from the $\alpha.H$ matrix. Whenever an automata 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 $\alpha.H$. Thus, at the end of the counting $\alpha.H$ gives the f_{ij}^{α} information.

2.7 Simulation Results

This section discusses the results obtained upon testing our algorithms on synthetic data containing embedded partial orders. We demonstrate the effectiveness of the algorithms in mining frequent episodes with general partial orders. The utility of the two new measures of interestingness introduced in Section 2.6 is analyzed.

2.7.1 Synthetic Data Generation

Synthetic data is generated by embedding occurrences of partial orders (episodes) in varying levels of noise. Input to the data generator is a set of episodes,$\mathcal{A}$, that we want to embed in the data. The data generator embeds an occurrence of a partial order in the data stream by picking one of its serial extensions (uniformly at random) and by embedding it in the data interspersed with some noise. There are two noise parameters in the data generation model, ρ and η, which are referred to as between-pattern noise probability and within-pattern noise probability respectively. A counter keeps track of the current time, t. We start with $t = 1$. With probability ρ, the generator embeds a ‘noise-event’ in the data stream and with probability $(1 - \rho)$, it embeds a ‘pattern-event’ from one of the partial orders in $\mathcal{A}$. A noise-event is generated by picking an event-type uniformly from the alphabet. To embed a pattern-event, the generator first randomly picks one of the patterns, say α, out of $\mathcal{A}$ and then randomly picks one of its serial extensions (out of $\mathcal{M}(\alpha)$). The pattern-event embedded in the data is simply the first event-type in the chosen serial extension. Every time an event is generated, the time-counter is incremented by 1. At the next time instant, the generator again needs to make a choice of whether to next embed a noise-event or a pattern-event. If currently the generator has partially embedded some serial extension in the data stream, then this choice (of whether to next embed a noise-event or a pattern-event) is made with probability η (which is the within-parameter noise probability), otherwise, the choice is made with probability ρ (like we did at $t = 1$). Noise-events, as always, are generated by picking event-types randomly from the alphabet. Embedding a pattern-event depends on whether or not, currently, we have partially embedded a serial extension; if we have, then the next pattern-event is simply the next event-type from the corresponding serial extension; otherwise, we start a fresh occurrence of a partial order (which, as before, is done by randomly picking a partial order and one of its serial extensions, and then by embedding the first event-type of the selected serial extension).

2.7.2 Effectiveness of Partial Order Mining

To show the effectiveness of our algorithm in mining for frequent episodes, synthetic data is generated by embedding the two different sized patterns $\alpha = (A \rightarrow (BCD) \rightarrow E)$ and $\beta = (F \rightarrow ((G \rightarrow (IJ))\,(L \rightarrow K)))$ in the event stream. Tables 2.1-2.2 present the results obtained (using *Algorithm 2* and *Algorithm 3* respectively) when data is generated with $\eta = 0.5$ and ρ, is varied in the range $[0.2, 0.9]$. A fixed value of η implies that the spread of serial extension embedding is roughly the same. Higher values if ρ results in higher amount of inter-embedding noise. For *Algorithm 3*, the value of T_X is set to 12. The tables show, for various values of ρ, the ranks and relative frequencies, f_{rel}, for α and β. Rank is the position of the required episode in the frequency sorted list of episodes of same size and f_{rel} is the ratio of frequency of episode to that of the episode with highest frequency. We use the the rank and f_{rel} of the embedded patterns as measures to show the effectiveness of the algorithms. These measures do not depend upon the frequency threshold of mining. Since there exists no statistical method to directly determine the frequency threshold (f_{th}), we set the value at reasonable levels. We ensure that most of the times the pattern embedded shows up in the result.

The ranks of α and β in tables 2.1 and 2.2 are quite deceiving. We expect the embedded partial orders to be ranked high in the frequent episode list. The reason for the poor ranks is as follows. We point out that every occurrence of α in the data stream, is also an occurrence of all partial orders less-specific than α. All these partial orders have frequencies at least as much as α. The parallel episode $(ABCDE)$ is always the most frequent amongst them.

In spite of the poor ranks, high f_{rel} values show that the frequencies of the embedded patterns

Table 2.1: Effectiveness of partial order mining: Performance of *Algorithm 2* for fixed η and varying ρ. (Patterns: $\alpha = (A \to (BCD) \to E)$ and $\beta = (F \to ((G \to (IJ))\,(L \to K)))$, n = 10000 , $\eta = 0.5$, $f_{th} = 150$)

	Pattern α		Pattern β	
ρ	Rank	f_{rel}	Rank	f_{rel}
0.2	352	0.920	128	0.955
0.6	361	0.904	143	0.944
0.8	730	0.814	183	0.889
0.85	612	0.823	190	0.879
0.9	3178	0.748	2072	0.790

Table 2.2: Effectiveness of partial order mining: Performance of *Algorithm 3* for fixed η and varying ρ. (Patterns: $\alpha = (A \to (BCD) \to E)$ and $\beta = (F \to ((G \to (IJ))\,(L \to K)))$, n = 10000 , $\eta = .5$, $f_{th} = 150$, $T_X = 12$)

	Pattern α		Pattern β	
ρ	Rank	f_{rel}	Rank	f_{rel}
0.2	189	0.941	176	0.907
0.6	102	0.929	175	0.910
0.8	112	0.907	176	0.891
0.85	221	0.872	176	0.898
0.9	97	0.925	179	0.886

Table 2.3: Effectiveness of partial order mining: Performance of *Algorithm 2* for fixed ρ and varying η. (Patterns: $\alpha = (A \to (BCD) \to E)$ and $\beta = (F \to ((G \to (IJ))\,(L \to K)))$, n = 10000 , $\rho = .8$, $f_{th} = 150$, $T_X = 12$)

	Pattern α		Pattern β	
η	Rank	f_{rel}	Rank	f_{rel}
0.2	139	0.944	138	0.940
0.3	186	0.915	124	0.926
0.5	730	0.8144	183	0.889
0.6	1796	0.777	236	0.870

Table 2.4: Effectiveness of partial order mining: Performance of *Algorithm 3* for fixed ρ and varying η. (Patterns: $\alpha = (A \rightarrow (BCD) \rightarrow E)$ and $\beta = (F \rightarrow ((G \rightarrow (IJ)) (L \rightarrow K)))$, n = 10000 , $\rho = .8$, $f_{th} = 150$, $T_X = 12$)

	Pattern α		Pattern β	
η	Rank	f_{rel}	Rank	f_{rel}
0.2	51	0.991	21	0.99
0.3	91	0.993	105	0.995
0.5	112	0.907	176	0.891
0.6	112	0.876	256	0.870

(α and β) are always close to the frequencies of the most-frequent episodes (in the corresponding lists of 5-node and 6-node frequent episodes). For example, in case of $\rho = 0.2$, frequency of α was 402, while the highest frequency, that of the parallel episode, $(ABCDE)$, was marginally higher at 427. We see that for ρ varying from 0.2 to 0.9 the f_{rel} values for α and β remain fairly constant.

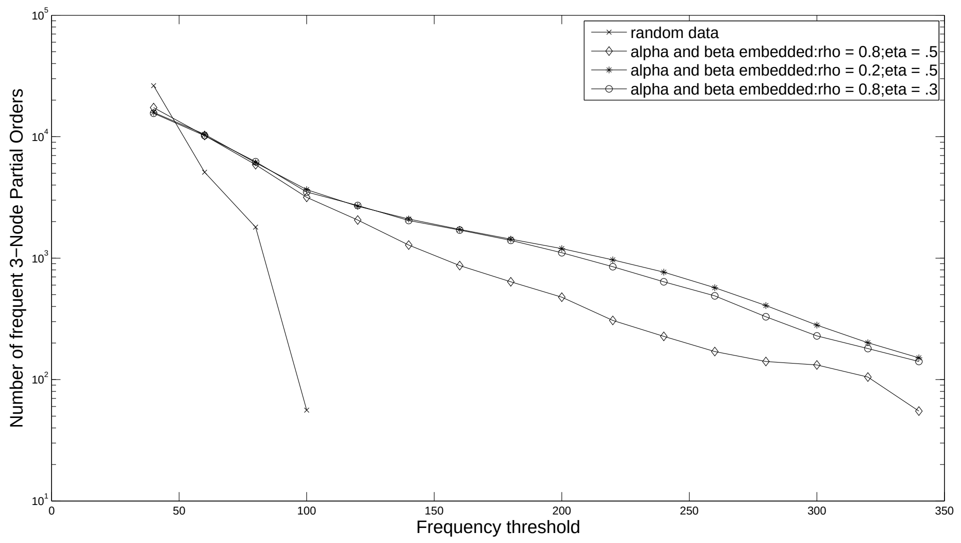
At higher noise levels ($\rho = 0.9$) performance of *Algorithm 2 is worse than* Algorithm 3. This is because, when counting without expiry-time constraints, many spurious patterns and their cyclic permutations are also found frequent. These get eliminated when suitable expiry-time constraints are imposed. Similar results were obtained when ρ was fixed and η varied (cf. Tables 2.3-2.4). Though at very high η (say 0.9) the embedded pattern was not mined because span of most of the embeddings exceeded T_X.

2.7.3 Effect of Frequency Threshold

We demonstrate the effectiveness of the algorithm by comparing the frequent partial orders discovered when the event sequence is just noise with those when some patterns were embedded. When $\rho = 1$, we see that no patterns are embedded and the entire sequence is just noise. In such a case we expect any partial order (of say 3-nodes) to be as frequent as any other partial order of 3-nodes. If the frequency threshold is increased starting from a low value, initially most of the episodes would be frequent and, after a critical threshold there wont be many frequent patterns reported. But if some patterns are embedded in the data, the 3-node sub-episodes of these patterns will have frequencies higher than the 'random' patterns and hence number of frequent episodes will not reduce drastically with increase in threshold.

Figure 2.5 plots the number of 3-node frequent partial orders versus the frequency threshold for data with no pattern (random data) and data streams with α and β embedded at different levels

Figure 2.5: Plot of Number of frequent episodes versus Frequency Threshold for data with patterns at different noise levels and entirely noise data.



of noise. All the data streams generated were of length 10,000. We can see that in the case of random data there are no 3-node partial orders with frequency greater than 100. Also, in the case of data with patterns embedded, at lower frequency thresholds, many random events were no longer frequent because the amount of noise in the data became small. That is why the graphs for data with patterns start lower on the Y-axis. The effect of noise parameters is also evident from the plots. At high noise levels the frequency of the sub episodes of embedded patterns are low. Thus at higher frequency thresholds we see that the curve for data with high noise stays well below the curves for low noise data.

2.7.4 Utility of Post Processing

Despite improvement in the ranks and f_{rel} of α and β by the incorporation of episode expiry-times (cf. Tables 2.1-2.2), the set of frequent episodes still contains several episodes whose ranks and f_{rel} are better than those for α and β. On inspecting the list of frequent episodes, we found that almost all higher frequency patterns reported are less-specific episodes with respect to α or β. We applied the specifity filter (described in Sec. 2.6.1) on the set of frequent episodes. In all cases, both the embedded patterns, α and β, appear among the top 3 frequent episodes. The results for α and β are presented in Table 2.5 and Table 2.6 respectively. Similar results were obtained when we fixed

Table 2.5: Effectiveness of Specificity Filtering: Comparison of frequency rank and f_{rel} for α before and after specificity filtering. Mining using *Algorithm 3* for fixed η and varying ρ. (Patterns: $\alpha = (A \rightarrow (BCD) \rightarrow E)$ and $\beta = (F \rightarrow ((G \rightarrow (IJ)) \ (L \rightarrow K)))$,n = 10000 , $\eta = .5$, $f_{th} = 150$, $T_X = 12$)

ρ	Rank		f_{rel}	
	Before	After	Before	After
0.2	189	1	0.9414	1
0.6	102	1	0.929	1
0.8	112	1	0.907	1
0.85	221	2	0.872	0.964
0.9	97	1	0.9248	1

Table 2.6: Effectiveness of Specificity Filtering: Comparison of frequency rank and f_{rel} for β before and after specificity filtering. Mining using *Algorithm 3* for fixed η and varying ρ. (Patterns: $\alpha = (A \rightarrow (BCD) \rightarrow E)$ and $\beta = (F \rightarrow ((G \rightarrow (IJ)) \ (L \rightarrow K)))$,n = 10000 , $\eta = .5$, $f_{th} = 150$, $T_X = 12$)

ρ	Rank		f_{rel}	
	Before	After	Before	After
0.2	176	2	0.90	0.993
0.6	175	2	0.91	0.99
0.8	176	2	0.891	0.984
0.85	176	1	0.898	1
0.9	179	3	0.886	0.993

η and varied ρ over a range (cf. Tables 2.7-2.8). In an another experiment, instead of the specificity filter, we used bidirectional-evidence (cf. Sec. 2.6.2) to filter the output set of frequent episodes discovered. The results are tabulated in Tables 2.9-2.10 for two different bidirectional-evidence thresholds, $H_{th} = 0.4$ and $H_{th} = 0.9$. For both thresholds, we see improvement in the ranks and f_{rel} of α with respect to Table 2.2. However, for β, the pattern is not found when filtering at $H_{th} = 0.9$. This is because of the inherent tree-like structure of β where L is just one level below the root, while I is two levels below (and consequently there are more serial extensions of β where L precedes I than there are the other way).

Table 2.7: Effectiveness of Specificity Filtering: Comparison of frequency rank and f_{rel} for α before and after specificity filtering. Mining using *Algorithm 3* for fixed ρ and varying η. (Patterns: $\alpha = (A \rightarrow (BCD) \rightarrow E)$ and $\beta = (F \rightarrow ((G \rightarrow (IJ))\ (L \rightarrow K)))$,n $= 10000$, $\rho = .8$, $f_{th} = 150$, $T_X = 12$)

η	Rank		f_{rel}	
	Before	After	Before	After
0.2	51	1	0.991	1
0.3	91	1	0.993	1
0.5	112	1	0.907	1
0.6	112	1	0.876	1

Table 2.8: Effectiveness of Specificity Filtering: Comparison of frequency rank and f_{rel} for β before and after specificity filtering. Mining using *Algorithm 3* for fixed ρ and varying η. (Patterns: $\alpha = (A \rightarrow (BCD) \rightarrow E)$ and $\beta = (F \rightarrow ((G \rightarrow (IJ))\ (L \rightarrow K)))$,n $= 10000$, $\rho = .8$, $f_{th} = 150$, $T_X = 12$)

η	Rank		f_{rel}	
	Before	After	Before	After
0.2	21	1	0.99	1
0.3	105	1	0.995	1
0.5	176	1	0.891	1
0.6	256	10	0.870	0.987

Table 2.9: Effectiveness of Bi-directional evidence based filtering: Improvement in frequency rank and f_{rel} for α after pruning episodes with bi-directional evidence less than H_{th}. Mining using *Algprithm 3* for fixed η and varying ρ. (Patterns: $\alpha = (A \rightarrow (BCD) \rightarrow E)$ and $\beta = (F \rightarrow ((G \rightarrow (IJ))\ (L \rightarrow K)))$,n $= 10000$, $\eta = .5$, $f_{th} = 200$, $T_X = 12$)

ρ	Rank		f_{rel}	
	$H_{th} = .4$	$H_{th} = .9$	$H_{th} = .4$	$H_{th} = .9$
0.2	11	1	0.94	1
0.6	12	1	0.93	1
0.8	22	1	0.92	1
0.85	38	1	0.87	1
0.9	13	1	0.92	1

Table 2.10: Effectiveness of Bi-directional evidence based filtering: Improvement in frequency rank and f_{rel} for β after pruning episodes with bi-directional evidence less than H_{th}. Mining using *Algorithm 3* for fixed η and varying ρ. (Patterns: $\alpha = (A \rightarrow (BCD) \rightarrow E)$ and $\beta = (F \rightarrow ((G \rightarrow (IJ))\ (L \rightarrow K)))$,n $= 10000$, $\eta = .5$, $f_{th} = 200$, $T_X = 12$)

ρ	Rank		f_{rel}	
	$H_{th} = .4$	$H_{th} = .9$	$H_{th} = .4$	$H_{th} = .9$
0.2	13	-	0.91	-
0.6	7	-	0.91	-
0.8	4	-	0.93	-
0.85	4	-	0.92	-
0.9	14	-	0.87	-

Table 2.11: Running times for mining 5,8,10 sized episodes at different values of H_{th}. (Patterns : α, γ, δ, $n = 10000$, $T_X = 15$, $f_{th} = 420$, $\rho = 0.8$)

	Mean Running time		
$H_{th} \rightarrow$	0	0.5	0.8
5-nodes	7 s	4 s	2 s
8-nodes	>2 hrs	63 s	14 s
10-nodes	>3 hrs	226 s	56 s

2.7.5 Level-wise filtering using bidirectional evidence

We now study the effectiveness of level-wise filtering based on bidirectional evidence. From the discussion in Sec. 2.6.2 we know that the number of less specific partial orders reported as frequent grows exponentially with the size of the embedded partial order(s). As a result of this, the number of candidates at higher levels and consequently the total run-times become very high. In order to curb the exponential growth in the number of candidates, after each level of counting (in Algorithm 3), we prune the candidates that have the entropy measure (H_X) less than a user-defined entropy threshold (H_{th}). We generated three different event streams, by embedding episodes of different sizes (a 5-node episode, α, an 8-node episode, $\gamma = (A \rightarrow (BCD) \rightarrow (EFG) \rightarrow H)$ and a 10-node episode, $\delta = (A \rightarrow (BCDE) \rightarrow (FGH) \rightarrow (IJ)))$. For all three data streams, we used the same ρ, but η was separately adjusted to ensure roughly the same span of occurrences for all three embedded patterns.

Table 2.11 shows the run-times comparison with level-wise filtering based on a threshold, H_{th}, on the bidirectional evidence. The values shown here are the run times obtained through an efficient

Table 2.12: Number of candidates and frequent episodes at each level of mining for various values of H_{th}. (Pattern: γ, $n = 10000$, $T_X = 15$, $f_{th} = 420$, $\rho = 0.8$, $\eta = 0.41$)

	#candidates (#frequent)		
$H_{th} \rightarrow$	0	0.5	0.8
Level 3	1064 (693)	1064 (512)	1064 (286)
Level 4	5288 (3474)	1744 (830)	330 (205)
Level 5	14556 (-)	671 (429)	85 (83)
Level 6	45418 (-)	162 (122)	34 (34)
Level 7	- (-)	25 (25)	9 (9)
Level 8	- (-)	2 (2)	1 (1)

implememtaion of the algorithms. The various runs were executed on the same machine. The column with $H_{th} = 0$ corresponds to no level-wise filtering using bidirectional evidence. The table clearly demonstrates the effectiveness of level-wise filtering in reducing the run-times.

Table 2.12 reports the number of candidates and number of frequent episodes at each level of counting and candidate generation. The data stream contains only one 8-node pattern embedded. With bi-directional filtering was not applied level-wise the algorithm, we were not able to mine beyond the 5-Node level because the number of candidates at the 6-node level was very high. The table shows the computational advantage (in terms of reducing the number of candidates) that level-wise bidirectional filtering can deliver.

2.8 Application of frequent episode discovery techniques to analyze multi-neuronal spike train data

Multi-Neuronal spike train data contains the recording of the activities of several neurons in a tissue. To understand and model the interactions of neurons in a network, neuro-biologists conduct experiments in which they record the activity of several neurons from a region in the brain, simultaneously, in response to external stimuli. These experiments are done both in-vivo (on live test subjects) and in-vitro (on cell cultures).

In-vitro experiments are conducted on networks of neurons cultured on, e.g., a petri-dish. A sophisticated tool called the Micro Electrode Array (MEA) is used for obtaining activities of groups of neurons. The neuron cells are cultured on top of a grid of micro-electrodes. Each electrode in the grid is capable of recording activity of one or more neurons around it. A typical MEA setup consists of 8×8 grid of 64 electrodes. These electrodes, besides recording cell activity, can also inject electric charge/external stimulus into the neurons.

A single electrode will have more than one neuron in its vicinity. Hence, these electrodes pick up voltage potentials generated by several neurons around them. From these voltage traces, the spike events or action potentials must be identified, the number of neurons being recorded must be determined, and each spike must be assigned to the neuron that produced it. This process is known as Spike Sorting, is an important step in multi-neuron spike data analysis. The accuracy of the spike sorting algorithm affects the accuracy of all subsequent analysis. There exists many spike sorting algorithms [9] each best suited to a particular experimental setup. These algorithms when applied to the same dataset can yield different results, illustrating the complexities of the spike-sorting problem. At present there is no consensus as to which are best.

The multi-neuron data, thus, consists of several labeled spikes occurring at different times. This data needs to be analyzed to unearth different types of interactions between neurons like finding which neurons affect which others, which group of neurons fire together or in sequence, etc. Such analysis can unearth the underlying connectivity patterns of micro circuits formed in ensembles of neurons. This will finally lead to much deeper understanding of functioning of the nervous system at the network level.

Some of the special areas of interest for such experiments have been the pre-motor and primary motor cortex which initiate and co-ordinates muscle activity; hippocampus or pre-frontal cortex responsible for cognitive tasks, long term memory etc.; and retinal ganglion cells to understand vision encoding.

2.8.1 Episodes in Multi-Neuronal Data

In this section we motivate the utility of frequent episode discovery framework for analyzing multi-neuronal data. The data obtained from MEA recordings can be seen as a single event stream where each event is a spike with a neuron label (assuming spike sorting is already done) and its time of occurrence. That is, each spike in a spike train is associated with the label of the neuron generating it and spikes from multiple spike trains can be merged into one long sequence. This is exactly the kind of data the frequent episode discovery requires.

The neurons in the cultures are connected through synapses to form circuits. The connections are such that a neuron firing can induce the firing of connected neurons. Along with such causal firings there would be random firings of neurons in the culture. Thus the information regarding the circuits in the cultures is embedded in the noise due to random firings. Frequent episodes, which are patterns that ocurr frequently in the data could very likely correspond to these circuits in the culture. The knowledge of the circuits present in the culture would lead to better understanding of more complex processes in the brain.

The neurons firing in a sequence are called synfire chains. Serial episodes are well suited to discover such synfire chains. The firing of a neuron causes the firing of a connected neuron with some delay. The knowledge of the timing order of this delay can be incorporated as an inter-event interval constraint to find only those episodes which could correspond to these synfire chains [10]. The other kind of activity present is the synchronous or co-spiking activity where in a group of neurons fire simultaneously with in a short interval of time. The parallel episodes with expiry time constraint is suitable to discover such synchronous activity [10].

Graph patterns like, $(A \rightarrow (BC) \rightarrow (DE) \rightarrow F)$, can also be mined using serial and parallel episode mining algorithm together [10]. First the parallel episode algorithm is used to mine for frequent parallel episodes with very small expiry time. In the case of this example pattern, the algorithm will return $(B\ C)$ and $(D\ E)$ are frequent. Then, at points of occurrence of the parallel episodes a new symbol is inserted to indicate an occurrence. Then, using the serial episode mining algorithm the patterns can be discovered. However this method is quite cumbersome to use. It is inefficient in terms of memory usage since we need to remember the time of occurrence of all the parallel episodes Also it does not employ a single algorithm to directly obtain frequent graph patterns.

2.8.2 Mining Graphs From Multi-Neuronal Data

In this section we illustrate the utility of partial order mining for directly obtaining graph-like patterns from multi-neuronal spike train data. We use data generated using a neuronal-spike simulator [10]. (The working details of the simulator is discussed in Appendix A .) 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 simulator keeps 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 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, the effect of one neuron on another is felt only after a time delay, an expiry time constraint for an episode is very natural.

We embed a 6-node pattern $\psi = (A \rightarrow (BC) \rightarrow (DE) \rightarrow F)$ at different values of E_s and see if our partial order mining algorithm is able to infer the connectivity pattern.

Table 2.13: Performance of the algorithm for different values of E_{strong}.(Pattern embedded:ψ,$T = 20s$,$\lambda_0 = 20Hz$,$h = 3ms$,$T_X = 10ms$)

E_{strong}	$H_{th} = 0$			$H_{th} = .7$		
	rank	f/f_{max}	Run time	rank	f/f_{max}	Run time
.9	2386	.90	3m 08s	4	.93	10s
.6	2426	.82	5m 54s	2	.97	1m 19s
.4	2463	.74	13m 7s	3	.86	6m 40s
.3	2467	.65	31m 25s	1	1	20m 59s

Table 2.14: Running times of the n-automata algorithm for different expiry time thresholds. (Pattern embedded:ϕ,$\Delta T = 1ms$, $f_{th} = 120$,$synaptic\ delay = 3$,$E_s = 0.9$,$H_{th} = .9$.)

T_X	Running Time	Pattern found
25ms	31 min 13 s	Yes
20ms	13 min 54 s	Yes
15ms	9 min 7 s	Yes
10ms	2 min 54 s	Yes
9ms	2 min 17 s	No

Table 2.13 shows the results obtained for different levels of strength of connectivity, E_s. Results are shown for both with and without level-wise filtering using bidirectional evidence (Columns $H_{th} = 0.7$ and $H_{th} = 0$ respectively). The table shows that even at low connection strengths our partial order mining algorithm is able discover the underlying connectivity correctly. Moreover, level-wise filtering based on bidirectional evidence delivers significant computational advantage.

In our second experiment we embed a large 11-node pattern $\phi = (A \rightarrow (BCDE) \rightarrow (FG) \rightarrow (HIJK)$, with each synaptic delay as 3ms. We analyze the effect of expiry-time, T_X, on the run-times of the algorithm (cf. Table 2.14). 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. It decreased from $31\,min$ at $T_X = 25ms$ to $3\,min$ at $T_X = 10ms$. This is because for high T_X more (random) patterns become frequent resulting in an increase in number of candidates.

2.9 Conclusions

In this chapter we presented algorithms for mining general partial orders from a single long stream of events. We showed how occurrences of partial orders in such data can be tracked using suitably defined finite state automata. As many as ℓ such automata may be required for each partial order whose occurrences are being tracked. Our algorithm also allows incorporation of time constraints on the span of occurrences (expiry-time constraints). We argue that frequency alone is not sufficient to measure interestingness of partial orders in data streams. To address this issue. we present two methods for filtering the output from frequent episode discovery. One is the so-called specificity filter which is based on a maximality notion for partial orders. The other filter we propose is based on what we call bidirectional evidence which attempts to capture uncertainty in the order between pairs of event-types in an episode. We validated our algorithms through extensive simulations.

Chapter 3

Statistical Significance of Frequent Episodes

3.1 Introduction

Detection of temporal firing patterns among groups of neurons is an important task as these patterns are potentially indicative of functional cell assemblies or microcircuits present in the underlying neural tissue. Several computational methods have been developed to discover repeating occurrences of such temporal patterns from multi neuronal spike train data [9, 11, 10]. These methods generally use correlation based techniques to obtain the frequency of occurrences of the patterns and are computationally not efficient. Some of the techniques only aim at finding patterns that occur at least once or twice in the data. In using the obtained patterns to infer connectivity information an important issue is that of statistical significance. Given that a pattern occurs some number of times, the question is how confident are we that the pattern is due to correlated firing of the set of neurons and is not a chance occurrence.

There have been many approaches for assessing the significance of detected firing patterns [15, 11] To assess significance, one generally employs a Null hypothesis that the different spike trains are generated by independent processes. In many cases one also assumes (possibly inhomogeneous) Bernoulli or Poisson processes. Then one can calculate the probability of observing the given number of repetitions of the pattern (or of any other statistic derived from such counts) under the null hypothesis of independent processes and hence calculate a minimum number of repetitions needed to conclude that a pattern is significant in the sense of being able to reject the null hypothesis.

There are also some empirical approaches, which may be called the *jitter* methods, suggested for assessing significance Here one creates many surrogate data streams from the experimentally observed data by perturbing (or jittering) the individual spikes while keeping certain statistics same. Then, by calculating the empirical distribution of pattern counts on the sample of surrogate data, one assesses the significance of the observed patterns. The main strength of these jitter methods is that they offer a lot of flexibility in the assumed model for the spike process of any neuron because the perturbations can be designed to preserve the required characteristics,for e.g., any assumed distribution for inter-spike intervals. Such jitter methods have been used for significance analyses by many researchers [11, 16, 17] In these jitter methods also, the implicit null hypothesis assumes independence because the spike trains of different neurons are jittered independently.

Chapter 2 presented frequent episode mining techniques that can be used to unearth graph-like patterns from spike train data. [10] introduced serial and parallel episode mining algorithms used to detect specific firing patterns like, sequential and synchronous firing patterns. These algorithms do not count the frequencies of all possible patterns. They employ the a level wise candidate generation and counting technique to curtail the combinatorial explosion in the number of patterns to be counted. At each level only patterns whose frequencies are above a user-defined threshold are reported as frequent. Such frequent patterns are then used to generate potentially frequent candidates for the next level. These algorithms have been discussed in detail in the previous chapters. However, these algorithms do not employ any statistical methods to automatically determine the frequency threshold of mining. Such efficient counting techniques together with proper methods for assessing the statistical significance of the observed counts will be very useful in analysing multi-neuronal spike train data. While there are no general techniques for assessing statistical significance for graph patterns, in this Chapter we discuss such techniques for serial and parallel episodes.

Recently, statistical techniques have been developed to assess the significance of sequential patterns with fixed inter-event delays [18]. The method is based on the serial episode mining algorithm described in [10]. It employs a compound Null hypothesis, that allows for weak dependencies among spike trains so that one can mine only for “strong” connections among neurons. The technique has been used effectively to automatically determine the frequency threshold for serial episode mining algorithm. In this chapter we extend this technique to assess the significance of parallel episodes.

Sec 3.2 discusses the statistical method used to determine the significance of sequential pattern. It also presents some simulation results to verify the effectiveness of the methods. Sec 3.3 extends the statistical technique to assess significance of parallel episodes. Sec 3.4 presents simulation results comparing the performance of parallel episode mining algorithms with a popular existing method, NeuroXidence.

3.2 Significance of Sequential Patterns

In this section we discuss how statistical significance of sequential pattern can be assessed. The material here follows [18]

3.2.1 Sequential Patterns

A pattern of ordered firing sequence is called a *sequential pattern* or a *precise firing sequence*. Symbolically, a sequential pattern is represented as $A \overset{T_1}{\rightarrow} B \overset{T_2}{\rightarrow} C$. The example pattern here, represents an ordered firing sequence of A followed by B followed by C with a delay of T_1 units between A & B and a delay of T_2 time units between B and C. The values T_1 and T_2 represent the synaptic delays between A & B and B & C. Such sequential patterns may occur frequently in the spike train data if there are strong excitatory connections between the neurons. For example, if the firing of A influences the firing of B and the firing of B in turn influences firing of C, the sequential pattern $A \overset{T_1}{\rightarrow} B \overset{T_2}{\rightarrow} C$ will be found to be frequent in the spike train data.

The frequency of a sequential pattern depends on the strength of the connections between the constituting neurons. Two neurons are said to be strongly connected if the spiking or not-spiking of one neuron strongly affects the other. In this thesis, we are concerned with only excitatory interactions among neurons. If two neurons are connected by an excitatory synapse then the spiking of one increases the chances of spiking of the other neuron after a particular time known as the synaptic delay of the connection. The strength of a connection between two neurons A and B with a synaptic delay of T time units can be represented [18] as the conditional probability,$e_s(A, B, T)$, of B firing at time $t + T$ given that A has fired at time t. The conditional probability enables us to "quantify" the strength of the interactions among neurons. For the analysis here it is assumed that the conditional probability is the same for all t and hence we use the notation $e_s(A, B, T)$ to denote it. The advantage of using such a conditional probability is that now we can specify the Null hypothesis to include many weak interactions. We say that any model of interacting neurons is in our composite null hypothesis if $e_s(x, y, T) \leq e_0 \ \forall \ x$,$y$ all delays T of interest, where e_0 is a user chosen parameter. Thee_0 here represents a level of interaction strength (in terms of conditional probability) below which we will agree to say that the interaction is 'weak'. Since e_0 is chosen by the user, our null hypothesis goes beyond the current method of analysis that assumes independence. Another important advantage of using such a probability measure is that we do not have to make any assumptions regarding the model of the spike train generation process. The strength of connection between two neurons is no longer dependent upon the parameters of any particular model.

The statistical method discussed in this section uses the threshold e_0 specified in the null hypothesis to determine the frequency threshold of the mining algorithm, so that, only patterns with strong interaction are reported as significant.

3.2.2 Serial Episode Mining

Serial episodes with inter-event time constraint are well suited to represent sequential patterns. The inter-event time constraint allows us to represent the synaptic delays between neurons. Suppose, let us say that, neuron A is connected to neuron B which, in turn, is connected to neuron C, through strong synaptic connections with respective delays T_1 and T_2. Then, in the spike train data obtained, we should be counting only those occurrences of the episode $A \rightarrow B \rightarrow C$, where the inter-event times satisfy the delay constraint. This would be counting occurrences of the sequential pattern, $A \overset{T_1}{\rightarrow} B \overset{T_2}{\rightarrow} C$.

An efficient algorithm for discovering all frequent serial episodes with inter-event time constraints is proposed in [10]. The algorithm employs a level wise procedure. A serial episode $A \rightarrow B \rightarrow C$ is counted at level 3 if and only if the patterns $A \rightarrow B$ and $B \rightarrow C$ are found to be frequent at the 2 node level. (We do not demand that $A \rightarrow C$ also be frequent.) This is justified because, if the sequential pattern $A \overset{T_1}{\rightarrow} B \overset{T_2}{\rightarrow} C$ is frequent then its sub-patterns $A \overset{T_1}{\rightarrow} B$ and $B \overset{T_2}{\rightarrow} C$ must also be frequent. The algorithm uses efficient techniques to calculate the frequencies of a set of candidate episodes through only one pass of the data. The exact details of the working of the algorithm are available in [10].

We can give an intuitive description of the algorithm as follows [18]. Suppose, we are operating at a time resolution of ΔT. (That is, the times of events or spikes are recorded to a resolution of ΔT). Then we discretize the time axis into intervals of length ΔT. For each episode whose frequency we want to find we do the following. Suppose the episode is the one mentioned above. We start with time instant 1. We check to see whether there is an occurrence of the episode starting from the current instant. For this, we need an A at that time instant and then we need a B and a C within appropriate time windows. If there are such B and C, then we take the earliest of the B and C to satisfy the time constraints, increment the frequency counter for the episode and start looking for the occurrence again starting with the next time instant (after C). On the other hand, if we can not find such an occurrence (either because A does not occur at the current time instant or because there are no B or C at appropriate times following A), then we move by one time instant and start the search again. Such a counting process gives the total number of non-overlapped occurrences (cf 2.2.1) of a serial episode with inter-event time constraints.

By modeling the above described process,we can obtain a bound on the probability that the frequency of an episode is above a given value, under null hypothesis.

3.2.3 Modeling the Counting of Non-overlapped Occurrences

We explain the method with reference to a 2-node episode $A \to B$. Let $p = \rho_A e_s(A,B,T)$, where ρ_A is the unconditional probability of A firing at an instant (ie., in an interval if length ΔT). Then p is the probability of the episode occurring starting at a given instant. Consider a sequence of *iid* random variables $\{X_i,\ i = 1,2,\ldots\}$ with distribution given by

$$\begin{aligned} P[X_i = T] &= p \\ P[X_i = 1] &= 1-p \end{aligned} \tag{3.1}$$

Let N be a random variable defined by

$$N = \min\{n \ : \ \sum_{i=1}^{n} X_i \geq L\} \tag{3.2}$$

where L is a fixed constant denoting the length of data in units of ΔT.

Let the random variable Z denote the number of X_i's out of the first N which have value T. Define the random variable M by

$$\begin{aligned} M &= Z \quad \text{if} \quad \sum_{i=1}^{N} X_i = L \\ M &= Z-1 \quad \text{if} \quad \sum_{i=1}^{N} X_i > L \end{aligned} \tag{3.3}$$

All the random variables N,Z and M depend on the parameters L,T,p. The random variable $M(L,T,p)$ actually represents the number of non-overlapped occurrences of the episode $A \xrightarrow{T} B$.

As described earlier., if at the first instant there is an occurrence of an episode (which happens with probability p) starting at that instant then, we a move T time units along the time axis and look for another occurrence. This is captured by assigning a value of T to X_1. If there is no occurrence of an episode at the first instant (which happens with probability $(1-p)$)then X_1 takes a value of 1. Thus, in a way, the variables $X_i's$ also indicate the occurrence of episodes at some time instants. Various $X_i's$ are independent of one another because, when we are counting non-overlapped occurrences, the occurrence of an episode at a given time instant is independent of the number of occurrences before that time instant.

The counting process is well captured by accumulating $X_i's$ till the end of data is reached.

Random variable N (ref Eqn. 3.2), denotes the number of such $X_i's$ that are accumulated since L represents the length of the data. It denotes the number of time instants at which we look for an occurrence of the episode before the event stream is exhausted. Random variable Z denotes the number of $X_i's$ out of the first N that take value T, i.e. the number of time instants at which we decided to move T time units along the data. The number of non-overlapped occurrences reported by the algorithm is either Z or $Z-1$ depending on whether the last occurrence was allowed to complete or not. Now it is clear from equation 3.3 that M is the number of non-overlapped occurrence counted.

The described model of counting can be extended to episodes of arbitrary length also. For example, if the episode is $A \overset{T_1}{\rightarrow} B \overset{T_2}{\rightarrow} C$, then $T = T_1 + T_2$ and $p = \rho_A e_s(A, B, T_1) e_s(B, C, T_2)$. In general, the value of T is obtained as sum of synaptic delays in the patterns and p is obtained from the product of pairwise conditional probabilities.

3.2.4 Significance Test

The Null hypothesis used in the significance test described in [18] is more complex than independence. The Null hypothesis includes all models of interacting neurons for which we have $e_s(x, y, T) < e_0$ for all pairs of neurons x,y and for a set of specific delays T, where e_0 is fixed used chosen number in the interval (0,1).

All models of inter-dependent neurons where the probability of A causing B to fire (after a delay) is less that e_0, would be in the Null hypothesis. The actual mechanism by which spikes from A affect the firing by B is immaterial. Whatever may be this mechanism of interaction, if the resulting conditional probability is less than e_0, then that model of interacting neurons would be in the null hypothesis. The model also includes independence among neurons. If two neurons A and B are independent, then the probability of B firing, a certain delay after $A's$ firing, is the unconditional probability(ρ_B) of B firing at any given time instant. If the average firing rate of B is 5 Hz and the time resolution $\Delta T = 1$ ms, then, $\rho_B = .005$. Thus if $e_0 = 0.5$, the null hypothesis includes only independence among neurons. If $e_0 = .05$, it means that only connections that have conditional probability 10 times that of independence can be called as "strong".

If it is possible to reject such a Null hypothesis then it is reasonable to say that the episodes discovered will indicate "strong" interactions among appropriate neurons. A method to bound the probability, that under the null hypothesis, the frequency (number of non-overlapped occurrences) of serial episode with inter event time constraints is more than a given threshold, is required. For this purpose the mean and the variance of the random variable(M) representing the number of non-overlapped occurrences is calculated.

Mean and Variance of $M(L,T,p)$

Let $F(L,T,p) = E\,M(L,T,p)$ where E denotes expectation. Fixing an episode fixes the value of p and T. The recurrence relation for F can be obtained as follows.

$$\begin{aligned} E\,M(L,T,p) &= E\ [\ E\,[M(L,T,p) \mid X_1]\ ] \\ &= E\,[M(L,T,p) \mid X_1 = 1](1-p)\ +\ E[M(L,T,p) \mid X_1 \neq 1]p \\ &= (1-p)E\,[M(L-1,T,p)]\ +\ p(1\ +\ E[M(L-T,T,p)]) \end{aligned} \tag{3.4}$$

In words what this means is: if the first X_i is 1 (which happens with probability $1-p$), then the expected number of occurrences is same as those in data of length $L-1$; on the other hand, if first X_i is not 1 (which happens with probability p) then the expected number of occurrences are 1 plus the expected number of occurrences in data of length $L-T$.

Hence the recurrence relation is:

$$F(L,T,p) = (1-p)F(L-1,T,p) + p(1 + F(L-T,L,p)) \tag{3.5}$$

The boundary conditions for this recurrence are:

$$F(x,y,p) = 0, \quad \text{if} \ \ x < y \ \ \text{and} \ \ \forall p. \tag{3.6}$$

Using the same idea as in the case of the mean, a recurrence relation for the second moment of M can also be obtained. Knowing $E\,[M]$ and $E\,[M^2]$, the variance of the number of non-overlapped occurrences can be obtained. Let $G(L,T,p) = E[M^2(L,T,p)]$. Then,

$$
\begin{aligned}
E\left[M^2(L,T,p)\right] &= E\ \left[E\left[M^2(L,T,p) \mid X_1\right]\right] \\
&= E\left[M^2(L,T,p) \mid X_1 = 1\right](1-p) \\
&\qquad + E[M^2(L,T,p) \mid X_1 \neq 1]p \\
&= (1-p)E\left[M^2(L-1,T,p)\right] \ + \ pE(1 \ + \ M(L-T,T,p))^2 \\
&= (1-p)E\left[M^2(L-1,T,p)\right] \ + \\
&\qquad pE(1 \ + \ M^2(L-T,T,p) \ + \ 2M(L-T,T,p))
\end{aligned} \tag{3.7}
$$

Simplifying,

$$
G(L,T,p) = (1-p)G(L-1,T,p) \ + \ p(1 \ + \ G(L-T,T,p) \ + \ 2F(L-T,T,p)) \tag{3.8}
$$

Solving the above recurrence give the value of $G(L,T,p)$. Let, $V(L,T,p)$ be the variance of $M(L,T,p)$. Then,

$$
V(L,T,p) = G(L,T,p) \ - \ (F(L,T,p))^2 \tag{3.9}
$$

For any $k > 0$, the number of non-overlapped occurrences can be bound using the Chebyshev inequality as follows

$$
\Pr\left[|M(L.T,p) - F(L,T,p)| > k\sqrt{V(L,T,p)}\right] \leq \frac{1}{k^2} \tag{3.10}
$$

Determining the frequency threshold for serial episodes

Let us consider an n-node episode. Let ρ be the random firing rate of the first neuron of the episode and e_0 the upper limit on the conditional probability for any pair of neurons. Let the allowed type-I error be ϵ. Let k be the smallest integer such that $k^2 \geq \frac{1}{\epsilon}$. Then from Equation 3.10,

$$
\Pr\left[M(L.T,p) > F(L,T,p) + k\sqrt{V(L,T,p)}\right] \leq \frac{1}{k^2} \leq \epsilon \tag{3.11}
$$

The Null hypothesis is that the conditional probabiliy for any pair of neurons is less than e_0. The random variable $M(L,T,p)$ is such that the probability of taking higher values increases with monotonically with p. Thus with $p = \rho e_0^{(n-1)}$, the probability of $M(L,T,p)$ being greater than any value is an upper bound on the probability of the episode frequency being greater than that value under any of the models in the null hypothesis.

Let $m_{th} = F(L,T,p)+k\sqrt{V(L,T,p)}$. From equation 3.11 it is clear that, if the episode frequency exceeds m_{th} we can reject the null hypothesis with a confidence of $1-\epsilon$.

For every episode, m_{th} is calculated. The episode is then reported as significant only if its frequency exceeds m_{th}. This way only patterns that do not belong to the null hypothesis are reported (ie., only "strong " connections are reported) The values of $F(L,T,p)$ and $M(L,T,p)$ are calculated using equations 3.5, 3.8 and 3.9.

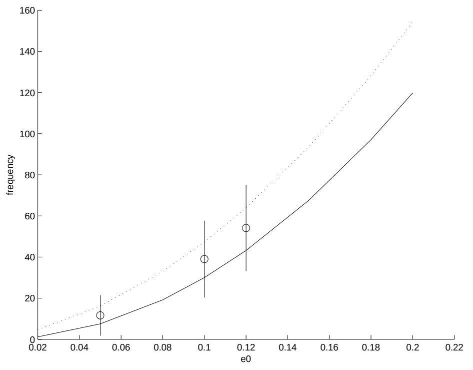
3.2.5 Simulation Results

In this sub-section we discuss the results of experiments designed to prove the effectiveness of the significance tests for sequential patterns. For the experiments, we use spike train data generated from a neuronal spike simulator. Strong connections between neurons can be embedded by specifying the conditional probability,$e_s(A,B,T)$, between pairs of neurons. The working of the simulator given in detail in Appendix A.

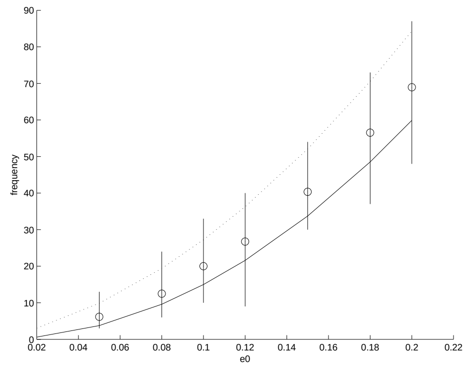
For the results reported here we used a network of 100 neurons with the nominal firing rate being 5 Hz. Each neuron is connected to 25 randomly selected neurons with the connection strengths ranging over [0.0025, 0.01]. With 5 Hz firing rate and 1ms time resolution, the effective conditional probability when two neurons are independent is about 0.005. Thus the random connections have conditional probabilities that vary by a factor of two on either side as compared to the independent case. We then incorporated some strong connections among some neurons. We have chosen the connection strengths of episodes to span the range 0.05 to 0.2. We have used three different delays: 5ms, 50ms and 120ms. Using the simulator with non-homogeneous Poisson model, we generated spike trains for 600 sec of time duration and obtained the counts of non-overlapped occurrences of episodes of all sizes using the algorithm for mining serial episode with inter-event delay [10]. The list of delays given to the algorithm are the intervals: $[4,6]$, $[49,51]$ and $[119,121]$. We had also generated another data set of 300 sec using the Gamma distribution (with time-varying parameters) as the inter-spike interval distribution. For this data set, we embedded only 3-node episodes all with delay of 5ms. On this data set also we obtained counts of all episodes using the datamining method with the same set of delay intervals. In all results presented below, all statistics are calculated using 500 repetitions of this simulation. Typically, on a data sequence for 600 Sec duration, the mining algorithms (run on a dual-core Pentium machine) takes about 25 minutes. Fig. 3.1 shows that the theoretical model for calculating the mean and variance of of the non-overlapped count (given by F and V determined through eqns. (3.5) and (3.9)) are good. The figure shows plot of the mean (F) and mean plus three times standard deviation ($F+3\sqrt{V}$) for different values of the connection strength in terms of conditional probabilities (denoted as e_0 in the figure), for the different episode sizes. Also shown are the actual counts obtained for episodes of that size with different e_0 values. We have shown plots for data generated with non-homogenous Poisson model (for 3-node episodes) as well as for data generated with non-homogeneous Gamma distribution for inter-spike intervals (3-node episodes). (For data generated with the Poisson model, we have shown the counts only for episodes with inter-event delay as 5 ms.) The actual counts obtained are a little more than what is predicted by our analysis because of two reasons. Firstly, in the data generation we assumed a fixed delay but the algorithm counts occurrences for delay in a small interval around the true value. Secondly, during data generation, the weights of connections are determined so that the conditional probability will be the specified value when there is no other input into the neuron; however, since all neurons also receive random output from others the the effective conditional probability and hence the count for the pattern would be a little higher. As is easily seen from the figure, the theoretically calculated mean and standard deviations are good and the method works equally well even when the data generation uses a Gamma distribution for inter-spike intervals. (Note that the analysis does not assume any model for the spike process). Notice that most of the observed counts are below the $F + k\sqrt{V}$ threshold for $k = 3$ even though this corresponds to a Type-I error of just over 10%. Thus our statistical test with $k = 3$ or $k = 4$ should be quite effective. Similar results were obtained for 4-node and 5-node pattern.

Using the formulation of the significance test we can infer a connection strength in terms of conditional probability based on the observed count. For this, given the count of a sequential pattern or episode, we ask what is the value of the strength or conditional probability at which this count is the threshold as per our significance test. This is illustrated in Fig. 3.2. For any episode if the inferred strength is q then we can assert (with the appropriate confidence) that it is highly unlikely for this episode to have this count if connection strength between every pair of neurons is less than q.

In Fig. 3.2 we show how good the mechanism is for inferring the strength of connection. Here we plot the actual value of the strength of connection in terms of the conditional probability as used in the simulation against the inferred value of this strength from the theory based on the actual observed value of count. For each value of the conditional probability, we have 500 replications and the ranges of inferred values are shown as a vertical line. The result in this figure shows the effectiveness of the approach in determining significance of sequential patterns based on counting the non-overlapped occurrences.

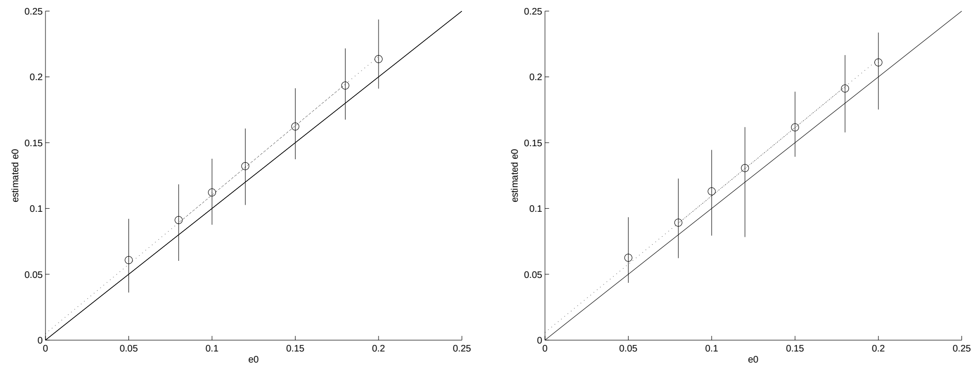


(a)



(b)

Figure 3.1: The analytically calculated values for the mean (i.e., F) and the mean plus 3 σ (i.e., $F + 3\sqrt{V}$), as a function of the connection strength in terms of conditional probabilities. (a) show plots for 3-node patterns with delay of 5ms when Poisson model is used in data generation. (b) shows the the plots when we used Gamma distribution for the inter-spike interval. For each value of the conditional probability, the actual counts as obtained by the algorithm are also shown. These are obtained through 500 replications. For these experimental counts, the mean value as well as the $\pm 3\sigma$ range (where σ is the data standard deviation) are also indicated. As can be seen, $F + 3\sqrt{V}$ line captures most of the count distribution even though this threshold corresponds to type-I error of about 10%.



(a) (b)

Figure 3.2: Plot of actual value of conditional probability used in the simulation versus the estimated value from the test of significance. Results are shown for episodes of size 3 for the cases of (a). Poisson and (b). Gamma models for data generation. (For each value we do 500 replications and the inferred value are represented as a point cloud. The mean of the inferred values and the line of best fit (dotted line) of the means are also plotted. Also a 45^o line is shown. Our method is quite effective in inferring the connection strength from the observed counts irrespective of the model of data generation.)

3.3 Significance of parallel episodes

Apart from the sequential patterns discussed above, the other kind of pattern that has been widely addressed in the literature is the *synchronous firing pattern* [11]. It corresponds to the synchronous firing of a group of neurons within a short duration of time. A synchronous pattern is represented by the set neurons that fire (in any order) to constitute its occurrence. For example, $(ABCDE)_T$ represents the synchronous firing of neurons A,B,C,D and E within a time T of one another. Synchronous firing patterns are captured by parallel episodes with expiry times as explained in Sec. 2.8.1. Efficient algorithms have been developed to unearth such firing patterns from spike train data [11, 19, 10]. The algorithm proposed in [10], discovers frequent parallel episodes with expiry time constraints to represent the frequently occurring synchronous patterns. The algorithm looks for non-overlapped occurrences of parallel episodes in the event stream. Like sequential pattern mining algorithm, it also employs apriori based techniques to count only certain patterns.

The algorithm counts the non-overlapped occurrence of an episode (say, $(ABCD)_T$ as follows. While going down the data stream it remembers the latest time of occurrence of all its constituent events. Once all the events are seen at least once, it checks if the span of the latest occurrences of all the events is less than T. If the expiry time constraint is satisfied, then frequency counter is incremented and all the events are marked as not seen. The algorithm then proceeds further to look for more occurrences. It is easy to see that such a method counts only non-overlapped occurrences of the parallel episodes.

The model, described in Section 3.2.3, can be extended to assess the significance of parallel episodes given the number of non-overlapped occurrences. However, unlike the sequential patterns, the span of the occurrence of the parallel episode under expiry time constraint is not fixed. The span can vary anywhere between 1 time unit to T time units. Thus the random variable X_i (ref Equation 3.1) which represents the number of time units to advance along the data stream can take multiple values. We modify the counting procedure such that it becomes compatible with the stochastic model in Section 3.2.3.

A synchronous pattern is said to occur at a given time instant, if there is a spike from atleast one of the constituent neurons at that time instant and there are spikes from all the other neurons within a time T. For example, an episode $(ABCD)$ is said to occur at time t, if there is a spike of A or B or C or D at t and at least a spike from each of the rest in the interval $[t, t + T]$. We note that the occurrence of the episode need not span the entire interval. The counting procedure described above is modified as follows. Whenever the algorithm finds a valid occurrence of an episode starting at time t we increment the frequency counter by one and ignore the further events till time $t + T$. We then start looking for other occurrence of the parallel episode starting from $t + T + 1$. Thus, similar to the counting of serial episodes, we need to move either T time instants or one time instants depending on whether the parallel episode has occurred or not. Using the same arguments as in the previous section, we can see that the random variables $X_i's$ clearly captures the modified counting procedure The random variable M (ref Equation 3.3) represents the number of occurrences of the parallel episode using above method. (We note that, after the modification we are not counting the maximum number of non-overlapped occurrences. Some occurrences of the data might be missed because we are ignoring some of the events in the data stream.)

The recursive methods (ref 3.2.4) to obtain the mean and variance of M also holds for the above proposed count. However the null hypothesis used to determine the significance of serial patterns is invalid. The notion of 'strength' between the two constituent neurons of a synchronous firing pattern is not well defined. Two neurons (say A and B) belonging a synchronous pattern may occur in any order in the occurrence. Synchronous firings of neurons may not be interactions that are related to the synaptic connections between them. Measure such as pairwise conditional probability that characterize the sequential firings so well cannot be clearly defined here.

Thus, the null hypothesis we employ is that of independence between spike trains. The hypothesis is that the spike trains of different neurons are generated by independent processes. The statistical theory discussed in the previous section can also be used here. Although here, the parameter p, represents the probability of occurrence of a parallel episode at a given time instant. Under the null hypothesis of independence it represents the probability of a random occurrence of an episode.

Let us consider episodes of size n. Let ρ be the unconditional probability that a neuron fires at any given time instant. (Assuming same random firing rate for all neurons) Then at a given time t the probability of occurrence of a random n-node parallel episode with expiry time T is given by.

$$p = \rho^n (\Delta T)^n \sum_{i=0}^{n-1} (T-1)^{n-1-i} T^i \tag{3.12}$$

Using this value of p, we can calculate values of $F(L,T,p)$ and $V(L,T,p)$ from equations 3.5, 3.8 and 3.9. For a given type-I error ϵ, the frequency threshold can then be obtained as $F(L,T,p) + k\sqrt{V(L,T,p)}$, where k is the smallest integer such that $k^2 \geq \frac{1}{\epsilon}$.

We use this frequency threshold for mining significant parallel episodes (synchronous firing patterns).

Effectiveness of Significance test

Here, we verify the effectiveness of the significance analysis discussed above. For the results reported here we generate spike train data from 100 neurons using the simulator given in Appendix A. No patterns are embedded in the data. Thus, the neurons fire independently of one another. We generate 500 repetitions of 20 seconds data for random firing frequencies of 5 and 10 Hertz. Then for an arbitrary 2-node and 3-node episode we find the frequencies in all the realizations using the modified parallel episode algorithm. Figure 3.3 shows the histogram plots of the frequencies for different expiry times and background firing rates. From the plots it is clear that for low values of expiry time ($T = 5$) the statistical model is able to capture the process very well. For 2-node and 3-node episodes, the calculated value of mean (F) lies at the centre of the distribution. However, for large values of $T (= 10)$, the model over-estimates F. This is because, unlike in the case of assessing significance of sequential patterns, the variables $X_i's$ here are not independent of one another. For example, consider the model for counting the occurrence of $(ABC)_T$. For some time instant t and some j,let X_j be the random variable that checks for the occurrence of the episode at t. Suppose we assume $X_j = 1$. This implies that at least one of the neurons among $A, B\ or\ c$ has not fired in the interval $[t, t+T]$. Now X_{j+1} will be a random variable that checks for the occurrence the episode in $[t+1, t+T+1]$. For $t > 1$, the intervals checked by X_j and x_{j+1} overlap. Hence, X_j and x_{j+1} are not independent. However, when $X_j = T$, X_{j+1} is independent of X_j. This is because the intervals of interest do not overlap in this case.

Thus the model does not exactly capture the counting process. Although, for small values of expiry time and normal firing frequencies like 5 or 10 Hertz the calculated value is a fair representation of the observed counts. For further experiments, we use the above discussed method of assessing statistical significance to determine the frequency of mining for synchronous patterns.

3.4 Simulation Results

In this section we compare the effectiveness of our parallel episode mining algorithm with NeuroXidence [11].

3.4.1 NeuroXidence

NeuroXidence is a popular tool used to detect an excess or a lack of synchronous firing in spike train data. This is done by counting the number of occurrences of synchronous firing patterns satisfying a given expiry time. Unlike our non-overlapped counts, NeuroXidence counts all occurrences of a pattern. For example, for the pattern (ABC), any set of spikes of A,B and C that satisfy the time constraint is considered as an occurrence. NeuroXidence counts the frequency of all patterns that occur at least once in the data. The counting process is essentially a correlation based technique.

Let the time axis be resolved into bins of size ΔT. The effect of a spike of a neuron in a given time bin is extended over τ_c neighboring bins , where τ_c is the expiry time of an episode (in units of ΔT. Thus if 3 neurons (say A, B, C) fire within τ_c bins of one another, then after extending the effect of the spike, there must be at least one time bin where the effect of the firing of all the three neurons is felt. The counting algorithm looks for such time bins to calculate the number of occurrences of an episode. However, employing such a procedure requires that two spikes of a given neuron should not occur within τ_c of one another. If such spikes occur then appropriate pre-processing is done to modify the data so that the counts of the episodes are retained and also that no two spikes of a neuron lie within τ_c of one another.

NeuroXidence employs a non-parametric method to assess the significance of the observed counts. The Null hypothesis is that the patterns occur by chance. The estimate of the chance frequency under null hypothesis is obtained by generating surrogate data. Surrogate data is created by jittering the spikes of the neurons independently of one another in timescales of the order of τ_c. This way the temporal cross structure in the data is destroyed while retaining the auto-structure of the spike trains. For every trial of the data obtained, around 25 surrogates are created. The patterns frequencies are found out in the surrogate data set. From the values so obtained, we get an empirical distribution of the chance frequencies. Using that the significance of the observed frequency counts are obtained.

NeuroXidence is found to be very effective in finding synchronous firing patterns [11].

3.4.2 Parallel Episode Mining Vs NeuroXidence

For the results provided in this section we use spike train data generated by using the Poisson simulator. No strong connections are embedded into the simulator. The neurons fire independently of one another. We include correlated firing in the data by means of external stimulation. At every time instant we decided with a low probability (0.0005) whether to embed a synchronous firing. Different sized patterns (upto 7 nodes) with various expiry times are embedded in the data.

Effectiveness of both the methods are assessed with respect to the running times and false positive rates. The methods are tested for varying parameters like random firing rate, different expiry times, different number of neurons. The running times and false positives rate reported for the parallel episode algorithm are average values obtained from 100 realizations of the data. In case of NeuroXidence, the values are averaged over 20 iterations. The results are reported in Tables 3.1-3.7..

NeuroXidence requires input data from various trials. For our experiments we split a single long data into 20 portions and give it as an input. For better statistical analysis, the number of surrogates is set at 25.

Both the methods were found to be very effective in mining the embedded patterns. All the patterns that are embedded in that data were discovered by the both the methods. However, the parallel episode mining algorithm has huge computational advantage over NeuroXidence (refer Table 3.1) Such difference in times are because the NeuroXidence calculates the frequencies of all possible patterns in the data. But the parallel episode mining algorithms uses an efficient level wise procedure to count candidates generated out of frequent sub-episodes. Also, the statistical test required for NeuroXidence requires it to find the frequencies of pattern in the surrogate data. If the number of surrogates is 25, then effectively NeuroXidence has to calculate the frequencies in data that is 25 times longer than the input data. This is the reason for the marked difference in running times of the algorithms.

In determining the false positive rates, we use a type-I error of 10%. This is because for lower type-I error the parallel episode algorithm reports only the embedded patterns as significant.

The change in expiry time of mining does not affect the running times of tha episodes mining algorithms(see Table 3.5). As we have already seen (c.f. Section 3.3) with increasing expiry time the estimate of the mean, and hence the threshold, is on the higher side. This the reason for the decrease in the reported false positive rates even though the frequency of random patterns may increase because of greater expiry times. The running time of NeuroXidence increases drastically with expiry times(see Table 3.2). Pre-processing stage becomes more complex to ensure that no two spikes of a neuron near one another. This is essentially done by increasing the data length. That is the reason the running times increase. Similar effects can also be seen because of increase in background firing rates(see Table 3.4). The number of false positives increase because more and more patterns start occurring more than once.

The increase in number of neurons has a huge effect on the running times of NeuroXidence(see Table 3.3). Infact for a network with 40 neurons the running time is as high as 20 minutes for only 50 sec data at 5 Hz. This is because of the exponential increase in the number of patterns to be counted.

Length of data (in seconds)	Average Running Times (in seconds)		False Positives (in %)	
	Parallel Episode Mining	NeuroXidence	Parallel Episode Mining	NeuroXidence
50	0.2	51	15%	31%
100	0.375	134	21%	47%
200	0.8	270	48%	79%

Table 3.1: Comparison of NeuroXidence and Parallel Episode Mining Algorithm : Running Times and False Positive Rates comparison for varying data lengths. (Parameters: Background Frequency = 5 Hz, $T = 5$, Number of Neurons = 20.)

Expiry Time (T) (in units of ΔT)	Average Running Times (in seconds)	False Positives (in %)	
	Type-I error = 10%	Type-I error = 10%	Type-I error = 5%
3	21	23%	12%
5	51	31%	17%
8	122	51%	29%
10	189	54%	29%

Table 3.2: NeuroXidence : Variation of Average Running Times and False Positives Rates for mining with different expiry times (Parameters: Background Frequency = 5 Hz, Data length = 50 seconds, Number of Neurons = 20.)

From Tables 3.5-3.7, it is clear that change in random firing frequency, expiry times, etc., does not affect the running times very much. The algorithm will be able to scale up for longer data with many interacting neurons.

Number of Neurons	Average Running Times (in seconds)	False Positives (in %)	
	Type-I error = 10%	Type-I error = 10%	Type-I error = 5%
20	51	31%	17%
30	233	49%	25%
40	1193	59%	35%

Table 3.3: NeuroXidence : Variation of Average Running Times and False Positives Rates for data with different number of neurons (Parameters: Background Frequency = 5 Hz, Data length = 50 seconds, $T = 5$.)

Random frequency (in Hz)	Average Running Times (in seconds)	False Positives	
	Type-I error = 10%	Type-I error = 10%	Type-I error = 5%
5	51	31%	17%
10	309	49%	26%

Table 3.4: NeuroXidence : Variation of Average Running Times and False Positives Rates for different firing rates of neurons (Parameters: $T = 5$, Data length = 50 seconds, Number of Neurons = 20.)

3.5 Conclusions

In this chapter, we extended the significance test for sequential patterns to suit parallel episodes. Though the statistical model is does not fit the counting process perfectly , we showed experimentally its effectiveness in mining synchronous patterns. The statistical methods discussed in this chapter assumed characteristics like a known model of spike generation and pairwise interaction among neurons. However, the main advantage of temporal datamining methods in unearthing patterns from spike train data is the computational efficiency over the more conventional methods. Experimental results discussed in this chapter proved the same. Combined with the proper statistical techniques the frequent episode discovery will be very useful in analyzing multi-neuronal data.

‘

Expiry Time T (in units of ΔT)	Average Running Time (in seconds)	False Positives
3	0.38	29%
5	0.38	22%
8	0.37	15%
10	0.37	14%

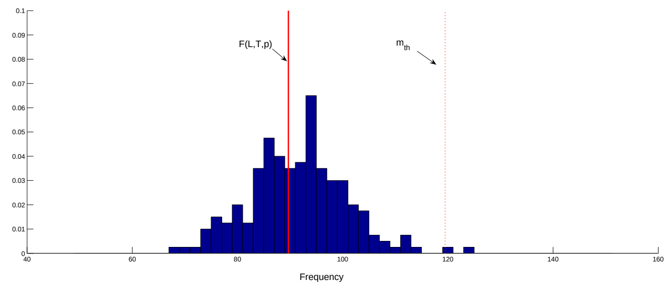
Table 3.5: Parallel Episode Mining Algorithm : Variation of Average Running Times and False Positives Rates for mining with different expiry times (Parameters: Background Frequency = 5 Hz, Data length = 100 seconds, Number of Neurons = 20.)

Number of Neurons	Average Running Time (in seconds)	False Positives
20	0.38	22%
30	0.44	27%
40	0.54	40%

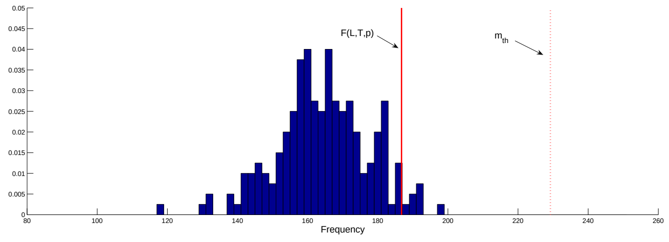
Table 3.6: Parallel Episode Mining Algorithm : Variation of Average Running Times and False Positives Rates for data with different number of neurons (Parameters: Background Frequency = 5 Hz, Data length = 100 seconds, $T = 5$.)

Random Frequency (in Hz)	Average Running Time (in seconds)	False Positives
5	0.38	22%
10	0.5	22%

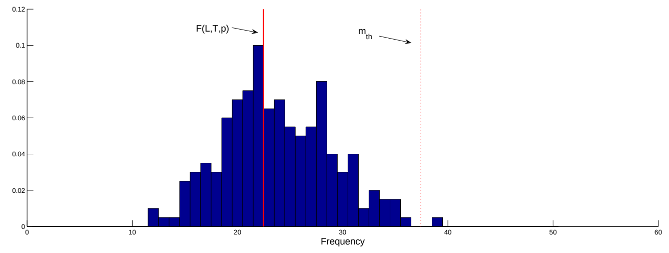
Table 3.7: Parallel Episode Mining Algorithm : Variation of Average Running Times and False Positives Rates for data with different number of neurons (Parameters: Data length = 100 seconds, $T = 5$, Number of Neurons = 20)



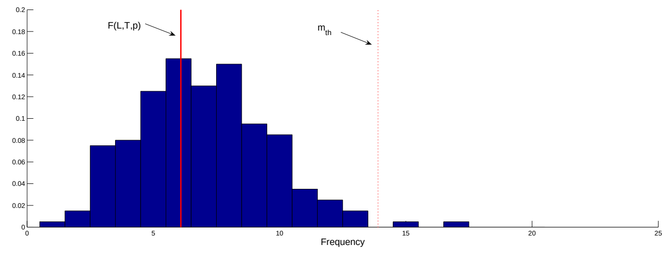
(a)



(b)



(c)



(d)

Figure 3.3: Distribution of the observed frequencies of Parallel Episodes. Plots are are shown for 2-node and 3-node episodes for different values of random frequency(f_{rand}) and expiry times (T). (a) & (c) show plots for a 2-node episode with $f_{rand}. = 10\ Hz$ and $T = 5$ & $10ms$. (b) shows the distribution for $f_{rand} = 5\ Hz$ and $T = 5\ ms$, (d) for a 3-node episode with $f_{rand}. = 10\ Hz$ and $T = 5\ ms$. The analytically calculated values of mean (F) and threshold ($m_{th} = F + 3 * \sqrt{V}$) are also shown. For small expiry times ($T = 5$), the calculated value matches the observed counts.

Chapter 4

Conclusions

In this project, we looke at frequent episode discovery framework in the context of temporal datamining. Efficient algorithms for discovering parallel and serial episodes were already available. We developed algorithms for discovering episodes with general partial orders from event streams. We thought that the graphical structure of partial orders can be used to represent the connectivity structures of neurons in a tissue. These partial orders should occur frequently in the spike train data. We used the frequent episode discovery algorithm to directly unearth the connectivity structure.

The frequent episode discovery techniques are computationally very efficient in discovering patterns from event streams. Thus if we have proper statistical techniques to determine the significance of episodes they can be very useful in analysing spike train data. We developed techniques to determine the significance of parallel episodes. We also proved the efficiency of datamining techniques in analysing spike train data by comparing it with an efficient existing tool, NeuroXidence.

Methods to determine the statistical significance of frequent graph patterns do not exist. If available, the technique will be very useful to directly unearth significant connections among neurons. Also, in this work we consider only pairwise interactions among neurons. Recent work reports existence of possible higher order interations among neurons. Application of frequent episode discovery techniques to unearth significant higher order interactions is still not explored.

Appendix A

Spike Train Data Generator

We use a simulator for generating the spike data from a network of interconnected neurons. For most of our simulations we model the spiking of each neuron as an inhomogeneous Poisson process where the instantaneous rates are time-varying because they depend on the input spikes received by the neuron through its synapses. We first describe this model. At the end of this section, we explain how we can modify the simulator to generate spike trians where inter-spike interval has Gamma distribution (with time-varying parameters).

Let N denote the number of neurons in the network and let $Z_i(t)$ denote the spiking process of the i^{th} neuron. Each Z_i is taken to be an inhomogeneous Poisson process whose rate is constant on each interval $[n\Delta T,\ (n+1)\Delta T)$, $n = 0, 1, \cdots, L$ where L represents the total time duration in units of ΔT. We normally take $\Delta T = 1ms$. Let $\lambda_i(k)$ denote the rate of $Z_i(t)$ for $t \in [(k-1)\Delta T,\ k\Delta T)$.

Let Y_{ik}, $k = 1, 2, \cdots T$, $i = 1, 2, \cdots, N$, be binary random variables defined as: $Y_{ik} = 1$ if there is a spike event of Z_i in the interval $[(k-1)\Delta T,\ k\Delta T)$; and $Y_{ik} = 0$ otherwise.

If we are given $\lambda_i(k)$ for all i, k then we can simulate the Z_i processes as follows. For each k, $k = 1, 2, \cdots$, we do the following: For $i = 1, 2, \cdots, N$, we generate random variables, ξ_i, exponentially distributed with parameter $\lambda_i(k)$; then, for each i, if $\xi_i < \Delta T$ then we put a spike for neuron i at time $(k-1)\Delta T + \xi_i$ else we put no spike for i in the interval $[(k-1)\Delta T,\ k\Delta T)$

The set of neurons are interconnected through synapses and the connection from i to j is characterized by a weight parameter w_{ij} and a delay parameter τ_{ij}. We take it that the delay τ_{ij} is specified in units of ΔT. The firing rates, $\lambda_i(k)$, of neurons are influenced by the inputs received from other neurons. Let Y_{ik}, $k = 1, 2, \cdots T$, $i = 1, 2, \cdots, N$, be binary random variables defined as: $Y_{ik} = 1$ if there is a spike event of Z_i in the interval $[(k-1)\Delta T,\ k\Delta T)$; and $Y_{ik} = 0$ The firing rates at k^{th} time-step are computed as

$$\lambda_i(k) = \frac{\lambda_m}{1 + \exp(-\theta_i - \sum_{j=1}^{N} w_{ji} Y_{j(k-\tau_{ji})})}. \tag{A.1}$$

Here θ_i is a parameter that fixes the nominal firing rate for the neuron i. (This is the firing rate when the neuron receives no input). The constant λ_m denotes the maximum possible firing rate attainable and its value is fixed so that the probability of atleast one spike in an interval of length ΔT is 0.99 and this value is the solution of the equation

$$1 - \exp(-\lambda_m \Delta T) = 0.99 \tag{A.2}$$

To get a specific network, the user specifies, for all i, the nominal firing rate, λ_0^i, of neuron i and the strengths and delays of all interconnections. Given the value of λ_0^i, the value of θ_i is fixed as

$$\theta_i = -ln\left(\frac{\lambda_m}{\lambda_0^i} - 1\right) \tag{A.3}$$

The strengths of interconnections (or synapses) are specified in terms of conditional probabilities. Let p_{ij} denote the strength of connection from i to j and it is taken to be the conditional probability that there is atleast one spike from j in an interval $[(k-1)\Delta T,\ k\Delta T]$ given that there is atleast one spike from i in the interval $[(k-\tau_{ij}-1)\Delta T,\ (k-\tau_{ij})\Delta T]$ and that all other input to j is zero. Here τ_{ij} is the delay associated with this connection. Given p_{ij}. we can calculate w_{ij} as

$$w_{ij} = -\theta_j - ln\left(\frac{\lambda_m}{\lambda'} - 1\right) \tag{A.4}$$

where λ' is the solution of the equation

$$1 - \exp(-\lambda' \Delta T) = p_{ij} \tag{A.5}$$

To use the simulator, we specify the nominal firing rate of each neuron and the strengths (in terms of conditional probability as explained above) and delays, τ_{ij} (in units of ΔT) for all connections. Then we can determine the parameters θ_i and w_{ij}. Then for each k, we obtain $\lambda_i(k)$ for all i and this is used to simulate the Z_i processes as explained earlier. We normally specify a network which has many random interconnections (i.e., with the strengths being set randomly) and some specific connections to constitute the patterns or microcircuits by giving high strength for these connections.

We note here that the nominal firing rate as well as the effective conditional probabilities in our system would have some small random variations. As explained above, we fix θ_j so that on zero input the neuron would have the nominal firing rate. However, all neurons would have synapses with randomly selected other neurons and the strengths of these synapses are also random. Hence, even in the absence of any strong connections, the firing rates of different neurons keep fluctuating around the nominal rate that is specified. Since we choose random connections in such a way that in an expected sense the input into a neuron is zero, the average rate of spiking should be close to the nominal rate specified. We also note that the way we calculate the effective weight for a given conditional probability is also approximate and we chose it for simplicity. If we specify a conditional probability for the connection from A to B, then, as given by (A.4)–(A.5), the weight of the connection is fixed so that the probability of B firing at least once in the next ΔT interval given that A has fired in an appropriate interval earlier is equal to this conditional probability *when all other input into B is zero*. But since B would be getting small random input from other neurons also, the effective conditional probability would also be fluctuating around the nominal value specified.

We have also used this simulator for generating spike trains where inter-spike intervals are (non-homogeneous) Gamma distributed instead of being exponential as in case of Poisson. (Our notation is: Gamma distribution with parameters α and β has density given by $f(x) = x^{(\alpha-1)}\beta^{\alpha}\exp(-\beta x)/\Gamma(\alpha)$). The changes needed in the simulator are as follows. We simulate the $Z_i(k)$ by generating ξ_i as before; but now the ξ_i has Gamma distribution with $\alpha = 2$ and $\beta = \lambda_{ik}$. Thus now, the spiking processes of neurons are such that inter-spike intervals are Gamma distributed with a time-varying rate parameter. The λ_{ik} are updated, as before, using eq. (A.1). But now λ_m is a solution of

$$1 - \exp(-\lambda_m \Delta T)(1 + \lambda_m \Delta T) = 0.99 \tag{A.6}$$

The θ_i are determined as before using the above λ_m. The weights w_{ij} are computed using eq. (A.4) as before; however the λ' in this equation is determined as the solution of

$$1 - \exp(-\lambda' \Delta T)(1 + \lambda' \Delta T) = p_{ij} \tag{A.7}$$

With these changes, we can still specify the connection strengths as conditional probabilities (p_{ij}) and with the weights determined as above we will have spike trains with the required embedded connection strengths. A minor difference is that the earlier equations, namely, eqs. (A.2) and (A.5) have simple closed-form solutions. Now we solve eqs. (A.6) and (A.7) using Newton-Raphson method.

Bibliography

[1] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques with JAVA implementations*. Morgan Kaufmann Publishers, 2000.

[2] D. Hand, H. Mannila, and P. Smyth. *Principles of data mining*. MIT Press, Cambridge, MA, USA, 2001.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, Taipei, Taiwan, March 1995. IEEE Computer Society, Washington, DC, USA.

[4] Srivatsan Laxman and P. S. Sastry. A survey of temporal data mining. *$S\bar{A}DH\bar{A}N\bar{A}$, Academy Proceedings in Engineering Sciences*, 31:173–198, April 2006.

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

[6] Srivatsan Laxman. *Discovering frequent episodes: Fast algorithms, Connections with HMMs and generalizations*. PhD thesis, Bangalore, India, September 2006.

[7] K.P.Unnikrishnan, B.Q.Shadid, P.S.Sastry, and Srivatsan Laxman. Root cause diagnostics using temporal data mining, 2009. US Patent Number 7509234.

[8] J. Pine D. A. Wagenaar and S. M. Potter. An extremely rich repertoire of bursting patterns during the development of cortical cultures. *BMC Neurosci*, 11:7–11, 2006.

[9] E.N Brown, R.E. Kass, and P.P.Mitra. Multiple neuronal spike train data analysis: state of art and future challenges. In *Natural Neurosciences*, pages 456–461, 2004.

[10] Debprakash Patnaik, P.S.Sastry, and K.P.Unnikrishnan. Inferring neuronal network connectivity from spike data: A temporal data mining approach. *Scientific Programming*, 16(1):49–77, 2008.

[11] Gordon Pipa, Diek W. Wheeler, Wolf Singer, and Danko Nikolic. Neuroxidence : reliable and efficient analysis of an excess or deficiency of joint spike events.

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

[13] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

[14] S. Laxman, P. S. Sastry, and K.P.Unnikrishnan. A fast algorithm for finding frequent episodes in event streams. In *ACM SIGKDD International Conference Knowledge Discovery and Datamining*, August 2007.

[15] M.Abeles and G.L.Gerstein. Detecting spatio temporal firing patterns among simultaneously recorded single neurons. *Journal of Neurophysiology*, 60(3):909–924, 1988.

[16] M. Abeles and I. Gat. Detecting precise firing sequences in experimental data. *Journal of Neuroscience Methods*, 107:141.

[17] A. Date, E. Bienenstock, and S. Geman. On the temporal resolution of neural activity. Technical report, Division of Applied Mathematics, Brown University, Providence, RI, USA, 1998.

[18] P.S.Sastry and K.P.Unnikrishnan. Conditional probability based significance tests for sequential patterns in multi-neuronal spike trains. *preprint*.

[19] Sonja Grun, Markus Diesmann, and Ad Aersten. Unitary events in multiple single neuron spiking acivity. *Neural Computation*, 14:43–80, 2002.