Conception of Semantic Complex Event Pattern Mining methods on Event Streams

Focussing on predictive Episode Mining

Leon Bornemann

Department of Mathematics and Computer Science
Freie Universität Berlin

This thesis is submitted for the degree of
Master of Science

June 2016

Declaration

Does the FU have a declaration text in which I declare that I worked on this alone, up to scientific standard, did not copy anything without citing etc? If yes I will put this here.

Leon Bornemann
June 2016

Acknowledgements

And I would like to acknowledge (TODO) ...

Abstract

It is a little early for an abstract, I guess I could already write a preliminary Abstract...

Table of contents

List of figures

List of tables

Chapter 1

Introduction

This chapter serves as a rather broad introduction to the topic of this thesis and provides motivation for the work.

1.1 General Introduction and Motivation

Almost any application domain of information systems has some data that is being generated. Data is available in many different forms. One of these forms are data streams. Data streams are not limited to the recent rise in popularity of video and audio streams. On the contrary the application domains that produce data in the form of streams are very diverse. They include for example constantly running business applications that log business activities and events, sensor networks that report usage data or devices that take measurements of physical quantities (such as temperature, pressure, humidity, etc...) at certain points of time.
The fact that streams generate a constant stream of data and thus lead to a constantly growing database is a significant difference to classic applications of data mining in which there is a static (training) database. Despite that significant difference in the data representation, many fields of interest in the context of static databases remain the same for data streams. Common areas of interest are frequent patterns, predictive patterns, association rules, clustering and classification of the data entities. Approaches and algorithms that solve these problems for static databases, while by no means fully researched, are rather well known and evaluated. Applying these methods to data streams can present challenges and may demand many modifications due to the large and possibly infinite amounts of data produced by streams. Naturally, data mining methods for stream data must be especially fast, scalable and memory efficient.
Apart from the additional, algorithmic constraints on memory and computation time, data streams also present conceptual challenges. In contrast to static databases streams may

evolve over time, which can make it very difficult for algorithms to assess which past data of the stream should be considered when analyzing the currently incoming data. Recognizing these so called concept drifts is one challenge among many when processing or mining data streams.
A suitable way to look at most data streaming scenarios is that of event streams. An event can be anything that happens in the real world, which can be represented as an element of the stream. These events are commonly referred to as basic or simple events. A frequent area of interest when processing event streams is to mine complex events that consist of multiple basic events with different relationships between each other. Discovering interesting complex event patterns can be tough, especially since there may be a lot of potential candidates. Often we are only interested in specific event combinations. One possible approach to improve the mining process is to use domain knowledge. If the domain knowledge about the underlying event stream contains semantic information about the different event types it is possible to use that knowledge directly in the mining process. Semantic knowledge can take many different forms, a common one being an RDF-graph that can be examined using queries.
Figure 1.1 presents the basic idea of semantic complex event mining algorithms. On the lowest level of abstraction we have the low-level event stream, which is the unrefined data coming directly from the sources (for example sensor data). The low-level stream needs to be transformed in some way to an annotated event stream, which then in turn gets mined to discover complex events. The mining algorithm's basic input is the stream of annotated events, but it can also use the previously mentioned semantic knowledge, an ontology, which contains additional information about the event types.

In terms of the very broad term of complex events this thesis will focus on the subtopic of episode pattern mining from stream data or very large log files or databases. Episodes are a specific kind of complex events and are formally defined in chapter 2. For now it is sufficient to know that episodes are essentially complex events in which single events or entire episodes can be combined using two operators: the conjunction and the sequence operator. Figure 1.2 visualizes a simple episode pattern and example occurrences in a data stream.

So why is the mining of episodes of interest? There are many real-life use cases in which episode discovery is relevant. The discovery of frequent episodes can for example be related to the discovery of underlying models for data generation, which can help to better understand the data generation process. Another application is to find predictive episodes, meaning episodes which can help to predict events occurring in the future. This has already been applied to predict outages in Finnish power grids (TODO: find a citable source for this). The discovery of predictive episodes is relevant for many domains, for example sensor networks (if a certain chain of events leads to failure, predictive episodes can be used to

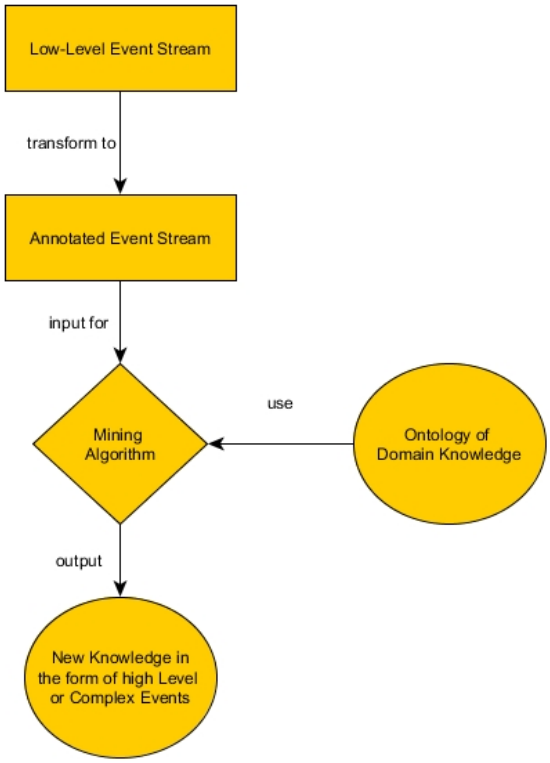


Fig. 1.1 General structure of a semantic mining process of complex events

preemptively expect failures and react accordingly). The domain that will be used in the evaluation of this thesis is stock market prediction. Predicting the overall direction of the stock market or whether individual stocks will rise or fall is a difficult problem that has the obvious application of generating investment strategies. Apart from this, predictive episodes also have use cases in the enforcement of regulations. Predictive episodes could for example be used to detect illegal price arrangements of certain companies, which have been known to happen between oil and gas stations of major companies (TODO: find and cite a source for this).

In contrast to other regression and forecasting methods, such as artificial neural networks, predictive episodes have the advantage that once they are discovered they can make ad-hoc predictions in a fast moving data-stream, whereas neural networks usually forecast the closing values of stock markets of the next day based on the values of previous days. This gives predictive episodes a niche: fast moving (real-time) data-streams that need quick predictions of future events.

The rest of the thesis is outlined as follows: Chapter 2 introduces basic terminology, gives an extensive introduction to episode mining and provides an overview over other relevant related work areas. TODO: enumerate the rest of the chapters

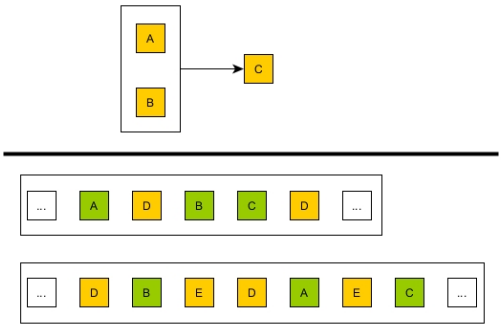


Fig. 1.2 The top half visualizes an example episode pattern, which consist of a conjunction (A and B must both occur, but the order does not matter) and a sequence (C must occur after A and B). The bottom half shows two windows of an example stream, in which occurrences of the episode are shown in green.

1.2 Contributions and Exact Research Question

This thesis aims to develop a semantic mining algorithm for predictive episodes. Since the algorithm will be used on data streams, the algorithm must have the following properties:

- The Algorithm must have an online learning mode
- The Algorithm must be able to adapt to a changing context (as the stream progresses, the underlying model may change completely)
- Fast prediction. Since streams, especially in the domain of stock markets, can have a high velocity it is important to be able to quickly predict events. If the prediction takes too long, the event which we want to predict may have already occurred before the algorithm outputs its prediction, thus robbing the user of the opportunity to take precautions.
- The Algorithm must not require to store the entire stream of events seen so far, since that is not feasible for most streams.

The algorithm will be evaluated on both synthetically generated data and real-life datasets. The latter ones will be from the domain of stock market prediction, in which the developed algorithm will be empirically compared to other approaches in terms of accuracy measures and execution time.

In summary the research question in particular is:

How can event streams be mined for episodes that effectively predict certain event

types and how can domain knowledge be used to improve the results or speed of the mining algorithm?

Chapter 2

Related Work

2.1 Basic Definitions and Terminology

This section introduces relevant definitions and terminology that was introduced in previous work and will be used in this thesis.

2.1.1 Event Processing Terminology

Most of the basic event terminology in this subsection is taken from the event processing glossary created by the Event Processing Technical Society [6]. Note that some of the definitions may be slightly altered or simplified. This is due to the fact that the event processing technical society uses these terms for a very general description of event processing and event processing architectures and thus some original definitions are more complex than what is needed in this thesis. The definitions given here aim to establish a clear terminology for this thesis.

Definition 1 ***Event*** *An event is either something that is happening in the real world or in the context of computer science an object that represents a real world event and records its properties. The latter can also be referred to as an event object or an event tuple. Note that the term is overloaded, but the context usually gives a clear indication of what is meant.*

Definition 2 ***Simple Event*** *A simple event is an event that is not viewed as summarizing, representing, or denoting a set of other events. Sometimes also referred to as a basic events.*

These two definitions can sometimes cause confusion. It is important to note that the term event is the most general term, since it can refer to any kind of event, be it simple, derived or complex (see definitions 3 and 4). A simple event however is the most basic form of an

event and often the ingredient for the creation of more complex events: Given simple events it is possible to derive events from those or the absence of those. For example the absence measurement events of a sensor could be used to derive the event of that very same sensor becoming defect. These events are called derived events:

Definition 3 ***Derived Event*** *A derived event or synthesized event is an event that is generated according to some method or based on some reasoning process.*

It is also possible to combine multiple simple events to form what we refer to as complex events:

Definition 4 ***Complex Event*** *A complex event is a derived event that is created by combining other events. The events can be combined by using certain operators, for example disjunction, conjunction or sequence. An example would be* $(A \wedge B) \rightarrow C$ *(event A and B in any order followed by event C).*

This is a very broad definition of complex events. The choice of allowed operators strongly impacts the expressiveness of complex events. A specific kind of complex events are episodes (see section 2.2), which will be the main focus of this thesis. The next notion that needs to be considered is that each individual event normally belongs to a certain class of events, which we refer to as the event type:

Definition 5 ***Event type*** *The event type, sometimes also referred to as event class, event definition, or event schema is a label that identifies events as members of an event class.*

Another important term that was not explicitly defined in the event processing glossary, but is very relevant to the topic at hand is the notion of a type alphabet:

Definition 6 ***Type Alphabet*** *The type alphabet, often simply called the event alphabet, is the set of all possible event types that can occur in the observed system.*

Event alphabets are often implicitly defined when mining frequent itemsets, patterns or episodes. So far we have looked at events without considering the scenario we are most interested in which are event streams. To do so we need the notion of timestamps:

Definition 7 ***Timestamp*** *A time value of an event indicating its creation or arrival time.*

Given that we can define an event stream:

Definition 8 ***Event Stream*** *An event stream is an ordered sequence of events, usually ordered by the event timings.*

Note that this rather broad definition of an event stream does not assume anything about the kind of event that is contained in it. A stream can contain very basic forms of events (simple events) but can also be made up out of derived events or even complex events. Also other properties for example that the stream is constantly updating (new events coming in) are not considered yet.

2.2 Episodes

This section gives a detailed introduction to the topic of episode mining and presents an overview over the most relevant related work.

2.2.1 Basic Definitions

As already mentioned, episodes are complex events whose basic building blocks are simple events. Note that in order to make use of episodes we require that all simple events have a type and we have a finite, previously known event alphabet, that contains these types, which we refer to as Σ. We follow up with a formal definition:

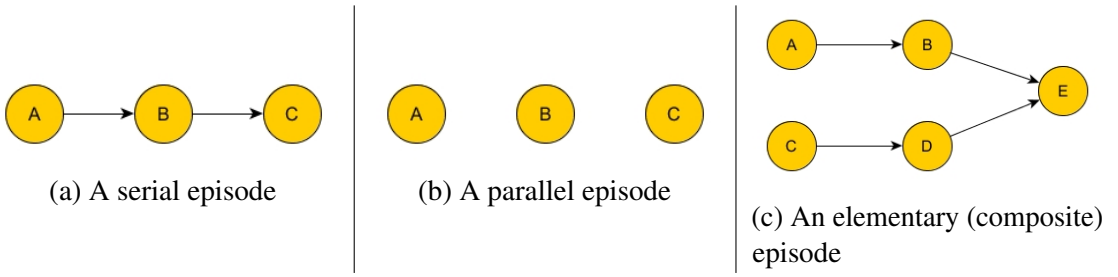
Definition 9 ***Episode*** *An episode (also sometimes called episode pattern or elementary episode) α of length m (also called m-episode) is defined as a triple: $\alpha = (V_\alpha, \leq_\alpha, g_\alpha)$ where $V_\alpha = \{v_1, ..., v_m\}$ is a set of nodes, $\leq_\alpha$ is a partial order over V_α and $g_\alpha : V_\alpha \rightarrow \Sigma$ is a mapping that maps each node of V_α to an event type [7].*

Put more simply an episode is a multiset of event types, whose elements can be, but do not have to be ordered by a relation ($\leq_\alpha$). Another way of putting it is that an episode is essentially a partially ordered sequence of events. Before we look at examples there are a two special types of episodes that need to be mentioned since they have received the most attention in the available literature. These are called serial and parallel episodes:

Definition 10 ***Serial Episode*** *An episode $\alpha = (V_\alpha, \leq_\alpha, g_\alpha)$ is called a serial episode if $\leq_\alpha$ is a total order [7].*

Definition 11 ***Parallel Episode*** *An episode $\alpha = (V_\alpha, \leq_\alpha, g_\alpha)$ is called a parallel episode if $\leq_\alpha = \emptyset$, in other words if there is no ordering imposed on V_α at all [7].*

Essentially, serial episodes are sequences, while parallel episodes are multisets. Figure 2.1 visualizes example episodes as directed acyclic graphs (DACs), which is a useful way to visualize episodes.



(a) A serial episode

(b) A parallel episode

(c) An elementary (composite) episode

Fig. 2.1 Different example episodes visualized as directed acyclic graphs

In fact each episode can be formally transformed to a DAC using the following simple procedure: Given an Episode $\alpha = (V_\alpha, \leq_\alpha, g_\alpha)$, create the corresponding DAC $G = (V, E)$ by executing the following:

1. For each $v \in V_\alpha$ add v to V and label v with $g_\alpha(v)$
2. For each pair $v, w \in V_\alpha$ where $v \leq_\alpha w$ add edge (v, w) to E

The original paper by Manilla et.al. [7] also introduces the notion of composite episodes. We repeat the definition here:

Definition 12 ***Composite Episode*** *An episode $\alpha = (V_\alpha, \leq_\alpha, g_\alpha)$ is called a composite episode if $g_\alpha : V_\alpha \to \Sigma \cup C^*$, where C^* is the set of all composite episodes.*

This recursive definition of composite episodes may be confusing at first but it has the advantage that any elementary episode can be represented as a composite episode which is exclusively a serial or parallel composition of serial, parallel or composite subepisodes (see definition 13).
Interestingly, there are other parts of the related work that use the term *composite episodes* but deviate from definition 12. For example Baathorn et. al. propose a method for finding composite episodes [2]. However they define composite episodes as a sequence of parallel episodes, which is more restrictive than the original definition. Also Baumgarten et. al. use this definition [3] when they present an approach to mine descriptive composite episodes. Note that not all elementary episodes can be represented as sequences of parallel episode. A simple example shown in figure 2.2 illustrates this. If the presented episode were to be represented as a sequence of parallel episodes obviously A and B would have to be in different parallel episodes in order to fulfill the requirement that B must be after C. After that the problem is that it is impossible to assign C to any of those sets. If it gets assigned to the same parallel episode set as A then this would prohibit A and B occurring first followed by C, which is allowed in the original definition. Likewise if C gets assigned to the same parallel

episode as B then that eliminates the possibility of C occurring before A and B, which once again was allowed in the elementary episode. In this thesis we will stick to the original definition as presented in definition 12.

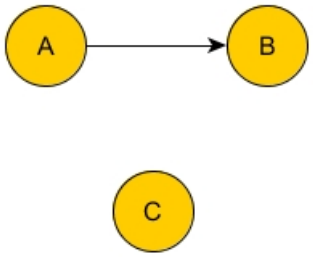


Fig. 2.2 An elementary episode that can not be represented as a sequence of parallel episodes

If we want to denote quick, simple episodes formally without drawing a DAC, we will use $\rightarrow$ as the sequence (ordering) operator. To show that there is no order specified between two nodes we use $\|$ as the parallel operator. For example $(A \,\|\, B) \rightarrow C$ denotes a composite episode of length 3, which specifies that it does not matter in which order A or B occur, but C must occur after both A and B. If we want to discuss more complex episodes we will visualize them graphically like in the previous figures
The notion of sub- and superpatterns, which is very important for most pattern mining applications also applies to episodes, as shown in the next definition.

Definition 13 ***Subepisode*** *An episode $\beta = (V_\beta, \leq_\beta, g_\beta)$ is said to be a subepisode of episode $\alpha = (V_\alpha, \leq_\alpha, g_\alpha)$ if all of the following conditions hold:*

1. $V_\beta \subseteq V_\alpha$
2. $\forall v \in V_\beta \ : \ g_\beta(v) = g_\alpha(v)$
3. $\forall v, w \in V_\beta \, , v \leq_\beta w \ : \ v \leq_\alpha w$

In this context we also refer to α as the superepisode of β [7] [5].

It is important to note that in the original definition of sub- and superepisodes by Mannila et. al. [7], the first property of the above definition was actually defined as $V_\beta \subset V_\alpha$, which implies that subepisodes would always have to consist of at least one less node than their superepisodes. This definition was changed by Laxman et. al. [5] to allow set equality as well. The implications of this change are that parallel episodes like $A \| B$ are now subepisodes of their serial counterparts of the same length $A \rightarrow B$. In this thesis we stick to the definition

that allows set equality between super- and subepisodes.
It is helpful to think of episodes as a template or pattern for concrete occurrences. In order to define what we mean by an episode occurance, we first need to formally introduce the notion of an event sequence.

Definition 14 ***Event sequence*** *An event sequence is defined as an ordered list of tuples* $S = [(T_1,t_1),...,(T_n,t_n)]$ *where* $T_i \in \Sigma$ *is the event type of the i-th event and* $t_i \in \mathbb{N}^+$ *is the timestamp of the i-th event. The sequence is ordered according to the timestamps, which means that* $\forall i,j \in 1,...,n\ i < j \implies t_i \leq t_j$.

Definition 15 ***Episode Occurrence*** *An event episode* $\alpha = (V_\alpha, \leq_\alpha, g_\alpha)$ *is said to occur in a sequence S if events of the types that the nodes in* V_α *are mapped to by* g_α*, occur in S in the same order that they occur in the episode. More formally if we are given a sequence of events* $S = [(T_1,t_1),...,(T_n,t_n)]$ *we can define an occurrence of* α *as an injective Map* $h : V_\alpha \to \{1,...,n\}$*, where* $g_\alpha(v) = T_{h(v)}$ *and* $\forall v,w \in V\alpha : v \leq_E w \implies t_{h(v)} \leq t_{h(w)}$ *holds [7].*

2.2.2 Episode Detection and general Mining Algorithm

When discovering episodes in a sequence one is usually interested in those episodes that occur frequently, meaning more often than a user defined threshold. This is similar to all the different kinds of pattern mining algorithms, such as the apriori algorithm for finding frequent itemsets [1]. A general algorithm for mining the episodes occurring frequently in a sequence is given in algorithm 1. The algorithm is very alike the basic apriori algorithms, since it uses a level-wise, breadth first search by first identifying all frequent episodes of a certain length i and then uses these frequent episodes to generate candidates (possibly frequent episodes) of length $i+1$. In order for this to be correct, episode frequency must follow the apriori principle [1], which formally means that if an episode is frequent, all its subepisodes must be frequent as well. Intuitively one would assume that this is always true for episodes but strictly speaking this depends on the definition of episode frequency. However, any frequency definition of episodes that does not satisfy the apriori principle would be highly questionable to say the least, since this would eliminate the possibility of an efficient candidate generation that prunes those episodes which have infrequent subepisodes. To the best of our knowledge all frequency definitions proposed in the literature satisfy the apriori principle.

Algorithm 1 General mining algorithm for frequent episodes

```
1:  function EPISODEMINING
2:      C_i ← Episodes of Size 1
3:      freq ← ∅
4:      i ← 1
5:      while C_i ≠ ∅ do
6:          Count frequencies of each Episode E ∈ C_i
7:          L_i ← {E | E ∈ C_i ∧ C_i is frequent}
8:          freq ← freq ∪ L_i
9:          C_{i+1} ← Generate Episode Candidates of length i+1 from L_i
10:         i ← i+1
11:     return freq
```

In summary, the general mining algorithm for frequent episodes requires:

- A definition of episode frequency, that does not violate the apriori principle
- An algorithm for counting episode frequency (of concrete candidates) according to this definition
- An algorithm to generate candidate episodes

It may be a bit confusing that we need a definition of episode frequency for such a mining algorithm. Since we have already defined what an occurrence of an episode looks like it would seem that counting all occurrences of an episode would yield its frequency. While this is a possible definition of frequency, it is important to note that finding all occurrences of an episode within a sequence is neither practical nor useful. An example will demonstrate the problem with this. Consider the simple serial episode $A \rightarrow B$ and a sequence of length $2 \cdot n$ which repeats the subsequence (A,B) n times. One quickly realizes that the number of episode occurrences is very large due to the possibility of overlapping episode occurrences. In this particular case there are already $\frac{n \cdot (n+1)}{2}$ possible occurrences. This number swiftly increases with the size of the episode pattern, since it introduces more potential overlappings. Naturally the number of possible parallel and composite episode occurrences is even larger, since they are less restrictive in the order of the events. Additionally, such a frequency definition would violate the apriori principle, since subepisodes can have less occurrences than their superepisodes. Consider the example sequence $[A,B,C,A,B,C]$ (timestamp values are left out). In this sequence there are 3 distinct occurences for the episode $A \rightarrow B$ and

four distinct occurrences for its superepisode $A \rightarrow B \rightarrow C$. These detrimental effects of this naive frequency definition are nicely summarized by Laxman et. al. in a paper presenting the non-overlapped frequency definition [5].
This leads to various frequency definitions of episodes in the literature, which will be dealt with in subsections 2.2.3 and 2.2.4. Each frequency definition comes with its own frequency counting algorithm. The procedure for generating candidates is independent of the frequency definition and is presented in 2.2.5.

2.2.3 Window based frequency

To the best of our knowledge the window based frequency was the first frequency definition for episodes to gain general popularity. It was conceived by Mannila et. al. [7], although the frequency counting algorithms were only mentioned in text form. The same authors specified the algorithms in a later paper [8], which acts as the primary source for the overview given in this subsection. In order to define the window based frequency we first need the notion of a time window:

Definition 16 ***Time Window*** *Given a sequence of events S we define the Time Window* $W(S,q,r)$ *with* $q,r \in \mathbb{N}^+$ *and* $q < r$ *as the ordered subsequence of S that includes all events of the annotated event stream S that have a timestamp t where* $q \leq t \leq r$*. We call* $w = r-q+1$ *the size of Window W.*

Definition 17 ***Episode Frequency - Window based Definition*** *Given a sequence of events S, a fixed window size of w and an episode* α*, we define the window based frequency* $w\_freq(\alpha)$ *as the number of windows W with size w of S in which* α *occurs:* $w\_freq(\alpha) = |\{W(S,q,r) \mid r-q+1 = w \wedge \alpha \text{ occurs in } W\}|$.

For example given a sequence $S = [(12,A),(14,B),(19,C),(22,A),(34,D)]$ the Episode $\alpha = B \rightarrow A$ occurs in window $W(S,14,22)$.
This definition can be confusing at first since it is intended that episode occurrences that are comprised of the exact same events count just as many times as there are windows in which the events appear. If we have a window size of $w = 11$ for the previously mentioned example, we can find the episode $B \rightarrow A$ in the consecutive windows $W(S,12,22)$, $W(S,13,23)$ and $W(S,14,24)$, which means we will get a frequency of 3 just for the two events $(14,B)$ and $(22,A)$. This effect obviously increases with the window size. Note that for each episode α we only count one occurrence per window W, no matter how many occurrences of α there are in W.

When determining the window based frequency the naive approach would be to check each window of the sequence separately. Since the windows are adjacent there is a better approach, which makes it possible to only iterate over the sequence once and determine the window based frequency for each candidate episode. Most papers focus purely on parallel and serial episodes and do not give an algorithm for composite episodes. The algorithms to determine the window based frequency of serial and parallel episodes are given in algorithm 2 and 3 respectively. The basic ideas are described below.

Frequency Counting of parallel Episodes

The algorithm for counting the frequency of parallel episodes uses the following data structures and variables:

- For each event type A store the number of occurrences of this event in the current window in $A.count$
- For each episode α store
  - $\alpha.freq$ - the number of windows in which alpha occurred so far
  - $\alpha.eventCount$ - the number of events of $alpha$ that are present in the current window
  - $\alpha.inwindow$ - this variable gets set to the current time whenever α becomes fully present ($alpha.eventCount = |\alpha|$).
- additionally maintain $contains$ which is a set of sets. Each set in $contains$ is identified by a tuple (T,i), where $T \in \Sigma$ is an event type and $i \in \mathbb{N}^+$. These sets are referred to as $contains(T,n)$. The set $contains(T,n)$ contains all candidate parallel episodes, which contain the event type T exactly n times. This is done to be able to efficiently access candidates, when certain events enter or leave the window.

The algorithm works as follows. First the above mentioned variables and index structures are initialized. Subsequently we perform the main loop which iterates through every point of time between the start and the end time of the sequence (TODO: they start earlier for some reason?). Essentially each iteration is sliding the window one step forwards. The only two things that need to be handled inside the loop are new events coming into the window and old events dropping out of the sliding window. If a new event comes in its count gets updated and each episode α that is affected by this will get its event count updated and, if it is completed, the current time will be saved in $\alpha.inwindow$.

If an event A drops out of the window, for all episodes α that occurred in the previous window(s) and now no longer have an occurrence in the current window (by losing A) the number of windows they were present in gets added to $\alpha.freq$.

Algorithm 2 Calculate Window based Frequency for parallel Episodes

Require: Let C be the set of candidate parallel episodes, and let $S = [(T_1, t_s), ..., (T_n, t_e)]$ be a sequence of events, let *win* be the window size and finally let *minS* be the minimum support.

1: *// Initialization*
2: **for** each $\alpha \in C$ **do**
3: **for** each $A \in \alpha$ **do**
4: $A.count \leftarrow 0$
5: **for** $i \in \{1, ..., |\alpha|\}$ **do**
6: $contains(A, i) \leftarrow \emptyset$
7: **for** each $\alpha \in C$ **do**
8: **for** each $A \in \alpha$ **do**
9: $a \leftarrow$ number of events of type A in α
10: $contains(A, a) \leftarrow contains(A, a) \cup \{\alpha\}$
11: $\alpha.eventCount \leftarrow 0$
12: $\alpha.freq \leftarrow 0$
13: *// Recognition*
14: **for** $start \leftarrow t_s - win + 1$ to t_e **do**
15: *//Bring new events to the window*
16: **for** each $(t, A) \in S$ where $t = start + win - 1$ **do**
17: $A.count \leftarrow A.count + 1$
18: **for** each $\alpha \in contains(A, A.count)$ **do**
19: $\alpha.eventCount \leftarrow \alpha.eventCount + A.count$
20: **if** $\alpha.eventCount = |\alpha|$ **then**
21: $\alpha.inwindow \leftarrow start$
22: *// Drop old events out of the window*
23: **for** each $(t, A) \in S$ where $t = start - 1$ **do**
24: **for** each $\alpha \in contains(A, A.count)$ **do**
25: **if** $\alpha.eventCount = |\alpha|$ **then**
26: $\alpha.freq \leftarrow \alpha.freq + \alpha.inwindow - start$
27: $\alpha.eventCount \leftarrow \alpha.eventCount - A.count$
28: $A.count \leftarrow A.count - 1$
29: **return** $\{\alpha \mid \alpha \in C \wedge \frac{\alpha.freq}{t_e - t_s + win - 1} \geq minS\}$

Frequency Counting of serial Episodes

The basic idea when determining the window based frequency of serial episodes is that a serial episodes can be recognized by using automaton that accepts the events of its corresponding serial episode in exactly the specified order and ignores all other input. For each serial episode α there can be several instances of the recognizing automaton (in different states) at the same time. This is necessary in order to replace old occurrences with newer ones whenever possible. The algorithm for counting the frequency of serial episodes uses the following data structures and variables:

- Each episode α is represented as an array in which the event types contained in α are stored in the correct order
- For each episode α the number of windows in which alpha occurred so far is stored in $\alpha.freq$
- An automaton is simply represented by a tuple (α, i), where α is the corresponding candidate serial episode and $i \in \{1, ..., |\alpha|\}$ is the state (position in the episode) in which the automaton currently is.
- The automata are grouped by the event type that will allow them to perform the next transition. These lists are referred to as $waits(T)$, where $T \in \Sigma$ is an event type.
- For each automata belonging to episode α that is currently in state i, the time at which it this automaton was initialized is stored in $\alpha.initialized[i]$.
- Additionally any automata that were initialized at point of time t will be contained in a list referred to as $beginsat(t)$.
- transitions to be made are stored in a list named transitions and are stored in the form (α, i, t), where α is the corresponding episode, i is the index of the state from which the automaton will transition to the next one and t is the time in which this automaton was initialized.

The structure of the algorithm is the same as the one of algorithm 2. First the variables are initialized, then the sequence is looped over and the sliding window moves by one time unit in each iteration. A new instance of an automaton is initialized, whenever an event A is the first event of an episode and enters the window. Additionally all automata that wait for A will be moved to the next state, while memorizing their initial starting time. If an automaton of episode α moves to a state which is already occupied by another automaton, the old automaton is discarded (since the newer one has a later starting time and thus will be present

in more windows). If an automaton reaches its final state the current time is memorized in $\alpha.inwindow$. If an automaton in its final state expires (its starting time drops out of the window) the number of windows it was present in is added to its corresponding sequence.

Algorithm 3 Calculate Window based Frequency for serial Episodes

Require: Let C be the set of candidate serial episodes, and let $S = [(T_1,t_s),...,(T_n,t_e)]$ be a sequence of events, let *win* be the window size and finally let *minS* be the minimum support. TODO: part of the algorithm is cut off

1: *// Initialization*
2: **for** each $\alpha \in C$ **do**
3: **for** $i \in \{1,...,|\alpha|\}$ **do**
4: $\alpha.initialized[i] \leftarrow 0$
5: $waits(\alpha[i]) \leftarrow \emptyset$
6: **for** each $\alpha \in C$ **do**
7: $waits(\alpha[1]) \leftarrow waits(\alpha[1]) \cup \{(\alpha,1)\}$
8: $\alpha.freq \leftarrow 0$
9: **for** $i \in \{t_s - win,...,t_s - 1\}$ **do**
10: $beginsat(t) \leftarrow \emptyset$
11: *// Recognition*
12: **for** $start \leftarrow t_s - win + 1$ to t_e **do**
13: *//Bring new events to the window*
14: $beginsat(start + win - 1) \leftarrow \emptyset$
15: $transitions \leftarrow \emptyset$
16: **for** each $(t,A) \in S$ where $t = start + win - 1$ **do**
17: **for** each $(\alpha, j) \in waits(A)$ **do**
18: **if** $j = |\alpha| \wedge \alpha.initialized[j] = 0$ **then**
19: $\alpha.inwindow \leftarrow start$
20: **if** $j = 1$ **then**
21: $transitions \leftarrow transitions \cup \{(\alpha, 1, start + win - 1\}$
22: **else**
23: $transitions \leftarrow transitions \cup \{(\alpha, j, initialized[j-1])\}$
24: Remove $(\alpha, j-1)$ from $beginsat(\alpha.initialized[j-1])$
25: $\alpha.initialized[j-1] \leftarrow 0$
26: Remove (α, j) from $waits(A)$
27: **for** each $(\alpha, j, t) \in transitions$ **do**
28: $\alpha.initialized[j] \leftarrow t$
29: $beginsat(t) \leftarrow beginsat(t) \cup \{(\alpha, j)\}$
30: **if** $j \leq |\alpha|$ **then**
31: $waits(\alpha[j+1]) \leftarrow waits(\alpha[j+1]) \cup \{(\alpha, j+1)\}$
32: *// Drop old events out of the window*
33: **for** each $(\alpha, l) \in beginsat(start - 1)$ **do**
34: **if** $l = |\alpha|$ **then**
35: $\alpha.freq \leftarrow \alpha.freq + start - \alpha.inwindow$
36: **else**
37: Remove $(\alpha, l+1)$ from $waits(\alpha[l+1])$

Naturally there are enhancements to these algorithms. For example both counting algorithms iterate over each point of time t between the start and end time of the sequence, regardless of the fact that there might not be anything to do at this point of time (no event drops out of the window and no new event comes in). Especially if the sequence of events is sparse, meaning that the time between events is usually large, this will become problematic. This issue can be fixed rather easily: instead of increasing *start* by one in each iteration, one can increase start by the amount of time that is needed until a change in the window occurs and update the data structures accordingly.
There is a notable absence of frequency counting algorithms for elementary (composite) episodes in literature. Mannila et. al. claim that each composite episode can be broken down into partial episodes, which are serial and/or parallel [8]. However they neither specify an algorithm for breaking down composite episodes into purely serial and parallel parts, nor do they specify a frequency counting algorithm for composite episodes. Subsequent research, such as alternate frequency definitions and counting algorithms has also mainly focused on parallel and serial episodes. If composite episodes have been studied they were usually studied in the above mentioned, more restrictive form of sequences of parallel episodes.

2.2.4 Other frequency definitions

In this subsection we briefly present alternative frequency definitions. However we do not present the respective counting algorithms, since they are not as relevant for this thesis. For the exact algorithms we refer the reader to the respective papers.
Most alternative definitions tried to move away from the ideas of fixed windows and tried to improve the performance of the counting algorithms.

Minimal Occurrence Based Frequency Definition

The first alternate definition does uses the concept of minimal occurrences:

Definition 18 ***Minimal Occurrence*** *An event episode α is said to occur minimally in a window $W(S,q,r)$ if α occurs in W and there is no subwindow of W in which α also occurs.*

Definition 19 ***Episode Frequency - Minimal Occurrence based Definition*** *Given a sequence of events S and an Episode α, we define the minimal occurrence based frequency $mo\_freq(\alpha)$ as the number of minimal occurrences of α in S. TODO: find and cite the original source*

The second alternative definition introduces the concept of non-overlapping occurrences:

Definition 20 ***Non-Overlapping Occurrences*** *Given a m-Episode $\alpha = (V_\alpha, \leq_\alpha, g_\alpha)$ where $V_\alpha = \{v_1, ..., v_m\}$, two occurrences h_1 and h_2 of α are non-overlapped if either*

- $\forall v_j \in V_\alpha : h_2(v_1) > h_1(v_j)$ *or*
- $\forall v_j \in V_\alpha : h_1(v_1) > h_2(v_j)$

A set of occurrences is non-overlapping if every pair of occurrences in it is non-overlapped [5].

This leads to the Definition:

Definition 21 ***Episode Frequency - Non-Overlapping Occurrences based Definition*** *Given a seqeunce of events S and an Episode α, we define the non-overlapping occurrence based frequency $noo\_freq(\alpha)$ as cardinality of the largest set of non-overlapped occurrences of α in S [5].*

When looking at these definitions in comparison to the window based frequency definition it is not clear whether any of these is always superior to or more useful than the other since they have different properties. We mention them briefly:

- As already mentioned the window based frequency counts an episode occurrence that is comprised of the same events in multiple windows. This might especially distort the count if the window size is high and the events in the episode happen with minimal delay between them.
- The minimal occurrence based definition of frequency does not suffer from the problem of the previous point
- The window based definition has the advantage that it already incorporates a fixed size during which episodes may occur, meaning there can not be episodes that stretch over a time period larger than the fixed window size w. This might be beneficial for potential algorithms, since it reduces the search space for episodes. On top of that it is also closer to reality, since in many domains episodes happen within a small time window [**?**].
- The non-overlapping occurrence based frequency offers the fastest counting algorithm of all three definitions. However when incorporating expiry times for serial episodes it looses this advantage. Additionally previous literature has not yet identified an efficient algorithm to count non-overlapped occurrences of parallel episodes with expiry times.

2.2.5 Candidate Generation

Most previous work generates candidates for serial and parallel episodes separately, using a levelwise approach for both cases. The candidate generation procedure presented below was originally specified by Manila et. al. [8]. A more detailed explanation can be found in Laxman's PHD thesis [4].
We first consider the case for parallel episodes. Given F_k as the set of all frequent parallel episodes of length k we can generate the candidate parallel episodes C_{k+1} of length $k+1$ by doing the following:

1. Represent each candidate $\alpha \in F_k$ as a lexicographically sorted array of length k.
2. For each unordered pair (α, β), $\alpha, \beta \in F_k$ where α and β share the same first $k-1$ nodes, generate candidate γ by copying α and appending $\beta[k]$.

For example the two frequent parallel episodes $A\|A\|B$ and $A\|A\|C$ will generate the candidate $A\|A\|C\|D$.
The same procedure can be applied to serial episodes, except that

- we do not order lexicographically, instead the serial episodes remain in the array in their natural order
- each pair (α, β) with the same properties as above now generates two candidates:
  - γ_1 by copying α and appending $\beta[k]$.
  - γ_2 by copying β and appending $\alpha[k]$.

Thus the two frequent serial episodes $A \to A \to B$ and $A \to A \to C$ will now generate the two candidates $A \to A \to B \to C$ and $A \to A \to C \to B$.
Since the mining of composite episodes has not received much attention, it is unsurprising that there little related work that mentions candidate generation strategies for these general types of episodes (TODO: refer to later chapter, since I will do that). For a more strict definition of composite episodes which only includes sequences of parallel episodes, Baumgarten et. al. use a tree growth strategy to generate candidates for composite episodes [3].

2.3 Related Work Overview

Needs rework, should I still give this kind of overview like I did in the last iteration? For now all

References

[1] Agrawal, R., Srikant, R., et al. (1994). Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499.

[2] Bathoorn, R. and Siebes, A. (2007). Finding composite episodes. In *International Workshop on Mining Complex Data*, pages 157–168. Springer.

[3] Baumgarten, M., Büchner, A. G., and Hughes, J. G. (2003). Tree growth based episode mining without candidate generation. In *IC-AI*, pages 108–114.

[4] Laxman, S. (2006). *Discovering frequent episodes: fast algorithms, connections with HMMs and generalizations*. PhD thesis, Indian Institute of Science Bangalore.

[5] Laxman, S., Sastry, P., and Unnikrishnan, K. (2007). A fast algorithm for finding frequent episodes in event streams. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 410–419. ACM.

[6] Luckham, D. and Schulte, R. (2011). Epts event processing glossary v2. 0. *Event Processing Technical Society*.

[7] Mannila, H., Toivonen, H., and Verkamo, A. I. (1995). Discovering frequent episodes in sequences extended abstract. In *Proceedings the first Conference on Knowledge Discovery and Data Mining*, pages 210–215.

[8] Mannila, H., Toivonen, H., and Verkamo, A. I. (1997). Discovery of frequent episodes in event sequences. *Data mining and knowledge discovery*, 1(3):259–289.