# Mining Partially-Ordered Episode Rules in an Event Sequence

Philippe Fournier-Viger[1][0000−0002−7680−9899], Yangming Chen[1], Farid Nouioua[2], and Jerry Chun-Wei Lin[3]

[1] Harbin Institute of Technology (Shenzhen), Shenzhen, China
philfv8@yahoo.com, 510717841@qq.com
[2] University of Bordj Bou Arreridj, Algeria
faridnouioua@gmail.com
[3] Department of Computing, Mathematics and Physics, Western Norway University of Applied Sciences (HVL), Bergen, Norway
jerrylin@ieee.org

**Abstract.** Episode rule mining is a popular data mining task for analyzing a sequence of events or symbols. It consists of identifying subsequences of events that frequently appear in a sequence and then to combine them to obtain episode rules that reveal strong relationships between events. But a key problem is that each rule requires a strict ordering of events. As a result, similar rules are treated differently, though they in practice often describe a same situation. To find a smaller set of rules that are more general and can replace numerous episode rules, this paper introduces a novel type of rules called partially-ordered episode rules, where events in a rule are partially ordered. To efficiently find all these rules in a sequence, an efficient algorithm named POERM (Partially-Ordered Episode Rule Miner) is presented. An experimental evaluation on several benchmark dataset shows that POERM has excellent performance.

**Keywords:** Pattern Mining · Sequence · Partially Ordered Episode Rule.

## 1 Introduction

Pattern mining is a sub-field of data mining, which aims at identifying interesting patterns in data that can help to understand the data and/or support decision-making. In recent years, numerous algorithms have been designed to find patterns in discrete sequences (a sequence of events or symbols) as this data type is found in many domains. For instance, text documents can be represented as sequence of words, customers purchases as sequences of transactions, and drone trajectories as sequence of locations. Whereas some pattern mining algorithms find similarities between sequences [6, 7, 15] or across sequences [16], others identify patterns in a single very long sequence. One of the most popular task of this type is *Frequent Episode Mining* (FEM) [9, 12, 11, 14]. It consists of finding all frequent episodes in a sequence of events, that is all subsequences

that have a *support* (occurrence frequency) that is no less than a user-defined *minsup* threshold. Two types of sequences are considered in FEM: *simple sequences* where events have timestamps and are totally ordered, and *complex sequences* where simultaneous events are allowed. Many algorithms were designed for discovering frequent episodes such as MINEPI and WINEPI [12], EMMA and MINEPI+ [9], and TKE [8]. While some algorithms find *serial episodes* (ordered lists of events), others find *parallel episodes* (sets of simultaneous events) or *composite episode* (a combination of serial/parallel episodes). Though finding frequent episodes is useful, episodes are only discovered on the basis of their support (occurrence frequencies) [1]. Thus, some events may only appear together in an episode by chance. Moreover, frequent episodes do not provide information about how likely it is that some events will occur following some other events. To address these issues, a post-processing step can be applied after FEM, which is to combine pairs of frequent episodes to create *episode rules*. An episode rule is a pattern having the form $E_1 \rightarrow E_2$, which indicates that if some episode $E_1$ appears, it will be followed by another episode $E_2$ with a given confidence or probability [4, 12].

Episode rule mining is useful as it can reveal strong temporal relationships between events in data from many domains [2–4, 12]. For example, a rule $R_1 : \langle\{a\},\{b\},\{c\}\rangle \rightarrow \langle\{d\}\rangle$ could be found in moviegoers data, indicating that if a person watches some movies $a$, $b$ and $c$ in that order, s/he will then watch movie $d$. Based on such rules, marketing decisions could be taken or recommendation could be done. However, a major drawback of traditional episode rule mining algorithms is that events in each rule must be strictly ordered. As a result, similar rules are treated differently. For example, the rule $R_1$ is considered as different from rules $R_2 : \langle\{b\},\{a\},\{c\}\rangle \rightarrow \langle\{d\}\rangle$, $R_3 : \langle\{b\},\{c\},\{a\}\rangle \rightarrow \langle\{d\}\rangle$, $R_4 : \langle\{c\},\{a\},\{b\}\rangle \rightarrow \langle\{d\}\rangle$, $R_5 : \langle\{c\},\{b\},\{a\}\rangle \rightarrow \langle\{d\}\rangle$ and $R_6 : \langle\{a\},\{c\},\{b\}\rangle \rightarrow \langle\{d\}\rangle$. But all these rules contain the same events. This is a problem because all these rules are very similar and may in practice represents the same situation that someone who has watched three movies (e.g. Frozen, Sleeping Beauty, Lion King) will then watch another (e.g. Harry Potter).Because these rules are viewed as distinct, their support (occurrence frequencies) and confidence are calculated separately and may be very different from each other. Moreover, analyzing numerous rules representing the same situation with slight ordering variations is not convenient for the user. Thus, it is desirable to extract a more general and flexible type of rules where ordering variations between events are tolerated.

This paper addresses this issue by introducing a novel type of rules called *Partially-Ordered Episode Rules* (POER), where events in a rule antecedent and in a rule consequent are unordered. A POER has the form $I_1 \rightarrow I_2$, where $I_1$ and $I_2$ are sets of events. A rule is interpreted as if all event(s) in $I_1$ appear in any order, they will be followed by all event(s) from $I_2$ in any order. For instance, a POER $R_7 : \{a, b, c\} \rightarrow \{d\}$ indicates that if someone watches movies $a$, $b$ and $c$ in any order, s/he will watch $d$. The advantage of finding POERs is that a single rule can replace multiple episode rules. For example $R_7$ can replace $R_1, R_2, \ldots, R_6$.

However, discovering POER is challenging as they are not derived from episodes. Thus, a novel algorithm must be designed to efficiently find POER in a sequence.

The contributions of this paper are the following. The problem of discovering POERs is defined and its properties are studied. Then, an efficient algorithm named POERM (Partially-Ordered Episode Rule Miner) is presented. Lastly, an experimental evaluation was performed on several benchmark datasets to evaluate POERM. Results have shown that it has excellent performance.

The rest of this paper is organized as follows. Section 2 defines the proposed problem of POER mining. Section 3 describes the POERM algorithm. Then, Section 4 presents the experimental evaluation. Finally, Section 5 draws a conclusion and discusses future work.

## 2 Problem Definition

The type of data considered in episode rule mining is a sequence of events with timestamps [9, 12]. Let there be a finite set of **events** $E = \{i_1, i_2, \ldots, i_m\}$, also called items or symbols. In addition, let there be a set of **timestamps** $T = \{t_1, t_2, \ldots t_n\}$ where for any integers $1 \leq i < j \leq n$, the relationship $t_i < t_j$ holds. A time-interval $[t_i, t_j]$ is said to have a duration of $t_j - t_i$ time. Moreover, two time intervals $[t_{i1}, t_{j1}]$ and $[t_{i2}, t_{j2}]$ are said to be **non-overlapping** if either $t_{j1} < t_{i2}$ or $t_{j2} < t_{i1}$. A subset $X \subseteq E$ is called an **event set**. Furthermore, $X$ is said to be a $k$-event set if it contains $k$ events. A **complex event sequence** is an ordered list of event sets with timestamps $S = \langle (SE_{t_1}, t_1), (SE_{t_2}, t_2), \ldots, (SE_{t_n}, t_n) \rangle$ where $SE_{t_i} \subseteq E$ for $1 \leq i \leq n$. A **simultaneous event set** in a complex event sequence is an event set where all events occurred at the same time. If a complex event sequence contains no more than one event per timestamp, it is a **simple event sequence**. Data of various types can be represented as an event sequence such as cyber-attacks, trajectories, telecommunication data, and alarm sequences [8].

For example, a complex event sequence is presented in Fig. 1. This sequence contains eleven timestamps ($\mathrm{T} = \{t_1, t_2, \ldots, t_{11}\}$) and events are represented by letters ($E = \{a, b, c, d\}$). That sequence indicates that event $c$ appeared at time $t_1$, followed by events $\{a, b\}$ simultaneously at time $t_2$, followed by event $d$ at time $t_3$, followed by $a$ at $t_5$, followed by $c$ at $t_6$, and so on.

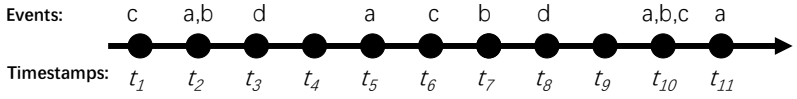


Fig. 1: An complex event sequence with 11 timestamps

This paper proposes a novel type of rules called **partially-ordered episode rule**. A POER has the form $X \rightarrow Y$ where $X \subset E$ and $Y \subset E$ are non empty

event sets. The meaning of such rule is that if all events from $X$ appear in any order in the sequence, they will be followed by all events from $Y$. To avoid finding rules containing events that are too far appart three constraints are specified: (1) events from $X$ must appear whithin some maximum amount of time $XSpan \in \mathbb{Z}^+$, (2) events from $Y$ must appear whithin some maximum amount of time $YSpan \in \mathbb{Z}^+$, and (3) the time between $X$ and $Y$ must be no less than a constant $XYSpan \in \mathbb{Z}^+$. The three constraints $XSpan, YSpan$ and $XYSpan$ must be specified by the user, and are illustrated in Fig. 2 for a rule $X \rightarrow Y$.

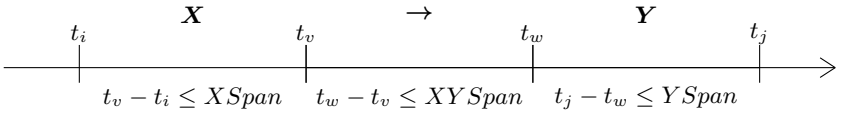


Fig. 2: The three time constraints on a POER

Furthermore, to select interesting rules, two measures are used called the support and confidence, which are inspired by previous work on rule mining. They are defined based on the concept of occurrence.

An **occurrence of an event set** $F \subset E$ in a complex event sequence $S$ is a time interval $[t_i, t_j]$ where all events from $F$ appear, that is $F \subseteq SE_i \cup SE_{i+1} \ldots \cup SE_j$. An **occurrence of a rule** $X \rightarrow Y$ in a complex event sequence $S$ is a time interval $[t_i, t_j]$ such that there exist some timestamps $t_v$, $t_w$ where $t_i \leq t_v < t_w \leq t_j$, $X$ has an occurrence in $[t_i, t_v]$, $Y$ has an occurrence in $[t_w, t_j]$, $t_v - t_i < XSpan$, $t_w - t_v < XYSpan$, and $t_j - t_w < YSpan$.

Analyzing occurrences of event sets or rules one can reveal interesting relationships between events. However, a problem is that some occurrences may overlap, and thus an event may be counted as part of multiple occurrences. To address this problem, this paper proposes to only consider a subset of all occurrences defined as follows. An occurrence $[t_{i1}, t_{j1}]$ is said to be **redundant** in a set of occurrences if there does exist an overlapping occurrence $[t_{i2}, t_{j2}]$ such that $t_{i1} \leq t_{i2} \leq t_{j1}$ or $t_{i2} \leq t_{i1} \leq t_{l2}$. Let $occ(F)$ denotes the **set of all non redundant occurrences of an event set** $F$ in a sequence $S$. Moreover, let $occ(X \rightarrow Y)$ denotes the **set of non redundant occurrences of a rule** $X \rightarrow Y$ in a sequence $S$.

The **support** of a rule $X \rightarrow Y$ is defined as $sup(X \rightarrow Y) = |occ(X \rightarrow Y)|$. The **support** of an event set $F$ is defined as $sup(F) = |occ(F)|$. The **confidence** of a rule $X \rightarrow Y$ is defined as $conf(X \rightarrow Y) = |occ(X \rightarrow Y)|/|occ(X)|$. It represents the conditional probability that events from $X$ are followed by those of $Y$.

**Definition 1 (Problem definition).** *XY Let there be a complex event sequence $S$ and five user-defined parameters: $XSpan$, $YSpan$, $XYSpan$, $minsup$ and $minconf$. The problem of mining POERs is to find all the valid POERs. A*

*POER $r$ is said to be frequent if $sup(r) \geq minsup$, and it is said to be valid if it is frequent and $conf(r) \geq minconf$.*

For instance, consider the sequence of Fig. 1, $minsup = 3$, $minconf = 0.6$, $XSpan = 3$, $XYSpan = 1$ and $YSpan = 1$. The occurrences of $\{a, b, c\}$ are $occ(\{a, b, c\}) = \{[t_1, t_2], [t_5, t_7], [t_{10}, t_{10}]\}$ The occurrences of the rule $R : \{a, b, c\} \rightarrow \{d\}$ are $occ(R) = \{[t_1, t_3], [t_5, t_8]\}$. Hence, $supp(R) = 3$, $conf(R) = 2/3$, and $R$ is a valid rule.

## 3 The POERM Algorithm

The problem of POER mining is difficult as if a database contains $m$ distinct events, up to $(2^m - 1) \times (2^m - 1)$ rules may be generated. Moreover, each rule may have numerous occurrences in a sequence.

To efficiently find all valid POERs, this section describes the proposed POERM algorithm. It first finds event sets that may be antecedents of valid rules and their occurrences in the input sequence. Then, the algorithm searches for consequents that could be combined with these antecedents to build POERs. The valid POERs are kept and returned to the user. To avoid considering all possible rules, the POERM algorithm utilizes the following search space pruning property (proof is omitted due to space limitation).

*Property 1 (Rule Event Set Pruning Property).* An event set $X$ cannot be the antecedent of a valid rule if $sup(X) < minsup$. An event set $Y$ cannot be the consequent of a valid rule if $sup(Y) < minsup \times minconf$.

The POERM algorithm (Algorithm 1) takes as input a complex event sequence $S$, and the user-defined $XSpan$, $YSpan$, $XYSpan$, $minsup$ and $minconf$ parameters. The POERM algorithm first reads the input sequence and creates a copy $XFres$ of that sequence containing only events having a support no less than $minsup$. Other events are removed because they cannot appear in a valid rule (based on Property 1). Then, POERM searches for antecedents by calling the *MiningXEventSet* procedure with $XFres$, $XSpan$ and $minsup$. This procedure outputs a list $xSet$ of event sets that may be antecedents of valid POERs, that is each event set having at least $minsup$ non overlapping occurrences of a duration not greater than $XSpan$.

The *MiningXEventSet* procedure is presented in Algorithm 2. It first scans the sequence $XFres$ to find the list of occurrences of each event. This information is stored in a map $fresMap$ where each pair indicates an event as key and its occurrence list as value. Then, scan all the pairs of $fresMap$ and put the pairs that have at least $minsup$ non-overlapping occurrences are added to $xSet$. At this moment, $xSet$ contains all 1-event sets that could be valid rule antecedents based on Property 1. Then, the procedure scans the sequence again to extend these 1-event sets into 2-event sets, and then extends 2-event sets into 3-event sets and so on until no more event sets can be generated. During that iterative

process, each generated event set having more than *minsup* non-overlapping occurrences is added to $xSet$ and considered for extensions to generate larger event sets. The *MiningXEventSet* procedure returns $xSet$, which contains all potential antecedents of valid POERs (those having at least *minsup* non-overlapping occurrences).

A challenge for implementing the *MiningXEventSet* procedure efficiently is that event sets are by definition unordered. Hence, different ordering of a same set represent the same event set (e.g. $\{a, b, c\}$,$\{b, a, c\}$ and $\{a, c, b\}$ are the same set). To avoid generating the same event set multiple times, event sets are in practice sorted by the lexicographic order (e.g. as $\{a, b, c\}$), and the procedure only extends an event set $F$ with an event $e$ if $e$ is greater than the last event of $F$.

A second key consideration for implementing $MiningXEventSet$ is how to extend an $l$-event set $F$ into $(l + 1)$-event sets and calculate their occurrences in the $XFres$ sequence. To avoid scanning the whole sequence, this is done by searching around each occurrence *pos* of $F$ in its $OccurrenceList$. Let *pos.start* and *post.end* respectively denote the start and end timestamps of *pos*. The algorithm searches for events in the time intervals $[pos.end - XSpan + 1, pos.start)$, $[pos.end + 1, pos.start + XSpan)$, and $[pos.start, pos.end]$. For each event $e$ that is greater than the last item of $F$, the occurrences of $F \cup \{e\}$ of the form $[i, pos.end]$, $[pos.start, i]$ or $[pos.start, pos.end]$ are added in $fresMap$. Then, $fresMap$ is scanned to count the non-overlaping occurrences of each $(l + 1)$-node event set, and all sets having more than *minsup* non-overlapping occurrences are added to $xSet$.

Counting the maximum non-overlapping occurrences of an $l$-event set $F$ is done using its occurrence list (not shown in the pseudocode). The procedure first applies the quick sort algorithm to sort the $OccurrenceList$ by ascending ending timestamps (*pos.end*). Then, a set $CoverSet$ is created to store the maximum non-overlapping occurrences of $F$. The algorithm loops over the $OccurrenceList$ of $F$ to check each occurrence from the first one to the last one. If the current occurrence does not overlap with the last added occurrence in $CoverSet$, it is added to $CoverSet$. Otherwise, it is ignored. When the loop finishes, $CoverSet$ contains the maximum non-overlapping occurrences of $F$.

After applying the $MiningXEventSet$ procedure to find antecedents, the POERM algorithm searches for event sets (consequents) that could be combined with these antecedents to create valid POERs. The algorithm first eliminates all events having less than $minsup \times minconf$ occurrences from the sequence $XFres$ to obtain a sequence $YFres$ (based on pruning Property 1). Then, a loop is done over each antecedent $x$ in $xSet$ to find its consequents. In that loop, the time intervals where such consequents could appear are first identified and stored in a variable $xOccurrenceList$. Then, a map $conseMap$ is created to store each event $e$ and its occurrence lists for these time intervals in $YFres$. The map is then scanned to create a queue $candidateRuleQueue$ containing each rule of the form $x \rightarrow e$ and its occurrence list. If a rule $x \rightarrow e$ is such that $|occ(x \rightarrow e)| \geq minconf \times |occ(x)|$, then it is added to the set $POERs$ of valid

**Algorithm 1:** POERM

**Input:** an event sequence $S$, the $XSpan$, $YSpan$, $XYSpan$, $minsup$ and $minconf$ parameters;
**Output:** the set $POERs$ of valid partially-ordered episode rules

```
1  XFres = loadFrequentSequence(S, minsup);
2  xSet = MiningXEventSet(XFres, XSpan, minsup);
3  YFres = loadFrequentSequence(XFres, minsup × minconf);
4  POERs ← ∅;
5  foreach event set x in xSet do
6      for i = 1 to (XYSpan + YSpan) do
7          xOccurrenceList ← {t | t = occur.end + i, occur ∈ x.OccurrenceList};
8      end
9      Scan each timestamp of YFres in xOccurrenceList to obtain a map
         conseMap that records each event e and its occurrence list;
10     Scan conseMap and put the pair (x → e, OccurrenceList) in a queue
         candidateRuleQueue (note: infrequent rules are kept because event e
         may be extended to obtain some frequent rules);
11     Add each rule x → e such that |occ(x → e)| ≥ minconf × |occur(x)| into
         the set POERs;
12     while candidateRuleQueue ≠ ∅ do
13         Pop a rule X → Y from candidateRuleQueue;
14         For each occurrence occur of X → Y, let;
15         start ← max(occur.X.end + 1, occur.Y.end − YSpan + 1);
16         end ← min(occur.X.end + XYSpan + YSpan,
             occur.X.start + YSpan);
17         Scan each timestamp in [start,end), add each candidate rule in
             candidateRuleQueue, and add each valid POER in POERs;
18     end
19 end
20 return POERs;
```

**Algorithm 2:** MiningXEventSet

**Input:** $XFres$: the sequence with only events having $support \geq minsup$; $XSpan$: maximum window size; $minsup$ threshold;
**Output:** a list of event sets that may be antecedents of valid POERs

```
1  Scan the sequence XFres to record the occurrence list of each event in a map
     fresMap (key = event set, value = occurrence list);
2  xSet ← all the pairs of fresMap such that |value| ≥ minsup;
3  start ← 0;
4  while start < |xSet| do
5  |  F ← xSet[start].getKey;
6  |  OccurrenceList ← xSet[start].getValue;
7  |  Clear fresMap;
8  |  start = start +1;
9  |  foreach occurrence pos in OccurrenceList do
10 |  |  pStart ← pos.end− XSpan +1; pEnd ← pos.start+ XSpan;
11 |  |  Search the time intervals [pStart, pos.start), [pos.end + 1, pEnd),
   |  |    [pos.start, pos.end] to add each event set F ∪ {e} such that
   |  |    e > F.lastItem and its occurrences of the forms [i, pos.end],
   |  |    [pos.start, i] or [pos.start, pos.end], in the map fresMap;
12 |  end
13 |  Add each pair of fresMap such that |value| ≥ minsup into xSet;
14 end
15 return xSet;
```

POERs. At this time, *candidateRuleQueue* contains all the candidate rule with a 1-event consequent.

The algorithm then performs a loop that pops each rule $X \rightarrow Y$ from the queue to try to extend it by adding an event to its consequent. This is done by scanning *conseMap*. The obtained rule extensions of the form $X \rightarrow Y \cup \{e\}$ and their occurrence lists are added to *candidateRuleQueue* to be further extended. Moreover, each rule such that $|occ(x \rightarrow e)| \geq minconf \times |occ(x)|$ is added to the set $POERs$ of valid POERs. This loop continues until the queue is empty. Then, the algorithm returns all valid POERs.

It is worth noting that an occurrence of a rule $X \rightarrow Y$ having an $l$-event set consequent may not be counted for its support while it may be counted in the support of a $(l + 1)$-event consequent rule that extends the former rule. For instance, consider the sequence of Fig. 1, that $XSpan = YSpan = 2$, and $XYSpan = 1$. The occurrence of $\{a\} \rightarrow \{b\}$ in $[t_5, t_7]$ is not counted in the support of $\{a\} \rightarrow \{b\}$ because the time between $a$ and $b$ is greater than $XYSpan$. But $\{a\} \rightarrow \{b\}$ can be extended to $\{a\} \rightarrow \{b, c\}$ and the occurrence $[t_5, t_7]$ is counted in its support. To find the correct occurrences of rules of the form $X \rightarrow Y \cup \{e\}$ extending a rule $X \rightarrow Y$, the following approach is used. For each occurrence *occur* of $X \rightarrow Y$, a variable *start* is set to max(occur.X.end+1,occur.Y.end-$YSpan$+1), a variable *end* is set to min(occur.X.end+$XYSpan$+$YSpan$, occur.Y.start+$YSpan$). Thereafter, three

intervals are scanned, which are $[start, occur.Y.end)$, $(occur.Y.end, end)$ and also $[occur.Y.start,\ occur.Y.end]$ to add each rule $X \rightarrow Y \cup \{e\}$ such that $e > Y.lastItem$ and its occurrences of the forms $[i, occur.Y.end]$, $[occur.Y.start, i]$ or $[occur.Y.start, occur.Y.end]$ in the $conseMap$. These three intervals are illustrated in Fig. 3 and allows to find the correct occurrences of $(l+1)$-node consequent rules. In that figure, $t_Y$ denotes $occur.Y$ and $t_X$ denotes $occur.X$.

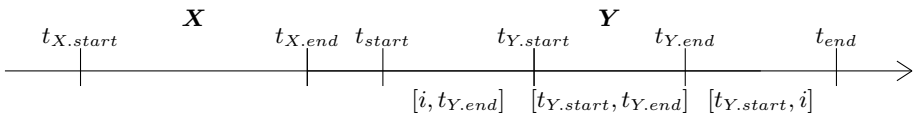


Fig. 3: The timestamps that are searched for a candidate rule $X \rightarrow Y$

The proposed POERM algorithm can find all valid POERs since it only prunes events that cannot be part of valid rules by Property 1. The following section presents an experimental evaluation of POERM, where its performance is compared with that of a baseline version of POERM, called POERM-ALL. The difference between POERM and POERM-ALL is that the latter finds all possible antecedents and consequents separately before combining them to generate rules, rather than using antecedents to search for consequents.

## 4 Experimental Evaluation

The proposed POERM algorithm's efficiency was evaluated on three benchmark sequence datasets obtained from the SPMF library [5], named *OnlineRetail, Fruithut* and *Retail*. *OnlineRetail* is a sequence of 541,909 transactions from a UK-based online retail store with 2,603 distinct event types indicating the purchase of items. *Fruithut* is a sequence of 181,970 transactions with 1,265 distinct event types, while *Retail* is a sequence of 88,162 customer transactions from an anonymous Belgian retail store having 16,470 distinct event types. Because there is no prior work on POER mining, the performance of POERM was compared with the POERM-ALL baseline, described in the previous section. Both algorithms were implemented in Java and source code and datasets are made available at `http://philippe-fournier-viger.com/spmf/`.

In each experiment, a parameter is varied while the other parameters are fixed. Because algorithms have five parameters and the space does not allow evaluating each parameter separately, the three time constraint parameters $XSpan$, $YSpan$ and $XYSpan$ were set to a same value called $Span$. The default values for ($minsup$, $minconf$, $Span$) on the *OnlineRetail*, *Fruithut* and *Retail* datasets are (5000, 0.5, 5), (5000, 0.5, 5) and (4000, 0.5, 5), respectively. These values were found empirically to be values were algorithms have long enough runtimes to highlight their differences.

**Influence of** *minsup*. Fig. 4(a) shows the runtimes of the two algorithms when *minsup* is varied. As *minsup* is decreased, runtimes of both algorithms increase. However, that of POERM-ALL increases much more quickly on the *OnlineRetail* and *FruitHut* datasets, while *minsup* has a smaller impact on POERM-ALL for *Retail*. The reason for the poor performance of POERM-ALL is that as *minsup* decreases, POERM-ALL considers more antecedent event sets ($sup(X) \geq minsup$) and more consequent event sets ($sup(Y) \geq minsup * minconf$). Moreover, POERM-ALL combines all antecedents with all consequents to then filter out non valid POERs based on the confidence. For instance, on *OnlineRetail* and $minsup = 6000$, there are 1,173 antecedents and 5,513 consequents, and hence 6,466,749 candidate rules. But for $minsup = 4000$, there are 2,813 antecedents, 11,880 consequents and 33,193,400 candidate rules, that is a five time increase. The POERM algorithm is generally more efficient as it does not combine all consequents will all antecedents. Hence, its performance is not directly influenced by the number of consequent event sets. This is why *minsup* has a relatively small impact on POERM's runtimes on the *OnlineRetail* and *FruitHut* dataset. However, *Retail* is a very sparse dataset. As a result for $minsup = 3000$, POERM-ALL just needs to combine 190 antecedents with 747 consequents, and thus spend little time in that combination stage. Because there are many consequents for *Retail*, POERM scans the input sequence multiple times for each antecedent to find all possible consequents. This is why POERM performs less well than POERM-ALL on *Retail*.

**Influence of** $minconf$. Fig. 4(b) shows the runtimes of the two algorithms for different $minconf$ values. It is observed that as $minconf$ is decreased, runtimes increase. POERM-ALL is more affected by variations of $minconf$ than POERM on *OnlineRetail* and *FruitHut* but the impact is smaller on *Retail*. The reason is that as $minconf$ decreases, POERM-ALL needs to consider more consequent event sets meeting the condition $sup(Y) \geq minsup \cdot minconf$, and thus the number of candidates rules obtained by combining antecedents and consequents increases. For instance, on *OnlineRetail* and $minconf = 0.6$, POERM-ALL finds 1,724 antecedents and 5,315 consequents. But for $minconf = 0.2$, the number of antecedents is the same, while POERM-ALL finds 46,236 consequents, increasing the number of candidate rules by nine times. On the other hand, $minconf$ does not have a big influence on POERM as it does not use a combination approach and hence is not directly influenced by the consequent count. On *Retail* the situation is different as it is a very sparse dataset. For each antecedent, POERM needs to scan the input sequence multiple time to find all the corresponding consequents. Thus, POERM performs less well on *Retail*.

**Influence of** *Span*. Fig. 4(c) compares the runtimes of both algorithms when *Span* is varied. As *Span* is increased, runtimes increase, and POERM-ALL is more affected by an increase of *Span* than POERM for *OnlineRetail* and *FruitHut*. But the impact on *Retail* is very small. This is because, as *Span* is increased, the time constraints are more loose and POERM-ALL needs to find more antecedents and consequent event sets, which results in generating more candidate rules by the combination process. For instance, on *OnlineRetail*

and $Span = 3$, POERM-ALL finds 44 antecedents and 2,278 consequents to generate 1,239,232 candidate rules. But for $Span = 7$, it finds 5,108 antecedents, 26,009 consequents and 132,853,972 rules (100 times increase). As POERM does not use this combination approach, its runtime is not directly influenced by the consequent count, and its runtime mostly increases with the antecedent count. Thus, changes in *Span* have a relatively small impact on POERM for *OnlineRetail* and *FruitHut*. But *Retail* is a sparse dataset. For each antecedent episode, POERM needs to scan the input sequence multiple times to make sure it finds all the corresponding consequents. Thus, POERM performs less well.

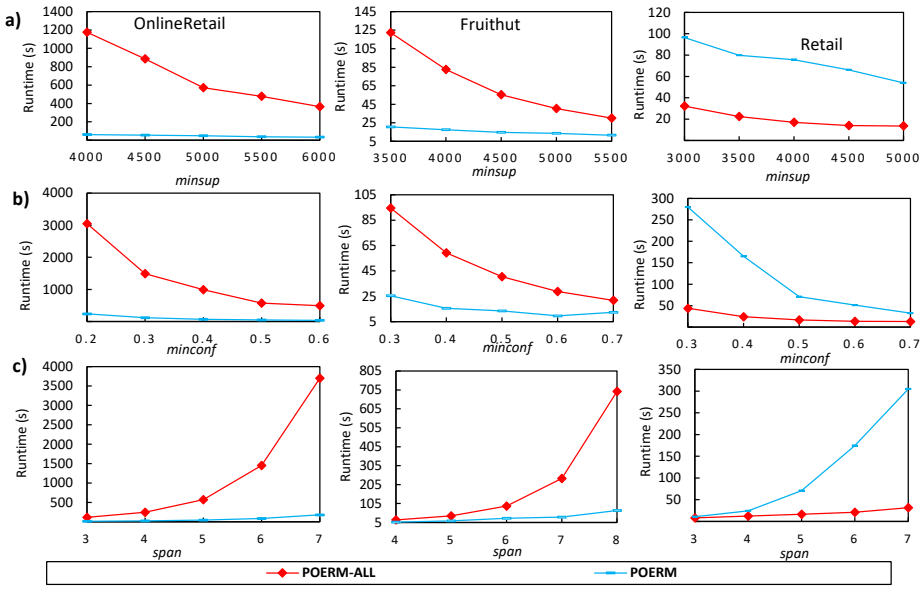


Fig. 4: Influence of (a) *minsup*, (b) *minconf* and (c) *span* on runtime

**Memory Consumption.** Fig. 5 shows the memory usage of both algorithms when *Span* is varied. Similarly to runtime, as *Span* is increased, memory usage increases because the search space grows. On *OnlineRetail* and *FruitHut*, POERM consumes from 25% to 200% less memory then POERM-ALL. But on sparse datasets like *Retail*, POERM needs to scan the input sequence multiple times and store antecedent episodes in memory to make sure it finds all the corresponding consequents. Thus, POERM consumes more memory then POERM-ALL on *Retail*.

**Discovered patterns.** Using the POERM algorithm, several rules were discovered in the data. For instance, some example rules found in the FruitHut dataset are shown in Table 1. Some of these rules have a high confidence. For example, the rule $CucumberLebanese, FieldTomatoes \rightarrow BananaCavendish$ has a confidence of $4910/6152 = 79.8\%$. Note that only a subset of rules in the

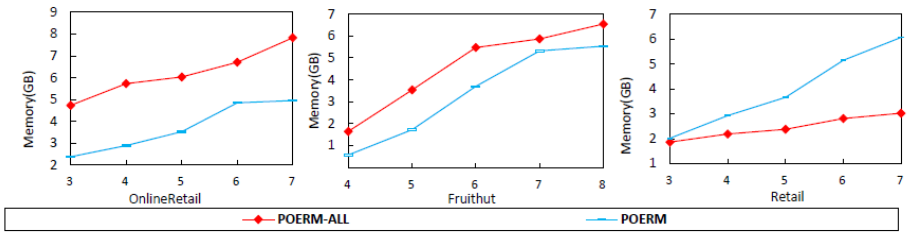


Fig. 5: Influence of *span* on Memory usage

table due to space limitation. The presented subset of rules is selected to give an overview of rules containing various items.

Table 1: Example rules from *FruitHut*

| Rule | occ(X → Y) | occ(X) |
|---|---|---|
| Cucumber Lebanese, Field Tomatoes→Banana Cavendish | 4910 | 6152 |
| Capsicum red, Field Tomatoes→Banana Cavendish | 5033 | 6352 |
| Broccoli, Capsicum red→Field Tomatoes | 2343 | 4043 |
| Nectarine White→Watermelon seedless | 2498 | 5687 |
| Garlic loose, Field Tomatoes→Capsicum red | 1752 | 4409 |
| Cucumber Lebanese, Capsicum red →Eggplant | 1236 | 4098 |

## 5 Conclusion

To find more general episode rules, this paper has proposed a novel type of rules called partially-ordered episode rules, where events in a rule are partially ordered. To efficiently find all these rules in a sequence, an efficient algorithm named POERM (Partially-Ordered Episode Rule Miner) was presented. An experimental evaluation on several benchmark dataset shows that POERM has excellent performance.

There are several possibilities for future work such as (1) extending POERM to process streaming data or run on a big data or multi-thread environment to benefit from parallelism, (2) considering more complex data such as events that are organized according to a taxonomy [3] or a stream [17], and (3) developing a sequence prediction model based on POERs. Other pattern selection functions will also be considered such as the utility [13, 18] and rarity [10].

## References


1. Ao, X., Luo, P., Li, C., Zhuang, F., He, Q.: Online frequent episode mining. In: 2015 IEEE 31st International Conference on Data Engineering. pp. 891–902. IEEE (2015)
2. Ao, X., Luo, P., Wang, J., Zhuang, F., He, Q.: Mining precise-positioning episode rules from event sequences. IEEE Transactions on Knowledge and Data Engineering **30**(3), 530–543 (2017)
3. Ao, X., Shi, H., Wang, J., Zuo, L., Li, H., He, Q.: Large-scale frequent episode mining from complex event sequences with hierarchies. ACM Transactions on Intelligent Systems and Technology (TIST) **10**(4), 1–26 (2019)
4. Fahed, L., Brun, A., Boyer, A.: Deer: Distant and essential episode rules for early prediction. Expert Systems with Applications **93**, 283–298 (2018)
5. Fournier-Viger, P., Lin, J.C.W., Gomariz, A., Gueniche, T., Soltani, A., Deng, Z., Lam, H.T.: The spmf open-source data mining library version 2. In: Joint European conference on machine learning and knowledge discovery in databases. pp. 36–40. Springer (2016)
6. Fournier-Viger, P., Lin, J.C.W., Kiran, U.R., Koh, Y.S.: A survey of sequential pattern mining. Data Science and Pattern Recognition **1**(1), 54–77 (2017)
7. Fournier-Viger, P., Wu, C.W., Tseng, V.S., Cao, L., Nkambou, R.: Mining partially-ordered sequential rules common to multiple sequences. IEEE Transactions on Knowledge and Data Engineering **27**(8), 2203–2216 (2015)
8. Fournier-Viger, P., Yang, Y., Yang, P., Lin, J.C.W., Yun, U.: Tke: Mining top-k frequent episodes. In: Proc. 33rd Intern. Conf. on Industrial, Engineering and Other Applications of Applied Intelligent Systems. Springer (2020)
9. Huang, K., Chang, C.: Efficient mining of frequent episodes from complex sequences. Inf. Syst. **33**(1), 96–114 (2008)
10. Koh, Y.S., Ravana, S.D.: Unsupervised rare pattern mining: a survey. ACM Transactions on Knowledge Discovery from Data (TKDD) **10**(4), 1–29 (2016)
11. Lin, Y.F., Huang, C.F., Tseng, V.S.: A novel methodology for stock investment using high utility episode mining and genetic algorithm. Applied Soft Computing **59**, 303–315 (2017)
12. Mannila, H., Toivonen, H., Verkamo, A.I.: Discovering frequent episodes in sequences. In: Proc. 1st Int. Conf. on Knowledge Discovery and Data Mining (1995)
13. Song, W., Huang, C.: Mining high average-utility itemsets based on particle swarm optimization. Data Science and Pattern Recognition **4**(2), 19–32 (2020)
14. Su, M.Y.: Applying episode mining and pruning to identify malicious online attacks. Computers & Electrical Engineering **59**, 180–188 (2017)
15. Truong, T., Duong, H., Le, B., Fournier-Viger, P.: Fmaxclohusm: An efficient algorithm for mining frequent closed and maximal high utility sequences. Engineering Applications of Artificial Intelligence **85**, 1–20 (2019)
16. Wenzhe, L., Qian, W., Luqun, Y., Jiadong, R., Davis, D.N., Changzhen, H.: Mining frequent intra-sequence and inter-sequence patterns using bitmap with a maximal span. In: Proc. 14th Web Inf. Syst. and Applications Conf. pp. 56–61. IEEE (2017)
17. You, T., Li, Y., Sun, B., Du, C.: Multi-source data stream online frequent episode mining. IEEE Access **8**, 107465–107478 (2020)
18. Yun, U., Nam, H., Lee, G., Yoon, E.: Efficient approach for incremental high utility pattern mining with indexed list structure. Future Generation Computer Systems **95**, 221–239 (2019)