Discovery of Frequent Episodes in Event Logs

Maikel Leemans and Wil M.P. van der Aalst

Eindhoven University of Technology, P.O. Box 513, 5600 MB, Eindhoven, The Netherlands. m.leemans@tue.nl,w.m.p.v.d.aalst@tue.nl

Abstract. Lion's share of process mining research focuses on the discovery of end-to-end process models describing the characteristic behavior of observed cases. The notion of a process instance (i.e., the case) plays an important role in process mining. Pattern mining techniques (such as frequent itemset mining, association rule learning, sequence mining, and traditional episode mining) do not consider process instances. An episode is a collection of partially ordered events. In this paper, we present a new technique (and corresponding implementation) that discovers frequently occurring *episodes* in event logs thereby exploiting the fact that events are associated with cases. Hence, the work can be positioned in-between process mining and pattern mining. Episode discovery has its applications in, amongst others, discovering local patterns in complex processes and conformance checking based on partial orders. We also discover episode rules to predict behavior and discover correlated behaviors in processes. We have developed a ProM plug-in that exploits efficient algorithms for the discovery of frequent episodes and episode rules. Experimental results based on real-life event logs demonstrate the feasibility and usefulness of the approach.

1 Introduction

Process mining provides a powerful way to analyze operational processes based on event data. Unlike classical purely model-based approaches (e.g., simulation and verification), process mining is driven by "raw" observed behavior instead of assumptions or aggregate data. Unlike classical data-driven approaches, process mining is truly process-oriented and relates events to high-level end-to-end process models [1].

In this paper, we use ideas *inspired by episode mining [2] and apply these to the discovery of partially ordered sets of activities in event logs. Event logs* serve as the starting point for process mining. An event log can be viewed as a multiset of *traces* [1]. Each trace describes the life-cycle of a particular *case* (i.e., a *process instance*) in terms of the *activities* executed. Often event logs store additional information about events, e.g., the *resource* (i.e., person or device) executing or initiating the activity, the *timestamp* of the event, or *data elements* (e.g., cost or involved products) recorded with the event.

Each trace in the event log describes the life-cycle of a case from start to completion. Hence, process discovery techniques aim to transform these event logs into *end-to-end process models*. Often the overall end-to-end process model is rather complicated because of the variability of real life processes. This results in "Spaghetti-like" diagrams. Therefore, it is interesting to also search for more local patterns in the event log – using episode discovery – while still exploiting the notion of process instances. Another useful application of episode discovery is conformance checking based on partial orders [3].

Since the seminal papers related to the Apriori algorithm [4, 5, 6], many pattern mining techniques have been proposed. These techniques do not consider the ordering of events [4] or assume an unbounded stream of events [5, 6] without considering process instances. Mannila et al. [2] proposed an extension of sequence mining [5, 6] allowing for partially ordered events. An episode is a partially ordered set of activities and it is frequent if it is "embedded" in many sliding time windows. Unlike in [2], our episode discovery technique does not use an arbitrary sliding window. Instead, we exploit the notion of process instances. Although the idea is fairly straightforward, as far as we know, this notion of frequent episodes was never applied to event logs.

Numerous applications of process mining to real-life event logs illustrate that *concurrency* is a key notion in process discovery [1, 7, 8]. One should avoid showing all observed *interleavings* in a process model. First of all, the model gets too complex (think of the classical "state-explosion problem"). Second, the resulting model will be overfitting (typically one sees only a fraction of the possible interleavings). This makes the idea of episode mining particularly attractive.

The remainder of this paper is organized as follows. Section 2 positions the work in existing literature. The novel notion of episodes and the corresponding rules are defined in Section 3. Section 4 describes the algorithms and corresponding implementation in the process mining framework *ProM*. The approach and implementation are evaluated in Section 5 using several publicly available event logs. Section 6 concludes the paper.

2 Related Work

The notion of frequent episode mining was first defined by Mannila et al. [2]. In their paper, they applied the notion of frequent episodes to (large) event sequences. The basic pruning technique employed in [2] is based on the frequency of episodes in an event sequence. Mannila et al. considered the mining of serial and parallel episodes separately, each discovered by a distinct algorithm. Laxman and Sastry improved on the episode discovery algorithm of Mannila by employing new frequency calculation and pruning techniques [9]. Experiments suggest that the improvement of Laxman and Sastry yields a 7 times speedup factor on both real and synthetic datasets.

Related to the discovery of episodes or partial orders is the discovery of end-to-end process models able to capture concurrency explicitly. The α algorithm [10] was the first process discovery algorithm adequately handling concurrency. Many other discovery techniques followed, e.g., *heuristic* mining [11] able to deal with noise and low-frequent behavior. The HeuristicsMiner is based on the notion of causal nets (C-nets). Several variants of the α algorithm have been proposed [12, 13]. Moreover, completely different approaches have been proposed, e.g., the different types of *genetic* process mining [14, 15], techniques based on *state-based regions* [16, 17], and techniques based on *language-based regions* [18, 19]. Another, more recent, approach is *inductive* process mining where the event log is split recursively [20]. The latter technique always produces a block-structured and sound process model. All the discovery techniques mentioned are able to uncover concurrency based on example behavior in the log. Additional feature comparisons are summarised in Table 1.

The episode mining technique presented in this paper is based on the discovery of frequent item sets. A well-known algorithm for mining frequent item sets and association rules is the Apriori algorithm by Agrawal and Srikant [4]. One of the

pitfalls in association rule mining is the huge number of solutions. One way of dealing with this problem is the notion of representative association rules, as described by Kryszkiewicz [21]. This notion uses user specified constraints to reduce the number of 'similar' results. Both sequence mining [5, 6] and episode mining [2] can be viewed as extensions of frequent item set mining.

	Exploits process instances	Mines end-to-end model	Soundness guaranteed	Sequence	Choice	Concurrency	Silent (tau) transitions	Duplicate Activities
Agrawal, Sequence mining [4]	-	-	n.a.	+	-	-	-	-
Manilla, Episode mining [2]	-	-	n.a.	+	-	+	-	-
Leemans M., Episode discovery	+	-	n.a.	+	-	+	-	+
Van der Aalst, α-algorithm [10]	+	+	-	+	+	+	-	-
Weijters, Heuristics mining [11]	+	+	-	+	+	+	-	-
De Medeiros, Genetic mining [14, 15]	+	+	-	+	+	+	+	+
Solé, State Regions [16, 17]	+	+	-	+	+	+	-	-
Bergenthum, Language Regions [18, 19]	+	+	-	+	+	+	-	-
Leemans S.J.J., Inductive [20]	+	+	+	+	+	+	+	-

Table 1. Feature comparison of discussed discovery algorithms

3 Event Logs, Episodes, and Episode Rules

This section defines basic notions such as event logs, episodes and rules. Note that our notion of episodes is different from the notion in [2] which does not consider process instances.

3.1 Event Logs

Activities and Traces Let $\mathcal{A}$ be the alphabet of activities. A trace is a list (sequence) $T = \langle A_1, \ldots, A_n \rangle$ of activities $A_i \in \mathcal{A}$ occurring at time index i relative to the other activities in T.

Event log An event log $L = [T_1, \ldots, T_m]$ is a multiset of traces T_i. Note that the same trace may appear multiple times in an event log. Each trace corresponds to an execution of a process, i.e., a *case* or *process instance*. In this simple definition of an event log, an event refers to just an *activity*. Often event logs store additional information about events, such as *timestamps*.

3.2 Episodes

Episode An episode is a partial ordered collection of events. Episodes are depicted using the transitive reduction of directed acyclic graphs, where the nodes represent events, and the edges imply the partial order on events. Note that the presence of an edge implies serial behavior. Figure 1 shows the transitive reduction of an example episode.

Formally, an episode $\alpha = (V, \leq, g)$ is a triple, where V is a set of events (nodes), $\leq$ is a partial order on V, and $g : V \mapsto \mathcal{A}$ is a left-total function from events to activities, thereby labelling the nodes/events [2]. For two vertices $u, v \in V$ we have $u < v$ iff $u \leq v$ and $u \neq v$. In addition, we define G to be the multiset of activities/labels used: $G = [\, g(v) \mid v \in V \,]$. Note that if $|V| \leq 1$, then we got an singleton or empty episode. For the rest of this paper, we ignore empty episodes. We call an episode *parallel* when $\leq = \emptyset$.

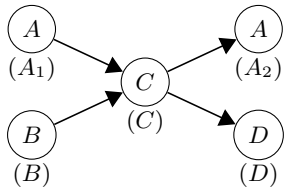


Fig. 1. Shown is the transitive reduction of the partial order for an example episode. The circles represent nodes (events), with the activity labelling imposed by g inside the circles, and an event ID beneath the nodes in parenthesis. In this example, events A_1 and B can happen in parallel (as can A_2 and D), but event C can only happen after both A_1 and B have occurred.

Subepisode and Equality An episode $\beta = (V', \leq', g')$ is a subepisode of $\alpha = (V, \leq, g)$, denoted $\beta \preceq \alpha$, iff there is an injective mapping $f : V' \mapsto V$ such that:

$$(\forall v \in V' : g'(v) = g(f(v))) \wedge (\forall v, w \in V' \wedge v \leq' w : f(v) \leq f(w))$$

An episode β equals episode α, denoted $\beta = \alpha$ iff $\beta \preceq \alpha \wedge \alpha \preceq \beta$. An episode β is a strict subepisode of α, denoted $\beta \prec \alpha$, iff $\beta \preceq \alpha \wedge \beta \neq \alpha$.

Episode construction Two episodes $\alpha = (V, \leq, g)$ and $\beta = (V', \leq', g')$ can be 'merged' to construct a new episode $\gamma = (V^*, \leq^*, g^*)$. $\alpha \oplus \beta$ is the smallest γ (i.e., smallest sets V^* and $\leq^*$) such that $\alpha \preceq \gamma$ and $\beta \preceq \gamma$. As shown below, such an episode γ always exists.

The smallest sets criteria implies that every event $v \in V^*$ and ordered pair $v, w \in V^* \wedge v \leq^* w$ must have a witness in α and/or β. Formally, $\gamma = \alpha \oplus \beta$ iff there exists injective mappings $f : V \mapsto V^*$ and $f' : V' \mapsto V^*$ such that:

$$G^* = G \cup G' \qquad \text{activity witness}$$

$$\leq^* = \{\, (f(v), f(w)) \mid (v, w) \in \leq \,\} \cup \{\, (f'(v), f'(w)) \mid (v, w) \in \leq' \,\} \qquad \text{order witness}$$

Occurrence An episode $\alpha = (V, \leq, g)$ occurs in an event trace $T = \langle A_1, \ldots, A_n \rangle$, denoted $\alpha \sqsubseteq T$, iff there exists an injective mapping $h : V \mapsto \{1, .., n\}$ such that:

$$(\forall v \in V : g(v) = A_{h(v)}) \wedge (\forall v, w \in V \wedge v \leq w : h(v) \leq h(w))$$

In Figure 2 an example of an "event to trace map" h for occurrence checking is given.

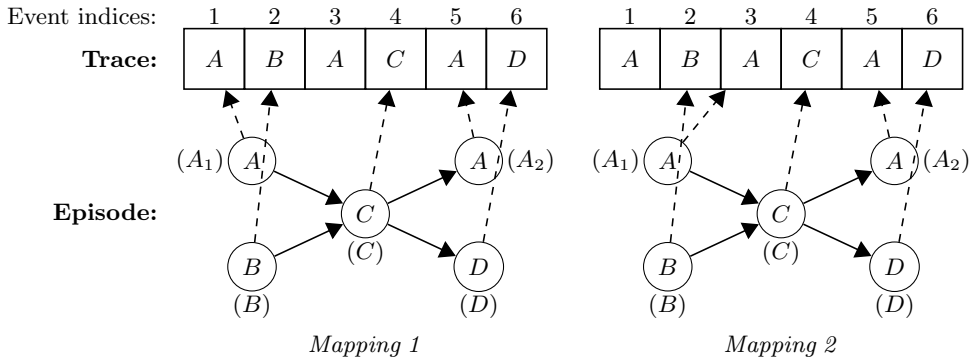


Fig. 2. Shown are two possible mappings h (the dotted arrows) for checking occurrence of the example episode in a trace. The shown graphs are the transitive reduction of the partial order of the example episode. Note that with the left mapping (*Mapping 1*) also an episode with the partial order $A_1 < B$ occurs in the given trace, in the right mapping (*Mapping 2*) the same holds for an episode with the partial order $B < A_1$.

Frequency The frequency $\mathit{freq}(\alpha)$ of an episode α in an event log $L = [T_1, \ldots, T_m]$ is defined as:

$$\mathit{freq}(\alpha) = \frac{|\,[\,T_i \mid T_i \in L \wedge \alpha \sqsubseteq T_i\,]\,|}{|L|}$$

Given a frequency threshold *minFreq*, an episode α is frequent iff $\mathit{freq}(\alpha) \geq \mathit{minFreq}$. During the actual episode discovery, we use the fact given in Lemma 1.

Lemma 1 (Frequency and subepisodes). *If an episode α is frequent in an event log L, then all subepisodes β with $\beta \preceq \alpha$ are also frequent in L. Formally, we have for a given α:*

$$(\forall \beta \preceq \alpha : \mathit{freq}(\beta) \geq \mathit{freq}(\alpha))$$

Activity Frequency The activity frequency $\mathit{ActFreq}(A)$ of an activity $A \in \mathcal{A}$ in an event log $L = [T_1, \ldots, T_m]$ is defined as:

$$\mathit{ActFreq}(A) = \frac{|\,[\,T_i \mid T_i \in L \wedge A \in T_i\,]\,|}{|L|}$$

Given a frequency threshold *minActFreq*, an activity A is frequent iff $\mathit{ActFreq}(A) \geq \mathit{minActFreq}$.

Trace Distance Given episode $\alpha = (V, \leq, g)$ occurring in an event trace $T = \langle A_1, \ldots, A_n \rangle$, as indicated by the event to trace map $h : V \mapsto \{1, .., n\}$. Then the trace distance $\mathit{traceDist}(\alpha, T)$ is defined as:

$$\mathit{traceDist}(\alpha, T) = \max\{\, h(v) \mid v \in V \,\} - \min\{\, h(v) \mid v \in V \,\}$$

In Figure 2, the left mapping yields $\mathit{traceDist}(\alpha, T) = 6 - 1 = 5$, and the right mapping yields $\mathit{traceDist}(\alpha, T) = 6 - 2 = 4$.

Given a trace distance interval $[minTraceDist, maxTraceDist]$, an episode α is accepted in trace T with respect to the trace distance interval iff $minTraceDist \leq traceDist(\alpha, T) \leq maxTraceDist$.

Informally, the conceptual idea behind a trace distance interval is that we are interested in a partial order on events occurring relatively close in time.

3.3 Episode Rules

Episode rule An episode rule is an association rule $\beta \Rightarrow \alpha$ with $\beta \prec \alpha$ stating that after seeing β, then likely the larger episode α will occur as well.

The confidence of the episode rule $\beta \Rightarrow \alpha$ is given by:

$$conf(\beta \Rightarrow \alpha) = \frac{freq(\alpha)}{freq(\beta)}$$

Given a confidence threshold $minConf$, an episode rule $\beta \Rightarrow \alpha$ is valid iff $conf(\beta \Rightarrow \alpha) \geq minConf$. During the actual episode rule discovery, we use Lemma 2.

Lemma 2 (Confidence and subepisodes). *If an episode rule $\beta \Rightarrow \alpha$ is valid in an event log L, then for all episodes β' with $\beta \prec \beta' \prec \alpha$ the event rule $\beta' \Rightarrow \alpha$ is also valid in L. Formally:*

$$(\forall \beta \prec \beta' \prec \alpha : conf(\beta \Rightarrow \alpha) \leq conf(\beta' \Rightarrow \alpha))$$

Episode rule magnitude Let the graph size $size(\alpha)$ of an episode α be denoted as the sum of the nodes and edges in the transitive reduction of the episode. The magnitude of an episode rule is defined as:

$$mag(\beta \Rightarrow \alpha) = \frac{size(\beta)}{size(\alpha)}$$

Intuitively, the magnitude of an episode rule $\beta \Rightarrow \alpha$ represents how much episode α 'adds to' or 'magnifies' episode β. The magnitude of an Episode rule allows smart filtering on generated rules. Typically, an extremely low (approaching zero) or high (approaching one) magnitude indicates a trivial episode rule.

4 Realization

The definitions and insights provided in the previous section have been used to implement a episode (rule) discovery plug-in in *ProM*. To be able to analyze real-life event logs, we need efficient algorithms. These are described next.

Notation: in the listed algorithms, we will reference to the elements of an episode $\alpha = (V, \leq, g)$ as $\alpha.V$, $\alpha.\leq$ and $\alpha.g$.

4.1 Frequent Episode Discovery

Discovering frequent episodes is done in two phases. The first phase discovers parallel episodes (i.e., nodes only), the second phase discovers partial orders (i.e., adding the edges). The main routine for discovering frequent episodes is given in Algorithm 1.

Algorithm 1: Episodes discovery

Input: An event log L, an activity alphabet $\mathcal{A}$, a frequency threshold *minFreq*.
Output: A set of frequent episodes Γ
Description: Two-phase episode discovery. Each phase alternates by generating new candidate episodes (C_l), and recognizing frequent candidates in the event log (F_l).
Proof of termination: Note that candidate episode generation with $F_l = \emptyset$ will yield $C_l = \emptyset$. Since each iteration the generated episodes become strictly larger (in terms of V and $\leq$), eventually the generated episodes cannot occur in any trace. Therefore, always eventually $F_l = \emptyset$, and thus we will always terminate.

EpisodeDiscovery(L, $\mathcal{A}$, *minFreq*)

(1) $\Gamma = \emptyset$
(2) *// Phase 1: discover parallel episodes*
(3) $l = 1$ *// Tracks the number of nodes*
(4) $C_l = \{ (V, \leq = \emptyset, g = \{v \mapsto a\}) \mid |V| = 1 \wedge v \in V \wedge a \in \mathcal{A} \}$
(5) **while** $C_l \neq \emptyset$
(6) $\quad F_l =$ RecognizeFrequentEpisodes(L, C_l, *minFreq*)
(7) $\quad \Gamma = \Gamma \cup F_l$
(8) $\quad C_l =$ GenerateCandidateParallel(l, F_l)
(9) $\quad l = l + 1$
(10) *// Phase 2: discover partial orders*
(11) $l = 1$ *// Tracks the number of edges*
(12) $C_l = \{ (V = \gamma.V, \leq = \{(v, w)\}, g = \gamma.g) \mid \gamma \in \Gamma \wedge v, w \in \gamma.V \wedge v \neq w \}$
(13) **while** $C_l \neq \emptyset$
(14) $\quad F_l =$ RecognizeFrequentEpisodes(L, C_l, *minFreq*)
(15) $\quad \Gamma = \Gamma \cup F_l$
(16) $\quad C_l =$ GenerateCandidateOrder(l, F_l)
(17) $\quad l = l + 1$
(18) **return** Γ

4.2 Episode Candidate Generation

The generation of candidate episodes for each phase is an adaptation of the well-known Apriori algorithm over an event log. Given a set of frequent episodes F_l, we can construct a candidate episode γ by combining two partially overlapping episodes α and β from F_l. Note that this implements the episode construction operation $\gamma = \alpha \oplus \beta$.

For phase 1, we have F_l contains frequent episodes with l nodes and no edges. A candidate episode γ will have $l + 1$ nodes, resulting from episodes α and β that overlap on the first $l - 1$ nodes. This generation is implemented by Algorithm 2.

For phase 2, we have F_l contains frequent episodes with l edges. A candidate episode γ will have $l + 1$ edges, resulting from episodes α and β that overlap on the first $l - 1$ edges and have the same set of nodes. This generation is implemented by Algorithm 3. Note that, formally, the partial order $\leq$ is the transitive closure of the set of edges being constructed, and that the edges are really only the transitive reduction of this partial order.

Algorithm 2: Candidate episode generation – Parallel

Input: A set of frequent episodes F_l with l nodes.
Output: A set of candidate episodes C_{l+1} with $l+1$ nodes.
Description: Generates candidate episodes γ by merging overlapping episodes α and β (i.e., $\gamma = \alpha \oplus \beta$). For parallel episodes, overlapping means: sharing $l-1$ nodes.

GenerateCandidateParallel(l, F_l)
(1) $C_{l+1} = \emptyset$
(2) **for** $i = 0$ **to** $|F_l| - 1$
(3) **for** $j = i$ **to** $|F_l| - 1$
(4) $\alpha = F_l[i]$
(5) $\beta = F_l[j]$
(6) **if** $\forall 0 \leq i \leq l-2 : \alpha.g(\alpha.V[i]) = \beta.g(\beta.V[i])$
(7) $\gamma = (V = (\alpha.V[0\,..\,l-1] \cup \beta.V[l-1]), \leq = \emptyset, g = \alpha.g \cup \beta.g)$
(8) $C_{l+1} = C_{l+1} \cup \{\gamma\}$
(9) **else**
(10) **break**
(11) **return** C_{l+1}

Algorithm 3: Candidate episode generation – Partial order

Input: A set of frequent episodes F_l with l edges.
Output: A set of candidate episodes C_{l+1} with $l+1$ edges.
Description: Generates candidate episodes γ by merging overlapping episodes α and β (i.e., $\gamma = \alpha \oplus \beta$). For partial order episodes, overlapping means: sharing all nodes and $l-1$ edges.

GenerateCandidateOrder(l, F_l)
(1) $C_{l+1} = \emptyset$
(2) **for** $i = 0$ **to** $|F_l| - 1$
(3) **for** $j = i+1$ **to** $|F_l| - 1$
(4) $\alpha = F_l[i]$
(5) $\beta = F_l[j]$
(6) **if** $\alpha.V = \beta.V \wedge \alpha.g = \beta.g \wedge \alpha.{\leq}[0\,..\,l-2] = \beta.{\leq}[0\,..\,l-2]$
(7) $\gamma = (V = \alpha.V, \leq = (\alpha.E[0\,..\,l-1] \cup \beta.E[l-1]), g = \alpha.g)$
(8) $C_{l+1} = C_{l+1} \cup \{\gamma\}$
(9) **else**
(10) **break**
(11) **return** C_{l+1}

4.3 Frequent Episode Recognition

In order to check if a candidate episode α is frequent, we check if $\mathit{freq}(\alpha) \geq \mathit{minFreq}$. The computation of $\mathit{freq}(\alpha)$ boils down to counting the number of traces T with $\alpha \sqsubseteq T$. Algorithm 4 recognizes all frequent episodes from a set of candidate episodes using the above described approach. Note that for both parallel and partial order episodes we can use the same recognition algorithm.

Algorithm 4: Recognize frequent episodes

Input: An event log L, a set of candidate episodes C_l, a frequency threshold $\mathit{minFreq}$.
Output: A set of frequent episodes F_l
Description: Recognizes frequent episodes, by filtering out candidate episodes that do not occur frequently in the log. **Note:** If $F_l = \emptyset$, then $C_l = \emptyset$.

RecognizeFrequentEpisodes$(L, C_l, \mathit{minFreq})$
(1) $\mathit{support} = [0, \ldots, 0]$ with $|\mathit{support}| = |C_l|$
(2) **foreach** $T \in L$
(3) **for** $i = 0$ **to** $|C_l| - 1$
(4) **if** Occurs$(C_l[i], T)$ **then** $\mathit{support}[i] = \mathit{support}[i] + 1$
(5) $F_l = \emptyset$
(6) **for** $i = 0$ **to** $|C_l| - 1$
(7) **if** $\frac{\mathit{support}[i]}{|L|} \geq \mathit{minFreq}$ **then** $F_l = F_l \cup \{C_l[i]\}$
(8) **return** F_l

Checking whether an episode α occurs in a trace $T = \langle A_1, \ldots, A_n \rangle$ is done via checking the existence of the mapping $h : \alpha.V \mapsto \{1, .., n\}$. This results in checking the two propositions shown below. Algorithm 5 implements these checks.

- Checking whether each node $v \in \alpha.V$ has a unique witness in trace T.
- Checking whether the (injective) mapping h respects the partial order indicated by $\alpha.\leq$.

For the discovery of an injective mapping h for a specific episode α and trace T we use the following recipe. First, we declare the class of models $H : \mathcal{A} \mapsto \mathcal{P}(\mathbb{N})$ such that for each activity $a \in \mathcal{A}$ we get the set of indices i at which $a = A_i \in T$. Next, we try all possible models derivable from H. A model $h : \alpha.V \mapsto \{1, .., n\}$ is derived from H by choosing an index $i \in H(f(v))$ for each node $v \in \alpha.V$. With such a model h, we can perform the actual partial order check against $\alpha.\leq$.

Algorithm 5: This algorithm implements occurrence checking via recursive discovery of the injective mapping h as per the occurrence definition.

Input: An episode α, a trace T.
Output: True iff $\alpha \sqsubseteq T$
Description: Implements occurrence checking based on finding an occurrence proof in the form of a mapping $h : \alpha.V \mapsto \{1, .., n\}$.
OCCURS($\alpha = (V, \leq, g), T$)
(1) **return** CHECKMODEL($\alpha, \{ a \mapsto \{ i \mid a = A_i \in T \} \mid a \in \mathcal{A} \}, \emptyset$)

Input: An episode α, a class of mappings $H : \mathcal{A} \mapsto \mathcal{P}(\mathbb{N})$, and an intermediate mapping $h : \alpha.V \mapsto \{1, .., n\}$.
Output: True iff there is a mapping h, as per the occurrence definition, derivable from H
Description: Recursive implementation for finding h based on the following induction principle: Base case (*if*-part): Every $v \in V$ is mapped ($v \in$ **dom** h). Step case (*else*-part): (IH) n vertices are mapped, step by adding a mapping for a vertex $v \notin$ **dom** h. (I.e., induction to the number of mapped vertices.)
CHECKMODEL($\alpha = (V, \leq, g), H, h$)
(1) **if** $\forall v \in V : v \in$ **dom** h
(2) **return** $(\forall (v, w) \in \leq : h(v) \leq h(w))$
(3) **else**
(4) pick $v \in V$ with $v \notin$ **dom** h
(5) **return** $(\exists i \in H(g(v)) :$
CHECKMODEL($\alpha, H[g(v) \mapsto H(g(v)) \setminus \{i\}], h[v \mapsto i]$))

4.4 Pruning

Using the pruning techniques described below, we reduce the number of generated episodes (and thereby computation time and memory requirements) and filter out uninteresting results. These techniques eliminate less interesting episodes by ignoring infrequent activities and skipping partial orders on activities with low temporal locality.

Activity Pruning Based on the frequency of an activity, uninteresting episodes can be pruned in an early stage. This is achieved by replacing the activity alphabet $\mathcal{A}$ by $\mathcal{A}^* \subseteq \mathcal{A}$, with
$(\forall A \in \mathcal{A}^* : \mathit{ActFreq}(A) \geq \mathit{minActFreq})$, on line 4 in Algorithm 1. This pruning technique allows the episode discovery algorithm to be more resistant to logs with many infrequent activities, which are indicative of exceptions or noise.

Trace Distance Pruning The pruning of episodes based on a trace distance interval can be achieved by adding the trace distance interval check to line 2 of Algorithm 5. Note that if there are two or more interpretations for h, with one passing and one rejected by the interval check, then we will find the correct interpretation thanks to the $\exists$ on line 5.

4.5 Episode Rule Discovery

The discovery of episode rules is done after discovering all the frequent episodes. For all frequent episodes α, we consider all frequent subepisodes β with $\beta \prec \alpha$ for the episode rule $\beta \Rightarrow \alpha$.

For efficiently finding potential frequent subepisodes β, we use the notion of "discovery tree", based on episode construction. Each time we recognize a frequent episode β created from combining frequent episodes γ and ε, we recognize β as a child of γ and ε. Similarly, γ and ε are the parents of β. See Figure 3 for an example of a discovery tree.

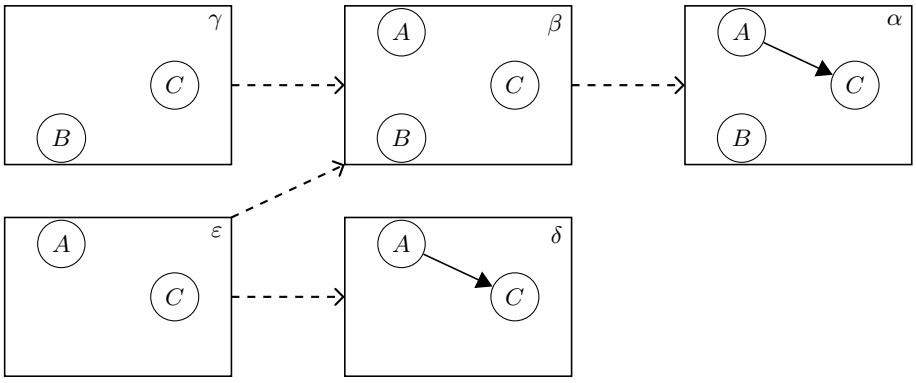


Fig. 3. Part of an example discovery tree. Each block denotes an episode. The dashed arrows between blocks denote a parent-child relationship. In this example we have, amongst others: $\beta \prec \alpha$, $\varepsilon \prec \beta$, $\varepsilon \prec \delta$ and $\delta \prec \alpha$ (not shown as a parent-child relation).

Using the discovery tree we can walk from an episode α along the discovery parents of α. Each time we find a parent β with $\beta \prec \alpha$, we can consider the parents and children of β. As result of Lemma 2, we cannot apply pruning in either direction of the parent-child relation based on the confidence $conf(\beta \Rightarrow \alpha)$. This is easy to see for the child direction. For the parent direction, observe the discovery tree in Figure 3 and $\delta \prec \alpha$. If for episode α we would stop before visiting the parents of β, we would never consider δ (which has $\delta \prec \alpha$).

4.6 Implementation Considerations

We implemented the episode discovery algorithm as a ProM 6 plug-in (see also Figure 4), written in Java. Since the OCCURS() algorithm (5) is the biggest bottleneck, this part of the implementation was considerably optimized.

5 Evaluation

This section reviews the feasibility of the approach using both synthetic and real-life event data.

5.1 Methodology

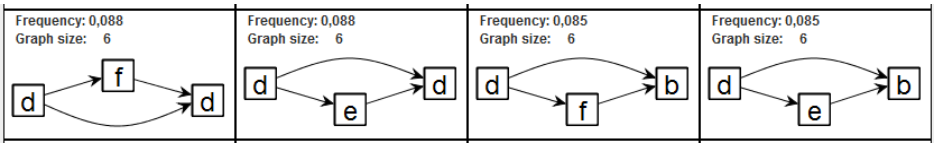
We ran a series of experiments on two type of event logs. The first event log, *bigger-example.xes*, is an artificial event log from the Chapter 5 of [1] and available via `http://www.processmining.org/event_logs_and_models_used_in_book`. The second event log, *BPI_Challenge_2012.xes*, is a real life event log available via `doi:10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f`. For these experiments we used a laptop with a Core i5-3570K CPU, 8 GB RAM and Java SE Runtime Environment 1.7.0_07-b11 (32 bit).

5.2 Performance Results

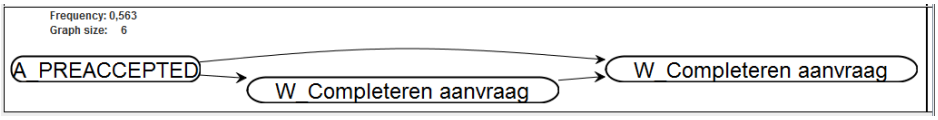
Table 2 some key characteristics for both event logs. We examined the effects of the parameters *minFreq*, *minActFreq* and *maxTraceDist* on the running time and the discovered number of episodes. In Figure 4 an indication (screenshots) of the ProM plugin output is given.

	# traces	*Avg. events/trace*	*Min. events/trace*	*Max. events/trace*
bigger-example.xes	1391	5	5	17
BPI_Challenge_2012.xes	13087	20	3	175

Table 2. Metadata for the used event logs



(a) Event log: bigger-example.xes – $minFreq = 0.05$, $minActFreq = 0.05$, $maxTraceDist = 3$



(b) Event log: BPI Challenge 2012 – $minFreq = 0.55$, $minActFreq = 0.55$, $maxTraceDist = 5$

Fig. 4. Screenshots of the results in the ProM plugin. Shown are the transitive reductions of the discovered episodes. Note that in the episodes in Figure 4(a), multiple nodes are allowed to have the same label.

As can be seen in all the experiments in Figure 5, we see that the running time is strongly related to the discovered number of episodes. Note that if some parameters

are poorly chosen, like high *maxTraceDist* in Figure 5(f), then a relatively large class of episodes seems to become frequent, thus increasing the running time drastically.

For a reasonably low number of frequent episodes (< 500, more will a human not inspect), the algorithm turns out to be quite fast (at most a few seconds for the Challenge log). We noted a virtual nonexistent contribution of the parallel episode mining phase to the total running time. This can be explained by a simple combinatorial argument: there are far more partial orders to be considered than there are parallel episodes.

An analysis of the effects of changing the *minFreq* parameter (Figure 5(a), 5(b)) shows that a poorly chosen value results in many episodes. In addition, the *minFreq* parameter gives us fine-grained control of the number of results. It gradually increases the total number of episodes for lower values. Note that, especially for the Challenge event log, low values for *minFreq* can dramatically increase the running time. This is due to the large number of candidate episodes being generated.

Secondly, note that for the *minActFreq* parameter (Figure 5(c), 5(d)), there seems to be a cutoff point that separates frequent from infrequent activities. Small changes around this cutoff point may have a dramatic effect on the number of episodes discovered.

Finally, for the *maxTraceDist* parameter (Figure 5(e), 5(f)), we see that this parameter seems to have a sweet-spot where a low – but not too low – number of episodes are discovered. Chosen a value for *maxTraceDist* just after this sweet-spot yields a huge number of episodes.

When comparing the artificial and real life event logs, we see a remarkable pattern. The artificial event log (*bigger-example.xes*), shown in Figure 5(a) appears to be far more fine-grained than the real life event log (*BPI_Challenge_2012.xes*) shown in Figure 5(b). In the real life event log there appears to be a clear distinction between frequent and infrequent episodes. In the artificial event log a more exponential pattern occurs. Most of the increase in frequent episodes, for decreasing $minFreq$, is again in the partial order discovery phase.

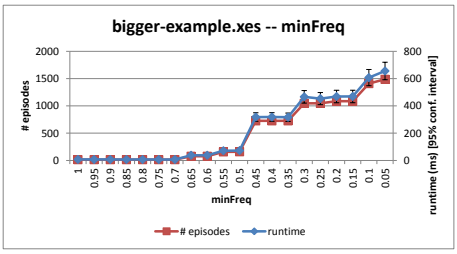
5.3 Comparison to existing discovery algorithms

As noted in the introduction, often the overall end-to-end process models are rather complicated. Therefore, the search for local patterns (i.e., episodes) is interesting. A good example of a complicated process is the *BPI Challenge 2012* log. In Figure 6 part of the "spaghetti-like" process models are shown, as an indication of the complexity. The episodes discovered over same log, depicted in Figure 4(b) gives us a simple and clear insight into important local patterns in the *BPI Challenge 2012* log. Hence, in these "spaghetti-like" process models, the episode discovery technique allows us to quickly understand the main patterns.

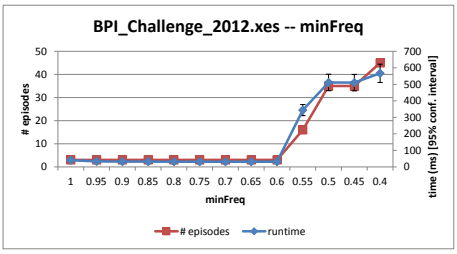
6 Conclusion and Future work

In this paper, we considered the problem of discovering frequently occurring episodes in an event log. An episode is a collection of events that occur in a given partial order. We presented efficient algorithms for the discovery of frequent episodes and episode rules occurring in an event log, and presented experimental results.

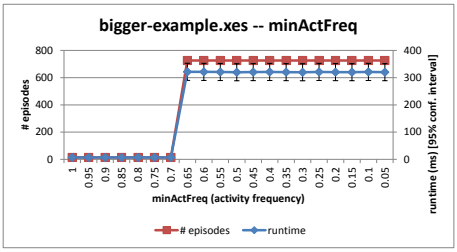
Our experimental evaluation shows that the running time is strongly related to the discovered number of episodes. For a reasonably low number of frequent episodes



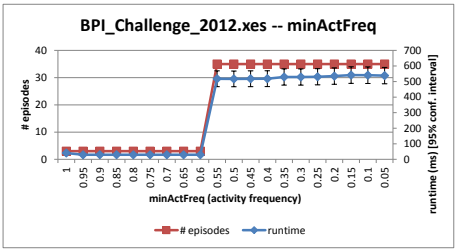
(a) Parameter: *minFreq*
Event log: bigger-example.xes
$minActFreq = 0.65$, $maxTraceDist = 4$



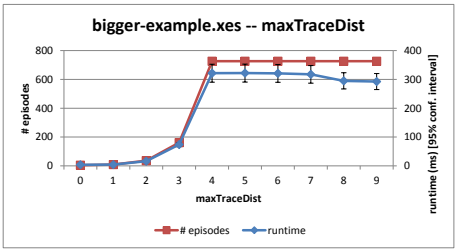
(b) Parameter: *minFreq*
Event log: BPI Challenge 2012
$minActFreq = 0.65$, $maxTraceDist = 4$



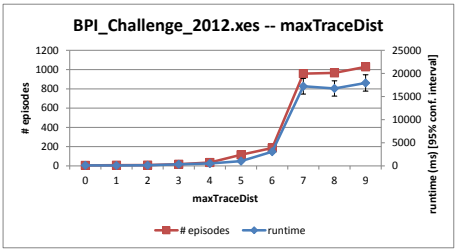
(c) Parameter: *minActFreq*
Event log: bigger-example.xes
$minFreq = 0.45$, $maxTraceDist = 4$



(d) Parameter: *minActFreq*
Event log: BPI Challenge 2012
$minFreq = 0.50$, $maxTraceDist = 4$



(e) Parameter: *maxTraceDist*
Event log: bigger-example.xes
$minFreq = 0.45$, $minActFreq = 0.65$



(f) Parameter: *maxTraceDist*
Event log: BPI Challenge 2012
$minFreq = 0.50$, $minActFreq = 0.55$

Fig. 5. Effects of the parameter on the performance and number of discovered episodes.

(< 500, more will a human not inspect), the algorithm turns out to be quite fast (at most a few seconds). The main problem is the correct setting of the episode pruning parameters *minFreq*, *minActFreq*, and *maxTraceDist*.

During the development of the algorithm for ProM 6, special attention was paid to optimizing the OCCURS() algorithm (Algorithm 5) implementation, which proved to be the main bottleneck. Future work could be to prune occurrence checking based on the parents of an episode, leveraging the fact that an episode cannot occur in a trace if a parent also did occur in that trace.

Another approach to improve the algorithm is to apply the *generic divide and conquer approach for process mining*, as defined in [22]. This approach splits the set of activities into a collection of partly overlapping activity sets. For each activity set, the log is projected onto the relevant events, and the regular episode discovery algorithm is applied. In essence, the same trick is applied as used by the *minActFreq*

(a) Event log: BPI Challenge 2012 – Discovery algorithm: α-algorithm [10].

(b) Event log: BPI Challenge 2012 – Discovery algorithm: [11].

Fig. 6. Screenshots of results in other ProM plugin. Shown are parts of the Petri-nets mined with the α-algorithm and the heuristics miner.

parameter (using an alphabet subset), which is to create a different set of initial 1-node parallel episodes to start discovering with.

The main bottleneck is the frequency computation by checking the occurrence of each episode in each trace. Typically, we have a small amount of episodes to check, but many traces to check against. Using the MapReduce programming model developed by Dean and Ghemawat, we can easily parallelize the episode discovery algorithm and execute it on a large cluster of commodity machines [23]. The MapReduce programming model requires us to define *map* and *reduce* functions. The *map* function, in our case, accepts a trace and produces [episode, trace] pairs for each episode occurring in the given trace. The *reduce* function accepts an episode plus a list of traces in which that episode occurs, and outputs a singleton list if the episode is frequent, and an empty list otherwise. This way, the main bottleneck of the algorithm is effectively parallelized.

References

[1] van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer-Verlag, Berlin (2011)

[2] Mannila, H., Toivonen, H., Verkamo, A.I.: Discovery of Frequent Episodes in Event Sequences. Data Mining and Knowledge Discovery **1**(3) (1997) 259–289

[3] Lu, X., Fahland, D., van der Aalst, W.M.P.: Conformance checking based on partially ordered event data. To appear in Business Process Intelligence 2014, workshop SBS (2014)

[4] Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases. In: Proceedings of the 20th International Conference on Very Large Data Bases. VLDB '94, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1994) 487–499

[5] Agrawal, R., Srikant, R.: Mining Sequential Patterns. In: Proceedings of the Eleventh International Conference on Data Engineering. ICDE '95, Washington, DC, USA, IEEE Computer Society (1995) 3–14

[6] Srikant, R., Agrawal, R.: Mining Sequential Patterns: Generalization and Performance Improvements. In: Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology. EDBT '96, London, UK, UK, Springer-Verlag (1996) 3–17

[7] Lu, X., Mans, R.S., Fahland, D., van der Aalst, W.M.P.: Conformance checking in healthcare based on partially ordered event data. To appear in Emerging Technologies and Factory Automation 2014, workshop M2H (2014)
[8] Fahland, D., van der Aalst, W.M.P.: Repairing process models to reflect reality. In: Proceedings of the 10th International Conference on Business Process Management. BPM'12, Berlin, Heidelberg, Springer-Verlag (2012) 229–245
[9] Laxman, S., Sastry, P.S., Unnikrishnan, K.P.: Fast Algorithms for Frequent Episode Discovery in Event Sequences. In: Proc. 3rd Workshop on Mining Temporal and Sequential Data. (2004)
[10] van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. IEEE Transactions on Knowledge and Data Engineering **16**(9) (2004) 1128–1142
[11] Weijters, A.J.M.M., van der Aalst, W.M.P., de Medeiros, A.K.A.: Process Mining with the Heuristics Miner-algorithm. BETA Working Paper Series, WP 166, Eindhoven University of Technology, Eindhoven (2006)
[12] de Medeiros, A.K.A., van der Aalst, W.M.P., Weijters, A.J.M.M.: Workflow mining: Current status and future directions. In Meersman, R., Tari, Z., Schmidt, C.D., eds.: On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE. Volume 2888 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2003) 389–406
[13] Wen, L., van der Aalst, W.M.P., Wang, J., Sun, J.: Mining process models with non-free-choice constructs. Data Mining and Knowledge Discovery **15**(2) (2007) 145–180
[14] de Medeiros, A.K.A., Weijters, A.J.M.M., van der Aalst, W.M.P.: Genetic Process Mining: An Experimental Evaluation. Data Mining and Knowledge Discovery **14**(2) (2007) 245–304
[15] Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: On the Role of Fitness, Precision, Generalization and Simplicity in Process Discovery. In Meersman, R., Rinderle, S., Dadam, P., Zhou, X., eds.: OTM Federated Conferences, 20th International Conference on Cooperative Information Systems (CoopIS 2012). Volume 7565 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2012) 305–322
[16] Solé, M., Carmona, J.: Process Mining from a Basis of State Regions. In: Applications and Theory of Petri Nets (Petri Nets 2010). Volume 6128 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2010) 226–245
[17] van der Aalst, W.M.P., Rubin, V., Verbeek, H.M.W., van Dongen, B.F., Kindler, E., Günther, C.W.: Process Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting. Software and Systems Modeling **9**(1) (2010) 87–111
[18] Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process Mining Based on Regions of Languages. In Alonso, G., Dadam, P., Rosemann, M., eds.: International Conference on Business Process Management (BPM 2007). Volume 4714 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2007) 375–383
[19] van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process Discovery using Integer Linear Programming. Fundamenta Informaticae **94** (2010) 387–412
[20] Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering Block-structured Process Models from Incomplete Event Logs. In Ciardo, G., Kindler, E., eds.: Applications and Theory of Petri Nets 2014. Volume 8489 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2014) 91–110
[21] Kryszkiewicz, M.: Fast Discovery of Representative Association Rules. In Polkowski, L., Skowron, A., eds.: Rough Sets and Current Trends in Computing. Volume 1424 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1998) 214–222
[22] van der Aalst, W.M.P.: Decomposing Petri Nets for Process Mining: A Generic Approach. Distributed and Parallel Databases **31**(4) (2013) 471–507
[23] Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM **51**(1) (2008) 107–113