Discovery of Frequent Episodes in Event Logs

Maikel Leemans and Wil M.P. van der Aalst
m.leemans@tue.nl,w.m.p.v.d.aalst@tue.nl

Eindhoven University of Technology,
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands.

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 traditional episode mining, i.e., mining collections of partially ordered events) do not consider process instances. 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, and apply our technique to other perspectives present in event logs. 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.

Keywords: Episode Discovery, Partial Order Discovery, Process Discovery

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 *from 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., the 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 discovering patterns while using other perspectives also present the event log. Lastly, we can use episode discovery as a starting point for 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 sized 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*, available through the *Episode Miner* package [9]. 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 [10]. 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

[11] was the first process discovery algorithm adequately handling concurrency. Several variants of the α algorithm have been proposed [12, 13]. Many other discovery techniques followed, e.g., *heuristic* mining [14] able to deal with noise and low-frequent behavior. The HeuristicsMiner is based on the notion of causal nets (C-nets). Moreover, completely different approaches have been proposed, e.g., the different types of *genetic* process mining [15, 16], techniques based on *state-based regions* [17, 18], and techniques based on *language-based regions* [19, 20]. A frequency-based approach is used in the *fuzzy* mining technique, which produces a precedence-relation-based process map [21]. Frequencies are used to filter out infrequent paths and nodes. Another, more recent, approach is *inductive* process mining where the event log is split recursively [22]. 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 summarized in Table 1. Based on the above discussion we conclude that *Episode Discovery* is the only technique whose results focus on local behavior while exploiting process instances.

Table 1. Feature comparison of discussed discovery algorithms

	Exploits process instances	Discovers end-to-end model	Focus on *local* behavior	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.	+	-	+	-	+
Maggi, DECLARE Miner [23, 24, 25]	+	+/-	-	n.a.	+	+	+	-	+
Van der Aalst, α-algorithm [11]	+	+	-	-	+	+	+	-	-
Weijters, Heuristics mining [14]	+	+	-	-	+	+	+	-	-
De Medeiros, Genetic mining [15, 16]	+	+	-	-	+	+	+	+	+
Solé, State Regions [17, 18]	+	+	-	-	+	+	+	-	-
Bergenthum, Language Regions [19, 20]	+	+	-	-	+	+	+	-	-
Günther, Fuzzy mining [21]	+	+	-	n.a.	+	+/-	+/-	-	-
Leemans S.J.J., Inductive [22]	+	+	-	+	+	+	+	+	-

The discovery of Declarative Process Models, as presented in [23, 24, 25], aims to discover patterns to describe an overall process model. The underlying model is the DECLARE declarative language. This language uses LTL templates that can be used to express rules related to the ordering and presence of activities. This discovery technique requires the user to limit the constraint search-space by selecting rule templates to search for. That is, the user selects a subset of pattern types (e.g., succession, not-coexists, etc.) to search for. However, the underlying discovery technique is pattern-agnostic, and simply generates all pattern instantiations (using apriori-based optimization techniques), followed by LTL

evaluations. The major downside of this approach is a relatively bad runtime performance, and we will also observe this in Section 5.4.

The discovery of patterns in the resource perspective has been partly tackled by techniques for organizational mining [26]. These techniques can be used to discover organizational models and social networks. A social network is a graph/network in which the vertices represent resources (i.e., a person or device), and the edges denote the relationship between resources. A typical example is the handover of work metric. This metric captures that, if there are two subsequent events in a trace, which are completed by resource a and b respectively, then it is likely that there is a handover of work from a to b. In essence, the discovery of handover of work network yields the "end-to-end" resource model, related to the discovery of episodes or partial orders on the resource perspective.

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 [27]. 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.

3 Definitions: 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 Preliminaries

Multisets Multisets are used to describe event logs where the same trace may appear multiple times.

We denote the set of all multisets over some set A as $\mathcal{B}(A)$. We define $B(a)$ for some multiset $B \in \mathcal{B}(A)$ as the number of times element $a \in A$ appears in multiset B. For example, given $A = \{x, y, z\}$, a possible multiset $B \in \mathcal{B}(A)$ is $B = [x, x, y]$. For this example, we have $B(x) = 2$, $B(y) = 1$ and $B(z) = 0$. The size $|B|$ of a multiset $B \in \mathcal{B}(A)$ is the sum of appearances of all elements in the multiset, i.e.: $|B| = \Sigma_{a \in A} B(a)$.

Note that the ordering of elements in a multiset is irrelevant.

Sequences Sequences are used to represent traces in an event log.

Given a set X, a sequence over X of length n is denoted as $\sigma = \langle a_1, a_2, \ldots, a_n \rangle \in X^*$. We denote the empty sequence as $\langle\rangle$.

Note that the ordering of elements in a sequence is relevant.

Functions Given sets X and Y, we write $f : X \mapsto Y$ for the function with domain $\mathbf{dom}\ f \subseteq X$ and range $\mathbf{ran}\ f = \{\, f(x) \mid x \in X \,\} \subseteq Y$. In this context, the $\mapsto$ symbol is used to denote a specific function.

As an example, the function $f : \mathbb{N} \mapsto \mathbb{N}$ can be defined as $f = \{ x \mapsto x + 1 \mid x \in \mathbb{N} \}$. For this f we have, amongst others, $f(0) = 1$ and $f(1) = 2$ (i.e., this f defines a succession relation on $\mathbb{N}$).

3.2 Event Logs

Activities and Traces Let $\mathcal{A} \subseteq \mathcal{U}_{\mathcal{A}}$ be the alphabet of activities occurring in the event log. A trace is a sequence $\sigma = \langle a_1, a_2, \ldots, a_n \rangle \in \mathcal{A}^*$ of activities $a_i \in \mathcal{A}$ occurring at time index i relative to the other activities in σ.

Event log An event log $L \in \mathcal{B}(\mathcal{A}^*)$ is a multiset of traces. 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 the *resource* (i.e., the person or device) executing or initiating the activity, and the *timestamp* of the event.

Note that, in this paper, we assumed simple event logs using the default activity classifier, yielding partial orders on activities. It should be noted that the technique discussed in this paper is classifier-agnostic. As a result, using alternative classifiers, partial orders on other perspectives can be obtained. An example is the flow of work between persons by discovering partial orders using a resource classifier on the event log.

3.3 Episodes

Episode An episode is a partially ordered collection of events. A partial order is a binary relation which is reflexive, antisymmetric and transitive. 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 labeling the nodes/events [2]. For two vertices $u, v \in V$ we have $u < v$ iff $u \leq v$ and $u \neq v$.

Note that if $|V| \leq 1$, then we got a singleton or empty episode. For the rest of this paper, we ignore empty episodes. We call an episode *parallel* when there are two or more vertices, and no edges.

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:

$$
\begin{array}{ll}
(\forall v \in V' : g'(v) = g(f(v))) & \text{All vertices in } \beta \text{ are also in } \alpha \\
\wedge\ (\forall v, w \in V' \wedge v \leq' w : f(v) \leq f(w)) & \text{All edges in } \beta \text{ are also in } \alpha
\end{array}
$$

An episode β equals episode α, denoted $\beta \equiv \alpha$ iff $\beta \preceq \alpha \wedge \alpha \preceq \beta$. An episode β is a strict subepisode of α, denoted $\beta \prec \alpha$, iff $\beta \preceq \alpha \wedge \beta \not\equiv \alpha$.

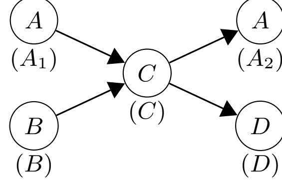


Fig. 1. Shown is the transitive reduction of the partial order for an example episode. The circles represent nodes (events), with the activity labeling imposed by g inside the circles, and an event ID beneath the nodes in parenthesis. In this example, events A_1 and B can happen in parallel (as can A_2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

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 a smallest γ (i.e., smallest sets V'' and $\leq''$) such that $\alpha \preceq \gamma$ and $\beta \preceq \gamma$.

The smallest sets criterion implies that every event $v \in V''$ and ordered pair $v, w \in V'' \land v \leq'' w$ must be represented in α and/or β (i.e., have a witness, see also the formulae below). Formally, an episode $\gamma = \alpha \oplus \beta$ iff there exists injective mappings $f : V \mapsto V''$ and $f' : V' \mapsto V''$ such that:

$$
\begin{aligned}
\gamma &= (V'', \leq'', g'') \\
\leq'' &= \{\, (f(v), f(w)) \mid (v, w) \in\ \leq \} \\
&\quad \cup \{\, (f'(v), f'(w)) \mid (v, w) \in\ \leq' \} && \text{order witness} \\
g'' &: (\forall v \in V : g(v) = g''(f(v))) \land (\forall v' \in V' : g'(v') = g''(f'(v'))) && \text{correct mapping} \\
V'' &: \forall v'' \in V'' : (\exists v \in V : f(v) = v'') \lor (\exists v' \in V' : f'(v') = v'') && \text{node witness}
\end{aligned}
$$

Observe that "order witness" and "correct mapping" are based on $\alpha \preceq \gamma$ and $\beta \preceq \gamma$. Note that via "note witness" it is ensured that every vertex in V'' is mapped to a vertex in either V or V'. Every vertex in V and V' should be mapped to a vertex in V''. This is ensured via "correct mapping".

Occurrence An episode $\alpha = (V, \leq, g)$ occurs in an event trace $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$, denoted $\alpha \sqsubseteq \sigma$, iff there exists an injective mapping $h : V \mapsto \{1, .., n\}$ such that:

$$
\begin{aligned}
&(\forall v \in V : g(v) = a_{h(v)} \in \sigma) && \text{All vertices are mapped correctly} \\
\land\ &(\forall v, w \in V \land v \leq w : h(v) \leq h(w)) && \text{The partial order } \leq \text{ is respected}
\end{aligned}
$$

In Figure 2 an example of an "event to trace map" h for occurrence checking is given. Note that multiple mappings might exists. Intuitively, if we have a trace t and an episode with $u \leq v$, then the activity $g(u)$ must occur before activity $g(v)$ in t.

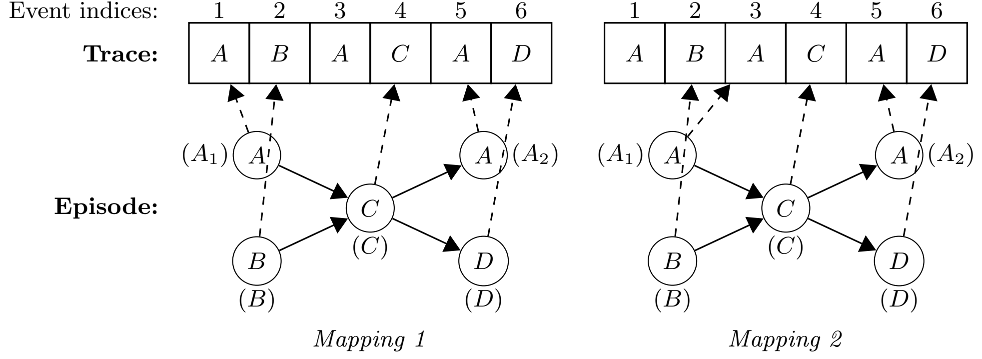


Fig. 2. Shown are two possible mappings h (the dotted arrows) for checking occurrence of the example episode in a trace. The shown graphs are the transitive reduction of the partial order of the example episode. Note that with the left mapping (*Mapping 1*) also an episode with the partial order $A_1 < B$ 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 $freq(\alpha)$ of an episode α in an event log $L \in \mathcal{B}(\mathcal{A}^*)$ is defined as:

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

Given a frequency threshold *minFreq*, an episode α is frequent iff $freq(\alpha) \geq minFreq$. During the actual episode discovery, we use the contrapositive of the fact given in Lemma 1. That is, we use the observation that if not all subepisodes β are frequent, then the episode α is also not frequent.

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:*

$$(\forall \beta \preceq \alpha : freq(\beta) \geq freq(\alpha))$$

3.4 Episode and Event Log Measurements

Activity Frequency The activity frequency$ActFreq(a)$ of an activity $a \in \mathcal{A}$ in an event log $L \in \mathcal{B}(\mathcal{A}^*)$ is defined as:

$$ActFreq(a) = \frac{|\,[\,\sigma \in L \mid a \in \sigma\,]\,|}{|L|}$$

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

Trace Distance Given episode $\alpha = (V, \leq, g)$ occurring in an event trace $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$, as indicated by an event to trace map $h : V \mapsto \{1, .., n\}$. Then the trace distance $traceDist(\alpha, h)$ is defined as:

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

In Figure 2, the left mapping h_1 yields $traceDist(\alpha, h_1) = 6 - 1 = 5$, and the right mapping h_2 yields $traceDist(\alpha, h_2) = 6 - 2 = 4$.

Given a trace distance interval $[minTraceDist, maxTraceDist]$, an episode α is accepted in trace σ with respect to the trace distance interval iff there exists a mapping h such that $minTraceDist \leq traceDist(\alpha, h) \leq maxTraceDist$.

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.

Eventually-follows Relation The eventually-follows relation $\gg_L$ for an event log L and two activities $a, b \in \mathcal{A}$ is defined as:

$$a \gg_L b = \left| \left\{ \sigma \in L \;\middle|\; \exists_{0 \leq i < j < |\sigma|} : \sigma(i) = a \wedge \sigma(j) = b \right\} \right|$$

Informally, the eventually-follows valuation for $a \gg_L b$ equals the amount of traces in which a happens (at timestamp i), and is followed by b at a later moment (at timestamp j with $i < j$).

If we evaluate the eventually-follows relation for every $a, b \in \mathcal{A}$, we obtain the eventually-follows matrix. In Table 2 the eventually-follows matrix is given for an example event log.

Table 2. The eventually-follows matrix for the following example event log: $L = [\langle a, b, a, c, a, d \rangle, \langle a, b, a, d \rangle, \langle b, d \rangle]$. Each cell gives the valuation for *row* $\gg_L$ *column*, where *row* is the activity shown to the left, and *column* is the activity shown on the top of the table.

$\gg_L$	a	b	c	d
a	2	2	1	2
b	2	0	1	3
c	1	0	0	1
d	0	0	0	0

Lemma 2 (Eventually-follows Relation and Episode Frequency).
The eventually-follows valuation $g(u) \gg_L g(v)$ for any two vertices $u, v \in V$ with $u \leq v$ is an upper bound for the frequency of the episode $\alpha = (V, \leq, g)$ in event log L. Formally:

$$(\forall u, v \in V \wedge u \leq v : \frac{g(u) \gg_L g(v)}{|L|} \geq freq(\alpha))$$

Consequently, if an episode $\alpha = (V, \leq, g)$ is frequent in an event log L, then for any two vertices $u, v \in V$ with $u \leq v$ also the eventually follows valuation for $g(u) \gg_L g(v)$ is frequent.

Based on Lemma 2, the eventually-follows relation can be used as a fast approximation of early occurrence checking. Concretely, by contraposition, we know that if there exists $u, v \in V$ with $u \leq v$ for which $\frac{g(u) \gg_L g(v)}{|L|} < minFreq$, then the episode α cannot be frequent. We use this fact as an optimization technique in the realization of our Episode Discovery technique.

3.5 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 3.

Lemma 3 (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 an episode (rule) discovery plug-in in the process mining framework *ProM*, available through the *Episode Miner* package [9]. To be able to analyze real-life event logs, we need efficient algorithms. These are described next.

4.1 Notation in realization

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

For the implementation, we rely on *ordered* sets, i.e., lists of unique elements. The order of a set is determined by the order in which elements are added to the sets, which is leveraged to make the algorithms efficient. We assume individual elements can be accessed via an index, with indexing starting at zero. We use the following operations and notations in the algorithms to come:

$A = \{x, y, z\}$ **with** $x < y < z$	Note: $n = \lvert A \rvert = 3$
$A[0] = x$	Access the first element
$A[n-1] = z$	Access the last element
$(A \cup \{v\}) = \{x, y, z, v\}$ **with** $x < y < z < v$	Adding new elements to a set
$(A \cup \{x\}) = A$	Every element is unique
$(A \cup \{v\})[n] = v$	Access the new last element
$A[0\,..\,n-2] = \{x, y\}$ **with** $x < y$	Access a subset of a set

4.2 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: Episode 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 between recognizing frequent candidates in the event log (F_l), and generating new candidate episodes (C_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, A, minFreq)
(1)     Γ = ∅
(2)     // Phase 1: discover parallel episodes
(3)     l = 1 // Tracks the number of nodes
(4)     // Initialize: create a candidate episode for every activity in A
(5)     C_l = { (V, ≤, g) | |V| = 1, ≤ = ∅, g = {v ↦ a}, v ∈ V, a ∈ A }
(6)     // Step: recognize and construct larger episodes from smaller episodes
(7)     while C_l ≠ ∅
(8)         F_l = RecognizeFrequentEpisodes(L, C_l, minFreq)
(9)         Γ = Γ ∪ F_l
(10)        C_l = GenerateCandidateParallel(l, F_l)
(11)        l = l + 1
(12)    // Phase 2: discover partial orders
(13)    l = 1 // Tracks the number of edges
(14)    // Initialize: create candidate episodes based on results from Phase 1
(15)    C_l = { (γ.V, ≤, γ.g) | γ ∈ Γ, ≤ = {(v, w)}, v, w ∈ γ.V, v ≠ w }
(16)    // Step: recognize and construct larger episodes from smaller episodes
(17)    while C_l ≠ ∅
(18)        F_l = RecognizeFrequentEpisodes(L, C_l, minFreq)
(19)        Γ = Γ ∪ F_l
(20)        C_l = GenerateCandidateOrder(l, F_l)
(21)        l = l + 1
(22)    return Γ
```

4.3 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.

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} = ∅
(2)   for i = 0 to |F_l| − 1
(3)       for j = i to |F_l| − 1
(4)           α = F_l[i]
(5)           β = F_l[j]
(6)           // Check if α and β overlap (see also description, index start at 0)
(7)           if ∀0 ≤ i ≤ l − 2 : α.g(α.V[i]) = β.g(β.V[i])
(8)               // Create candidate γ = α ⊕ β
(9)               γ = (V, ≤, g) where V = (α.V[0 .. l − 1] ∪ β.V[l − 1]), ≤ = ∅, g = α.g ∪ β.g
(10)              C_{l+1} = C_{l+1} ∪ {γ}
(11)          else
(12)              break
(13)  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} = ∅
(2)   for i = 0 to |F_l| − 1
(3)       for j = i + 1 to |F_l| − 1
(4)           α = F_l[i]
(5)           β = F_l[j]
(6)           // Check if α and β overlap (see also description, index start at 0)
(7)           sharingAllNodes = (α.V = β.V ∧ α.g = β.g)
(8)           overlappingEdges = (α.≤[0 .. l − 2] = β.≤[0 .. l − 2])
(9)           if sharingAllNodes ∧ overlappingEdges
(10)              // Create candidate γ = α ⊕ β
(11)              γ = (α.V, ≤, α.g) where ≤ = (α.E[0 .. l − 1] ∪ β.E[l − 1])
(12)              C_{l+1} = C_{l+1} ∪ {γ}
(13)          else
(14)              break
(15)  return C_{l+1}
```

4.4 Frequent Episode Recognition

In order to check if a candidate episode α is frequent, we check if $freq(\alpha) \geq minFreq$. The computation of $freq(\alpha)$ boils down to counting the number of traces σ with $\alpha \sqsubseteq \sigma$. 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.

Recall that an event log is a multiset of traces. Based on this observation, we note that particular trace variants typically occur more than once in an event log. We use this fact to reduce the number of iterations in Algorithm 4, and consequently the number of occurrence checks performed (i.e., Occurs() invocations). Instead of iterating over all the process instances on line 2 of the algorithm, we consider each trace variant σ only once. For the support count we use the $L(\sigma)$ multiset operation to get the correct number of process instances.

Algorithm 4: Recognize frequent episodes

Input: An event log L, a set of candidate episodes C_l, a frequency threshold *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($L, C_l, minFreq$)
(1) $support = [0, \ldots, 0]$ **with** $|support| = |C_l|$
(2) **foreach** $\sigma \in L$
(3) **for** $i = 0$ **to** $|C_l| - 1$
(4) **if** Occurs($C_l[i], \sigma$) **then** $support[i] = support[i] + L(\sigma)$
(5) $F_l = \emptyset$
(6) **for** $i = 0$ **to** $|C_l| - 1$
(7) **if** $\frac{support[i]}{|L|} \geq minFreq$ **then** $F_l = F_l \cup \{C_l[i]\}$
(8) **return** F_l

Checking whether an episode α occurs in a trace $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ 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 σ.
- 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 σ 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 $a = a_i \in \sigma$. 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: Occurrence checking for an episode

Input: An episode α, a trace σ.
Output: True iff $\alpha \sqsubseteq \sigma$
Description: Implements occurrence checking based on finding an occurrence proof in the form of a mapping $h : \alpha.V \mapsto \{1, .., n\}$.

```
Occurs(α = (V, ≤, g), σ)
(1)   // H indicates for each activity a all the indices i at which a = a_i ∈ σ
(2)   H = { a ↦ { i | a = a_i ∈ σ } | a ∈ A }
(3)   h = ∅
(4)   return checkModel(α, H, h)
```

Algorithm 6: This algorithm implements occurrence checking via recursive discovery of the injective mapping h as per the occurrence definition.

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 induction to the number of mapped vertices:
Base case (*if*-part): Every $v \in V$ is mapped ($v \in \mathbf{dom}\ h$).
Step case (*else*-part): (IH) n vertices are mapped, step by adding a mapping for a vertex $v \notin \mathbf{dom}\ h$.

```
checkModel(α = (V, ≤, g), H, h)
(1)   if ∀v ∈ V : v ∈ dom h
(2)       // Every v ∈ V is mapped, check the edge relation
(3)       return (∀(v, w) ∈ ≤ : h(v) ≤ h(w))
(4)   else
(5)       // Choose a mapping for a vertex v ∉ dom h
(6)       pick v ∈ V with v ∉ dom h
(7)       // Compute ∃i ∈ H(g(v)) : checkModel(v mapped to i)
(8)       exists = False
(9)       foreach i ∈ H(g(v)) do exists ∨ checkModel(α, H[g(v) ↦ H(g(v)) \ {i}], h[v ↦ i])
(10)      return exists
```

4.5 Time complexity analysis

The theoretical time complexity of the provided algorithms is dominated by two aspects: 1) the Apriori-style iterations in Algorithm 1, and 2) the occurrence checking in Algorithm 6. For the worst case time complexity we will first investigate the occurrence checking, and then briefly display the total time complexity.

Analysis of Occurence checking (Alg 6) Consider trace $\sigma = [a_1, a_2, \ldots, a_n]$ and episode with $V = \{v_1, v_2, \ldots, v_m\}$. Worst case, $m = n$.

Finding mapping h is done by, for each v_i find a a_j such that the order condition holds. Checking the order condition takes $O(|\leq|)$. Worst case, we check mappings in ascending order $(v_1 \to a_1, \ldots v_1 \to a_n)$ where only the last mapping is valid. Hence, we need $n!$ attempts, resulting in worst case complexity $O(n! \cdot |\leq|)$.

Total time complexity of Algorithm 1 The total worst case running time consists of $O(\mathit{Phase1}) + O(\mathit{Phase2})$, and is given by:

$$O({T_L}^2 \cdot |\mathcal{A}|^{T_L+1} \cdot \left(|\mathcal{A}|^{T_L+1} + |L| \cdot \Sigma_{l=1}^{T_L}(l-1)!\right)$$
$$+ {T_L}^5 \cdot \Sigma_{l=1}^{\frac{1}{2}{T_L}^2 - \frac{1}{2}T_L} \binom{T_L \cdot (T_L - 1)}{l} \cdot \left(\binom{T_L \cdot (T_L - 1)}{l} + |L| \cdot (T_L - 1)!\right))$$

Where: $T_L = \max \{ |\sigma| \mid \sigma \in L \}$ is the max trace size in log, $|L|$ is the size of event log (# trace variants), and $|\mathcal{A}|$ is the size of alphabet (# event classes).

Note that, despite the theoretical worst case time complexity, our episode discovery algorithm is very fast in practice. See also the evaluation in Section 5.

4.6 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 events not occurring relatively close in time. In addition, for pruning based on the antisymmetry of $\leq$ and the Eventually-follows Relation, we leverage the fact that it is cheaper to prune candidates during generation than to eliminate them via occurrence checking.

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 $\mathcal{A}$ with the largest set $\mathcal{A}' \subseteq \mathcal{A}$ satisfying $(\forall a \in \mathcal{A}' : \mathit{ActFreq}(a) \geq \mathit{minActFreq})$, on line 5 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. Note that, if *minActFreq* is set too high, we can end up with $\mathcal{A}' = \emptyset$. In this case, no episodes are discovered.

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 3 of Algorithm 6. 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 7.

Pruning Based on the Antisymmetry of $\leq$ During candidate generation in Algorithm 3 we can leverage the antisymmetry of $\leq$. Recall that in Algorithm 3 we generate candidate episodes γ from merging episodes α and β overlapping on the first $l-1$ edges. If we extend the predicate on line 9 with the check $\mathit{reverse}(\beta.{\leq}[l-1]) \notin \alpha.{\leq}$ we ensure that we don't generate candidate episodes γ that violate the antisymmetry of $\leq$. (Note: $\mathit{reverse}((a, b)) = (b, a)$.)

Pruning Based on the Eventually-follows Relation During seeding the partial order candidates in Algorithm 1 on line 15 we can utilize the eventually-follows relation as a fast approximation of early occurrence checking. Using this relation, we can extend the predicate on line 15 with the check $\frac{a \gg_L b}{|L|} \geq \mathit{minFreq}$, where $a = g(v) \wedge b = g(w)$.

In practice, we pre-calculate the eventually-follows matrix, having a space-complexity of $|\mathcal{A}|^2$, where $|\mathcal{A}|$ the number of unique activities in the event log. This allows us to compute the eventually-follows values only once in a linear scan over the log, and reuse values, accessing them in constant time.

4.7 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.

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 a result of Lemma 3, 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 $\delta \prec \alpha$).

This principle of traversing the discovery tree is implemented by Algorithm 7. This implementation uses a discovery *front* queue for traversing the discovery tree, similar to the queue used in the Breadth-first search algorithm. The discovery tree is traversed for each discovered episode (each $\alpha \in \Gamma$). Hence, we consider the discovery tree as a partial order on the set Γ, and use that structure to efficiently find sets of subsets.

Algorithm 7: Discovering episode rules

Input: A list of episodes Γ, a confidence threshold $minConf$ and a magnitude interval specified by $minMag$ and $maxMag$.
Output: A set of valid episode rules R
Description: Episode rule discovery. For each discovered episode (each $\alpha \in \Gamma$), the discovery tree is traversed in a Breadth-first search style, searching for candidate β episodes yielding episode rules $\beta \Rightarrow \alpha$.

```
RuleDiscovery(Γ, minConf, minMag, maxMag)
(1)    R = ∅
(2)    foreach α ∈ Γ
(3)        discovered = ∅
(4)        Let front be an empty FIFO queue
(5)        foreach parent ∈ α.parents
(6)            discovered = discovered ∪ {parent}
(7)            front.enque(parent)
(8)        while front ≠ ∅
(9)            β = front.deque()
(10)           foreach parent ∈ β.parents
(11)               discovered = discovered ∪ {parent}
(12)               front.enque(parent)
(13)           if β ⪯ α
(14)               // prune siblings of α
(15)               if β ∉ α.parents
(16)                   foreach child ∈ β.children ∧ child ∉ discovered
(17)                       discovered = discovered ∪ {child}
(18)                       front.enque(child)
(19)               if conf(β ⇒ α) ≥ minConf ∧ minMag ≤ mag(β ⇒ α) ≤ maxMag
(20)                   R = R ∪ {β ⇒ α}
```

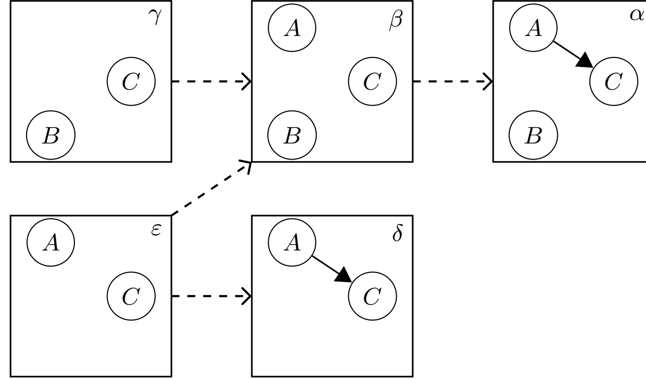


Fig. 3. Part of an example discovery tree. Each block denotes an episode. The dashed arrows between blocks denote a parent-child relationship. In this example we have, amongst others: $\beta \prec \alpha$, $\varepsilon \prec \beta$, $\varepsilon \prec \delta$ and $\delta \prec \alpha$ (not shown as a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

4.8 Implementation Consideration

We implemented the episode discovery algorithm as a ProM 6 plug-in (see also Figure 7), 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 used three different event logs for our experiment. The first event log, *bigger-example.xes*, is an artificial event log from Chapter 5 of [1] and available via `http://www.processmining.org/event_logs_and_models_used_in_book`. The second and third event logs, *BPI_Challenge_2012.xes* and *BPI_Challenge_2013.xes*, are real life event logs available via `doi:10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f` and `doi:10.4121/uuid:500573e6-accc-4b0c-9576-aa5468b10cee` respectively. The experiment consists of two parts: first a series of tests focused on performance and the number of discovered episodes, and second, a case study focused on comparing our technique with existing discovery techniques. For these experiments we used a laptop with a Core i7-4700MQ CPU (2.40 GHz), Java SE Runtime Environment 1.7.0_67 (64 bit) with 4 GB RAM.

5.2 Performance and Number of Discovered Episodes

In Table 3 some key characteristics of the event logs are given. We examined the effects of the parameters *minFreq*, *minActFreq* and *maxTraceDist* on the running

time, the discovered number of episodes (number of results), and the total number of intermediate candidate episodes. In Figure 7 an indication (screenshots) of the ProM plugin output is given.

Table 3. Metadata for the used event logs.

	# traces	*# variants*	*# activities*	*events / trace* *avg.*	*min.*	*max.*
bigger-example.xes	1,391	21	8	5.42	5	17
BPI_Challenge_2012.xes (BPIC 2012)	13,087	4,366	36	20.05	3	175
BPI_Challenge_2013.xes (BPIC 2013)	7,554	2,278	13	8.68	1	123

In Figures 4, 5, and 6 the results of the experiments are given.

The metric "# Episodes (result)" indicates the size of the end result. This metric is given by $|\Gamma|$ in Algorithm 1. The metric "# Candidate episodes" indicates the size of the intermediate results, after episode construction and pruning, but before occurrence checking. This metric is calculated by summing $|C_l|$ across iterations in both discovery phases in Algorithm 1. The "runtime", indicates the average running time of the algorithm, and its associated 95% confidence interval. Note that the scale of the runtime is in milliseconds.

As can be seen in the experimental results, 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 *minFreq* in Figure 4(b), then a relatively large class of episodes seems to become frequent, thus increasing the running time dramatically.

For a reasonably low number of frequent episodes (< 500, more will a human not inspect), the algorithm turns out to be quite fast (under one second). 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. Also note the increasing number of candidate episodes in Figure 5(b), which consists solely of parallel episodes, but there is no significant change in the runtime.

An analysis of the effects of changing the *minFreq* parameter (Figure 4(a), 4(b), and 4(c)) 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 BPIC 2012 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(a), 5(b), and 5(c)), there seems to be a cutoff point that separates frequent from infrequent activities. Small changes around this cutoff point may have a noticeable effect on the number of episodes discovered.

Finally, for the *maxTraceDist* parameter (Figure 6(a), 6(b), and 6(c)), 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 large 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 4(a) appears to be far more fine-grained than the real life event log (*BPIC 2012*) shown in Figure 4(b) and 4(c). 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 fine-grained pattern occurs. Most of the increase in frequent episodes, for decreasing $minFreq$, is again in the partial order discovery phase.

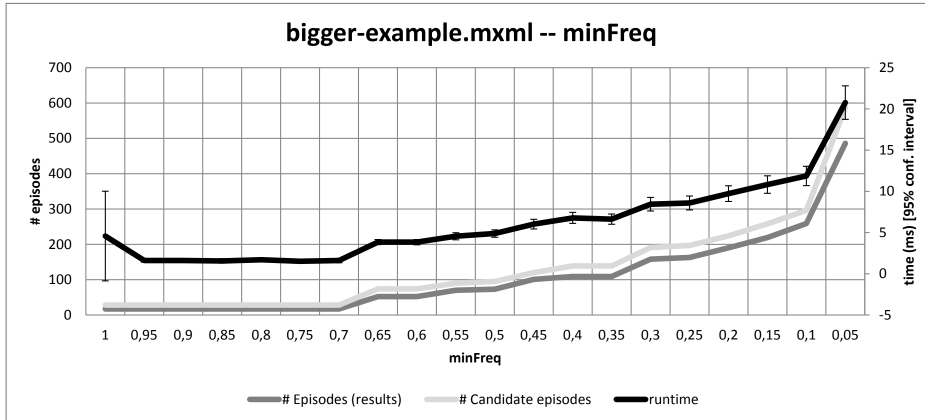
Table 4. Case Study results – Comparison of discovered sub-patterns per discovery algorithm. In the top part of this table, an x in two consecutive rows a and b indicate a sub-pattern $a \leq b$. In the bottom part of this table, a + indicates the corresponding patterns was revealed by the corresponding discovery algorithm output.

Activities and pattern	`A_SUBMITTED+COMPLETE`	x				
	`A_PARTLYSUBMITTED+COMPLETE`	x	x			x
	`A_PREACCEPTED+COMPLETE`		x	x		
	`W_Complementeren_aanvraag+SCHEDULE`			x	x	
	`W_Complementeren_aanvraag+START`				x	
	`A_DECLINED+COMPLETE`					x
Discovery algorithms	**Episode Discovery**	+	+	+	+	+[a]
	α-algorithm [11]	+				
	Heuristics miner [14]	+		+	+	
	Inductive miner [22]	+	+[b]	+[b]	+	+[b]
	DECLARE miner [23]	+	+[c]	+	+	+[c]

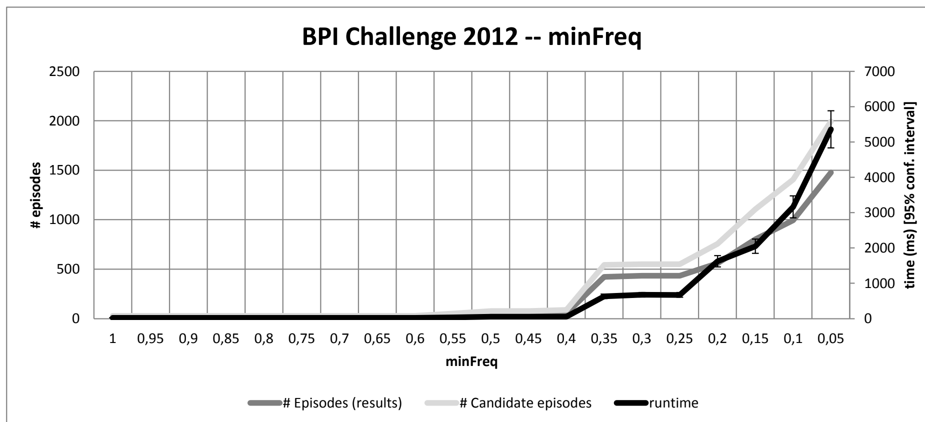
[a] Indicates the pattern was revealed, but only after increasing $maxTraceDist$.
[b] Indicates the pattern was revealed, but obfuscated by choice constructs.
[c] Due to the aggregated overview of the DECLARE model, it is not immediately clear that these patterns are disjoint.

5.3 Case Study – Pattern Discovery Compared with Existing Algorithms

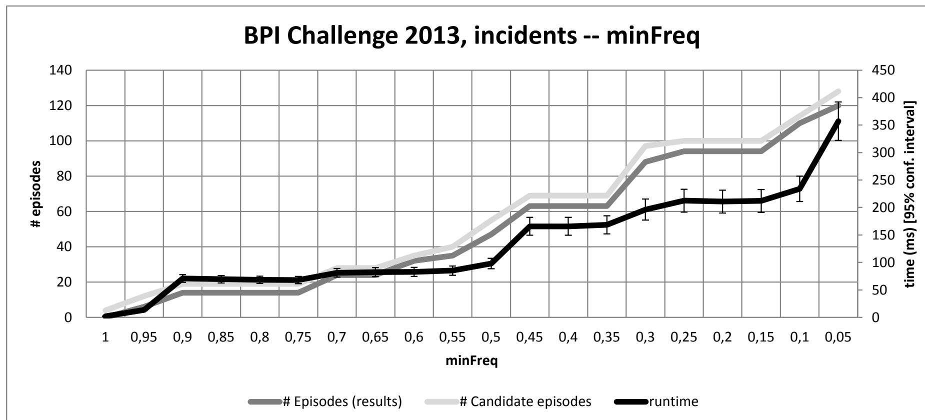
As noted in the introduction, often the overall end-to-end process models are rather complicated. Therefore, the search for local patterns (i.e., episodes) is interesting. In this section we perform a short case study using the BPI Challenge 2012, an event log of a loan application process. We explored this event log using: the α-algorithm [11], Heuristics miner [14], Inductive miner [22], DECLARE Miner [23], and our Episode Discovery technique. For this case study, we assume no prior knowledge about this event log. Instead, we want to get initial insight into the recorded behavior, and are interested in the most important patterns. For all the algorithms we use the default parameter settings and the "Activity classifier" defined in the event log (the default values are provided in



(a) Event log: bigger-example.mxml , $minActFreq = 1.0$, $maxTraceDist = 4$

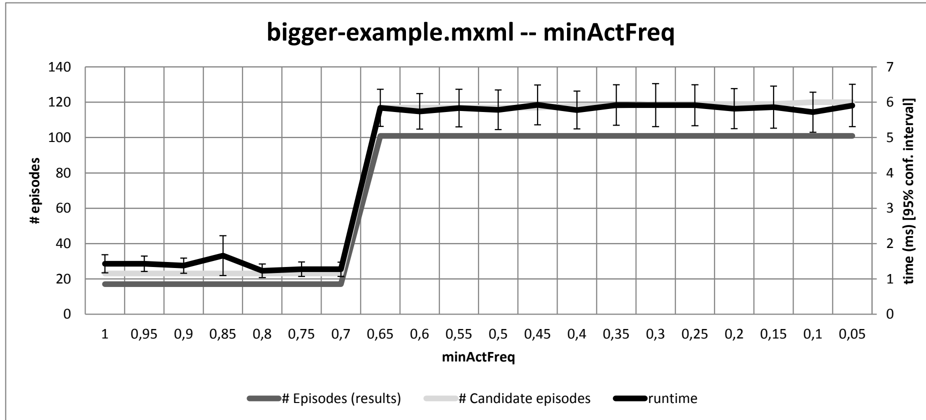


(b) Event log: BPI Challenge 2012 , $minActFreq = 1.0$, $maxTraceDist = 4$

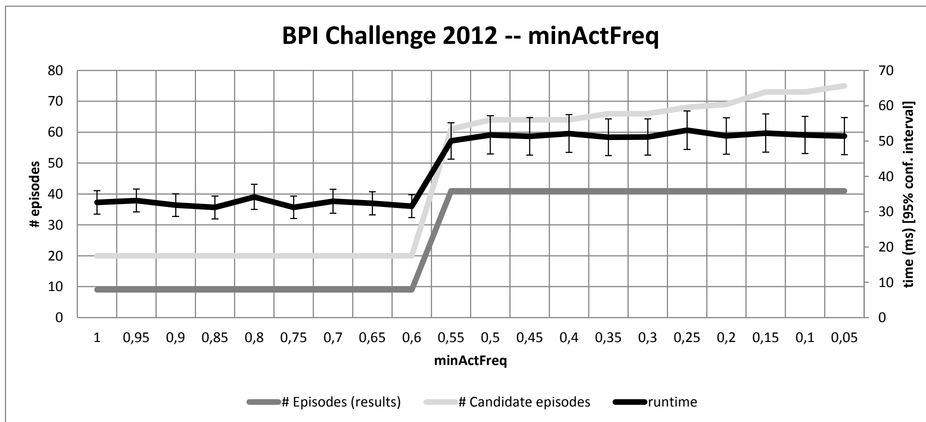


(c) Event log: BPI Challenge 2013, incidents , $minActFreq = 1.0$, $maxTraceDist = 4$

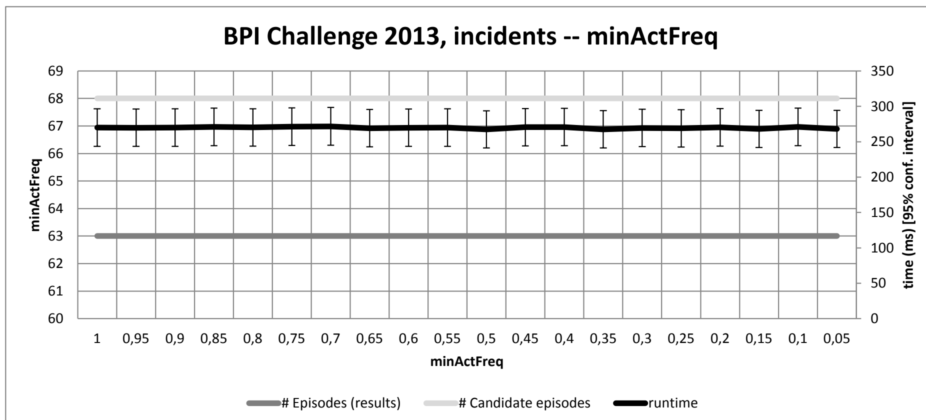
Fig. 4. Effects of the parameter *minFreq* on the number of results and candidate episodes. Observe that the *minFreq* parameter gives us fine-grained control of the number of results. Note that for less than 500 result episodes, the runtime is less than one second.



(a) Event log: bigger-example.mxml , $minFreq = 0.45$, $maxTraceDist = 4$

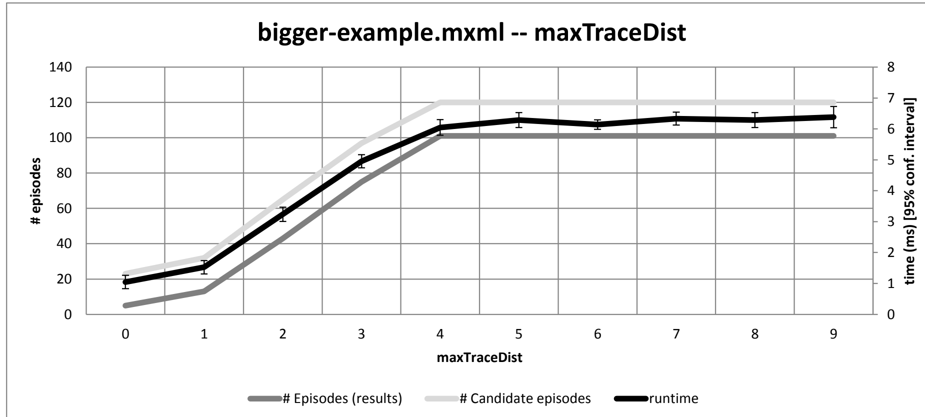


(b) Event log: BPI Challenge 2012 , $minFreq = 0.50$, $maxTraceDist = 4$

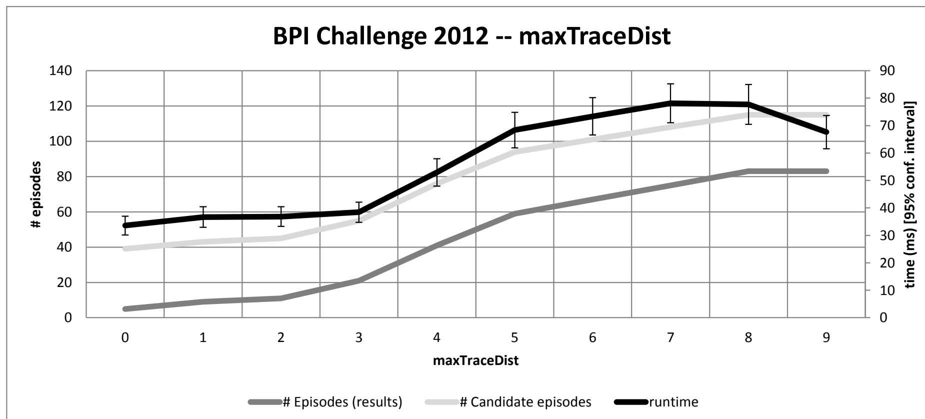


(c) Event log: BPI Challenge 2013, incidents , $minFreq = 0.45$, $maxTraceDist = 4$

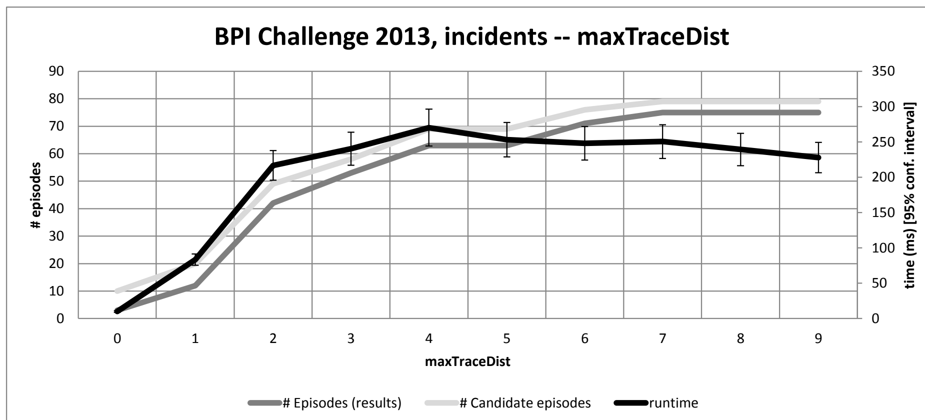
Fig. 5. Effects of the parameter *minActFreq* on the number of results and candidate episodes. Observe that there seems to be a cutoff point that separates frequent from infrequent activities. Note that the runtime is never greater than a third of a second.



(a) Event log: bigger-example.mxml , $minFreq = 0.45$, $minActFreq = 0.65$



(b) Event log: BPI Challenge 2012 , $minFreq = 0.50$, $minActFreq = 0.55$



(c) Event log: BPI Challenge 2013, incidents , $minFreq = 0.45$, $minActFreq = 1.00$

Fig. 6. Effects of the parameter *maxTraceDist* on the number of results and candidate episodes. Observe that *maxTraceDist* seems to have a sweet-spot where a low – but not too low – number of episodes are discovered. Note that the runtime is never greater than a third of a second.

the footnotes). The observations made below are summarized in Table 4. Experiments show that only the Episode Discovery was able to unobfuscated and unambiguously discover all the mentioned patterns.

Episode Discovery With our Episode Discovery technique we get a small overview of twelve frequent episodes (Figure 7(a)). Inspecting these episodes more closely, we find two frequent patterns: the order `A_SUBMITTED+COMPLETE` ≤ `A_PARTLYSUBMITTED+COMPLETE` ≤ `A_PREACCEPTED+COMPLETE`, and the order `A_PREACCEPTED+COMPLETE` ≤ `W_Complementeren_aanvraag+SCHEDULE` ≤ `W_Complementeren_aanvraag+START` (Figure 7(b)). The interpretation of these patterns is twofold. One, frequently whenever a loan application is submitted it either preaccepted or declined. And two, frequently whenever a loan application is preaccepted, additional information is requested ("Complementeren aanvraag"). Clearly, we found a simple overview of the most important patterns in the event log. After increasing the $maxTraceDist$ parameter to fifty (50), we also discover the pattern`A_PARTLYSUBMITTED+COMPLETE` ≤ `A_DECLINED+COMPLETE` (see Figure 7(c)). In the remaining paragraph, we focus on finding patterns using other discovery techniques, and we are particularly interested in finding similar patterns.

α-algorithm [1] Figure 8(a) shows the overall Petri net model produced by the α-algorithm [11]. Closer inspection of the bottom-left part (Figure 8(b)) reveals the sub-pattern `A_SUBMITTED+COMPLETE` ≤ `A_PARTLYSUBMITTED+COMPLETE`. The remaining of the previously discovered frequent patterns are not clearly visible in this model. No other patterns were discovered.

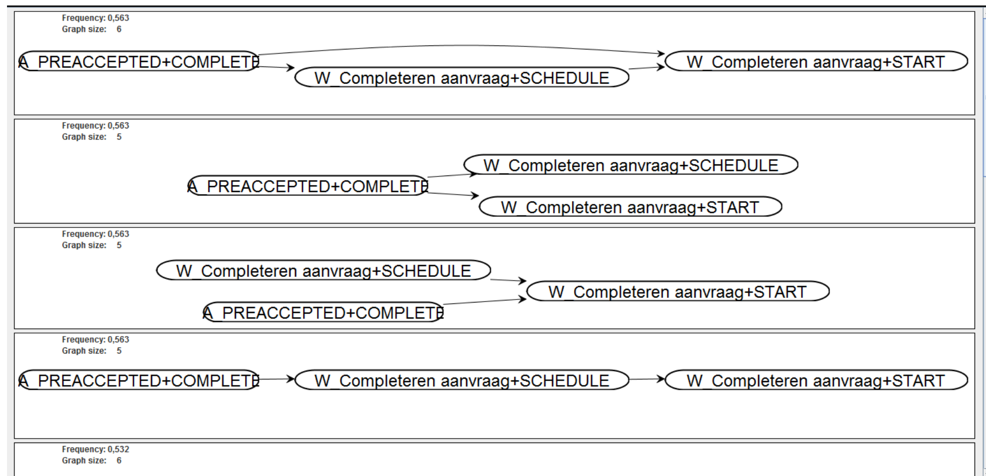
Heurisitcs miner [2] The heuristics net in Figure 9(a) is produced by the Heuristics miner [14]. Closer inspection of this net (Figure 9(b)) reveals two sub-patterns: the order `A_SUBMITTED+COMPLETE` ≤ `A_PARTLYSUBMITTED+COMPLETE`, and the order `A_PREACCEPTED+COMPLETE` ≤ `W_Complementeren_aanvraag+SCHEDULE` ≤ `W_Complementeren_aanvraag+START`. However, the sub-pattern `A_PARTLYSUBMITTED+COMPLETE` ≤ `A_PREACCEPTED+COMPLETE` and `A_PARTLYSUBMITTED+COMPLETE` ≤ `A_DECLINED+COMPLETE` were not clearly visible in this model. No other patterns were discovered.

Inductive miner [3] Figure 10(a) shows the overall process model (a process tree) produced by the Inductive miner [22]. All frequent patterns can be found in this model. However, as can be seen in the close-up in Figure 10(b), the choice constructs obfuscate these patterns. After detailed inspection of this model, and armed with our results from the Episode Discovery technique, we discovered one

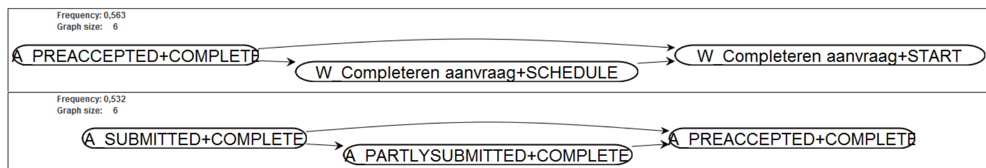
[1] **Plugin action:** "Mine for a Petri Net using Alpha-algorithm".
Parameters: n/a

[2] **Plugin action:** "Mine for a Heuristics Net using Heuristics Miner".
Parameters: Activity classifier, Relative-to-best = 5.0, Dependency = 90.0, Length-one-loops = 90.0, Length-two-loops = 90.0, Long distance = 90.0, All tasks connected = On, Long distance dependency = Off, Ignore loop dependency tresholds = On

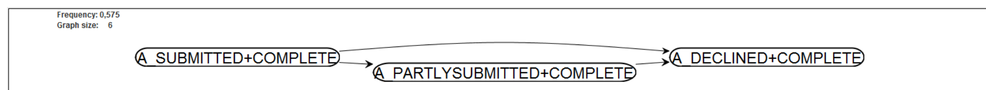
[3] **Plugin action:** "Mine process tree with Inductive Miner".
Parameters: Variant = Inductive Miner - infrequent, Noise threshold = 0.20, Event classifier = Event Name



(a) View of discovered episodes (twelve in total)

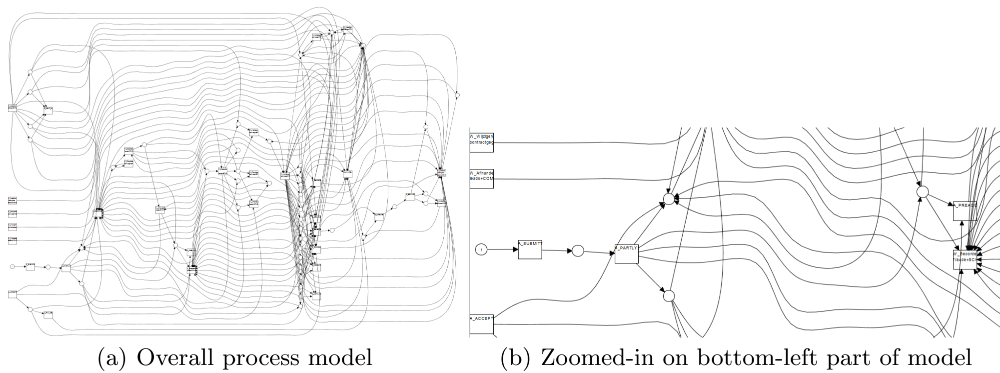


(b) The two most interesting episodes



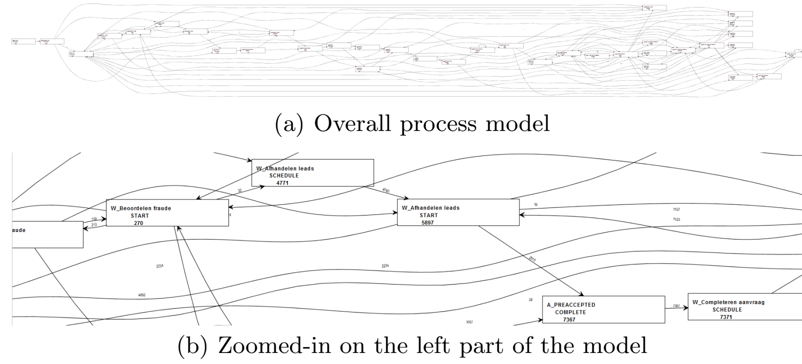
(c) Additional pattern, discovered after increasing the $maxTraceDist$ parameter to fifty (50).

Fig. 7. Algorithm: Episode Discovery. Result in ProM for the BPIC 2012 event log.



(a) Overall process model (b) Zoomed-in on bottom-left part of model

Fig. 8. Algorithm: α-algorithm [11]. Result in ProM for the BPIC 2012 event log.

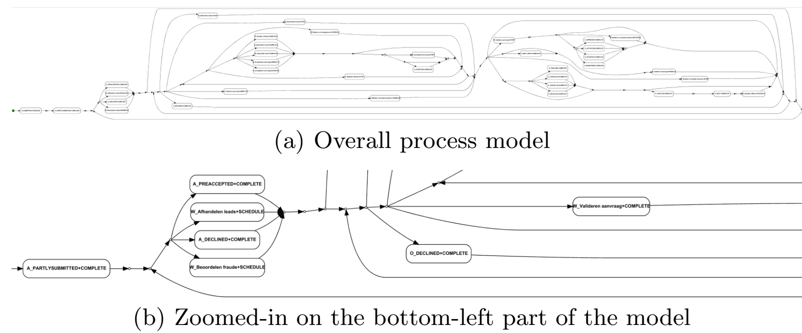


(a) Overall process model

(b) Zoomed-in on the left part of the model

Fig. 9. Algorithm: Heuristics miner [14]. Result in ProM for the BPIC 2012 event log.

less frequent pattern. We rephrase our first interpretation of the Episode Discovery results as: “whenever a loan application is submitted it frequently either preaccepted or declined, or in some rare cases followed by a fraud detection” (“Beoordelen fraude”).



(a) Overall process model

(b) Zoomed-in on the bottom-left part of the model

Fig. 10. Algorithm: Inductive miner [22]. Result in ProM for the BPIC 2012 event log.

DECLARE Miner [4] Finally, in Figure 11, the DECLARE model is given, as produced by the DECLARE Miner [23]. In this case we did change the following parameters: we chose the *succession* template and set the min support to 50 (comparable to the default settings of Episode Miner). As can be observed, all the frequent patterns can be found. However, note that due to the aggregated overview of the DECLARE model, it is not immediately clear that

[4] **Plugin action:** “Declare Maps Miner”.
Parameters: Selected Templates = {succession}, All Activities (considering Event Types), Min. support = 50, Alpha = 0, Control_Flow = On, Time = Off

the patterns A_PARTLYSUBMITTED+COMPLETE $\leq$ A_PREACCEPTED+COMPLETE and A_PARTLYSUBMITTED+COMPLETE $\leq$ A_DECLINED+COMPLETE are disjoint. No other patterns were discovered.

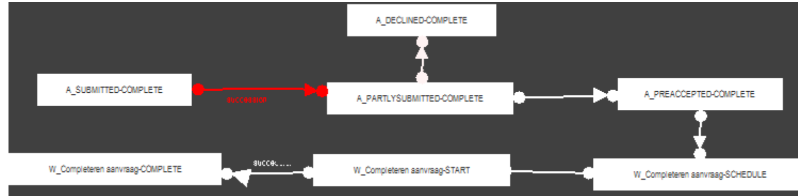


Fig. 11. Algorithm: DECLARE Miner [23]. Result in ProM for the BPIC 2012 event log, using the *succession* template and a min support of 50.

As demonstrated in this case study, and summarized in Table 4, overall end-to-end process models can be rather complicated, and the search for local patterns (i.e., episodes) quickly reveals important insight into recorded behavior.

5.4 Case Study – Runtime Compared with Existing Algorithms

After showing the insights that can be gained by our algorithm, we now compare the running time of our approach with existing algorithms. We revisit the same set of algorithms, and investigate the average running time on all three event logs. The same (default) parameter settings are used as in the previous section (see footnotes footnotes 1–4).

The resulting running times are compared in Figure 12. Note that the runtime is shown in milliseconds, on a logarithmic scale. Broadly speaking, the discovery algorithms can be grouped in three classes, based on their runtime. Our episode miner and the alpha miner form the fastest class of discovery algorithms. Next is the class of algorithms to which the heuristics and inductive miner belong. These algorithms are roughly ten times slower than the first class. Finally, there is the class of the declare miner. This algorithm is roughly a hundred times slower than the first class.

Looking at the difference between the BPIC 2012 and 2013 logs, we see observe the 2012 log has more event classes (36 for 2012, 13 for 2013), more traces (13,087 for 2012, 7,554 for 2013), and longer traces (avg. 20.05 for 2012, avg. 8.68 for 2013). This increase in size is directly observable in terms of running time for the existing algorithms, but has a less effect on the running time of the episode miner (with default settings).

We conclude that our Episode Discovery realization is among the fastest of algorithms. In particular, it is orders of magnitude faster than the Declare Miner configured to discover only succession relations.

5.5 Case Study – Episode Rules

Continuing with our case study of the BPI Challenge 2012 event log, we also take a look at the discovery of association rules. Here we use the episode rule generation feature of our Episode Discovery ProM plugin, and used the default settings.

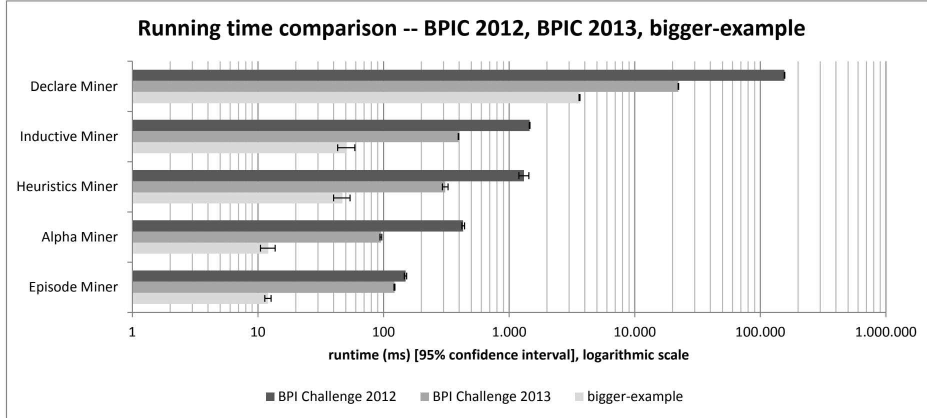


Fig. 12. Comparison of the running time for the different discovery algorithms used in the case study. The runtime is shown in milliseconds, on a logarithmic scale. We distinguish three classes based on runtimes: 1) our Episode miner and the α-miner, 2) the class of algorithms to which the Heuristics and Inductive miner belong, and 3) the class of the Declare miner.

The result consists of six episode rules, one of which is shown in Figure 13. The interpretation of the shown episode rule is as follows: "If we saw `A_PARTLYSUBMITTED+COMPLETE` $\leq$ `A_PREACCEPTED+COMPLETE` occurring, we likely will also see `W_Complementeren_aanvraag+SCHEDULE` occurring next". In other words, whenever a partially submitted request was preaccepted, it is likely that we will request additional information ("Complementeren aanvraag").

Similar, episode rules can be used in an online setting to predict likely follow-up activities using episodes discovered in historical data.

Fig. 13. Episode rules discovered in ProM for the BPIC 2012 event log. The black solid line indicates the assumed partial order (the β in $\beta \Rightarrow \alpha$), the red dashed line indicates the added pattern (the α).

5.6 Case Study – Alternative Perspective: Resources

We conclude our case study of the BPI Challenge 2012 event log with mining patterns in the flow of work between persons. For this we used the Resource classifier defined in the event log. We explored this perspective using: the Inductive miner [22], Handover of Work Social Network miner [26], and our Episode Discovery technique.

The discovered episodes are shown in Figure 14. The vertices in these results represent resources instead of activities. The first pattern shows that the resource 112 is present in all traces (based on the observation that $freq(\texttt{112} \leq \texttt{112} \leq \texttt{112}) = 1.0$). Furthermore, we also discover that in most cases work is passed from the resource 112 to tasks without a recorded resource (e.g., automated tasks). Activities conducted by "no recorded resource" can be observed in Figure 14 as empty vertices.

Figure 15(a) shows the overall process model (a process tree) for the resource perspective, produced by the Inductive miner [22]. At first glance no obvious pattern is visible. In the close-up in Figure 15(b), the resource 112 and "no recorded resource"/"empty resource" are visible, but no clear patterns are visible.

In Figure 15(c) the handover of work social network is given, as produced by the organizational miner [26]. Most of the resources are forming one big tightly-connected cluster. The "no recorded resource"/"empty resource" is completely disconnected, but the resource 112 is not easily found (it is in the top-left corner). The patterns found by the Episode Miner cannot be deduced from this social network.

By using the resource perspective in combination with Episode Discovery, we gained insight into the most important resources, and the flow of work between resources. This demonstrates that Episode Discovery is not only useful in the activity-focused control-flow perspective, but also in other perspectives. While we only showed pattern discovery in the control-flow and resource domain, other perspectives are possible. One example is discovering the flow of work between event locations (e.g., system components or organization departments generating the events). Another example is discovering the relations between data attributes (e.g., which information is used in which order).

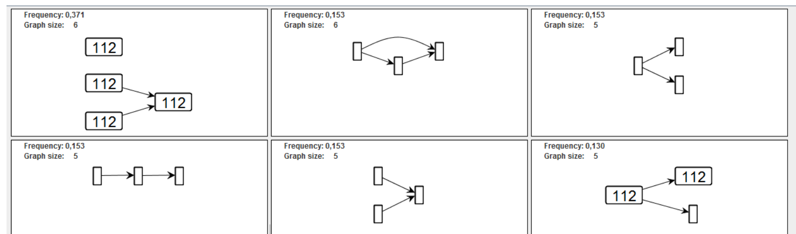
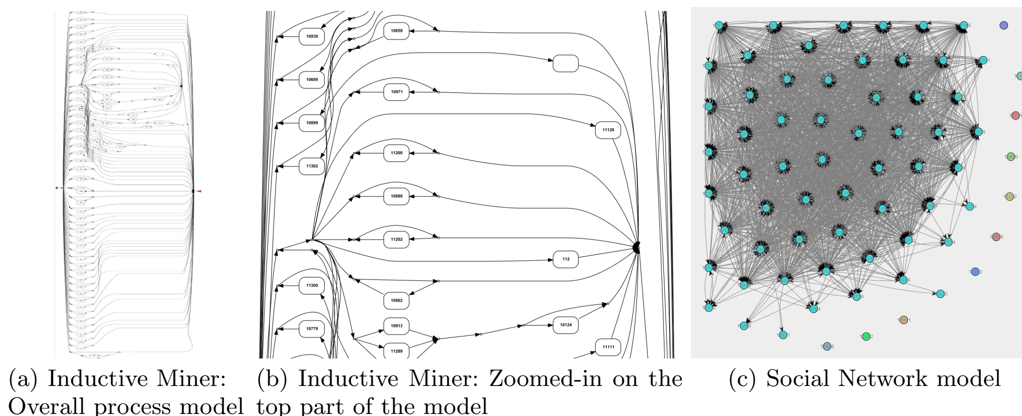


Fig. 14. Episodes discovered in ProM for the BPIC 2012 event log, using the Resource classifier. In total, forty episodes were discovered. Note that the vertices in these results represent resources instead of activities. The empty vertices indicate the absence of a recorded resource (e.g., automated tasks).

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



(a) Inductive Miner: Overall process model (b) Inductive Miner: Zoomed-in on the top part of the model (c) Social Network model

Fig. 15. Result in ProM for the BPIC 2012 event log, using the Resource classifier. Algorithms: Inductive miner [22], Handover of Work Social Network miner [26].

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, for a reasonably low number of frequent episodes, the algorithm turns out to be quite fast (under one second); typically faster than existing many algorithms. The main problem is the correct setting of the episode pruning parameters *minFreq*, *minActFreq*, and *maxTraceDist*. In addition, comparison with existing discovery algorithms has shown the benefit of episode mining in getting insight into recorded behavior. Moreover, we have demonstrated the usefulness of episode rules that can be discovered. Finally, the applicability of Episode Discovery for other perspectives (like the resources perspective) was shown.

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 [28]. 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* 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 [29]. 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 can be 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. In Grau, A., Zurawski, R., eds.: IEEE Emerging Technology and Factory Automation (ETFA 2014), IEEE Computer Society (2014) 1–8
[8] Fahland, D., van der Aalst, W.M.P.: Model Repair: Aligning Process Models to Reality. Volume 47. (January 2015) 220–243
[9] Leemans, M.: Episode Miner. `https://svn.win.tue.nl/repos/prom/Packages/EpisodeMiner/` [Online, accessed 9 Januari 2015].
[10] Laxman, S., Sastry, P.S., Unnikrishnan, K.P.: Fast Algorithms for Frequent Episode Discovery in Event Sequences. In: Proceedings of the 3rd Workshop on Mining Temporal and Sequential Data. SIGKDD, Seattle, WA, USA, Association for Computing Machinery, Inc. (August 2004)
[11] 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
[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] 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)
[15] 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
[16] 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
[17] 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
[18] 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
[19] 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
[20] 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
[21] Günther, C.W., van der Aalst, W.M.P.: Fuzzy mining - -adaptive process simplification based on multi-perspective metrics. In: Business Process Management. Springer (2007) 328–343
[22] 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
[23] Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: User-guided discovery of declarative process models. In: Computational Intelligence and Data Mining (CIDM), 2011 IEEE Symposium on, IEEE (2011) 192–199
[24] Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: Efficient discovery of understandable declarative process models from event logs. In: Advanced Information Systems Engineering, Springer (2012) 270–285
[25] Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: A knowledge-based integrated approach for discovering and repairing declare maps. In: Advanced Information Systems Engineering, Springer (2013) 433–448
[26] Song, M., van der Aalst, W.M.P.: Towards comprehensive support for organizational mining. Decision Support Systems **46**(1) (2008) 300–317
[27] 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
[28] van der Aalst, W.M.P.: Decomposing Petri Nets for Process Mining: A Generic Approach. Distributed and Parallel Databases **31**(4) (2013) 471–507
[29] Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM **51**(1) (2008) 107–113