Discovering Infrequent Behavioral Patterns in Process Models

David Chapela-Campa, Manuel Mucientes, and Manuel Lama

Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS)
Universidade de Santiago de Compostela. Santiago de Compostela, Spain
{david.chapela, manuel.mucientes, manuel.lama}@usc.es

Abstract. Process mining has focused, among others, on the discovery of frequent behavior with the aim to understand what is mainly happening in a process. Little work has been done involving uncommon behavior, and mostly centered on the detection of anomalies or deviations. But infrequent behavior can be also important for the management of a process, as it can reveal, for instance, an uncommon wrong realization of a part of the process. In this paper, we present WoMine-i, a novel algorithm to retrieve infrequent behavioral patterns from a process model. Our approach searches in a process model extracting structures with sequences, selections, parallels and loops, which are infrequently executed in the logs. This proposal has been validated with a set of synthetic and real process models, and compared with state of the art techniques. Experiments show that WoMine-i can find all types of patterns, extracting information that cannot be mined with the state of the art techniques.

Keywords: infrequent patterns, process mining, process discovery

1 Introduction

One of the aims of process mining during the past years has been, among others, the study of frequent behavior in order to focus on the more common parts of a process during the different tasks of process mining —discovery, monitoring, and enhancement. Under this scope, several algorithms have been proposed to discover process models covering the most common behavior [1,2,3], and to search frequent structures either directly in the logs [4,5] or extracting the structures from the process models [6]. The search of infrequent cases —deviations or anomalous traces— has been also used during the discovery of a process model, removing them to reduce the complexity of the model without a high decrease in the fitness [7,8]. Nevertheless, the discovery of infrequent behavior can be also interesting in order to monitor and enhance a process, and discarding it might not be a proper solution.

There are scenarios where an infrequent subprocess in the model can hint a wrong behavior which must be examined. For instance, in insurance companies, infrequent behavior can be used to recognize fraudulent claims [9]. It can be also useful to detect intrusions in networks [10], or failures in software behavior [11].

Additionally, in well-structured processes, the behavior supported by a model is designed and expected to be executed. A substructure of the model with a low frequency of execution can hint a path in the process that must be reinforced in order to increase its frequency or, conversely, where the assigned resources could be restructured to optimize the process.

There are few approaches related with the search of infrequent behavior, most of them focused on the detection of uncommon anomalous traces in process logs [7,8,12,13]. Nevertheless, these techniques focus on the identification of infrequent traces considering the whole trace as a unit. Further knowledge can be obtained searching for infrequent patterns, as they allow to focus on infrequent subprocesses, instead of discovering infrequent full traces —an infrequent trace can contain frequent behavior, hindering the study of infrequent behavior.

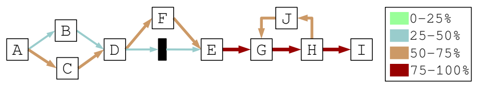
In this paper we present WoMine-i, a novel algorithm to detect infrequent behavioral patterns from a process model, measuring their frequency with the instances of the log. The main novelty of our approach is that it can detect infrequent substructures of the process model, i.e., behavioral patterns, with all type of structures —sequences, selections, parallels and loops. The ability to work with these structures prevents WoMine-i of interpreting the traces as sequences of events. Furthermore, the extracted information allows to focus on infrequent subprocesses, and not to analyze infrequent full traces. The algorithm has been qualitatively compared using various synthetic process models with all related techniques, showing our algorithm finds the correct infrequent patterns and estimates precisely its frequency while related techniques do not. Experiments have also been run with real logs of two Business Process Intelligence Challenges, 2012 and 2013.

2 Related Work

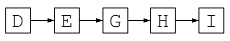
There are no approaches in the literature focused directly on the extraction of infrequent substructures in a process. There are a few techniques, like Heat Maps, that can be used to detect infrequent behavior in a process model, although that is not their main objective. Heat Maps provide a simple way to highlight the frequent structures of a process model considering the individual frequency of each arc. If the arcs with a frequency higher than a threshold are removed, the remaining structures are formed by infrequent arcs. The drawback of this approach is that the frequency of each arc is measured individually. An infrequent pattern can be composed by arcs that are individually frequent and, therefore, they will not be part of the result of this technique. Fig. 1 shows an example of a process model represented by a C-net, and the result of the Heap Maps technique over this model (Fig. 1a). As can be seen, there are arcs —e.g. $(E \rightarrow G)$— executed in all the traces of the log which are part of an infrequent pattern (Fig. 1b).

In [7], a state automaton with each state representing an activity of the log is built. A valuated arc between two states is added when one of them is followed by the other one in the log. Its value increases as this relation appears in the log.

A: Start course	B: Take class with Dominic
C: Take class with Marcus	D: Study first chapter
E: Study second chapter	F: Take optional class
G: Do exam	H: Revise correction of exam
I: Pass exam	J: Retake class (failed exam)



(a) C-net with the arcs highlighted depending on its individual frequency (Heat Maps).



(b) Infrequent pattern with a frequency of 5%.

Id	Trace	
0	ABDFEGHI	(×16)
1	ABDEGHI	(×3)
2	ABDFEGHJGHI	(×6)
3	ABDFEGHJGHJGHI	(×4)
4	ABDEGHJGHI	(×8)
5	ABDEGHJGHJGHJGHI	(×11)
6	ACDFEGHI	(×19)
7	ACDEGHI	(×2)
8	ACDFEGHJGHI	(×3)
9	ACDFEGHJGHJGHJGHJGHI	(×8)
10	ACDEGHJGHI	(×11)
11	ACDEGHJGHJGHI	(×9)

(c) 100-trace log.

Fig. 1: An example of a process model with an infrequent pattern that cannot be discovered through related techniques.

Afterwards, the infrequent arcs are used to filter infrequent traces. The drawback of this technique is the same as the Heat Maps approach, because the frequency is measured individually. Furthermore, the automaton interprets the log as a sequence, without parallels nor other dependencies.

The technique used by Lu et al. in [13] also performs a filtering of traces using the infrequent parts of a process model. In this case, models are built by merging the behavior in a subset of traces. The drawback of this approach is also its inability to measure the frequency of a structure as a whole, analyzing instead the number of individual executions of an arc.

Bezerra et al. search in [12] for infrequent or anomalous traces in the log analyzing the whole trace. They present three approaches to filter infrequent traces depending on their frequency and conformance. The drawback of this technique is that it takes into account the whole trace, and instead of an infrequent pattern, this approximation returns a set of traces. This makes impossible, without further analysis, to know which parts of the traces are infrequent. Fig. 1c shows an example: this log contains four instances with a frequency under 5% (1, 3, 7, 8). Two of them (1, 7) contain the pattern from Fig. 1b, but they also contain frequent patterns as A–B–D, A–C–D or G–H–J–G–H–I. Trace-clustering techniques [14] could also be used to obtain traces containing infrequent behavior, but the problem would be the same.

Finally, techniques searching for frequent structures could be adapted to search infrequent behavior, inverting the main search. The drawback of this alternative lies in the way these algorithms measure the frequency of the patterns. For instance, the approach of Tax et al. [5] performs an alignment-based method to detect if the pattern is executed in a trace. When an activity from the trace does not appear in the pattern, the method performs a *move on log* without a penalty because this activity might belong to a parallel branch in the model. This method gets a frequency of 40% for the pattern in Fig. 1b (traces 0, 1, 6, 7) which is far from the real value (5%). The shortcoming of this method is that it analyses the frequency based in the order of the activities in the log, not in the real path that is being executed in the model. Similarly, pattern-based search techniques that only use the log to extract frequent behavior would present the same drawback. Furthermore, the search space of these approaches would be extremely large, as they do not use the model to build the patterns.

As far as we know there is not algorithms to retrieve infrequent patterns from a process model. The algorithm presented in this paper, WoMine-i, is able to retrieve infrequent subgraphs ensuring the low frequency of the entire structure. This allows to focus on infrequent subprocesses and to abstract from the traces containing them, simplifying the analysis of the process.

3 Preliminaries

In this paper, we will represent the examples with place/transition Petri nets [15] due to its comprehensibility. Nevertheless, our algorithm represents the process with a Causal net (Def. 1).

Definition 1 (Causal net [16]). A Causal net (C-net) is a tuple $C = (A, a_i, a_o, D, I, O)$ where:

- A is a finite set of activities;
- $a_i \in A$ is the start activity;
- $a_o \in A$ is the end activity;
- $D \subseteq A \times A$ is the dependency relation,
- $AS = \{X \subseteq \mathcal{P}(A) \mid X = \{\emptyset\} \vee \emptyset \notin X\}$;[1]
- $I \in A \rightarrow AS$ defines the set of possible input bindings per activity;
- $O \in A \rightarrow AS$ defines the set of possible output bindings per activity,

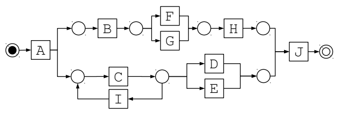
such that:

- $D = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup_{as \in I(a_2)} as\}$;
- $D = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup_{as \in I(a_1)} as\}$;
- $\{a_i\} = \{a \in A \mid I(a) = \{\emptyset\}\}$;
- $\{a_o\} = \{a \in A \mid O(a) = \{\emptyset\}\}$;
- all activities in the graph (A, D) are on a path from a_i to a_o.

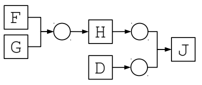
Definition 2 (Trace). Let A be the set of activities of a process model, and ε an event —the execution of an activity $\alpha \in A$. A trace is a list (sequence) $\tau = \varepsilon_1, ..., \varepsilon_n$ of events ε_i occurring at a time index i relative to the other events in τ. Each trace corresponds to an execution of the process, i.e., a process instance.

Definition 3 (Log). An event log $L = [\tau_1, ..., \tau_m]$ is a multiset of traces τ_i. In this simple definition, events only specify the name of the activity, but usually, event logs store more information as timestamps, resources, etc.

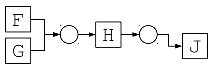
[1] $\mathcal{P}(A) = \{A' \mid A' \subseteq A\}$ is the powerset of A. Hence, elements of AS are *sets of sets* of activities.



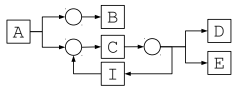
(a) Petri net, with parallels, selections, and a loop.



(b) Valid pattern with a selection and a parallel.



(c) Invalid pattern. `J` has incomplete input combinations —{{`H`}} $\not\subseteq$ {{`H,D`}, {`H,E`}}.



(d) Valid pattern with a parallel, a selection and a loop.

Fig. 2: Examples of a process model, valid and invalid patterns.

Definition 4 (Pattern). Let $C = (A, a_i, a_o, D, I, O)$ be a C-net representing a process model M. A connected subgraph represented by the C-net $P = (A', A'_i, A'_o, D', I', O')$, where $A'_i \subseteq A'$ and $A'_o \subseteq A'$ represent respectively the start and end activities, is a pattern of M if and only if:

- $A' \subseteq A$;
- $D' \subseteq D$;
- for any $\alpha \in A'$: $I'(\alpha) \subseteq I(\alpha), O'(\alpha) \subseteq O(\alpha)$

A *pattern* (Def. 4) is a subgraph of the process model that represents the behavior of a part of the process. For each activity α in the pattern, its inputs, $I'(\alpha)$, must be a subset of $I(\alpha)$; and the outputs, $O'(\alpha)$, must be also a subset of $O(\alpha)$. This ensures that a pattern has not a partial parallel connection. Fig. 2 shows some examples of valid and invalid patterns.

Definition 5 (Simple pattern). A pattern $P = (A', A'_i, A'_o, D', I', O')$ is a simple pattern if and only if, for all activities $\alpha \in A'$:

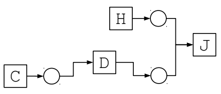
- $[\exists! \Phi \in I'(\alpha) \colon \Phi \not\subseteq R^+_\alpha] \vee [\forall \Phi \in I'(\alpha) \colon \Phi \subseteq R^+(\alpha)]$;
- $[\exists! \Theta \in O'(\alpha) \colon \Theta \not\subseteq R^-_\alpha] \vee [\forall \Theta \in O'(\alpha) \colon \Theta \subseteq R^-(\alpha)]$

Being R^+_α the set of successors[2] of an activity α, and R^-_α the set of predecessors [3] of an activity α.

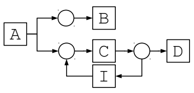
The simple patterns (Def. 5) are those patterns whose behavior can be entirely executed in at least one trace. If the inputs or outputs of an activity have a selection, it must be able to execute each path in the same trace —at most, one of the paths is not a loop. For this, the inputs of each activity α must have all activities reachable from α except, at most, the activities of one path. The outputs present the same constraint, but in this case they must reach α, not be reachable from α. Fig. 3 shows two valid simple patterns and an invalid one.

[2] The successors of an activity α are the activities with a path from α to them, e.g., the successors of `B` in Fig. 2a are `F`, `G`, `H` and `J`.

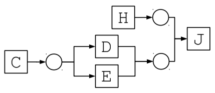
[3] The predecessors of an activity α are the activities with a path from them to α, e.g., the predecessors of `C` in Fig. 2a are `C`, `I` and `A`.



(a) Valid simple pattern. The pattern is executed in the instance `[C D H J]`.



(b) Valid simple pattern with a loop. The pattern is executed in the instance `[A B C I C D]`.



(c) Invalid simple pattern. `D` and `E` cannot be in the same instance of the pattern.

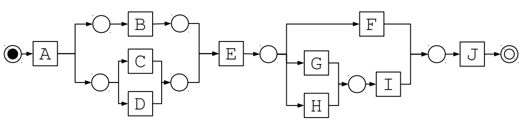
Fig. 3: Examples of valid and invalid simple patterns of the process model shown in Fig. 2a.

Definition 6 (Minimal pattern, *M*-pattern). Each activity of the process model belongs to, at least, one minimal pattern. The M-pattern of an activity α corresponds to the closure of α, i.e., the structure that is going to be executed when α is executed. An exception is made with parallel structures: if α has a parallel in its inputs or outputs, there must be an M-pattern containing each parallel path.

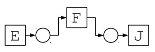
Given a C-net $C = (A, a_i, a_o, D, I, O)$ representing a process model M and an activity $\alpha' \in A$, a pattern $P = (A', A'_i, A'_o, D', I', O')$ is a Minimal Pattern of α' if and only if is a maximum simple pattern containing α' and fulfilling the following rules:

- if $|I(\alpha')| > 1$ then $[I'(\alpha') = \emptyset] \vee [|I'(\alpha')| = 1, \Phi \in I'(\alpha') \colon |\Phi| > 1]$;
- if $|O(\alpha')| > 1$ then $[O'(\alpha') = \emptyset] \vee [|O'(\alpha')| = 1, \Theta \in O'(\alpha') \colon |\Theta| > 1]$;
- $\forall \alpha \in R^+_{\alpha'} \colon$ if $|O(\alpha)| \neq 1$ then $O'(\alpha) = \emptyset$;
- $\forall \alpha \in R^-_{\alpha'} \colon$ if $|I(\alpha)| \neq 1$ then $I'(\alpha) = \emptyset$;
- $\forall \alpha \in A', \alpha \neq \alpha', \alpha \notin (R^+_{\alpha'} \bigcup R^-_{\alpha'}) \colon$ if $|I(\alpha)| \neq 1$ then $I'(\alpha) = \emptyset$, and if $|O(\alpha)| \neq 1$ then $O'(\alpha) = \emptyset$

In WoMine-i each activity α' is associated, at least, to an M-pattern. The M-patterns of an activity α' are obtained through an expansion process that starts in α' and continues through its inputs and outputs fulfilling the following rules: *i)* the process will not expand through the inputs of α' with size 1 and being part of a selection; *ii)* the same stands for the outputs of α'; *iii)* for all the successors of α' the expansion stops if the outputs are formed by a selection; *iv)*

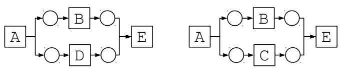


(a) Petri net of the process model.



(b) M-pattern of `F`.

(c) M-pattern of `J`.



(d) M-patterns of `A`.

Fig. 4: A process model and three examples of M-patterns.

the same stands for the inputs of the predecessors of α'; *v)* finally, the process does not expand either through the inputs or outputs of the activities not fitting the previous constraints if those are formed by an XOR structure in the model.

Fig. 4 shows some M-patterns of a model. Fig. 4b shows the M-pattern of `F`: the process starts in `F` and expands the M-pattern through `F` inputs and outputs, because both are formed by only one path. The backwards expansion stops in `E` because its inputs are part of a selection. Fig. 4c depicts the M-pattern of `J`. It is formed only by itself, because its inputs are part of a selection and its outputs are empty. Finally, Fig. 4d presents the two M-patterns of `A`. As `A` is an AND-split with a selection, two M-patterns are created, each one related to one of the possible paths.

Definition 7 (Candidate arcs). Let $C = (A, a_i, a_o, D, I, O)$ be a causal net representing a process model M. An arc $\langle \alpha_i \rightarrow \alpha_j \rangle \colon \alpha_i, \alpha_j \in A$ is part of the $A^<$ set, i.e., a candidate arc, if and only if:

- $O(\alpha_i) = \{\Theta \in AS \mid \Theta = \{\alpha_j\} \vee \alpha_j \notin \Theta\}$
- $I(\alpha_j) = \{\Phi \in AS \mid \Phi = \{\alpha_i\} \vee \alpha_i \notin \Phi\}$

The set of candidate arcs, or $A^<$, is a subset of the arcs in the model which are not part of an AND structure. For instance, all arcs of Fig. 4a, but those starting in `A` or ending in `E`, are included in the $A^<$ set.

Definition 8 (Compliance). Given a trace $\tau \in L$ and a simple pattern SP belonging to the process model, the trace is compliant with SP, denoted as $SP \vdash \tau$, when the replay of the trace in the process model contains the replay of the pattern, i.e., all the arcs and activities of SP are executed in a correct order, and each activity fires the execution of its output activities in the pattern.

Definition 9 (Frequency of pattern and simple pattern). Let L be the set of traces of the process log. The frequency of a simple pattern SP is the number of traces compliant with SP divided by the size of the log:

$$\mathit{freq}(SP) = \frac{|\{\tau \in L \colon SP \vdash \tau\}|}{|L|} \tag{1}$$

And the frequency of a pattern P is the maximum frequency of the simple patterns it represents:

$$\mathit{freq}(P) = \max_{\forall SP \in P} \mathit{freq}(SP) \tag{2}$$

Definition 10 (Infrequent Pattern). Given a frequency threshold $\sigma \in \mathbb{R} \colon 0 < \sigma \leq 1$, a pattern P is an infrequent pattern if and only if $\mathit{freq}(P) < \sigma$.

4 Infrequent Pattern Mining Algorithm

Given a process model and a set of instances, i.e., traces, the objective is to extract the subgraphs of the process model that are executed in a percentage of

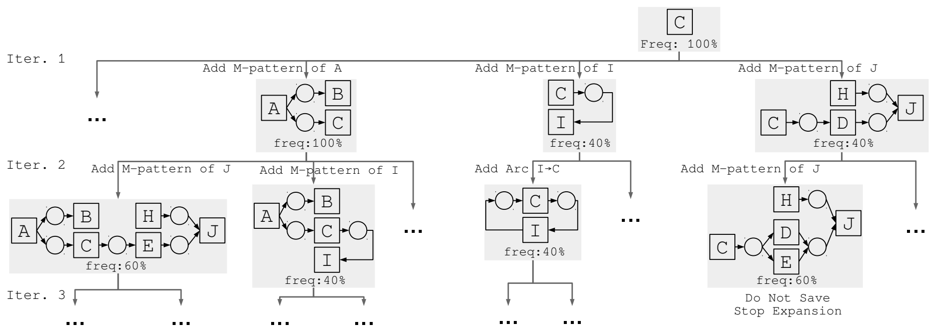


Fig. 5: Example of a part of the expansion process starting with the M-pattern of C. The example shows only three branches of expansion and two iterations. Some of the expansions have been omitted for the sake of clarity.

the traces under a threshold. A naive approach might be a brute-force algorithm, checking the frequency of every existent subgraph inside the process model, and retrieving the infrequent ones. The computational cost of this approach makes it a non-viable option. The algorithm presented in this paper performs an a priori search[4] starting with the minimal patterns (Def. 6) of the model. In this search, there is an expansion stage done in two ways: *i)* adding M-patterns not contained in the current pattern, and *ii)* adding arcs of the $A^{<}$ set (Def. 7). This expansion is followed by a pruning strategy that verifies the upward-closure property of support —also known as monotonicity [17]. This property ensures that if a pattern is infrequent, all patterns containing it will be infrequent and, thus, it is no necessary to continue expanding it —the minimum pattern itself expresses all the infrequent behavior containing it.

This pruning presents an exception in order to simplify the results: if a pattern is infrequent and maintains the value of its frequency with the expansion, it is not removed from the expansion stage —it means the pattern is being expanded with a selection branch with less frequency (cf. Def. 9). In this way, WoMine-i returns the largest patterns expressing the minimum infrequent behavior.

Fig. 5 shows an example of a part of the expansion process, assuming a threshold under 40%. The example starts with the M-pattern of C and shows three expansions of the first iteration: the M-pattern of A, one of the M-patterns of I and one of the M-patterns of J. Each of the patterns obtained in the first iteration is again expanded in the second iteration with the M-patterns of J, an M-pattern of I, and the arc $\langle I \rightarrow C \rangle$.

The pseudocode in Alg. 1 shows the main structure of the search made by the algorithm. First, the candidate arcs and the minimal patterns are initialized (Alg. 1:2). These M-patterns will be the used to start the iterative process. Then,

[4] An a priori search uses the previous —a priori— knowledge. It reduces the search space by pruning the exploration of the paths that will not finish in a valuable result.

Algorithm 1. Main structure of WoMine-i.

```
Input: A process model $W$, a set $T = \{T_1, T_2, \dots, T_n\}$ of traces of $W$ and a threshold $tr$.
Output: A set of maximum infrequent patterns of $W$ w.r.t. $T$.
Algorithm infrequentSearch($W$, $T$, $tr$)
    $M \leftarrow \{m \mid m \in W, m$ is an $M$-pattern $\}$ // Def. 6
    $A^{<} \leftarrow \{a \mid a \in W, a$ is a Candidate Arc $\}$ // Def. 7
    $currentPatt \leftarrow M$
    $infreqPatt \leftarrow \{m \mid m \in M,\ m$ is infrequent w.r.t. $T\}$ // using Alg. 2
    while $currentPatt \neq \emptyset$ do
        $candPatt \leftarrow \emptyset$
        forall $p \in currentPatt$ do
            $candPatt \leftarrow candPatt \cup$ addArcs($p$)
            $complementaryM \leftarrow \{m \mid m \in M,\ m \notin p\}$
            forall $m \in complementaryM$ do
                $candPatt \leftarrow candPatt \cup$ addMPattern($p$, $m$)
            end
        end
        $currentPatt \leftarrow$ filterCandidatePatterns($candPatt$, $infreqPatt$)
    end
    Delete the redundant patterns of $infreqPatt$
    return $infreqPatt$
Function filterCandidatePatterns($candPatt$, $infreqPatt$)
    $currentPatt \leftarrow \emptyset$
    forall $p \in candPatt$ do
        measure current frequency of $p$ // using Alg. 2
        if $p$ has no previous frequency || $p$'s frequency has not increased then
            if $p$ is frequent then
                $currentPatt \leftarrow currentPatt \cup p$
            else if $p$ is infrequent then
                if $p$ has no previous frequency || $p$ was frequent || $p$'s frequency has
                  maintained then
                    $currentPatt \leftarrow currentPatt \cup p$
                    $infreqPatt \leftarrow infreqPatt \cup p$
                end
            end
        end
    end
    return $currentPatt$
```

using the algorithm described in Section 5, the infrequent patterns are included in the final set (Alg. 1:5).

Afterwards, the iterative part starts (Alg. 1:6). In this stage, an expansion of each of the current patterns is done, followed by a filtering of the patterns. The expansion by adding arcs from the $A^{<}$ set (Alg. 1:9) is done with the function `addArcs`. The other expansion, the addition of M-patterns that are not in the current pattern (Alg. 1:10-13), is done with the function `addMPattern`.

Once the expansion is completed, the obtained patterns are filtered (Alg. 1:15) to distinguish the promising from the unpromising ones. Firstly, the frequency of the new pattern is measured, comparing it with the frequency of the pattern before the expansion (Alg. 1:22). If this expansion has caused its frequency to grow, the pattern is discarded, otherwise the analysis continues (Alg. 1:23). Then, if the pattern is frequent, it is saved for the next iteration (Alg. 1:25) —because any frequent pattern can become infrequent by expanding it. And otherwise, if the pattern is infrequent, it is saved in the results as infrequent one (Alg. 1:28). But, this is only done if *i)* it is the first iteration and the pattern has no previous frequency, *ii)* the pattern was frequent before the expansion, i.e., it has become infrequent or *iii)* the frequency has maintained, i.e., the pattern was already infrequent and the expansion has not changed its frequency (Alg. 1:27).

Finally, once the iterative process finishes, a simplification is made to delete the patterns which provide redundant information (Alg. 1:17). This redundancy is because there are patterns in the k-th iteration which are expanded and thus are subpatterns of those in the $k+1$-th iteration. A naive approach to reduce the redundancy generated by the expansion might be to remove the patterns from iteration k-th that are expanded in iteration $k+1$-th but, with the existence of loops, there is no assurance that the behavior of a pattern is contained in all its superpatterns.

The simplification process consists in the deletion of the patterns that are contained into others, but whose difference is not a loop. For this, each pattern is compared with its previous patterns in the expansion. If the arcs and activities of a pattern are contained into the other, and the difference between them does not contain a complete closed loop, one of the two patterns must be deleted. The subpattern is deleted if its frequency is higher or equal to the frequency of the pattern under analysis. Otherwise the pattern under analysis is deleted.

5 Measuring the Frequency of a Pattern

In each step of the iterative process, WoMine-i reduces the search space by pruning the infrequent patterns (Alg. 1:15). For this, an algorithm to check the frequency of a pattern is needed (Alg. 2). Following Defs. 9 and 10, the algorithm generates the simple patterns of a pattern and checks the frequency of each one (Alg. 2:2-6). After calculating the frequency of the simple patterns, the function checks if this is considered infrequent w.r.t. the threshold (Alg. 2:12). The frequency of a simple pattern is measured in function `getPatternFrequency` by parsing all the traces and checking how many of them are compliant with it (Alg. 2:15-19). Finally, to check if a trace is compliant with a simple pattern, function `isTraceCompliant` is executed: it goes over the activities in the trace (Alg. 2:22), replaying its execution in the model, and retrieving the activities that have fired the current one (Alg. 2:23-24). The simulation (`simulateExecutionInPattern`) consists in a replay of the trace, checking if the pattern is executed correctly (Alg. 2:25).

With the current activity —the fired one— and the activities that have fired it —the firing activities, retrieved by the simulation—, the executed activities and arcs are saved, in order to analyze and to detect if the execution of the pattern is being disrupted before it is completed. Fig. 6 shows an example of this process. The algorithm starts (#0) with the empty sets of *executed arcs* and *last executed activities.* The first step (#1) executes `A`. There are no firing activities because `A` is the initial activity of the process model. As `A` is also one of the initial activities of the pattern, it is saved correctly in the *last executed activities* set.

Algorithm 2. Check if a pattern is infrequent.

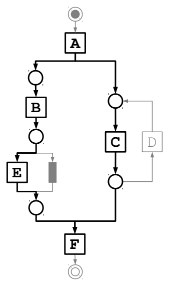
```
   Input: A set $T = \{T_1, T_2, \ldots, T_n\}$ of traces, a pattern pattern to measure its frequency
          w.r.t. $T$ and a threshold to establish the bound of frequency.
   Output: A Boolean value indicating if the pattern is infrequent or not.
1  Algorithm isInfrequentPattern(pattern, T, threshold)
2      simplePatterns ← generate the simple patterns of pattern
3      frequencies ← ∅
4      forall simplePattern ∈ simplePatterns do
5          frequencies ← frequencies ∪ getPatternFrequency(simplePattern, T)
6      end
7      maxFreq ← 0
8      if frequencies.length > 0 then
9          maxFreq ← maximum of frequencies
10     end
11     realFreq ← maxFreq/T.length
12     return realFreq < threshold
13 Function getPatternFrequency(pattern, T)
14     executed ← 0
15     forall trace ∈ T do
16         if isTraceCompliant(pattern, trace) then
17             executed ← executed + 1
18         end
19     end
20     return executed
21 Function isTraceCompliant(pattern, trace)
22     forall activity ∈ trace do
23         Replay activity in the process model
24         sources ← get the activities that fired the execution of activity
25         simulateExecutionInPattern(sources, activity, pattern)
26         if pattern has been successfully executed then
27             return true
28         end
29     end
30     return false
```

The following activity (#2) in the trace is B. As there is only one firing activity (A), a single arc is executed ($\langle A \rightarrow B \rangle$). The arc is added to the *executed arcs* set, and the activity B to the *last executed activities* set. The A activity is not deleted because the set of outputs is formed by {B, C}, and C is still pending.

The next step, activity E (#3), has the same behavior. There is only one firing activity, i.e., one executed arc. The arc is in the pattern and its source activity is in the *last executed activities* set. Hence, the *executed arcs* set is updated and B replaced by E in the *last executed activities* set. After this process, the following activity is C (#4). Its execution has the same behavior as the execution of B, but with the deletion of A from the *last executed activities*, because the set of outputs {B, C} has been fired.

Finally (#5), F has two firing activities and, thus, two arcs are executed. In both cases, the source activity of the arcs —C and E— is in the *last executed activities* set, and the arc is in the pattern. Thus, a simple addition of F to the *last executed activities* set is done when the last of its branches is executed.

At the end of each step, the algorithm checks if the pattern has been correctly executed (Alg. 2:26), i.e., all its arcs have been correctly executed and the *last executed activities* set corresponds with the end activities of the pattern (A_o).



(a) Petri net of a process model with a pattern highlighted in black (the unnamed activity is an invisible activity).

Trace: `A B E C F`			
Initial activities: {`A`}			
End activities: {`F`}			
#	**executed activities**	**executed arcs**	**last executed activities**
0	-	$\emptyset$	$\emptyset$
1	A	$\emptyset$	A
2	B	$\langle A \rightarrow B \rangle$	A, B
3	E	$\langle A \rightarrow B \rangle, \langle B \rightarrow E \rangle$	A, E
4	C	$\langle A \rightarrow B \rangle, \langle B \rightarrow E \rangle, \langle A \rightarrow C \rangle$	E, C
5	F	$\langle A \rightarrow B \rangle, \langle B \rightarrow E \rangle, \langle A \rightarrow C \rangle, \langle C \rightarrow F \rangle, \langle E \rightarrow F \rangle$	F

(b) Check of the execution of a trace for the pattern highlighted in Fig. 6a: '#' is the step of the algorithm; *'executed activity'* is the activity currently executed; *'executed arcs'* is the set with the arcs belonging to the pattern which execution was correctly saved; *'Last executed activities'* is the set of activities which have not fired an entire set of their outputs.

Fig. 6: An example that shows how the algorithm checks if a trace is compliant with a pattern of the process model.

Unlike the other steps, this testing has a positive result when `F` is executed. Thus, the trace is compliant with the pattern.

The process of saving the executed arcs and activities has to be restarted when the executed arc is disrupting the execution of the pattern. For instance, in step #5, if the arc $\langle C \rightarrow D \rangle$ was executed, this would cause this saving process to go back by removing the arcs and activities of the failed path and to continue with the trace to check if the execution of the pattern is resumed later. This analysis is able to detect the correct execution of a pattern in 1-safe Petri nets[5].

6 Experimentation

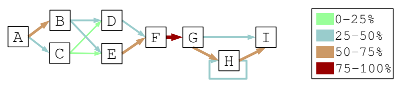
In this section we evaluate the performance of WoMine-i. First (Sec. 6.1), we qualitatively compare WoMine-i with the related techniques and, then (Sec. 6.2), we test WoMine-i on four logs from two Business Process Intelligence Challenges. These experiments have been executed in a laptop with an Intel i7-3612QM (2.1 GHz) processor and 8GB of RAM (1600 MHz)[6].

6.1 Qualitative Comparison Between WoMine-i and the State of the Art Approaches

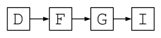
We present a qualitative comparison between WoMine-i and related techniques through a set of illustrative synthetic models. We have classified the related tech-

[5] A Petri net is 1-safe when there can be only one mark in a place at the same time.

[6] The algorithm and datasets can be downloaded from `http://tec.citius.usc.es/processmining/womine/`



(a) Heat Maps technique (all XOR).



(b) Infrequent pattern (frequency of 6%).

Id	Trace	Id	Trace
0	ABDFGI (×4)	6	ACDFGHI (×12)
1	ABEFGI (×17)	7	ACEFGHI (×5)
2	ACDFGI (×2)	8	ABDFGHHI (×11)
3	ACEFGI (×15)	9	ABEFGHHHI (×7)
4	ABDFGHI (×10)	10	ACDFGHHHI (×9)
5	ABEFGHI (×4)	11	ACEFGHHI (×4)

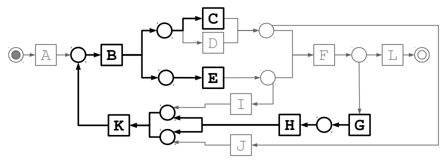
(c) 100-trace log.

Fig. 7: Process model, infrequent pattern and event log of a process.

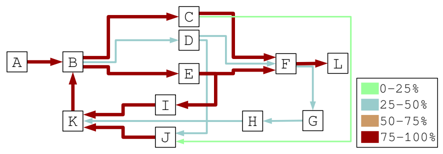
niques into three groups: *i)* individual frequency-based, *ii)* pattern extraction-based and *iii)* trace-based.

The first process model (Fig. 7) presents several selections, an optional task and a loop. WoMine-i finds the pattern in Fig. 7b appearing in the 6% of the traces. On the contrary, individual frequency-based techniques —e.g. Heat Maps (Fig. 7a)— detect parts of the pattern as frequent. Pattern-based techniques —as Local Process Models— get a frequency of 48% (traces 0, 2, 4, 6, 8 and 10) for the pattern —the correct value is 6%. Finally, trace-based techniques retrieve full traces, being necessary a post analysis to extract infrequent patterns —e.g. traces 0, 2, 5, 7 and 11 have a frequency under 6% but contain both frequent and infrequent behavior.

The second example presents a more complex model with loops, parallels and selections (Fig. 8). The approach presented in this paper discovers, with a 5% of frequency, an infrequent pattern denoting as uncommon the execution of the C–E parallel structure after the loop of G–H. As can be seen, based in the individual frequency of the arcs is impossible to extract this infrequent behavior. Local Process Models can extract this pattern successfully but the obtained frequency is not reliable. Also, the search space is larger because they do not rely on the process model. Trace-based techniques present the same problem as in the previous example but, as traces are longer, the post analysis becomes more difficult.



(a) Infrequent pattern detected by WoMine-i (frequency 5%).



(b) Heat Maps technique applied to the process model.

Fig. 8: Results of WoMine-i and Heat Maps for a process model composed by a sequence with a selection, and two loops.

Table 1: Behavioral structure of the infrequent patterns extracted for a threshold of 5% from the process models of the BPICs. It shows the results for two process models (ProDiGen and Inductive Miner) on each log.

			Threshold : 5%								
			runtime (secs)		#patt	frequency	#activities	#sequences	#choices	#parallels	#loops
			pre	alg							
PDG	2012	a	0.208	16.439	1	$0{\pm}0$	$11.00{\pm}0.00$	$2.00{\pm}0.00$	$4.00{\pm}0.00$	$0{\pm}0$	$0{\pm}0$
		o	0.202	343.106	21	$0.02{\pm}0.01$	$6.67{\pm}0.91$	$0.52{\pm}0.51$	$3.33{\pm}1.62$	$0{\pm}0$	$0.38{\pm}0.50$
	2013	clo	0.056	0.700	3	$0.03{\pm}0.02$	$2.00{\pm}1.73$	$0{\pm}0$	$0{\pm}0$	$0.33{\pm}0.58$	$0.33{\pm}0.58$
		op	0.036	4.806	12	$0.02{\pm}0.02$	$6.00{\pm}0.43$	$0{\pm}0$	$1.58{\pm}1.08$	$1.50{\pm}0.52$	$1.50{\pm}1.09$
IM	2012	a	0.208	16.766	2	$0{\pm}0$	$10{\pm}0$	$1.50{\pm}0.71$	$1.50{\pm}2.12$	$0{\pm}0$	$0{\pm}0$
		o	0.202	79.027	4	$0.02{\pm}0.02$	$5.25{\pm}2.87$	$0{\pm}0$	$2.25{\pm}3.30$	$1.75{\pm}1.26$	$1.25{\pm}1.89$
	2013	clo	0.056	6.630	1	$0.01{\pm}0.00$	$1.00{\pm}0.00$	$0{\pm}0$	$0{\pm}0$	$0{\pm}0$	$0{\pm}0$
		op	0.036	0.474	5	$0.02{\pm}0.01$	$3.00{\pm}2.74$	$0{\pm}0$	$0{\pm}0$	$0.40{\pm}0.55$	$0.60{\pm}0.55$

6.2 Infrequent Patterns for the BPI Challenges

The objective of this section is twofold: on the one hand, to test WoMine-i on complex real logs from the Business Process Intelligence Challenge (BPIC)[7][8][9] demonstrating the ability to retrieve all type of structures and, on the other hand, to analyze the influence of the model in the retrieved patterns. We used 4 BPIC logs, and we mined the process models with two different discovery algorithms, ProDiGen (PDG) [3] and the Inductive Miner (IM) [2].

A series of experiments have been run for these logs and models with different thresholds. Table 1 shows the structural characteristics of the mined infrequent patterns for a threshold of 5%. As explained in Sec. 5, the algorithm needs to replay the trace in the model to retrieve the executed arcs. This process is independent of the threshold —it only depends on the traces (log) and on the model. Thus, the runtime is divided in two parts to distinguish this preprocessing time and the time spent by the algorithm. As can be seen, WoMine-i is able to retrieve infrequent patterns with all type of structures. Regarding the runtime, the preprocessing time is short, being 208 ms the longest time. The time spent by the algorithm is longer, and depends on the model and patterns extracted. Log *2012_o* shows a difference in the runtime due to the number of patterns extracted —a model with more uncommon structures will return as infrequent this behavior, increasing the runtime. Nevertheless, the other runtimes are under 20 seconds (*2012_a*), and 7 seconds (*2013*).

Besides, we have compared the number of patterns discovered for the PDG and the IM models. As can be seen, except for one log (*2012_a*), the algorithm retrieves more patterns with the PDG model than with the IM one. This is due to the structure of the models: the higher number of relations in the IM model

[7] BPIC 2012 - `10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f`. This dataset has been split into two logs: 2012_a contains the events related with the state of an application process, while 2012_o has the events related with the state of an offer belonging to an application process.

[8] BPIC 2013 clo - `10.4121/uuid:c2c3b154-ab26-4b31-a0e8-8f2350ddac11`

[9] BPIC 2013 op - `10.4121/uuid:3537c19d-6c64-4b1d-815d-915ab0e479da`

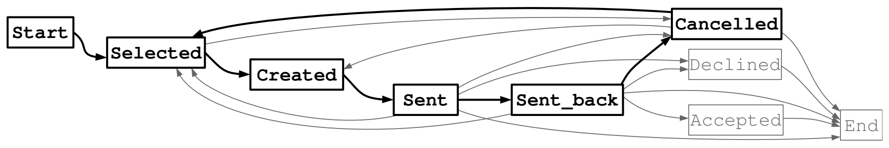


Fig. 9: Infrequent pattern (2%) retrieved from the BPIC 2012_o. All relations are selections (XOR)

allows to embrace more infrequent behavior with few small patterns, while with the PDG model is necessary to build larger patterns —smaller patterns with IM. Results with *2012_a* show a case where the infrequent patterns represent behavior not recorded in the log, but allowed by the models —frequency 0.

Fig. 9 shows an example of a pattern extracted by WoMine-i from the PDG model of the *BPIC 2012_o* log, which corresponds to a Dutch Financial Institute. The extracted pattern appears in the 2% of the traces, and models the selection of a procedure, followed by the creation and shipment of it, and ended by sending it back and canceling the procedure, but with a return to the selection, instead of a finalization of the instance. This behavior might be from a illegal execution where the procedure is restarted after a cancellation, while the normal execution should be the ending of it. Trace-based approaches extract complete traces —the traces of the log have up to 35 activities—, hindering the identification of the pattern. On the other hand, pattern-based approaches might consider the infrequent pattern as executed although other activities, that are not part of the pattern, are executed before the end of the it.

7 Conclusion and Future Work

We have presented WoMine-i, an algorithm designed to search infrequent behavioral patterns in an already discovered process model, being able to discover patterns with the most common control structures, including loops. This structures allow to discover, for instance, subprocesses executed less than the expected, or uncommon wrong behavior. We have compared WoMine-i with other proposals, showing that our approach discovers uncommon behavior that other techniques are not able to detect. Moreover, we have also tested our algorithm with complex real logs from the BPICs. Results show the importance of the infrequent patterns to analyze and optimize the process model.

Acknowledgments.

This research was supported by the Spanish Ministry of Economy and Competitiveness (grant TIN2014-56633-C3-1-R) and the Galician Ministry of Education, Culture and Universities (grants GRC2014/030 and accreditation 2016-2019, ED431G/08). These grants are co-funded by the European Regional Development Fund (ERDF/FEDER program).

References

1. Weijters, A., van Der Aalst, W.M., De Medeiros, A.A.: Process mining with the heuristics miner-algorithm. Technische Universiteit Eindhoven, Tech. Rep. WP **166** (2006) 1–34
2. Leemans, S.J., Fahland, D., van der Aalst, W.M.: Discovering block-structured process models from event logs-a constructive approach. In: International Conference on Applications and Theory of Petri Nets and Concurrency, Springer (2013) 311–329
3. Vázquez-Barreiros, B., Mucientes, M., Lama, M.: Prodigen: Mining complete, precise and minimal structure process models with a genetic algorithm. Information Sciences **294** (2015) 315–333
4. Han, J., Pei, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In: proceedings of the 17th international conference on data engineering. (2001) 215–224
5. Tax, N., Sidorova, N., Haakma, R., van der Aalst, W.M.: Mining local process models. Journal of Innovation in Digital Ecosystems (2016)
6. Greco, G., Guzzo, A., Manco, G., Pontieri, L., Saccà, D.: Mining constrained graphs: The case of workflow systems. In: Constraint-Based Mining and Inductive Databases. Springer (2006) 155–171
7. Conforti, R., La Rosa, M., ter Hofstede, A.H.: Filtering out infrequent behavior from business process event logs. IEEE Transactions on Knowledge and Data Engineering (2016)
8. Ghionna, L., Greco, G., Guzzo, A., Pontieri, L.: Outlier detection techniques for process mining applications. In: International Symposium on Methodologies for Intelligent Systems, Springer (2008) 150–159
9. Yang, W.S., Hwang, S.Y.: A process-mining framework for the detection of healthcare fraud and abuse. Expert Systems with Applications **31**(1) (2006) 56–68
10. Münz, G., Li, S., Carle, G.: Traffic anomaly detection using k-means clustering. In: GI/ITG Workshop MMBnet. (2007)
11. Lo, D., Cheng, H., Han, J., Khoo, S.C., Sun, C.: Classification of software behaviors for failure detection: a discriminative pattern mining approach. In: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM (2009) 557–566
12. Bezerra, F., Wainer, J.: Algorithms for anomaly detection of traces in logs of process aware information systems. Information Systems **38**(1) (2013) 33–44
13. Lu, X., Fahland, D., van den Biggelaar, F.J., van der Aalst, W.M.: Detecting deviating behaviors without models. In: International Conference on Business Process Management, Springer (2015) 126–139
14. De Weerdt, J., vanden Broucke, S., Vanthienen, J., Baesens, B.: Active trace clustering for improved process discovery. IEEE Transactions on Knowledge and Data Engineering **25**(12) (2013) 2708–2720
15. Desel, J., Reisig, W.: Place/transition petri nets. In: Lectures on Petri Nets I: Basic Models. Springer (1998) 122–173
16. Van Der Aalst, W., Adriansyah, A., Van Dongen, B.: Causal nets: a modeling language tailored towards process discovery. In: International Conference on Concurrency Theory, Springer (2011) 28–42
17. Leung, C.K.S. In: Monotone Constraints. Springer US, Boston, MA (2009) 1769–1769