Efficient Discovery of Compact Maximal Behavioral Patterns from Event Logs

Mehdi Acheli[1][0000−0001−9649−7127](✉), Daniela Grigori[1], and Matthias Weidlich[2]

[1] Univ. Paris-Dauphine, CNRS UMR[7243], LAMSADE, 75016 Paris, France
mehdi.acheli@dauphine.fr
daniela.grigori@dauphine.fr
[2] Humboldt-Universität zu Berlin, Germany
matthias.weidlich@hu-berlin.de

Abstract. Techniques for process discovery support the analysis of information systems by constructing process models from event logs that are recorded during system execution. In recent years, various algorithms to discover end-to-end process models have been proposed. Yet, they do not cater for domains in which process execution is highly flexible, as the unstructuredness of the resulting models renders them meaningless. It has therefore been suggested to derive insights about flexible processes by mining behavioral patterns, i.e., models of frequently recurring episodes of a process' behavior. However, existing algorithms to mine such patterns suffer from imprecision and redundancy of the mined patterns and a comparatively high computational effort. In this work, we overcome these limitations with a novel algorithm, coined COBPAM (COmbination based Behavioral PAttern Mining). It exploits a partial order on potential patterns to discover only those that are compact and maximal, i.e. least redundant. Moreover, COBPAM exploits that complex patterns can be characterized as combinations of simpler patterns, which enables pruning of the pattern search space. Efficiency is improved further by evaluating potential patterns solely on parts of an event log. Experiments with real-world data demonstrates how COBPAM improves over the state-of-the-art in behavioral pattern mining.

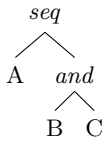
Keywords: Behavioral Patterns · Process Discovery · Pattern Mining.

1 Introduction

Process mining connects the research areas of data mining with process modeling and analysis [1]. Specifically, the analysis of information systems may be supported by exploiting the event logs recorded during their execution. Techniques for process discovery use such event logs and construct a model of the underlying end-to-end process. Recently, a plethora of process discovery algorithms has been proposed [3]. These algorithms impose varying assumptions on the event log used as input, e.g., in terms of the event model [17]; adopt different target languages, e.g., Petri-nets [18], process trees [14], or BPMN [8]; and differ in how they cope with noise and incompleteness, e.g., avoiding over-fitting [24] or filtering noise [6].

Trace ID	Event Sequence
1	U R Y G A K F B C
2	R U T H A C E D B
3	X Y T J A H C I B J
4	X T Y H A D C E K B
5	X Y U G J H K F T

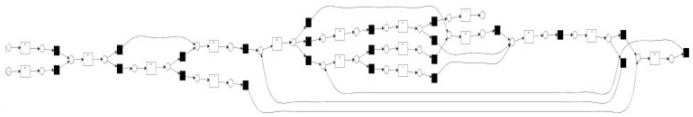
(a)



(b)

Sequential Patterns
AB
AC

(c)



(d)

Fig. 1: (a) Event log; (b) behavioral pattern; (c) 2-length sequential patterns extracted with PrefixSPAN [16]; (d) end-to-end model mined by FHM [23].

Most existing discovery algorithms, however, aim to unify *all* the behavior observed in the log into an end-to-end model. As such, they are not suited for domains in which process execution is highly flexible, as the resulting models are unstructured and are subject to over-generalization [1]. The reason being that there is a large variability of the behavior of different process instances and a model capturing all variations tends to be complex. It was therefore suggested to derive insights about flexible processes by mining behavioral patterns [21, 22]. These patterns are formalized as process models, yet they capture only comparatively small episodes of a process' behavior that occur frequently. The basic idea is illustrated in Fig. 1. For the example log, given as a set of traces, i.e., sequences of events that denote the executions of different activities, a traditional discovery algorithm such as the Flexible Heuristics Miner (FHM) [23] would yield a complex model. However, one may observe that the traces show a specific behavioral pattern: An execution of activity A is followed by B and C in parallel. Detecting such a pattern provides a general understanding of the regularities in process execution. Note though, that such a pattern cannot be detected using standard techniques for sequential pattern mining, such as PrefixSPAN [16], as those would miss complex behavioral dependencies such as concurrency and exclusive choices.

Existing algorithms [21,22] to mine behavioral patterns suffer from imprecision and redundancy of the mined patterns, and a comparatively high computational effort. That is, even though certain behavior is frequent, patterns may capture (i) only a part of the frequent behavior (i.e., they are not maximal), or (ii) a combination of frequent behavior with infrequent behavior (i.e., patterns are not compact). For instance, in Fig. 1, the pattern *seq(a, and(b,c))* is frequent. Arguably, discovery of further patterns *seq(a,b))* and *seq(a, xor(b,k))* would not lead to any new insights on the process, so that it is sufficient to discover the former one. At the same time, existing algorithms suffer from high run-times since pattern candidates are evaluated based on the complete event log.

In this paper, we overcome the above limitations with COBPAM, a novel combination-based algorithm to mine behavioral patterns that are formalized as process trees. It identifies all trees of which the behavior can be found in a certain number of traces of the event log, which takes up the well-established notion of support for patterns in sequence databases [10]. Moreover, we consider a notion of language fitness to assess how strongly a tree materializes. Based thereon, the contributions of COBPAM are threefold:

(1) It defines a partial order on pattern candidates to discover only those that are maximal and compact, thereby improving effectiveness of pattern mining.
(2) It efficiently explores the pattern search space by pruning strategies, exploiting that complex patterns are combinations of simpler patterns.
(3) It further improves efficiency by considering only a subset of traces, when evaluating the support and language fitness of a pattern candidate.

The paper is structured as follows. Section 2 reviews related work on process discovery and pattern mining. Preliminaries are given in Section 3. We then define algebraic operations and structures on potential behavioral patterns in Section 4, while quality metrics for them are presented in Section 5. Our novel mining algorithm for behavioral patterns, COBPAM, is introduced in Section 6. We evaluate the algorithm in experiments with real-world event logs in Section 7, before we conclude and discuss future research directions in Section 8.

2 Related Work

The discovery of behavioral patterns defined with respect to their frequency in a log connects two research areas: sequential pattern mining and process mining. In this section, we will mention the algorithms in the former area that inspired our work and then proceed with an overview of process mining algorithms that aim to derive insights for event logs that have been recorded for flexible processes.

GSP [20] is a sequential pattern mining algorithm that combines pairs of sequential patterns of length k to obtain patterns of length $k+1$. As it will be discussed later, we adopt this principle for COBPAM when generating behavioral patterns. We also borrow the concept of a projected database in the form of log projections from the PrefixSPAN algorithm [16] to evaluate the pattern candidates on the minimal number of traces possible. Moreover, we adopt the maximality principle as discussed for sequential patterns in [10].

Trace clustering is an active research area concerned with inferring insights from logs of flexible processes [4, 5, 11, 19]. These techniques group traces into homogeneous clusters, so that process discovery is applied to each cluster to obtain comparatively structured models. Such techniques are well-suited if a log contains few groups of similar traces. Yet, they do not cater for scenarios, where a partitioning of a log into groups of similar traces is not possible.

Targeting flexible processes, the Fuzzy miner [12] discovers abstract models that describe only the most significant behavior of a log. It enables control of the level of aggregation and abstraction of events and relations between them. Yet, it fails to mine certain behavioral structures, such as concurrency.

The Declare Miner [15] discovers a set of rules that are satisfied by a certain share of traces. These rules come in the form of relations between activities, e.g., two activities being always executed together in a trace, potentially in a fixed order. These rules relate to the presence or the absence of behavior. Each rule, however, is limited to a relation between at most two activities, while our approach considers patterns with an arbitrary number of activities.

The Episode Miner [13] is another algorithm that discovers frequent patterns. The results are partial orders over activities. The method, however, does not support loops and choice constructs.

Our work is inspired by the discovery of Local Process Models (LPMs) [22]. Specifically, we also adopt the notion of process trees to represent behavioral patterns that are observed in event logs in unstructured domains. However, existing algorithms for LPM discovery limit the size of patterns and are not grounded in the traditional definition of support, as known from sequence databases. Rather, when mining LPMs, a trace may account for multiple occurrences of a pattern. Moreover, the algorithm of [22] follows a generate-and-test approach, where only frequent trees are expanded by replacing an activity with some structured behavior. This way, a single tree may be evaluated multiple times in the discovery process. Also, the discovery neglects certain types of patterns, e.g., two infrequent trees that become frequent when joined by a choice operator. Moreover, the existing discovery algorithm relies on the computation of alignments on the entire log, which turns out to be computationally heavy for large-scale event logs.

Compared to the mining of LPMs, our COBPAM algorithm adopts a well-established definition of support for behavioral patterns. COBPAM further provides several innovations. Mined patterns are guaranteed to show desirable properties (maximality and compactness), while the discovery algorithm also leverages pruning strategies and explores pattern candidates solely on a subset of the traces of a log. Note that the initial approach to discover LPMs [22] has been extended with goal-driven strategies to mine patterns based on notions of utility and constraint satisfaction [21]. Yet, these extensions are orthogonal to our work.

3 Preliminaries

This section presents basic definitions. We begin with the notion of an event log.

Definition 1. *Let A be a set of* activity identifiers *(*activities*), and A^* the set of all sequences over A. A* trace $\sigma \in A^*$ *is a finite sequence of activities. An* event log L *is a multiset of traces. $|L|$ denotes its size, i.e., the number of traces it contains.*

A process discovery technique takes as input an event log and constructs process models. Process trees [7] are a language to capture such process models.

Definition 2. *A* process tree *is an ordered tree structure, such that leaf nodes represent activities and non-leaf nodes represent operators. Considering a set of activities A, a set of binary operators $\Omega = \{seq, and, loop, xor\}$, a process tree is recursively defined as follows:*

- $a \in A$ *is a process tree.*
- *considering an operator* $x \in \Omega$ *and two process trees* P_1, P_2, $x(P_1, P_2)$ *is a process tree having* x *as root,* P_1 *as left child, and* P_2 *as right child.*

The language of a process tree $\Sigma(P)$ *is the set of traces it generates. The language is also defined recursively. We exemplify the language of each operator for two activities* $a, b \in A$*:* $\Sigma(seq(a,b)) = \{\langle a,b\rangle\}$*;* $\Sigma(and(a,b)) = \Sigma(and(b,a)) = \{\langle a,b\rangle, \langle b,a\rangle\}$*;* $\Sigma(xor(b,a)) = \{\langle a\rangle, \langle b\rangle\}$*; and* $\Sigma(loop(a,b)) = \{\langle a\rangle, \langle aba\rangle, \langle ababa\rangle, \ldots\}$*. Since the language associated with a loop operator is infinite, we define the n-language of a tree* $\Sigma_n(P)$ *as the set of traces obtained when traversing each loop n-times, e.g.,* $\Sigma_1(loop(a,b)) = \{\langle a\rangle, \langle aba\rangle\}$*.*

4 Algebraic Operations and Structures on Process Trees

We now devise a method for constructing process trees incrementally. We propose to combine two process trees composed of n activities to derive process trees of $n+1$ activities. The process trees combined must be identical except for a single leaf node. We further impose conditions on these leaves, as follows:

Definition 3. *Given a process tree* P *of depth* i*, a leaf node* a *of depth* d *is called* potential combination leaf*, if* $d \geq i-1$ *and there is no leaf* b *of depth* d' *on the left of* a *such that* $d' > d$*.*

Two process trees that can be combined are called seeds.

Definition 4. *Process trees* P_1 *and* P_2 *are called* seeds*, if they contain two potential combination leaves,* a *in* P_1 *and* a' *in* P_2*, such that by replacing* a *in* P_1 *with* a'*, we obtain* P_2*.*

The above notion requires that both process trees are identical except at the level of the leaves a and a'. For instance, *seq(a, b)* and *seq(a, c)* are seeds. Next, we formally define the algebraic operation of combination.

Definition 5. *A* combination *of two seeds* P_1 *and* P_2 *through an operator* x *is an operation generating two process trees. Starting from* P_1*, the combination leaf* a *is replaced by the operator* x*, whose children are set to* a *and* a'*.* a *becomes the left child in one resulting tree, and the right child in another one.* a *and* a' *are called the combination leaves and* x *is called the combination operator.*

Fig. 2a shows an example of a combination of two process trees *seq(a, b)* and *seq(a, c)* through the concurrency operator, which results in two trees: *seq(a, and(b,c))* and *seq(a, and(c,b))*.

Thanks to the conditions characterizing the potential combination leaves, the following theorem holds true:[3]

Theorem 1. *For a process tree* P *of depth* $i \geq 1$*, there is a unique pair of seeds* P_1 *and* P_2*, whose combination through an operator* x *results in* P*.* P_1 *and* P_2 *are called 'the' seeds of* P *and* x *is called the defining operator of* P*.*

[3] The proof can be found in the accompanying technical report at: http://www.lamsade.dauphine.fr/~macheli/behavioralPatterns/paper.pdf

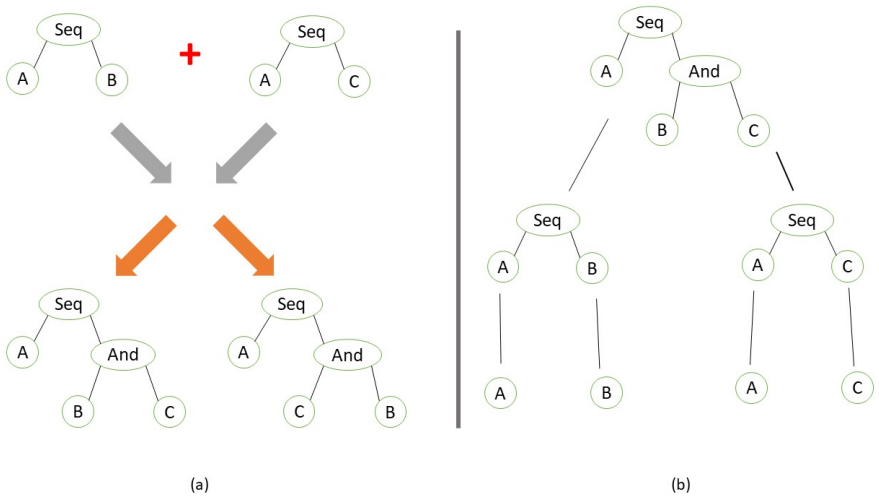


Fig. 2: (a) a combination operation of two process trees (b) the construction tree of the process tree *seq(a, and(b,c))*

Given that every process tree of depth larger than zero results from the combination of two unique seeds, we introduce additional structures.

Definition 6. *Given a process tree P of depth $i \geq 1$, we define its* construction tree*. The nodes of this tree are process trees: The root is P, the leaves are trees with single activity nodes; the children of a non-leaf node are its seeds.*

Fig. 2b exemplifies the construction tree of the process tree *seq(a, and(b,c))*.

Definition 7. *We define the* construction graph *over the set of activities A. It is a directed acyclic graph. Its (infinite) set of nodes is given by all possible process trees. An edge is defined between nodes n_1 and n_2, if n_1 is a seed of n_2. We say that n_2 contains n_1 through the defining operator of n_2.*

To identify a tree, COBPAM uses the concept of representative word.

Definition 8. *Each process tree P is assigned a* representative word *$RW(P)$, a sequence of characters. It is constructed by pre-order traversal of its nodes, outputting activities and operators.*

For example, the representative word of $seq(a, and(b, c))$ is '(a (b c and) seq)'.

5 Quality Metrics

This section defines metrics to evaluate the quality of a behavioral pattern that is formalized as a process tree. These metrics rely on a Boolean function $\epsilon(\sigma, P)$ that returns one, if $\sigma \in \Sigma(P)$, i.e., trace σ fits the process tree P; otherwise, it returns zero. To determine whether this predicate holds true, we compute an

alignment [2] between the trace and the process tree. Such an alignment is a sequence of steps that are defined for the trace σ and one of the traces of the process tree P. Each step in the alignment is either a synchronous move (both traces show the same activity) or an asynchronous move (only one of the traces shows an activity, while a placeholder is introduced for the other one). These steps need to be defined such that the sequence of moves yields the original traces when ignoring placeholders introduced as part of asynchronous moves. Techniques for alignment computation strive for the construction of an optimal alignment, i.e., a sequence of steps of minimal costs, where costs are assigned solely to asynchronous moves.

Following the reasoning given in [22], we consider solely alignments with asynchronous moves that introduce placeholders for the trace of the process tree P. As such, the exact behavior exhibited by the trace σ among all the traces of the process tree P is identified. We capture this by a function $\upsilon(\sigma, P)$. For example, in Fig. 1, the behavior exhibited by the trace 1 is $\langle ABC \rangle$, while trace 2 exhibits $\langle ACB \rangle$. Both traces are part of the language of the discovered pattern.

We employ these functions to define the concept of projection and several quality metrics that provide the foundation for the COBPAM algorithm.

Definition 9. *A* projection *is a subset of an event log associated with a process tree P that contains the traces that can be aligned with P:*

$$proj(P) = \{\sigma \in L \mid \epsilon(\sigma, P) = 1\}.$$

Definition 10. *Given an event log L, the* frequency *of a process tree P is the number of traces that exhibit its behavior:*

$$frequency(P) = \sum_{\sigma \in L} \epsilon(\sigma, P) = |proj(P)|.$$

Its support *is the frequency over the size of the log:*

$$support(P) = \frac{frequency(P)}{|L|}.$$

Definition 11. *Given an event log L, the* language fitness *of a process tree P is the ratio of the behavior seen in the log and all the behavior allowed for by the model. If P does not contain loop operators, it is defined as:*

$$language\_fitness(P) = \frac{|\{\upsilon(\sigma, P) | \sigma \in L \wedge \epsilon(\sigma, P) = 1\}|}{|\Sigma(P)|}.$$

If P contains loop operators, its language will be infinite, so that its language fitness will tend to zero. In this case, we use the n-language of P:

$$language\_fitness(P) = \frac{|\{\upsilon(\sigma, P) | \sigma \in L \wedge \epsilon(\sigma, P) = 1\}|}{|\Sigma_n(P)|}$$

6 Behavioral Pattern Discovery with COBPAM

This section presents a new algorithm to discover process trees that represent frequent behavior in a log. Our idea is to explore a construction graph, starting from process trees of single activities. Each process tree is evaluated against a part of the log that may exhibit its behavior to calculate the aforementioned quality metrics. We also introduce a projection based optimization and pruning rules to limit the number of process trees to evaluate and the number of traces used for evaluation.

In Section 6.1, we discuss a monotonicity property that is later exploited in our pattern search. We then define what we consider compact and maximal process trees in Section 6.2, and introduce optimization based on projections in Section 6.3. In Section 6.4, a detailed view of the algorithm is given.

6.1 A Monotonicity Property

The combination operation introduced in Section 4 replaces a potential combination leaf with a sub-tree representing a portion of a behavior that either extends the behavior of the original tree (when using the choice operator) or constrains it (when using a sequence, loop, or concurrency operator). When evaluating a process tree whose defining operator is a constraining operator (sequence, loop, concurrency), we essentially want that the trace exhibits all the behavior of its seeds except at the position of the combination leaf. At this position, additional behavior shall replace the appearance of an activity in the trace. The shared behavior between a process tree and its seeds represents a context to which the additional behavior is joined. Hence, if a trace does not exhibit the context, there is no need to evaluate the added behavior.

From the above, it follows that, if one of the seeds is not frequent, there is no need to evaluate the tree, as it will be infrequent too. This is a monotonicity property of the support metric. Based thereon, we specify a **first pruning rule**: If a seed is infrequent, it should not be combined using a constraining operator.

6.2 Compact and Maximal Process Trees

We further direct our search for behavioral patterns towards process trees that are useful from an analysis point of view. We therefore define compactness of process trees, as follows:

Definition 12. *Given an event log L, a process tree P is* compact*, if it satisfies all of the following conditions:*

(1) P does not exhibit the choice operator as a root node. If this condition is violated, the process tree would be the union of completely separate behaviors. While this may result in a frequent tree, the tree is arguably of little interest.

(2) P does not result from a combination through a choice operator, where, given L, one of the seeds is frequent. This is motivated as follows: If a tree P_1 is frequent, combining it with any other tree P_2 through the choice operator

results in a frequent tree. Yet, P_2 adds complexity by means of behavior that may not even appear in the log.

(3) P does not contain a loop operator $loop(P_1, P_2)$, such that only the behavior of P_1 appears in L. While having only the behavior of P_1 yields a valid trace of the respective process tree, the derivation of an operator $loop(P_1, P_2)$ is not meaningful, if L does not contain the behavior of P_2.

Note that from condition (2), we immediately derive a **second pruning rule** for the exploration of candidate patterns: When performing a combination through the choice operator, both seeds must be infrequent.

In addition to compactness, there is a second property that is desired for behavioral patterns. It is motivated by the monotonicity property. The latter states that a frequent process tree P whose defining operator is a constraining operator must have two frequent seeds. Hence, we shall return solely P, as the seeds can simply be derived from P and are known to be frequent. In other words, we consider P to be the representative of its seeds. Furthermore, by transitivity, P is a representative of the paths in the construction tree composed solely of trees defined by constraining operators. As a consequence, discovery shall be limited to the largest representatives, which we call maximal behavioral patterns.

Definition 13. *Considering all behavioral patterns of at most depth i, a behavioral pattern is* maximal*, if it is frequent and not contained through a constraining operator in another frequent process tree of depth smaller or equal to i.*

In the example of Fig. 1, the trees *seq(a,b)* and *seq(a,c)* are frequent, but not maximal, since they are contained in *seq(a,and(b,c))* as shown through the construction tree in Fig. 2b. When *seq(a,and(b,c))* is discovered, all the frequent trees it represents, such as *seq(a,c)*, can be deduced.

6.3 Optimization Based on Projections

Recall that we aim at the discovery of frequent process trees. The runtime complexity of a method to solve this problem is governed by the size of the construction graph, which increases exponentially when the number of activities increases, and by the size of the log used to evaluate the quality of the trees. To cope with the latter, we present an optimization that complements the two pruning rules introduced in Section 6.1 and Section 6.2. Our optimization uses projections to assess the frequency of a tree based on a small number of traces:

- When performing a combination through a constraining operator, the behavior associated with the resulting trees may only appear in the intersection of the projections of the seeds. As a result, quality metrics are calculated solely based on the said intersection. Moreover, the size of the intersection of the seeds projections represents an upper bound for the frequency of the resulting trees. This yields a **third pruning rule**: If the upper bound is less than the frequency threshold, the combination is not considered further.
- When performing a combination using the choice operator, the projection associated with the resulting trees is the union of the projections of the seeds.

Algorithm 1: COBPAM: Function *addFreq*

input : P, a process tree; Γ, set of frequent process trees;
Θ, a set of frequent compact maximal process trees;
τ_S, a support threshold; τ_L, a language fitness threshold.

```
Γ' ← ∅;
for P' ∈ Γ do
    Γ' ← combine P, P' through operators {and,loop,seq} with pruning rules;
for R ∈ Γ' do
    if τ_S < support(R) (Definition 10) then
        if τ_L < language_fitness(R) (Definition 11) then
            Θ ← Θ ∪ R;
            Θ' ← trees on constraining operat. paths in construction tree of R;
            Θ ← Θ \ Θ' ;
        for each potential combination leaf a in R do
            RW ← create representative word, replace a with '_' in RW(R);
            Γ_RW ← set containing frequent trees identified by RW;
            addFreq(R, Γ_RW, Θ, τ_S, τ_L);
    else
        for each potential combination leaf a in R do
            RW ← create representative word, replace a with '_' in RW(R);
            γ_RW ← set containing infrequent trees identified by RW;
            addInfreq(R, γ_RW, Θ, τ_S, τ_L);
Γ ← Γ ∪ P;
```

Moreover, the frequency of the resulting trees can be precisely derived and corresponds to the size of the union of the seeds projections. On another hand, the language of the new trees is the union of the languages of the seeds.

6.4 The COBPAM Algorithm

Now we are ready to present COBPAM, an algorithm that strives for efficient discovery of behavioral patterns that are frequent, compact and maximal. In order to achieve high efficiency, it largely neglects infrequent activities. More precisely, it discovers process trees that are built from frequent activities as well as frequent combinations of two infrequent activities through the choice operator. Here, a frequent combination of two infrequent activities is considered as a single activity in the remainder of the algorithm.

Note that pruning of infrequent trees, in general, implies a certain loss of patterns. Due to the choice operator, trees that are infrequent at some point can be combined to get frequent ones at a later stage. Hence, pruning infrequent trees potentially leads to missing some frequent patterns that comprise a choice operator. Despite this, COBPAM applies the respective pruning, since without it, a large number of infrequent process trees would need to be evaluated. Moreover, the loss of frequent behavioral patterns in the discovery process is limited to

Algorithm 2: COBPAM: Function *addInfreq*

input : P, a process tree; γ, set of infrequent process trees;
Θ, a set of frequent compact maximal process trees;
τ_S, a support threshold; τ_L, a language fitness threshold.

```
1  γ' ← ∅;
2  for P' ∈ γ do
3      γ' ← combine P and P' through choice operator;
4  for R ∈ γ' do
5      if τ_S < support(R) (Definition 10) then
6          if τ_L < language_fitness(R) (Definition 11) then
7              Θ ← Θ ∪ R;
8          for each potential combination leaf a in R do
9              RW ← create representative word, replace a with '_' in RW(R);
10             Γ_RW ← set containing frequent trees identified by RW;
11             addFreq(R, Γ_RW, Θ, τ_S, τ_L);
12     else
13         for each potential combination leaf a in R do
14             RW ← create representative word, replace a with '_' in RW(R);
15             γ_RW ← set containing infrequent trees identified by RW;
16             addInfreq(R, γ_RW, Θ, τ_S, τ_L);
17 γ ← γ ∪ P;
```

process trees that comprise the choice operator. Completeness of the discovery result for trees built of constraining operators is not affected.

The idea of the COBPAM algorithm is to incrementally build up sets of process trees. In the light of the pruning rules, we maintain two kinds of sets, containing only frequent and infrequent trees, respectively. The former kind serves as the basis for combinations through the constraining operators, whereas the latter serves for choice-based combinations. All trees inside a set are identical except for a single leaf node. In fact, any two trees in a set are seeds and can be combined. Moreover, since the difference between two seeds is a single leaf, we associate each set with an identifier in the form of a representative word that applies to any of the representative words of the contained trees. Take, for example, the tree *seq(a,b)*. Its representative word is *(a b seq)*. The process tree can be added to the set defined by *(a _ seq)*, where the underscore is a placeholder for any activity. So, any other tree, e.g., *seq(a,c)*, can be added to the set by replacing the placeholder with an activity. The placeholder is always at the position of a potential combination leaf.

The algorithm revolves around two functions, *addFreq*, defined in Alg. 1, that adds the process tree P to a set Γ containing only frequent trees; and *addInfreq*, defined in Alg. 2, that adds P to a set γ containing only infrequent trees. By Θ, we further denote the set of frequent compact maximal trees, which represents the actual result of our algorithm. As such, the respective trees must satisfy a given language fitness threshold. Moreover, since the result shall contain only

maximal trees, each time a frequent tree defined by a constraining operator is added to it, parts of its construction tree are deleted.

The COBPAM algorithm starts by creating the set of frequent process trees identified by the word '_', i.e., any frequent tree with a single activity is added to it. We apply function *addFreq* on this set for each frequent activity. The algorithm then proceeds recursively, switching between *addFreq* and *addInfreq*. Note that one may use a maximum recursion depth d, which then also limits the maximum depth for the discovered trees to force termination.

7 Experimental Evaluation

In this section, we evaluate the efficiency and effectiveness of COBPAM by comparing it to LPM discovery on real-life datasets. We first present details on the used datasets and the experimental setup. We then compare the algorithms in terms of their efficiency and effectiveness (quantitative and qualitative).

7.1 Setup and Datasets

COBPAM has been implemented as a plugin in the ProM framework [9] as the package *BehavioralPatternMining*. We ran the experimental evaluation on a PC with an i5-1.8 Ghz processor, 8GB RAM and MacOS High Sierra.

We ran COBPAM and LPM discovery [22] on four real-world event logs. COBPAM was parameterized with a value of 0.7 for the support and language fitness thresholds, and a value of 2 for the maximal depth. The implementation of LPM discovery in ProM has a single parameter, i.e., the bound for the number of LPMs to discover. We set this bound to 500, the maximal possible value.

We used the following event logs, which are publicly available:[4]

- Sepsis: A log of a treatment process for Sepsis cases in a hospital. It contains 1050 traces with 15214 events that have been recorded for 16 activities.
- Traffic Fines: A log of an information system managing road traffic fines, containing 150370 traces, 561470 events and 11 activities.
- Hospital: A log of a dutch hospital containing 1143 traces, 150291 events and 624 activities.
- WABO: A log of a building permit application process in the Netherlands. It contains 1434 traces with 8577 events, recorded for 27 activities.

7.2 Results

Efficiency. Running both algorithms for behavioral pattern discovery, we observed the execution times reported in Table 1. They depend on the size of the log, the number of activities and events, and the complexity of the behavioral patterns in the log. COBPAM generally turns out to be more efficient.

Quantitative effectiveness. Next, we assess the relevance of patterns discovered by our algorithm and LPM discovery. While COBPAM guarantees that

[4] https://data.4tu.nl/repository/collection:event_logs_real

Table 1: Execution times of COBPAM and LPM Discovery

	Sepsis	Traffic Fines	Hospital	WABO
COBPAM	22m	28m	6h	75s
LPM discovery	88m	68m	>48h	26h

Table 2: Pattern statistics of LPM Discovery and COBPAM

	COBPAM	**LPM Discovery**		
		Relevant	Non-Maximal	Non-Compact
Sepsis	386	125	17	194
Traffic Fines	4	1	0	0
WABO	28	21	5	235

discovered patterns are compact and maximal, we check how many patterns derived by LPM discovery also satisfy these properties. Given the above execution times, we focus on three logs: Sepsis, Traffic fines, and WABO.

The results are summarized in Table 2. For instance, for the Sepsis log, among the 500 patterns mined by the LPM discovery, 336 patterns satisfy the support and language fitness thresholds set by COBPAM. Among these, 194 are not compact and 17 are not maximal. So, only 125 of the patterns derived by LPM discovery are maximal and compact, whereas COBPAM discovered 386 such patterns. Similar results are obtained for the other datasets. We conclude that the patterns derived by LPM discovery contain much redundant information, whereas COBPAM yields many more relevant patterns.

Qualitative effectiveness. Lastly, we conducted a qualitative analysis on the patterns derived by COBPAM and LPM discovery for the Sepsis log. This log is well-suited for our analysis as its end-to-end process model obtained with the FHM algorithm shows that the process is highly unstructured with intertwined execution paths, a high number of choice and loop constructs, and many edges.

In Fig. 3, we show some patterns derived by LPM discovery that satisfy the thresholds set to COBPAM. Patterns found by COBPAM are shown in Fig. 4. We notice the difference in the trees derived by the two algorithms. For instance, the tree (12) was not extracted by COBPAM, because it is not maximal. In fact, it is contained in tree (17). Knowing that (17) is frequent, one knows that *"ER Registration"* followed by *"CRP"* followed by *"Leucocytes"* is frequent. Hence, it follows that *"CRP"* followed by *"Leucocytes"*, as in tree (12), is frequent, too.

The trees (14), (15), and (16) are not discovered by COBPAM either, as they are not compact. Trees (14) and (15) are obtained from (12) by replacing activity *"CRP"* by a choice that includes activity *"CRP"* and some other behavior. Then, knowing that tree (12) is frequent, one concludes that the trees (14) and (15) are also frequent. Hence, they do not give new information. Similarly, tree (16) can be constructed from tree (13). The construction and evaluation (support, language fitness) of such trees lead to execution time being wasted without gaining further information about the process.

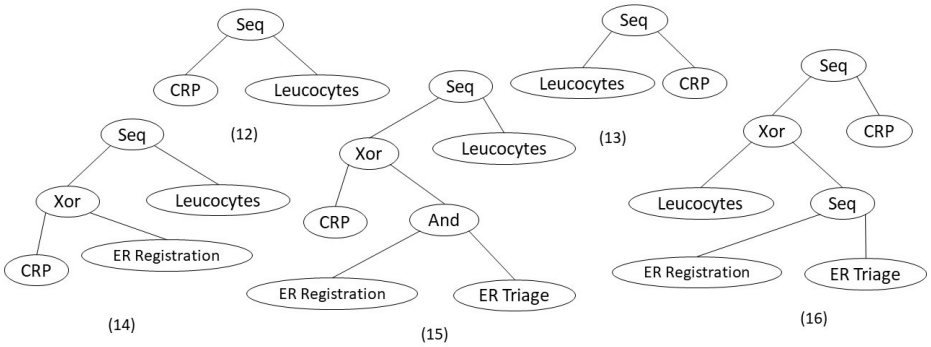


Fig. 3: Behavioral patterns mined by LPM discovery

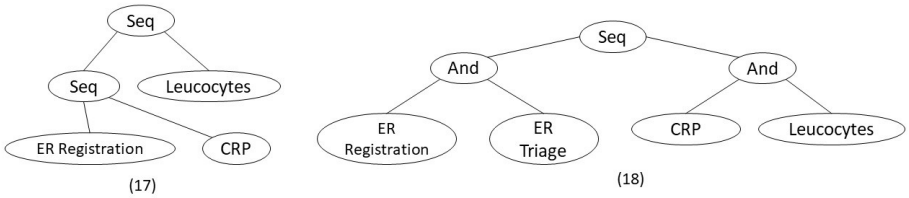


Fig. 4: Behavioral patterns mined with COBPAM

8 Conclusion

In this paper, we proposed COBPAM, an efficient algorithm for behavioral pattern mining. Potential patterns are obtained by combining simpler patterns using algebraic operations on process trees. Compared with an exhaustive search, the efficiency of the algorithm is improved by pruning the search space, evaluating the candidates solely on parts of the event log, and exploiting calculations already done for smaller trees. Moreover, the algorithm exploits a partial order on potential patterns to discover only those that are maximal while respecting a compactness property. An experimental evaluation with real-world event logs demonstrated how COBPAM improves over the state-of-the-art in behavioral pattern mining in terms of execution time and relevance of extracted patterns.

In a future work, we plan to investigate the relation between operators in terms of their semantics. For instance, if *and(a,b)* is infrequent then *seq(a,b)* is also infrequent. That is because the behavior of the sequence operator is included in that of the concurrency operator. Moreover, we intend to further improve efficiency by using frameworks for parallel computing.

References

1. Van der Aalst, W.: Process mining: Data science in action. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
2. Adriansyah, A.: Aligning Observed and Modeled Behavior. Ph.D. thesis (2014)
3. Augusto, A., Conforti, R., Dumas, M., La Rosa, M., Maggi, F.M., Marrella, A., Mecella, M., Soo, A.: Automated Discovery of Process Models from Event Logs: Review and Benchmark (2018)

4. Bose, R.P.J.C., van der Aalst, W.M.: Context Aware Trace Clustering: Towards Improving Process Mining Results. In: 2009 SIAM International Conference on Data Mining (2009)
5. Bose, R.P.C., Van Der Aalst, W.M.: Trace clustering based on conserved patterns: Towards achieving better process models. In: LNBIP (2010)
6. vanden Broucke, S.K., De Weerdt, J.: Fodina: A robust and flexible heuristic process discovery technique. Decision Support Systems **100**, 109–118 (8 2017)
7. Buijs, J.C., Van Dongen, B.F., Van Der Aalst, W.M.: A genetic algorithm for discovering process trees. In: CEC 2012. pp. 1–8. IEEE (6 2012)
8. Conforti, R., Dumas, M., García-Bañuelos, L., La Rosa, M.: BPMN Miner: Automated discovery of BPMN process models with hierarchical structure. Information Systems **56**, 284–303 (3 2016)
9. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The ProM Framework: A New Era in Process Mining Tool Support. pp. 444–454. Springer, Berlin, Heidelberg (2005)
10. Fournier-Viger, P., Chun, J., Lin, W., Kiran, R.U., Koh, Y.S., Thomas, R.: A Survey of Sequential Pattern Mining. Ubiquitous International **1**(1), 54–77 (2017)
11. Greco, G., Guzzo, A., Pontieri, L., Saccà, D.: Discovering expressive process models by clustering log traces. IEEE TKDE (2006)
12. Günther, C.W., van der Aalst, W.M.P.: Fuzzy Mining – Adaptive Process Simplification Based on Multi-perspective Metrics. pp. 328–343. Springer, Berlin, Heidelberg (2007)
13. Leemans, M., van der Aalst, W.M.: Discovery of frequent episodes in event logs. In: Lecture Notes in Business Information Processing. vol. 237, pp. 1–31. Springer, Cham (11 2015)
14. Leemans, S.J.J., Fahland, D., Van Der Aalst, W.M.P.: LNCS 7927 - Discovering Block-Structured Process Models from Event Logs - A Constructive Approach pp. 311–329 (2013)
15. Maggi, F.M., Mooij, A.J., Van Der Aalst, W.M.: User-guided discovery of declarative process models. In: CIDM 2011. pp. 192–199. IEEE (4 2011)
16. Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.C.: Mining sequential patterns by pattern-growth: The prefixspan approach. IEEE Transactions on Knowledge and Data Engineering (2004)
17. Senderovich, A., Weidlich, M., Gal, A.: Temporal Network Representation of Event Logs for Improved Performance Modelling in Business Processes. BPM **1**, 3–21 (2017)
18. Solé, M., Carmona, J.: Process mining from a basis of state regions. In: Applications and Theory of Petri Nets. vol. 6128 LNCS, pp. 226–245. Springer, Berlin, Heidelberg (2010)
19. Song, M., Günther, C.W., Van Der Aalst, W.M.: Trace clustering in process mining. In: Lecture Notes in Business Information Processing (2009)
20. Srikant, R., Agrawal, R.: Mining sequential patterns: Generalizations and performance improvements. pp. 1–17. Springer, Berlin, Heidelberg (1996)
21. Tax, N., Dalmas, B., Sidorova, N., van der Aalst, W.M., Norre, S.: Interest-driven discovery of local process models. Information Systems **77**, 105–117 (9 2018)
22. Tax, N., Sidorova, N., Haakma, R., van der Aalst, W.M.: Mining local process models. Journal of Innovation in Digital Ecosystems **3**(2), 183–196 (2016)
23. Weijters, A.J.M.M., Ribeiro, J.T.S.: Flexible heuristics miner (FHM). In: CIDM 2011. pp. 310–317 (2011)
24. van Zelst, S.J., van Dongen, B.F., van der Aalst, W.M.P.: Avoiding Over-Fitting in ILP-Based Process Discovery. In: BPM, pp. 163–171. Springer, Cham (2015)