Enhancing Discovered Processes with Duplicate Tasks

Borja Vázquez-Barreiros[a,*], Manuel Mucientes[a], Manuel Lama[a]

[a]*Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS)*
Universidade de Santiago de Compostela, Santiago de Compostela, Spain

Abstract

Including duplicate tasks in the mining process is a challenge that hinders the process discovery, as it is also necessary to find out which events of the log belong to which transitions. To face this problem, we propose SLAD (Splitting Labels After Discovery), an algorithm that uses the local information of the log to enhance an already mined model, by performing a local search over the tasks that have more probability to be duplicated in the log. This proposal has been validated with 54 different mined models from three process discovery algorithms, improving the final solution in 45 of the cases. Furthermore, SLAD has been tested in a real scenario.

Keywords: Duplicate tasks, Process discovery, Petri nets.

1. Introduction

In the recent years, a lot of work has been made for developing technologies to automate the execution of processes in different application domains such as industry, education or medicine [20]. In particular, for business processes, there has been an incredible growth on the amount of process-related data, i.e, execution traces of business activities. Within this context, process mining has emerged as a way to analyze the behavior of an organization based on these data —event logs— offering techniques to discover, monitor and enhance real processes, i.e., to understand *what is really happening in a business process* [25].

Based on this idea, and in order to model what is really happening in an organization, *process discovery* techniques aim to find the process model that *better* portraits the behavior recorded in an event log. There are four quality dimensions to measure *how good* is a model and, hence, identify which model is the *best*: fitness replay, precision, generalization and simplicity. *Fitness replay* measures how much of the behavior recorded in the log can be reproduced in the process model. On the other hand, *precision* and *generalization* measure if the model overfits —it disallows for new behavior not recorded in the log— or underfits —it allows additional behavior not recorded in the log— the data, respectively. Finally, *simplicity*, quantifies the complexity of the model, for instance, the number of arcs and tasks. Hence, the idea behind process discovery is to maximize these metrics in order to obtain an *optimal* solution that better describes the flow of the events that occur within the process.

[*]Corresponding author

Email addresses: borja.vazquez@usc.es (Borja Vázquez-Barreiros), manuel.mucientes@usc.es (Manuel Mucientes), manuel.lama@usc.es (Manuel Lama)

In order to discover the optimal solution, one key question is *how to describe the ordering and flow of events that occur in a process* [28]. From the control-flow perspective, a model can be represented with many different workflow patterns, such as *sequences*, *parallels*, *loops*, *choices*, etc. Furthermore, to improve the quality of the solution, models can be extended with more behavior: *duplicate activities*, *non-free-choice constructs*[1], etc. Within the scope of this paper, the notion of duplicate tasks —or activities— [30] refers to situations in which multiple tasks in the process have the same label, i.e., they can appear more than once in the process. As previously said, the inclusion of duplicate tasks is useful to improve the precision and simplicity of a model, and, hence, enhance its comprehensibility [3, 10]. Figure 1 shows an example on how the addition of duplicate tasks to a model improves its understandability and structural clarity. In this example, considering the sample log of Fig. 1a, the events *Quiz* and *Check Bibliography*, are executed multiple times —twice in each trace. Between the multiple possibilities of modeling the behavior of the log, we can assume i) an injective relation between the events in the log and the activities in the model (Fig. 1b); or ii) that multiple activities can share the same label, i.e. a model with duplicate activities (Fig. 1c). In this example, although both models perfectly reproduce all the behavior recorded in the log —both have a perfect replay fitness—, the model depicted in Fig. 1b allows to execute both *Quiz* and *Check Bibliography* as many times as we want at any time in the process, hence, this model is not a precise picture of the recorded behavior of the log —its precision is lower. On the other hand, if both activities are duplicated, the resulting model (Fig. 1c) is more suitable, i.e., more precise with respect to the recorded behavior in the log, as it does not allow, for example, to check the bibliography —*Check Bibliography*— during the exam —*Quiz*. Hence, the ability to discover these duplicate tasks *may* greatly enhance the comprehensibility of the final solution, and create a more specific process model.

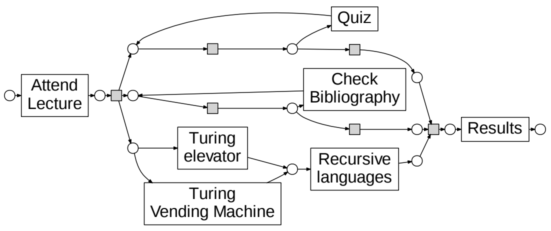
From the perspective of process discovery, including duplicate tasks in the mining process is a well known challenge [29, 30] as, usually, duplicate tasks are recorded with the same label in the log, hindering the discovery of the model that better fits the log —algorithms need to find out which events of the log belong to which tasks. To face this issue, handling duplicate tasks is usually considered as a pre-mining step, i.e., the potential duplicate tasks are identified and accordingly labeled *before* mining the log, or as part of the process discovery algorithm. Within this context, there are several techniques in the state of the art that allow to mine duplicate tasks [3, 6, 8, 9, 13, 14, 15, 19]. However, some approaches can retrieve worse solutions than without duplicate activities [3], others can duplicate any activity of the log without imposing any limit to the number of activities that may be duplicated [6, 8, 13], or they have problems when dealing with duplicate activities involved in certain constructs, such as loops [9, 19].

In this paper we analyze the possibility of tackling duplicate tasks *after* mining a process model, in particular, after mining a causal net [27] or heuristic net [38], without adversely affecting the quality of the initial solution. Hence, we present SLAD (Splitting Labels After Discovery) a novel algorithm that enhances an already discovered process model by splitting the behavior of its activities. Firstly, a process discovery technique mines a log without considering

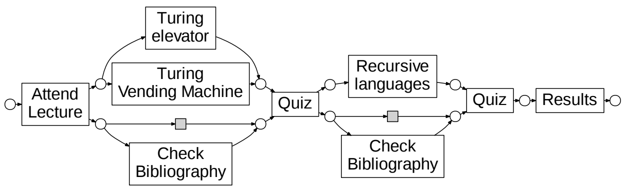
[1]A non-free choice (NFC) construct is a special kind of choice, where the selection of a task depends on what has been executed before in the process model.

	Events
case$_1$	Attend lecture, Turing Elevator, Check Bibliography, Quiz, Recursive Languages, Quiz, Results.
case$_2$	Attend lecture, Turing Vending Machine, Check Bibliography, Quiz, Recursive Languages, Check Bibliography, Quiz, Results.
case$_3$	Attend lecture, Check Bibliography, Turing Vending Machine, Quiz, Check Bibliography, Recursive Languages, Quiz, Results.
case$_4$	Attend lecture, Turing Elevator, Check Bibliography, Quiz, Check Bibliography, Recursive Languages, Quiz, Results.
case$_n$	...

(a) *Log*.



(b) *Model without duplicate tasks.*



(c) *Model with duplicate tasks.*

Figure 1: A log and two process models —Petri nets— exemplifying a lecture of Automata Theory and Formal Languages.

duplicate tasks, generating a causal net or a heuristic net. Then, SLAD, using the local information of the log and the retrieved model, tries to improve the quality of the model by performing a local search over the tasks that have more probability to be duplicated in the log. The contributions of this proposal are: i) the discovering of the duplicate activities is performed *after* the discovery process, in order to *unfold* the overly connected nodes than may introduce extra behavior not recorded in the log; ii) new heuristics to focus the search of the duplicated tasks on those activities that better improve the model and iii) new heuristics to detect potential duplicate activities involved in loops.

The remainder of this paper is structured as follows. Section 2 describes the current state of the art of process discovery algorithms dealing with duplicate tasks. Then, Section 3 describes in detail the SLAD algorithm to tackle duplicate tasks. Section 4 shows the obtained results with 54 different mined models from three process discovery algorithms as well as a comparison with other state of the art approaches. Finally, Section 5 points out the conclusions.

2. State of the art

In the state of the art of process discovery, many techniques [11, 18, 32, 36, 37, 38, 39, 40] assume an injective relation between tasks and events in the log, considering that there cannot be two different activities with the same label. Therefore, these algorithms, when dealing with logs with duplicate tasks, usually give as a result models with overly connected nodes or needless loops, decreasing the precision and simplicity of the model. On the other hand, there are techniques that do not make such a restrictive assumption [3, 6, 7, 8, 9, 13, 14, 15, 19]. Typically, all these techniques identify the potential duplicate activities in a pre-mining step, or during the mining process. One example is the α^*-algorithm [19], an extension of the α-algorithm [31] to mine duplicate tasks. However, the heuristic rules used in this algorithm require a noise-free and complete log [34]. Fodina [3] is an algorithm based on heuristics that infers the duplicate tasks transforming the event log into a *task log* following the heuristics defined in [9]. Other solutions, like DGA [9] and ETM [6] —based on evolutionary algorithms—, or AGNES [13] —an approach based on inductive programming— include the possibility to mine duplicate tasks. However, these techniques do not allow to *unfold* loops [9], or they are very permissive allowing to duplicate any activity in the log [6]. State-based region theory algorithms [7, 8] are also able to mine duplicate tasks, but, when searching for regions, they usually allow to split any label in the log without any bound. Herbst et al. also developed a set of algorithms [14, 15] that infer the duplicate activities in a pre-mining step. However, the algorithms developed by Herbst et al. only focus their search on block-structured representations of a process model, constraining the expressiveness of a process model. For example, he only way to represent a non-free-choice construct, if possible, is through duplicate labels, which can lead to a more complex model.

In summary, although very valuable results have been achieved in this field, the state of the art algorithms have different weaknesses. Some obtain, in specific logs, worse solutions than without duplicated tasks [3]. Others allow to duplicate any activity in the log [6], or generate solutions with a lower simplicity [8] —more complex solutions. Finally, other proposals use heuristics that do not consider duplicate activities in some workflow patterns such as loops [9, 19]. Within this context, we propose SLAD, an algorithm to tackle duplicate tasks *after* mining a model, with the objective of improving its quality, i.e., its replay fitness, precision and simplicity. SLAD combines the actual dependencies of a model mined by a process discovery technique, and a set of heuristics that use the information of the behavior recorded in the log, in order to enhance the precision and simplicity —and hence its comprehensibility— of the already mined model, trying to split its overly connected tasks that are more suitable to be duplicated.

3. Splitting Labels After Discovery

Algorithm 1 describes SLAD[2], an algorithm to tackle duplicate tasks on an already mined causal matrix (Definition 2). Usually, when applying a process mining technique, duplicate events in the event log (Definition 1) are represented as i) overly connected nodes where all the behavior from different contexts of the model piles up, and with ii) needless loops to allow the execution of the same label multiple times. Therefore, with the presented approach, we try to reduce the density of these nodes by delegating some of their inputs and outputs to other activities with the same label. First, using heuristics, the algorithm detects which activities may be split into multiple tasks. Then, based on the local information of the event log and the causal dependencies of the input model, the algorithm splits the behavior of the original tasks among the new tasks with the same label. Finally, the original model is replaced with the new one if its quality —*how good* is the solution— is better than the previous solution. Note that we measure the quality of a process based on three criteria: *fitness replay*, *precision* and *simplicity*. Hence, the presented algorithm tries to improve the quality of an already mined solution by unfolding the overly-connected activities —through duplicate activities— and, therefore, enhancing the comprehensibility of the model.

[2]`http://tec.citius.usc.es/processmining/SLAD`

Definition 1 (Trace, Event log)**.** *Let T be a set of tasks. A trace $\sigma \in T^*$ is a sequence of tasks. Let $\mathbb{B}(A)$ denote the set of all multisets over some set A. An event log $L \in \mathbb{B}(T^*)$ is a multiset of traces.*

For the sake of the argument, we will use the example in Figure 2 to illustrate the behavior of the presented approach. Fig. 2a shows a log with three traces and 6 *different* tasks. On the other hand, Fig. 2b shows the initial solution —a causal matrix and its respective Petri net— mined by the process discovery algorithm ProDiGen [36] without considering duplicate tasks. Fig. 2c shows the model obtained after the execution of SLAD to the model of Fig. 2b. Finally, Fig. 2d shows each of the steps —described in the next sections— involved in the process.

Algorithm 1: Local search Algorithm.

```
input: A log $L$
$ind_0 \leftarrow$ initial_solution($L$)          // Causal matrix retrieved by a process discovery technique.
$T \leftarrow$ finite set of tasks of $L$
$potentialDuplicates \leftarrow \emptyset$
foreach activity $t \in T$ do
    if $\max(\min(|t >_L t'|, |t' >_L t|), 1) > 1$ then
        $potentialDuplicates \leftarrow potentialDuplicates \cup \{t\}$
$ind_0 \leftarrow$ localSearch ($ind_0$, $L$, $potentialDuplicates$, $true$)
Function localSearch($ind_0$, $L$, $potentialDuplicates$, $unfoldL2L$)
    $ind_{best} \leftarrow ind_0$
    $potentialDuplicatesL2L \leftarrow \emptyset$
    foreach activity $t \in potentialDuplicates$ do
        subsequences $\leftarrow$ Retrieve all the subsequences $(t_1 t t_2)$ where $t_1 \in I(t)$ and $t_2 \in O(t)$ from parsing $ind_0$
        combinations $\leftarrow$ calculateCombinations ($subsequences$, $t$)
        foreach combination $c \in combinations$ do
            $t' \leftarrow$ activity $t$ from $ind_0$
            $t.inputs = (t.inputs \setminus c.inputs) \cup c.sharedInputs$
            $t'.inputs = c.inputs$
            $t.outputs = (t.outputs \setminus c.outputs) \cup c.sharedOutputs$
            $t'.outputs = c.outputs$
            if $(I(t') \neq \emptyset$ && $O(t') \neq \emptyset$ && $I(t) \neq \emptyset$ && $O(t) \neq \emptyset)$ then
                Add task $t'$ to $ind_0$ and update $t$ in $ind_0$
                Repair $ind_0$
                Prune unused arcs
                Evaluate $ind_0$
                if $ind_0 < ind_{best}$ then
                    $ind_0 \leftarrow ind_{best}$
                else
                    $ind_{best} \leftarrow ind_0$
                    $potentialDuplicatesL2L = potentialDuplicatesL2L \cup \{\bigcup O(t)\}$
                                        /* $\bigcup O(t)$ is the union of the subsets in $O(t)$ */
            else
                $ind_0 \leftarrow ind_{best}$
    if $potentialDuplicatesL2L \neq \emptyset$ && $unfoldL2L$ then
        $ind_{best} \leftarrow$ localSearch ($ind_{best}$, $L$, $potentialDuplicatesL2L$, $unfoldL2L$)
    return $ind_{best}$
```

Definition 2 (Causal matrix)**.** *A Causal matrix is a tuple (T, I, O) where:*

T is a finite set of tasks,

$I : T \rightarrow \mathbb{P}(\mathbb{P}(T))$ is the input condition function, where $\mathbb{P}(X)$ denotes the powerset of some set X. Hence, I represents a set of sets of the tasks T.

$O : T \rightarrow \mathbb{P}(\mathbb{P}(T))$ is the output condition function.

If $e \in T$ then $I(e)$ denotes the input tasks of e, i.e., a set o sets of tasks, and $O(e)$ denotes the output tasks of e.

3.1. Discovering duplicate tasks

The first step of the algorithm is the discovery of the potential duplicate tasks. One naive solution to detect if a task is a potential duplicate is to set the upper bound for that task to the number of times it appears in the log. This makes the search space finite, covering all the possible solutions with duplicate tasks, i.e., *all the tasks* are identified as potential duplicates. The problem with this solution is that within this search space is also included the overly-specific *trace-model*[3]. A variant to this approach is to set the upper bound to the maximum number of times a task is repeated in a trace, instead of considering the complete log. This will reduce the search space, but at the expense of dismissing the possible duplicity of a task between traces.

In SLAD, instead of going through a blind search over all the tasks of the input model —ind_0—, we decided to apply heuristics to identify and retrieve more information about the duplicate tasks of the event log. This strategy follows the heuristics defined in [9], where the duplicate activities can be distinguished based on their local context, reducing the search space by stating that two tasks with the same label cannot share the same input and output dependencies. Within this context, the duplicate tasks are locally identified based on the *follows relation* ($>_L$) — Definition 3 [31]. Thus the heuristics to detect potential duplicates can be formalized as described in Definition 4 [9]. In summary, this definition states that if for a task t the upper bound is greater than 1, then t is considered as a potential task for being duplicated and, hence, it is added to *potentialDuplicates* (Alg.1:4-6).

Definition 3 (Follows relation)**.** *Let T be a set of tasks. Let L be an event log over T, i.e., $L \in \mathbb{B}(T^*)$. Let $t, t' \in T$:*

$t >_L t'$ iff: there is a trace $\sigma = t_1 t_2 \dots t_n$ and $i \in \{1, \dots, n-1\}$ such that $\sigma \in L$, $t_i = t$ and $t_{i+1} = t'$.

Definition 4 (Duplicate task)**.** *Let L be an event log over T. Let $t, t' \in T$, $|t >_L t'|$ the total number of times that task t' follows t, and $|t' >_L t|$ the total number of times that task t' precedes t. A task t is considered as a duplicate task iff:*

$max(min(|t >_L t'|, |t' >_L t|), 1) > 1$.

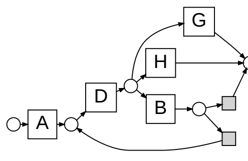
We use this strategy as a first step to detect the potential duplicates, and as a way to retrieve the local context of the activities. Step 1 in Fig. 2d shows the potential duplicate tasks detected after applying the described heuristic over the log of Fig. 2a. In this case, only D is detected as a potential duplicate task: task D is *preceded* by tasks A

[3] A *trace-model* creates a path for each trace of the log. This kind of model has a perfect precision and replay fitness as it only allows the specific behavior recorded in the log, but it is not a desirable solution.

$case_1$	$case_2$	$case_3$
A D G J	A D B D H J	A D B D B J

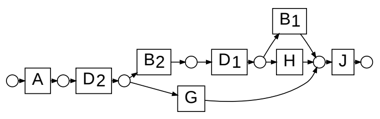
(a) *Log*

t	I(t)	O(t)
A	{}	{{D}}
D	{{A,B}}	{{G,B,H}}
G	{{D}}	{{J}}
J	{{G,B,H}}	{}
B	{{D}}	{{D,J}}
H	{{D}}	{{J}}



(b) *Initial solution*

t	I(t)	O(t)
A	{}	{{D_2}}
D_1	{{B_2}}	{{B_1,H}}
G	{{D_2}}	{{J}}
J	{{G,B_1,H}}	{}
B_1	{{D_1}}	{{J}}
H	{{D_1}}	{{J}}
D_2	{{A}}	{{G,B_2}}
B_2	{{D_2}}	{{D_1}}



(c) *Final solution*

Step	t	Local variables	Unrepaired Solution		Repaired Solution
1		*potentialDuplicates* = [D]			
2	D	**Subsequences from parsing** ind_0: *ADG*, *ADB*, *BDB* and *BDH*			
3	D	**Combinations**: $c[0].inputs$ = [A] $c[0].outputs$ = [G, B] $c[0].sharedOutputs$ = [B]; $c[0].sharedInputs$ = []; $c[1].inputs$ = [B] $c[1].outputs$ = [B, H] $c[1].sharedOutputs$ = [B]; $c[1].sharedInputs$ = [];			
4	D	**Duplicate D into D_1 and D_2 using the previous combinations and repair the causal dependencies**	→		→

t	I(t)	O(t)
A	{}	{{D_1}}
D_1	{{B}}	{{B,H}}
G	{{D_1}}	{{J}}
J	{{G,B,H}}	{}
B	{{D_1}}	{{D_1,J}}
H	{{D_1}}	{{J}}
D_2	{{A}}	{{G,B}}

→

t	I(t)	O(t)
A	{}	{{D_2}}
D_1	{{B}}	{{B,H}}
G	{{D_2}}	{{J}}
J	{{G,B,H}}	{}
B	{{D_1,D_2}}	{{D_1,J}}
H	{{D_1}}	{{J}}
D_2	{{A}}	{{G,B}}

Step	t	Local variables	Unrepaired Solution		Repaired Solution
5		**After duplicating** *D*: *potentialDuplicates* = [] *potentialDuplicatesL2L* = [B,G,H]			
6	B	**Subsequences from parsing** ind_0: D_2BD_1 and D_1BJ			
7	B	**Combinations**: $c[0].inputs$ = [D_2] $c[0].outputs$ = [D_1] $c[0].sharedInputs$ = [] $c[0].sharedOutputs$ = [] $c[1].inputs$ = [D_1] $c[1].outputs$ = [J] $c[1].sharedInputs$ = [] $c[1].sharedOutputs$ = []			
8	B	**Duplicate B into B_1 and B_2 using the previous combinations and repair the causal dependencies**	→		→

t	I(t)	O(t)
A	{}	{{D_2}}
D_1	{{B_1}}	{{B_1,H}}
G	{{D_2}}	{{J}}
J	{{G,B_1,H}}	{}
B_1	{{D_1}}	{{J}}
H	{{D_1}}	{{J}}
D_2	{{A}}	{{G,B_1}}
B_2	{{D_2}}	{{D_1}}

→

t	I(t)	O(t)
A	{}	{{D_2}}
D_1	{{B_2}}	{{B_1,H}}
G	{{D_2}}	{{J}}
J	{{G,B_1,H}}	{}
B_1	{{D_1}}	{{J}}
H	{{D_1}}	{{J}}
D_2	{{A}}	{{G,B_2}}
B_2	{{D_2}}	{{D_1}}

(d) Steps of SLAD.

Figure 2: An example on how SLAD works.

and B and directly followed by B, G and H. Hence, as $max(min(3,2),1) > 1$, activity D is a potential candidate for label splitting. In the next steps of the algorithm, we extend this heuristic in order to get more information about the duplicate activities and perform the adequate split of the tasks (Section 3.2). Moreover, we improve this heuristic to detect potential duplicate activities in loops (Section 3.3).

One particular scenario of this initial approach to detect the potential duplicate tasks is related to the *start* and *end* activities of a trace. For instance, if we consider the trace $\sigma = \{A, B, C, A\}$, the activity A is never going to be selected as a potential duplicate task through this method, as A is only followed by B and preceded by C, i.e., $max(min(1,1),1) = 1$. To overcome this situation, one possible solution is to add a *dummy* start and end activity to each trace (Definition 5). Thus, by changing σ to $\sigma' = \{start, A, B, C, A, end\}$, it is possible to add the activity A into *potentialDuplicates*, as $max(min(2,2),1) > 1$.

Definition 5 (Dummy tasks)**.** *Let L be an event log over T. Let 'start' be a dummy task with no predecessors, i.e. $|t' >_L start| = 0$, and 'end' be a dummy task with no successors, i.e., $|end >_L t'| = 0$. Then L^+ is an event log over T^+ with the dummy activities such as:*

$T^+ = T \cup \{start, end\}$,

$L^+ = \{start\ \sigma\ end \mid \sigma \in L\}$.

3.2. Extending the model with duplicate tasks

Once all the potential duplicates are identified, the algorithm splits the input and output dependencies of these activities of the model into multiple tasks with the same label through the function *localSearch* (Alg. 1:8). Within this function, the algorithm calculates the input and output combinations for each activity in *potentialDuplicates* (Alg. 1:11-13), in order to split their behavior among new tasks. To compute these combinations, the algorithm firstly finds all the subsequences —Definition 6— (Alg. 1:12) in the log L that match the pattern $t_1 t t_2$ —window of size 1— where $t_1 \in \mathrm{I}(t)$ and $t_2 \in \mathrm{O}(t)$ in the model —being $\mathrm{I}(t)$ and $\mathrm{O}(t)$ the inputs and outputs of task t, respectively. We add this input/output constraint to focus only on those patterns that can be reproduced by the initial model. Following with the example shown in Figure 2, when iterating over *potentialDuplicates*, the algorithm must find the subsequences that match the pattern $t_1 \mathrm{D} t_2$ in the traces of the log L. The resultant subsequences are shown in the Step 2 in Fig. 2d. Note that all these subsequences satisfy the input and output dependencies of the task D in the initial model of Fig. 2b.

Definition 6 (Subsequences)**.** *Let CM be a causal matrix (T, I, O). Let L be an event log. Let $\sigma = t_1 t_2 \ldots t_n$ and $i \in \{1, \ldots, n-1\}$ be a trace such that $\sigma \in L$. The subsequences of a task $t \in T$ are defined by:*

$S = \{(t_{i-1} t_i t_{i+1}) \mid t_{i-1}, t_i, t_{i+1} \in \sigma \wedge t_{i-1} >_L t \wedge t >_L t_{i+1} \wedge t_{i-1} \in I(t) \wedge t_{i+1} \in O(t)\}$.

After obtaining such subsequences, the algorithm creates the combinations —through the function *calculateCombinations* (Alg.2)— which will serve as the basis to split the original task into multiple activities. These combinations —Definition 7— represent the *context* in which the original task is involved —in terms of relations— with the other tasks of the log. *calculateCombinations* builds such combinations of a task t following three rules (Alg.2:7-14). First,

Algorithm 2: Algorithm to compute the combinations of a task.

```
1  Function calculateCombinations(subsequences, t)
2      combinations ← ∅
3      forall the ((t_1 t t_2) ∈ subsequences) do
4          c ← ∅
5          Create a set c.inputs with the subsequences that share the same $t_1$ and add in c.outputs their respective $t_2$
6          Add c to combinations
7      foreach c ∈ combinations do
8          if c.outputs = c'.outputs where c' ∈ combinations then
9              c.inputs = c.inputs ∪ c'.inputs and c.outputs = c.outputs ∪ c'.outputs
10             combinations = combinations \ c'
11         if c.outputs shares an element e with another c'.outputs then
12             c.sharedOutputs ← c.sharedOutputs ∪ {e}
13         if c.inputs shares an element e with another c'.inputs then
14             c.sharedInputs ← c.sharedInputs ∪ {e}
15     return combinations
```

given two subsequences $t_1 \mathrm{t} t_2$ and $t_3 \mathrm{t} t_4$, if $t_1 = t_3$, then SLAD merges both subsequences into a new combination (Alg.2:3-6). Second, and after identifying all the combinations, given two different combinations c and c', if they share the same *outputs*, i.e., *c.outputs* = *c'.outputs*, these two combinations are merged (Alg.2:8-10). With these two rules SLAD can split the behavior of the task into different contexts combining the mined causal dependencies and the local information of the log. Finally, if the intersection between two combinations is not the empty set, SLAD records which inputs and outputs are shared by both combinations (Alg.2:11-14) —this information is later used in the repairing step. Step 3 in Fig. 2d shows the combinations generated from the subsequences in which task D is involved. For instance, one of the combinations is generated based on the subsequeces ADG and ADB; and the other one is based on the subsequences BDB and BDH. In this case, task B is shared in the *outputs* of both combinations, therefore $c[0]$.*sharedOutputs* = $[B]$ and $c[1]$.*sharedOutputs* = $[B]$; in the same way, none of the combinations share an element in their inputs, therefore both *sharedInputs* subsets are empty.

Definition 7 (Combinations)**.** *Let S be a set of subsequences. We define a combination c as a group of subsequences such as:*

$C = \{c \in \mathbb{P}(S) \mid \forall (xyz), (ijk) \in S : x = i \wedge y = j\}$.

Let n be the total number of combinations $c \in C$ and m the total number of subsequences $s \in c$. Hence, for a task t, each $c \in C$ represents a set of k grouped subsequences $(t'tt'')$ where:

c.inputs $= \{\bigcup_{i=1}^{m} t' \mid t' >_L t \in s_m\}$

c.outputs $=\{\bigcup_{i=1}^{m} t'' \mid t >_L t'' \in s_m\}$

c.sharedInputs $=\{\bigcap_{i=1}^{n} c_i.inputs \mid c_i \in C\}$

c.sharedOutputs $=\{\bigcap_{i=1}^{n} c_i.outputs \mid c_i \in C\}$

After building all the possible combinations, the next steps in Alg. 1 are straightforward. For each combination c

(Alg.1:14), SLAD creates a new task t' equal to the original task t of the current model (Alg.1:15). Then, it removes from *Input*(t) all the tasks shared with *c.inputs*, but keeping the tasks that are in *c.sharedInputs* (Alg.1:16). On the other hand, for the new task t', it retains only the elements in *Input*(t') that are contained in *c.inputs* (Alg.1:17). The same process is applied for the outputs of both t and t' but with *c.outputs* and *c.sharedOutputs* (Alg.1:18-1:19). With this process the algorithm redistributes the inputs and outputs of the original task among the new task. If these two tasks are compliant with the model, i.e. both their inputs and outputs are not empty[4] (Alg.1:20), the new task is included in ind_0 —the actual model—, and the original task of ind_0 is updated (Alg. 1:21). Otherwise the model goes back to its previous state and tries a new combination. Following with the example, SLAD, using the combinations previously calculated (Step 3 of Fig. 2d), splits the inputs and outputs of the original task D among two new tasks D_1 and D_2 —for the sake of the argument we label the duplicate tasks with different subscripts. This process gives as a result the unrepaired model shown in Step 4 of Fig. 2d.

Definition 8 (Repairing process)**.** *Being t the original task, t' a new task split from t, a Causal Matrix (T, I, O) is repaired as follows:*

$\forall t'' \in O(t') \rightarrow I(t'') = I(t'' : t \rightarrow t')$

$\forall t'' \in I(t') \rightarrow O(t'') = O(t'' : t \rightarrow t')$

When including this new task, the model has to be repaired (Alg.1:22), as some of the dependencies of the original task t are now in a new task t' with the same label, resulting in an inconsistent model, e.g., the activity D_2 has the activity B as output, but activity B has no activity D_2 as input, i.e., it still has the activity D_1. The repairing process works as stated in Definition 8. In summary, being t the original task and t' the new task, for each task t" that was eliminated from O(t), the process checks if $t \in$ I(t''). If that is true, t has to be replaced *in each subset* of I(t'') with t'. This process is repeated also for the input sets. Then, the algorithm performs a post-pruning of the model removing the unused arcs, i.e., arcs whose frequency of use is zero (Alg.1:23). In order to evaluate the models (Alg.1:24), we based their quality on three criteria: *fitness replay*, *precision* and *simplicity*. To measure these criteria we used the hierarchical metric defined in [36], that first compares the fitness replay, then the precision and last the simplicity of the process models:

Definition 9 (Process models dominance)**.** *Let x, x' be two process models. Let $F(x)$, $P(x)$ and $S(x)$ be, respectively, the replay fitness, precision and simplicity of the process model x. The process model x is better than the process model x' iff:*

$$\begin{aligned} x \geq x' \iff \quad & [F(x) > F(x')] \\ & \vee [F(x) = F(x') \wedge P(x) > P(x')] \\ & \vee [F(x) = F(x') \wedge P(x) = P(x') \wedge S(x) > S(x')] \end{aligned}$$

[4]Based on Definition 5, only the dummy activities *start* and *end* should have an empty input and output set, respectively, as they are the only initial and final activities of the event log L.

Thus, replay fitness is the primary ordering criteria. When two solutions have the same replay fitness, precision is used to decide the best solution. When two solutions have the same replay fitness and precision, simplicity is the decisive criterion. Finally, if the new model with duplicate tasks is better, it means that the task t was correctly duplicated. Therefore, the best solution ind_{best} is replaced with ind_0 (Alg.1:28). Otherwise, the model goes back to its previous state and repeats the process with a new combination.

In the previous example, after splitting D into two new activities, SLAD ends up with an inconsistent model (Step 4 of Fig. 2d). Therefore, the dependencies of the model have to be repaired as explained. Note the task B is shared by both outputs of tasks D_1 and D_2. This has to be taken into account when repairing the process model, as B must now contain both tasks as inputs. When this situation occurs, the repairing process must add a new relation into the model. In other words, if O(t) and O(t') share an activity t'' when repairing the model, t' should be added *in the same subset* as t in $I(t'')$. Thus in the repaired solution shown in the Step 4 of Fig. 2d, I(B) contains as input both tasks D_1 and D_2. Finally, in this example, the new repaired model after duplicating task D has a better precision than the initial solution, and therefore the best model is updated.

3.3. Handling length-two-loops

One of the novelties of SLAD is its ability to duplicate tasks in loops, specially in Length-two-Loops (L2L). In process mining, there is a well known relation between duplicate tasks and loops [30]. For example, in Fig. 2a the sequence of activities BD is executed multiple times, i.e., twice, but, as shown in the model depicted in Fig. 2c, it can be considered that it is not a loop. Depending on the perspective, modeling this behavior as a loop could be a valid solution —for instance, by generalizing the model and hence allowing more executions of the loop than the recorded number in the log. With SLAD, we are going to consider that this type of behavior, i.e., situations where a sequence of activities is executed multiple times —twice as maximum—, will be modeled with duplicate tasks with the aim of improving the *precision* of the model. Otherwise, these situations —more than two repetitions— will be modeled as a Length-one-Loop (L1L) or Length-two-Loop (L2L) —we do not want to unfold a loop by converting it into a large sequence. Detecting if the behavior can be modeled as a L1L or with two tasks with the same label can be detected with the previous process (Section 3.2). The problem arises when deciding if a L2L can be modeled with duplicate tasks.

The main limitation of the heuristic followed to detect the possible duplicate tasks of the log —Definition 4— is that it does not cover all the search space, particularly with tasks involved in a Length-two-Loop situation, as it breaks the rule of two tasks sharing the same input and output dependencies (Definition 3). For instance, considering the log of Fig. 2a, only the activity D is identified as a potential duplicate activity, making it impossible to include in the search space the model depicted in Fig. 2c, as B is not identified as a duplicate task: based on the log, B is only preceded by D and followed by D and J, hence $max(min(2, 1), 1) = 1$.

Making this process recursive can solve this drawback: when a task t is detected as a duplicate activity, the upper bound for all the tasks t' that directly follow t must be updated, because these tasks will now have multiple tasks with the same label as input. Note that updating only the tasks t' that directly follow t is sufficient, as this situation only occurs when loops of length two are involved in the computation of this heuristic —a task in a L2L usually have the same activity as input and output. To avoid a infinite recursion —Theorem 1—, the maximum number of times a task t can be modified by this recursive operation is twice. This bound is set also to avoid ending with a potential trace-model by unfolding a loop into a large sequence. In order to mimic this behavior in the proposed algorithm, if a task t is correctly duplicated in the model (Alg.1:28), we add the tasks that directly follow t into $potentialDuplicatesL2L$ (Alg.1:29). Following with the previous example, when duplicating D, the algorithm also needs to include B as a potential duplicate task. Therefore, after splitting the behavior of D among the new tasks D_1 and D_2, the algorithm adds to $potentialDuplicatesL2L$ the outputs of D, i.e. $potentialDuplicatesL2L = [B, G, H]$ as shown in the Step 5 of Fig. 2d.

Theorem 1 (Infinite recursion)**.** *Given a trace with a very long length-two-loop, recursively applying Definition 2 after splitting an activity can lead to a trace-model.*

Proof. Let T be a set of tasks. Let L be a log over T and $\sigma = \ldots t_1 t_2 t_3 t_4 t_5 \ldots$ be a fragment of trace where label(t_2) = label(t_4) = a, and label(t_3) = b, i.e., $\sigma = \ldots t_1 abat_5 \ldots$. Based on Definition 4, the task a is selected to be duplicated. Assuming that this task is correctly split, this results in $\sigma = ...t_1 a_1 ba_2 t_5....$ As $b \in O(a)$, task b is selected as a potential duplicate task. If the task b is correctly split, as $a_2 \in O(b)$, a_2 is going to be added again as a potential duplicate task. Let's consider now that label(t_5) = b_2, i.e., $\sigma = ...t_1 a_1 b_1 a_2 b_2....$ As $b_2 \in O(a_2)$, b_2 is selected again as a duplicate task. If the L2L between tasks a and b in σ is infinite, i.e., $\{\ldots ababab a \ldots\}$, this could lead to an infinite recursion between these two tasks as long as they are correctly split. □

Once the algorithm ends its iteration over the main loop (Alg.1:11-31), it checks if there are tasks that were affected by the duplication of other activities, i.e., it checks if $potentialDuplicatesL2L$ is empty (Alg.1:32). If this condition is false, the algorithm makes a recursive call but considering $potentialDuplicatesL2L$ as input (Alg.1:33) —following with the example, now the algorithm has to iterate over $[B, G, H]$. In this new iteration, the process is the same as explained before.

Considering that the new task to be duplicated in this new iteration is B, the algorithm first retrieves the context in which this task is involved. In this new particular case, we have to take into account that the original task D from the log was duplicated in the model, generating D_1 and D_2. Therefore, when reproducing the solution over the log, we can check which activities with the same label $D \in \mathrm{I}(B)$ were executed just before B and which activities $t \in \mathrm{O}(B)$ were executed after B. With this process, the algorithm creates the sequences needed to generate the combinations to split B: when D_2 is executed before B, then D_1 is always executed, and when D_1 is executed before B, J is always executed later —where $J, D_1 \in \mathrm{O}(B)$. These subsequences are shown in the Step 6 of Fig. 2d.

Then, with these subsequences the algorithm builds the combinations considering that the tasks with the same label are different (Step 6 in Fig. 2d), and create two different combinations as explained in Section 3.2. The resultant combinations are shown in the Step 7 in Fig. 2d. Note that, as these combinations do not share neither *inputs* nor *outputs*, both *sharedInputs* and *sharedOutputs* are empty. With this process, the algorithm is able to split B into two tasks B_1 and B_2, obtaining the model shown in the Step 8 in Fig. 2d. After repairing the solution —for instance, $\text{I}(D_1)$ has to be updated—, this model achieves a better quality than the best solution so far (Step 8 in Fig. 2d), allowing to unfold the loop into a sequence of tasks, and retrieving the model depicted in Fig. 2c. If we would have extended the log with a new trace repeating the sequence *BD* more than twice, i.e., a case like $<A,D,B,D,B,D,H,J>$, the obtained model would have a lower fitness replay than the actual best solution —it would be impossible to execute the tasks more than twice— and therefore this behavior would be modeled as a L2L instead of unfolding the loop.

At this point the algorithm will continue to check if the other tasks can be duplicated —and even make another recursive call with the outputs of the already split task B. However, the model cannot be further improved by SLAD, as it perfectly models the behavior of the log.

4. Experimentation

The validation of SLAD has been done with a set of synthetic models from [9, 19]. Table 1 summarizes the original known models on the basis of their activities and the workflow patterns that each net contains[5], For example, the *FlightCar* model has eight *different tasks* structured in sequences, choices and parallel constructs. It also has duplicate tasks in sequence[6], meaning that, although the model contains eight different labels, it can be represented with more than eight activities. For each of these models, there is log with 300 traces. Table 1 shows the total number of events in each log. Note that, in these logs, we also included two additional dummy activities —a start and end activity—, as some algorithms used in the experimentation, including SLAD, are very sensitive when handling event logs with more than one start and/or end points [3, 9, 36, 38].

4.1. Metrics

The quality of the models retrieved by the proposed approach were measured by taking into account three objectives: fitness replay, precision and simplicity. To measure the fitness replay (F), we use the *proper completion* metric [22]:

$$F = \frac{PPT}{|L|} \tag{1}$$

where PPT is the number of properly parsed traces, and $|L|$ is the total number of traces in the event log. Hence, *proper completion* takes a value of 1 if the mined model can process all the traces without having missing tokens or tokens left behind. Also, the precision (P) is evaluated as follows:

$$P = 1 - max\{0, P'_o - P'_m\} \tag{2}$$

[5]All the datasets and experiments can be found in `http://tec.citius.usc.es/processmining/SLAD`.

[6]Models with duplicate tasks in parallel mean that the activities with the same label are executed in different branches, whereas duplicate tasks in sequence are executed in the same branch.

Table 1: Process models used in the experimentation.

		Activity structures										Log content	
Model Name	#Labels	Sequence	Choice	Parallelism	Length-One Loop	Length-Two Loop	Structural Loop	Non-local NFC	Invisible tasks	Duplicates in Sequence	Duplicates in Parallel	#traces	#events
betaSimpl [9]	13	✓	✓					✓	✓	✓		300	4,209
FlightCar [9]	8	✓	✓	✓							✓	300	2,385
Fig5p1AND [9]	5	✓		✓						✓		300	2,400
Fig5p1OR [9]	5	✓	✓							✓		300	2,100
Fig5p19 [9]	8	✓	✓	✓					✓	✓		300	2,428
Fig6p9 [9]	7	✓	✓						✓	✓		300	2,592
Fig6p10 [9]	11	✓	✓	✓					✓	✓		300	4,376
Fig6p25 [9]	21	✓	✓			✓	✓		✓	✓		300	5,661
Fig6p31 [9]	9	✓	✓							✓		300	2,400
Fig6p33 [9]	10	✓	✓							✓		300	2,504
Fig6p34 [9]	12	✓	✓	✓			✓		✓	✓		300	5,406
Fig6p38 [9]	7	✓		✓							✓	300	3,000
Fig6p39 [9]	7	✓		✓					✓		✓	300	2,684
Fig6p42 [9]	14	✓	✓	✓					✓	✓		300	3,420
RelProc [9]	16	✓	✓	✓						✓		300	4,155
Alpha [19]	11	✓	✓	✓						✓	✓	300	3,978
Loop	7	✓	✓		✓					✓		300	2,175
FoldedOr	6	✓	✓							✓		300	2,200

where P'_o and P'_m are, respectively, the precision of the original model and the precision of the mined model, both evaluated through *alignments*. More specifically, we used the metric defined in [1], considering all possible optimal alignments. It takes a value of 1 if all the behavior allowed by the model is observed in the log. As the original model is the optimal solution, we use it to normalize the precision. Therefore, P will be equal to 1 if the mined model has a precision (P'_m) equal or higher than the original model (P'_o). When the precision of the mined model is worse than that of the original model, P will take a value under 1 —the lower the precision of the mined model, the closer the value of P to 0. Finally, for the simplicity (S) we use:

$$S = \frac{1}{1 + max\{0, S'_m - S'_o\}} \tag{3}$$

where S'_m and S'_o are, respectively, the simplicity of the mined model and the simplicity of the original model, both calculated with the *weighted P/T average arc degree* defined in [23] —the higher the value of S' the lower the simplicity. As explained with the precision, we use the original model —which is the optimal model— to normalize the simplicity. S takes a value of 1 if the simplicity of the mined model is equal or higher than that of the original model, i.e., $S'_m \leq S'_o$. If the simplicity of the mined model is worse than that of the original model ($S'_m > S'_o$), S will take a value under 1 —the worse the simplicity of the mined model, the closer the value of S to 0.

We used the tool CoBeFra [4] to compute the different metrics. Note that the model input representation for this tool is a Petri net, therefore we had to map each heuristic net retrieved by SLAD into its equivalent Petri net.

4.2. Setup

As explained in Section 3, SLAD takes as starting point a model and a log. Therefore, the first step in the validation process is to apply a process discovery technique over the previously described logs (Table 1) to retrieve a heuristic net or causal net as input. In order to test how the solutions of different algorithms affect the behavior of SLAD, we repeated the experiments taking as starting point the solutions of three different algorithms that retrieve solutions in the aforementioned format: ProDiGen [36], Heuristics Miner (HM) [38], and Fodina [3] —all these algorithms, by default, do not handle label splitting. To identify the solutions retrieved by these algorithms after applying SLAD, we have added the suffix "+*SLAD*". Additionally, we also used Inductive Miner [18] (IMi) in the experimentation[7].

On the other hand, we wanted to compare the results of mining the duplicate activities *before* or *after* the discovery process. Therefore, we also repeated the experiments with Duplicate Genetic Miner (DGM) [9], Evolutionary Tree Miner (ETM) [6], a state-based region theory algorithm (TS) [2, 24] and Fodina [3], as it can mine duplicate activities if enabled —we use the name Fodina+D as a way to reference the algorithm with this parameter enabled. Additionally, we also tested if it is possible to apply SLAD to an already mined solution with duplicate activities (with Fodina and DGM). Note that for all these algorithms we used the default settings specified by the authors. More specifically: i) for Inductive Miner we guarantee a perfect replay fitness; ii) for DGM we set the *maximum number of iterations* to 5,000, and a *population size* of 50; iii) for the ETM we generate the pareto front for each log, retrieving the solution with the highest fitness replay and precision; and iv) for the state-based region theory algorithm, when discovering the transition system, we set *no limit* in the *set* size, and the inclusion of all activities. We used the ProM framework [33] to execute each of these algorithms.

4.3. Results

Table 2 shows the results —in terms of fitness replay (F), precision (P) and simplicity (S)— retrieved for each algorithm over each log. Moreover, Table 2 also shows information about which algorithm retrieves better results for each metric and log —highlighted in grey. In Section 4.3.1 we prove that applying SLAD to the mined models improves the solutions retrieved by the algorithms that do not take into account duplicate tasks. In Section 4.3.2 we show that the models obtained with SLAD, considering this case study, are better than those mined with other algorithms that consider duplicated tasks —Fodina+D, ETM, DGM.

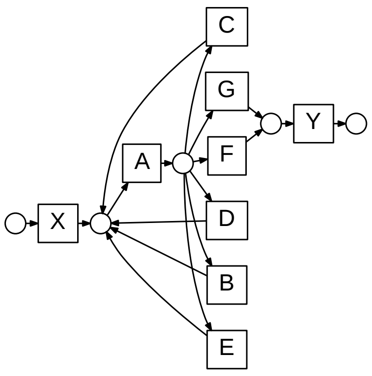
4.3.1. Improvement of the mined models through SLAD.

First, we analyze the results of the algorithms used to retrieve the initial solution for SLAD: ProDiGen, HM, Fodina and IMi —rows 1-4 in Table 2. With ProDiGen and IMi, all the solutions have a perfect fitness replay, whereas HM and Fodina do not achieve a perfect value for this quality dimension in some of the cases. In general, for

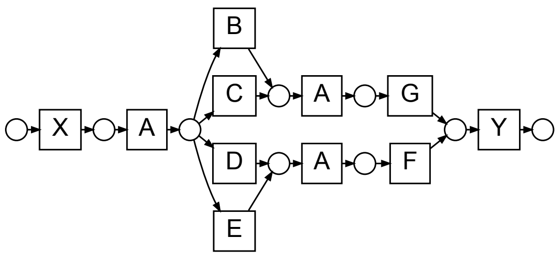
[7]We did not apply SLAD over IMi and ETM, as these algorithms retrieve process trees as a solution, and, in some situations, this kind of models cannot be easily translated to heuristic nets without changing its internal behavior.

Table 2: Results for the 18 logs.

			Logs																	
			Alpha	Folded	Loop	Fig6p25	betaSimpl.	flighCar	Fig5p19	Fig5p1AND	Fig5p1OR	Fig6p10	Fig6p31	Fig6p33	Fig6p34	Fig6p38	Fig6p39	Fig6p42	Fig6p9	RelProc
1	*ProDiGen*	*F*	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
		P	0.85	0.75	0.8	0.78	0.92	0.81	0.9	0.76	0.73	0.82	0.61	0.65	0.8	0.93	0.93	0.7	0.79	0.94
		S	0.82	0.84	0.86	0.88	0.92	0.82	0.88	0.69	0.67	0.82	0.61	0.73	0.82	0.84	0.92	0.79	0.79	0.94
2	*HM*	*F*	1.0	1.0	1.0	1.0	1.0	0.32	1.0	0.67	1.0	1.0	1.0	1.0	0.41	0.0	0.0	0.07	0.21	1.0
		P	0.77	0.75	0.8	0.76	0.92	0.81	0.9	0.75	0.67	0.82	0.56	0.6	0.83	0.57	0.6	0.64	0.95	0.94
		S	0.85	0.84	1.0	0.86	0.99	0.81	0.94	1.0	0.8	0.82	0.63	0.68	0.78	0.62	0.82	0.82	0.99	0.94
3	*Fodina*	*F*	1.0	1.0	1.0	1.0	0.23	0.32	0.28	0.32	1.0	1.0	1.0	1.0	1.0	1.0	0.52	0.31	0.21	0.72
		P	0.77	0.75	0.8	0.76	0.97	0.88	0.82	0.73	0.67	0.82	0.56	0.6	0.8	0.93	1.0	0.93	0.86	0.96
		S	0.85	0.84	1.0	0.86	1.0	1.0	1.0	1.0	0.8	0.82	0.63	0.68	0.82	0.91	1.0	1.0	1.0	0.98
4	*IMi*	*F*	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
		P	0.78	0.83	0.73	0.79	0.68	0.81	0.73	0.83	0.7	0.66	0.63	0.67	0.54	0.74	0.95	0.34	0.7	0.76
		S	0.84	0.88	0.98	0.81	0.98	0.89	0.87	0.92	0.83	0.77	0.69	0.73	0.82	0.86	1.0	0.9	0.84	0.85
5	*ProDiGen + SLAD*	*F*	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
		P	0.97	1.0	1.0	1.0	1.0	1.0	1.0	0.98	1.0	1.0	1.0	1.0	1.0	1.0	0.94	1.0	1.0	1.0
		S	0.91	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.89	1.0	1.0	1.0
6	*HM + SLAD*	*F*	1.0	1.0	1.0	1.0	1.0	0.32	1.0	0.67	1.0	1.0	1.0	1.0	0.72	1.0	0.53	0.36	0.21	1.0
		P	0.86	1.0	1.0	1.0	0.99	0.82	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.94	0.95	0.95	1.0
		S	0.96	1.0	1.0	1.0	1.0	0.84	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.99	1.0
7	*Fodina + SLAD*	*F*	1.0	1.0	1.0	1.0	1.0	0.45	0.28	0.32	1.0	1.0	1.0	1.0	1.0	1.0	0.52	0.31	0.21	0.72
		P	0.86	1.0	1.0	1.0	1.0	0.88	0.82	0.73	1.0	1.0	1.0	1.0	1.0	0.93	1.0	0.93	0.86	1.0
		S	0.85	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.91	1.0	1.0	1.0	0.99
8	*Fodina + D*	*F*	1.0	1.0	1.0	1.0	1.0	1.0	0.52	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.22	0.75	1.0	0.62
		P	1.0	0.75	0.74	1.0	1.0	1.0	1.0	1.0	1.0	0.91	1.0	1.0	0.85	0.93	0.91	0.97	1.0	0.98
		S	1.0	0.84	0.84	1.0	1.0	1.0	1.0	1.0	1.0	0.92	1.0	1.0	0.85	0.91	0.92	1.0	1.0	0.98
9	*DGM*	*F*	1.0	0.67	0.33	0.38	1.0	1.0	1.0	1.0	1.0	0.43	1.0	0.66	0.41	1.0	0.23	0.38	1.0	0.28
		P	1.0	0.8	0.87	0.98	1.0	1.0	1.0	1.0	1.0	0.67	1.0	0.85	0.7	0.93	0.78	0.62	1.0	0.78
		S	1.0	0.78	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.91	1.0	0.84	0.99	0.91	0.81	0.88	1.0	0.83
10	*ETM*	*F*	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
		P	1.0	0.9	0.83	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.9	0.73	0.91	1.0	0.95	0.11	0.92	0.14
		S	1.0	0.93	0.92	0.47	1.0	1.0	1.0	1.0	1.0	1.0	0.87	0.7	0.9	1.0	0.58	0.41	0.93	0.41
11	*TS*	*F*	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
		P	0.51	0.62	0.87	0.89	0.95	1.0	0.82	0.78	0.75	1.0	0.82	0.82	1.0	0.84	0.94	0.91	0.77	0.84
		S	0.35	0.74	0.62	0.46	0.55	1.0	0.41	0.7	0.83	0.37	0.59	0.61	0.35	0.61	0.61	0.55	0.46	0.36
12	*Fodina + D + SLAD*	*F*	1.0	1.0	1.0	1.0	1.0	1.0	0.52	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.36	0.75	1.0	0.62
		P	1.0	1.0	0.98	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.88	0.93	1.0	1.0	1.0	1.0
		S	1.0	1.0	0.98	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.91	1.0	1.0	1.0	1.0



(a) *Mined model by Heuristics Miner without duplicate tasks.*



(b) *Mined model by Heuristics Miner and SLAD.*

Figure 3: A Petri net mined for the log *Fig6p31* before and after SLAD.

all the algorithms, the mined nets contain overly connected nodes that make the models hardly readable and complex, i.e, models with a poor precision and activities with a high density of incoming/outgoing arcs, needless loops, or too many invisible activities.

Then, we applied SLAD to the models mined by ProDiGen, HM and Fodina. Our proposal was able to enhance the results in 45 out of 54 solutions —rows 5-7 in Table 2. One example of this improvement can be seen in Figure 3, where Figure 3a shows the original Petri net mined by HM for the log *Fig6p31*, and Figure 3b shows the Petri net after finding the duplicate tasks through SLAD —task *A* in this case. In this example, the process model of Figure 3a has a very low precision (0.56) as the number of different traces it can generate is infinite due to the loop with the activity A. However, through the inclusion of duplicate labels (Figure 3b), we can limit its behavior to only four different paths, i.e., we create a more specific process model.

Furthermore, for those initial models mined by ProDiGen, SLAD was able to improve the precision in all the cases. Taking as starting point the initial solutions retrieved by HM, 17 out of 18 solutions were improved not only in terms of precision, but also the fitness replay achieves a higher value in four of those models. After applying SLAD over the results retrieved by Fodina, 12 out of 18 solutions were improved in terms of both precision and fitness replay. Analyzing the simplicity of the models, we have to take into account that when performing the label splitting over the initial models, there is a reduction in the number of invisible activities —used as control structures— as we are reducing most of the needless loops and overly connected nodes. For example, with the log *Fig6p10*, ProDiGen, Fodina and HM retrieve a solution —the three algorithms retrieve the same solution— with 46 arcs, and 22 tasks —including the invisible activities. After SLAD, the final solution has, in total, 36 arcs and 17 tasks.

Table 3: Wilcoxon test for each algorithm with and without SLAD.

Comparison	p-value	Result
ProDiGen+SLAD vs ProDiGen	0.0002	**Rejected**
HM+SLAD vs HM	0.0002	**Rejected**
Fodina+SLAD vs Fodina	0.0019	**Rejected**
Fodina+D+SLAD vs Fodina+D	0.014	**Rejected**

We have compared the results of applying SLAD by means of non-parametric statistical tests —through the web platform STAC [21]—, checking if each algorithm with SLAD improves its correspondent algorithm without SLAD. The Wilcoxon test [41] has been applied, using as null hypothesis that the medians of the quality of the solutions are equal with a given significance level (α). However, as we are using 3 different metrics, the comparison must be done in a multi-objective way. In order to perform a fair comparison, we have used the criterion of Pareto dominance. We have applied the *fast-non-dominated-sort* [12] in order to rank the solutions of the algorithms for each log. With this method, a mined model a dominates other mined model b, i.e., $a \succ b$, if the model a is not worse than the model b in all the objectives —the 4 metrics— and better in at least one objective. Thereby, for each log, all the solutions in the first non-dominated front will have a rank equal to 1 —these are the solutions more similar to the original model and, therefore, the best ones—, the solutions in the second non-dominated front will have a rank equal to 2, and the process continues until all fronts are identified. Therefore, to perform the non-parametric statistical tests, we first ranked all the solutions for each log based on the Pareto dominance and then we used these ranks as input for the tests.

Table 3 summarizes such test. The hypothesis —which states that the solutions retrieved before and after applying SLAD are equal— is rejected in all the cases, as the *p-value* for each comparison is lower than the given confidence level (α = 0,05). This means that SLAD was able to i) *significantly* improve the precision of the models, and ii) enhance their structural clarity by splitting the behavior of the overly connected activities. In summary, SLAD significantly enhances the solutions of the process discovery algorithms.

4.3.2. Comparison of process discovery algorithms with duplicate activities.

Rows 8-11 in Table 2 show the results of the algorithms that take into account duplicate activities in the mining process. Fodina, when mining duplicate activities, is able to enhance the solution on 13 out of 18 solutions, however, this algorithm also retrieves a worse solution in three of the cases —*Loop*, *Fig6p39* and *RelProc*— than without duplicate activities. On the other hand, DGM is able to retrieve the original model in eight of the 18 cases, and ETM in seven of the 18 cases. In particular, these algorithms were able to retrieve to original solution in cases where two activities with the same label are executed in different branches of a parallel construct. For instance, the three algorithms were able to retrieve the original model with the *Alpha* log. On the other hand, TS was only able to retrieve the original solution in one case. In all the cases the state-based region theory algorithm results process models with a guaranteed perfect replay fitness. However, this type of algorithms tend to overfit the event log, resulting in solutions with a very low simplicity.

Comparing the solutions of performing the label splitting after —rows 5-7 in Table 2— and before —rows 8-11 in Table 2— we can extract that with SLAD, the most difficult scenario is related to those situations where two different activities with the same label are executed in different branches —log *Alpha* or *Fig5p1AND*. To achieve this duplicity of a label in different branches of a parallel construct, it is necessary to perform the label splitting *before* or during the mining step, otherwise the process mining techniques usually isolate the affected activity into only one branch, precluding the label splitting in a post-processing step. Additionally, SLAD is highly dependant on the input solution, more specifically, on the fitness replay of the initial model. As can be seen in the results after applying SLAD over Fodina and HM —rows 5-6, respectively, of Table 2—, if the input model avoids the overly connected nodes by means of reducing its fitness replay, it is more difficult to perform the label splitting through SLAD.

We also compared in more detail the solutions retrieved with Fodina detecting the duplicate activities *after* (Fodina+SLAD) and *before* (Fodina+D) mining the model —rows 7-8 of Table 2. Within this context, detecting the duplicate activities through SLAD improves 14 out of 18 solutions, whereas detecting the duplicate activities before the mining process, improves 13 out of 18 solutions. In particular, there are some solutions that were improved performing the label splitting before —for instance, log *Fig6p9*—, and after —for instance, log *Fig5p19*— the process mining. On the other hand, as previously indicated, in three of the solutions Fodina+D retrieved a worse model than the one without duplicate activities, whereas with SLAD the final solution is always the same or better than the original.

It should be noted that performing the label splitting before and after the mining process is not exclusive, i.e. it is possible to apply SLAD to a solution that already contains duplicate activities. Therefore, we carried out this exper-

Table 4: Friedman ranking for all the algorithms with SLAD.

Algorithm	Ranking
ProDiGen+SLAD	25.528
Fodina+D+SLAD	35.583
HM+SLAD	40.611
Fodina+SLAD	44.278
p-value: 0.0941	

iment with the solutions retrieved by Fodina+D (Fodina+D+SLAD) and DGM (DGM+SLAD). Fodina+D+SLAD enhances a total of 7 out of 18 models —row 12 of Table 2— of the solutions retrieved by Fodina+D. On the other hand, SLAD was unable to enhance any of the models mined by DGM, therefore we omitted this row in the table, as the results where the same as the achieved by the DGM —row 9 of Table 2.

Finally, we compared the algorithms that include the duplicate activities in the mining process with the best algorithm with SLAD. In order to select the best algorithm with SLAD, we applied the Friedman Aligned Ranks test [16], using the rank solutions based on the Pareto dominance. This test computes the ranking of the results of the algorithms rejecting the null hypothesis —which states that the results of the algorithms are equivalent— with a given confidence or significance level (α). Then we applied the Holm's post-hoc test [17] for detecting significant differences among the results. Table 4 summarizes this test, ranking ProDiGen+SLAD as the best algorithm. We omitted the Holm's post-hoc test, as based on the p-value of the previous test, the differences are not significant.

Table 5: Non-parametric test.

(a) *Friedman ranking.*

Algorithm	Ranking
ProDiGen+SLAD	22.527
Fodina+D	40.833
ETM	44.666
DGM	50.277
TS	69.194
p-value: 0.0007	

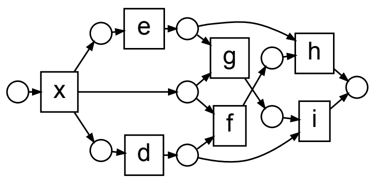
(b) *Holm post-hoc, α = 0.05.*

Comparison	Adjusted p-value	Result
ProDiGen+SLAD vs TS	0.0000	**Rejected**
ProDiGen+SLAD vs DGM	0.0043	**Rejected**
ProDiGen+SLAD vs ETM	0.0220	**Rejected**
ProDiGen+SLAD vs Fodina+D	0.0355	**Rejected**

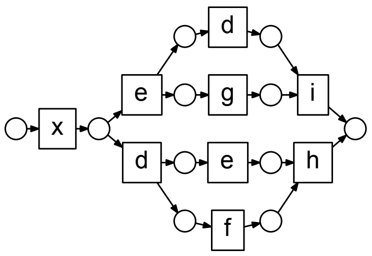
The comparison of Fodina+D, ETM, DGM, TS and ProDiGen+SLAD is summarized in Table 5. After applying the Friedman Aligned Ranks test ProDiGen+SLAD has the best ranking (Fig. 5a). Based on the results of the Friedman Aligned Ranks, we performed a Holms post-hoc test (Fig. 5b), starting with the initial hypothesis that all the tested algorithms are equal to ProDiGen+SLAD. The test rejects the null hypothesis in all the cases for a confidence level of $\alpha = 0.05$ —the p-value of each algorithm has to be lower than α in order to reject the hypothesis. This means that ProDiGen+SLAD outperforms all the other algorithms, and that the difference is statistically significant for that confidence level.

ProDiGen+SLAD obtains the original model in 15 out of 18 logs. The most difficult situations for SLAD are the duplicates in parallel. Nevertheless, as previously said, this problem is more related to the input solution, rather than to the performance of the proposed algorithm (Figs. 4 and 5) as it cannot create new behavior in the mined model. More specifically:

- For log *Alpha* (Fig. 4), although the algorithm correctly detects the tasks *e* and *d* as potential duplicates, due to the original model retrieved by ProDiGen (Fig. 4a), it cannot construct a model with those tasks repeated in the different branches (Fig. 4b). This is because ProDiGen finds a non-free-choice pattern that mimics the same behavior as the model with duplicate tasks disabling the possibility to split the causal dependencies of these activities. In other words, both tasks *d* and *e* are always executed in the same trace. Therefore, ProDiGen mines a parallel construct involving these two tasks. Then, depending on the execution of *g* or *f*, the next task to be executed can be *h* or *i*, respectively.



(a) *Detail of the mined model by ProDiGen without duplicate tasks. SLAD does not improve this part of the model.*

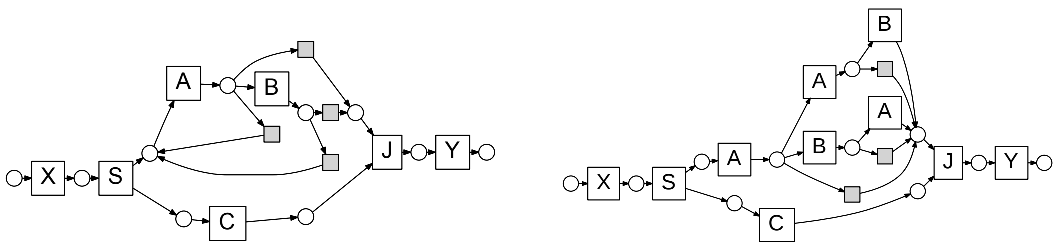


(b) *Detail of the original model with duplicate tasks.*

Figure 4: Two models depicting the same behavior for the log *Alpha*, but with and without duplicate tasks.

- For model *Fig6p39* (Fig. 5) the algorithm cannot correctly obtain the parallelism with the task *A*, which is again executed multiple times in different branches. Based on this, ProDiGen tries to reproduce as much behavior as possible overly connecting the same task *A* in only one branch (Fig. 5a). For this reason, SLAD tries to split the task *A*(Fig. 5b), improving the precision but still allowing for more behavior than the recorded in the log. This is one example of how duplicate activities executed in different branches are isolated when mining the log without considering the duplicity.

One of the objectives of SLAD is to retrieve high values of precision, i.e., to create a more specific process model and, therefore, to reduce the generalization that overly-connected nodes and unnecessary loops can introduce to the process model. As precision and generalization are opposed objectives [5, 25], and SLAD is trying to increase the precision and reduce the generalization, we evaluated the generalization separately to analyse how much this dimension decreases when SLAD introduces duplicate labels. Table 6 presents the values for the generalization —*G*—, for the models retrieved by ProDiGen, HM, and Fodina, before and after SLAD, using the *Alignment Based Probabilistic Generalization* metric [26]. Table 6 also shows the values of generalization for the original model as a baseline. Note that a generalization value closer to 1 means that the process model is too general —a less specific process model— whilst a value closer to 0 means that there is no room for reproducing unseen behavior



(a) *Initial solution retrieved by ProDiGen without duplicate tasks.*

(b) *The same model after the local search.*

Figure 5: Two models from the log *Fig6p39* with and without duplicate tasks.

Table 6: Generalization values for 18 logs before and after SLAD.

		Logs																	
		Alpha	Folded	Loop	Fig6p25	betaSimpl.	flighCar	Fig5p19	Fig5p1AND	Fig5p1OR	Fig6p10	Fig6p31	Fig6p33	Fig6p34	Fig6p38	Fig6p39	Fig6p42	Fig6p9	RelProc
ProDiGen	*G*	0.99991	0.99997	0.99991	0.99945	0.70556	0.51282	0.75333	0.0	0.0	0.9992	0.41667	0.38462	0.99986	0.6375	0.94324	0.96336	0.62308	0.99796
ProDiGen + SLAD	*G*	0.99991	0.99995	0.99991	0.99945	0.38889	0.51282	0.75333	0.0	0.0	0.99887	0.41667	0.38462	0.99982	0.6375	0.9301	0.95229	0.62308	0.99591
HM	*G*	0.99993	0.99997	0.99991	0.9995	0.79722	0.47619	0.75333	.44444	0.33333	0.9992	0.68254	0.69597	0.99984	0.7	0.97596	0.96493	0.725	0.99796
HM + SLAD	*G*	0.99992	0.99995	0.99991	0.99945	0.72639	0.2381	0.75333	0.0	0.0	0.99887	0.41667	0.38462	0.99976	0.5875	0.92241	0.96183	0.725	0.99591
Fodina	*G*	0.99993	0.99997	0.99991	0.9995	0.7911	0.40000	0.5297	0.1363	0.33333	0.9992	0.68254	0.69597	0.99986	0.6375	0.93467	0.97421	0.83334	0.99797
Fodina + SLAD	*G*	0.99991	0.99995	0.99991	0.99945	0.72639	0.40000	0.5297	0.1363	0.0	0.99887	0.41667	0.38462	0.99982	0.6375	0.93467	0.97421	0.83334	0.99797
Fodina + D	*G*	0.99993	0.99997	0.99993	0.99945	0.72639	0.51282	0.85278	0.0	0.0	0.99901	0.41667	0.38462	0.99987	0.6375	0.9783	0.96115	0.62308	0.9976
Fodina + D + SLAD	*G*	0.99993	0.99995	0.99993	0.99945	0.72639	0.51282	0.85278	0.0	0.0	0.99887	0.41667	0.38462	0.99987	0.6375	0.96799	0.96014	0.62308	0.9971
Original Model	*G*	0.99993	0.99995	0.99991	0.99945	0.38889	0.51282	0.75333	0.0	0.0	0.99887	0.41667	0.38462	0.99982	0.6375	0.9246	0.95229	0.62308	0.99591

—a more specific process model. Furthermore, for each pair of algorithms, we have shadowed those cases where the generalization changes after applying SLAD. As can be seen, for ProDiGen the generalization was lower after applying SLAD in 7 out of 18 cases. The cases where SLAD retrieved the same generalization is because ProDiGen, through its hierarchical fitness function —Definition 9—, already tries to reduce this dimension of a process model by focusing in precise process models —generally through the inclusion of non-free-choice constructs. On the other hand, for HM, in 15 out of 18 cases the label splitting process retrieved a more specific process model. With Fodina, SLAD reduced the generalization in 9 of the cases —both Fodina and HM usually introduce a high number of unnecessary loops and overly connected nodes that can be executed any time in the process. Finally, with the Fodina extension to mine duplicate labels —Fodina+D—, SLAD retrieved a more specific process model in 6 out of 18 cases —Fodina+D already mines models with duplicate labels, therefore it retrieves solutions reducing the generalization. It should be noticed that in all the cases SLAD retrieved equal or lower generalization than its respective process model without duplicate labels.

In summary, SLAD was able to detect the duplicate tasks and *significantly* improve the precision in 45 out of 54 solutions —considering the initial solutions of ProDiGen, HM and Fodina— and also the fitness replay in some of the cases. More specifically, after applying SLAD, 35 out of 54 solutions were equal to the original model. Furthermore, the algorithm was able to unfold those situations detected as a loop, i.e., situations with repeated sequences —at most two times— that can be represented with duplicate tasks in order to improve the precision of the model. Additionally, in *none of the cases* the presented approach retrieved the trace-model after the label splitting process, avoiding the overfitted models with one path per trace. Also, none of the resulting models after applying SLAD were worse than its respective initial solution.

4.3.3. Complexity analysis.

With respect to the runtime of SLAD, we can identify two main time consuming parts in the algorithm: *discovering the duplicate tasks* (Alg. 1:4-6), and *evaluating* the solutions (Alg. 1:24). On the one hand, discovering the duplicate tasks is a costly function as, in order to detect the follows relation of each activity, it is necessary to build a dependency graph, which requires one pass through the log and, for each trace, one scan through the trace. This translates to a complexity $O(n)$, where n represents the total number of events in the log. Fortunately, most process discovery algorithms [3, 9, 36, 38] already compute this dependency graph from an event log in order to mine a process model. Therefore SLAD can reuse this information without the need to recompute it, which reduces this process to check only *once* —at the beginning of the algorithm— each of the different activities (T) of the log, i.e $O(T)$. On the other hand, evaluating each solution is the main *bottleneck* of SLAD. Each time SLAD generates a new potential solution after splitting a task, the algorithm needs to evaluate its replay fitness, precision and simplicity. Unfortunately, with current state of the art conformance checking metrics, such as alignments or token replay techniques, this is a very time consuming process when dealing with large event logs. Nevertheless, we can approximately relate the complexity of a *greedy* conformance checking technique to the number (t) and length (l) of the traces in an event log, hence $O(t \cdot l)$. In fact, this is equal to traverse all the events of the log, therefore we can set $O(t \cdot l)$ to $O(n)$. Regardless the complexity of the conformance checking metric, SLAD has to check and split each one of the activities detected as a potential duplicate. Thereby, the complexity of this part can be set as $O(d \cdot c)$ where d is the total number of potential duplicate activities and c is the number of times the activities can be split. In summary, the complexity of SLAD is $O(d \cdot c \cdot n)$. This does not include the complexity of building the initial dependency graph.

Regarding the logs used in the experimentation, the runtime of SLAD was, on average, 9 seconds —considering the time needed to compute the dependency graphs. In particular, the highest time was achieved when applying SLAD over the solution retrieved by ProDiGen with the log *Fig6p42*, that took 12 seconds.

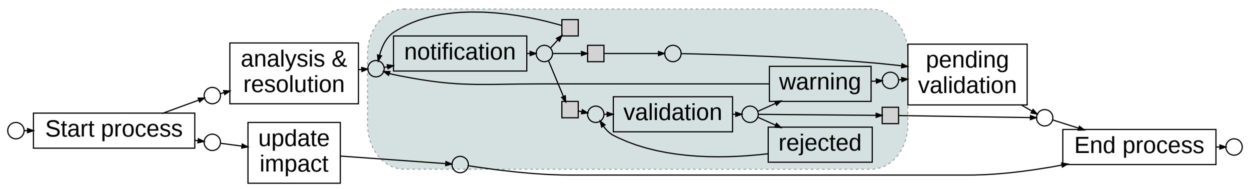
4.3.4. Experiments in a real-life scenario.

Finally, we have tested ProDiGen+SLAD in a real-life scenario: a process model within an IT Service Management platform [35]. This process is related to handling incidents and requests, henceforth tickets, in a *service desk*, i.e., a central point of communications between users and staff in an organization. The process model has 7 different activities. However one of these activities, *notification*, is executed multiple times during the process. In particular, albeit its purpose is to notify different involved staff during the process of handling a ticket, it is always recorded with the same label. Hence, using this process model we generated an even log containing 300 traces to check if it is possible to obtain again the original process.

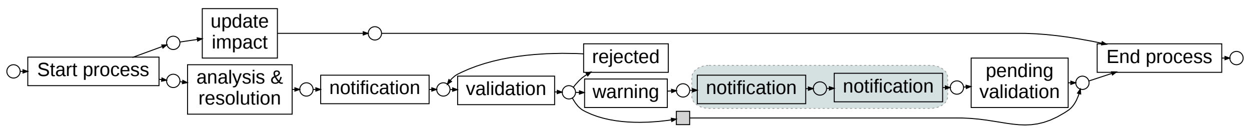
Figure 6a shows the process model discovered with ProDiGen, and Figure 6b displays the same process model after applying SLAD. In this example, both process models have a perfect replay fitness. However, the precision of the model without duplicate labels is lower —0.58— than the precision of the process model after applying SLAD —0.87. The low precision of the model without duplicate labels (Fig. 6a) is due the overly-connected activity *notification*, that enables to repeat the shadowed part of the model without any limit. On the other hand, the model retrieved after applying SLAD (Fig. 6b) removes this loop by duplicating *notification* three times. Furthermore, the only difference between the designed process model, and the one retrieved by SLAD, is the shadowed part in Fig. 6b. In reality, there are always two notifications that are performed in parallel: one of the notifications is for the *Incidents manager* and the other one is for the *Senior manager*. This two activities always happen as a sequence sharing the same label. As we do not have any other information regarding this situation, it is impossible to distinguish them, thus its representation as a sequence.

5. Conclusions

We have presented SLAD, an algorithm to tackle duplicate tasks in an already discovered model. Our proposal takes as starting point a model without duplicate tasks and its respective log and, based on heuristics, the local information of the log, and the causal dependencies of the input mined model, it improves the fitness replay, precision and simplicity of the model. SLAD has been validated with 18 different logs and 54 different initial solutions from three different process mining algorithms. Results show that the algorithm was able to enhance the initial solutions in 45 of the 54 tested scenarios. Moreover, we have compared SLAD with the state of the art process discovery algorithms with duplicate tasks. Statistical test have shown that the best SLAD algorithm (ProDiGen+SLAD) is better, and that



(a) *Initial solution retrieved by ProDiGen without duplicate tasks.*



(b) *The same model after applying SLAD.*

Figure 6: Process models mined for a real event log before and after SLAD.

the differences are statistically significant. As a future work, and based on the obtained results, we want to extend SLAD with the possibility to not only redistribute the already mined dependencies of the model, but introduce new relations based on the combinations of the potential duplicate activities, in order to enhance those models with a lower fitness.

Acknowledgments

This work was supported by the Spanish Ministry of Economy and Competitiveness under project TIN2014-56633-C3-1-R, and the Galician Ministry of Education under the projects EM2014/012 and CN2012/151.

References

[1] Adriansyah, A., Munoz-Gama, J., Carmona, J., van Dongen, B. F., van der Aalst, W. M. P., 2012. Alignment based precision checking. In: Rosa, M. L., Soffer, P. (Eds.), Proceedings of the 10th International Workshops on Business Process Management, BPM. Vol. 132 of Lecture Notes in Business Information Processing. Springer, Tallinn, Estonia, pp. 137–149.

[2] Badouel, E., Darondeau, P., 1998. Theory of regions. In: Reisig, W., Rozenberg, G. (Eds.), Lectures on Petri Nets I: Basic Models. Vol. 1491 of Lecture Notes in Computer Science. Springer, Dagstuhl, pp. 529–586.

[3] Broucke, S. K. L. M. v., 2014. Advances in process mining: Artificial negative events and other techniques. Ph.D. thesis, Katholieke Universiteit Leuven.

[4] Broucke, S. K. L. M. v., Weerdt, J. D., Vanthienen, J., Baesens, B., 2013. A comprehensive benchmarking framework (CoBeFra) for conformance analysis between procedural process models and event logs in ProM. In: Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, CIDM. IEEE, Singapore, Singapore, pp. 254–261.

[5] Buijs, J., van Dongen, B., van der Aalst, W., 2012. On the role of fitness, precision, generalization and simplicity in process discovery. In: Meersman, R., Panetto, H., Dillon, T., Rinderle-Ma, S., Dadam, P., Zhou, X., Pearson, S., Ferscha, A., Bergamaschi, S., Cruz, I. (Eds.), OTM Federated Conferences, Proceedings of the 20th International Conference on Cooperative Information systems, CoopIS. Vol. 7565 of Lecture Notes in Computer Science. Springer, Rome, Italy, pp. 305–322.

[6] Buijs, J. C. A. M., van Dongen, B. F., van der Aalst, W. M. P., 2014. Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity. International Journal of Cooperative Information Systems 23 (1), 1–39.

[7] Carmona, J., 2012. The label splitting problem. Transactions on Petri Nets and Other Models of Concurrency 6, 1–23.

[8] Carmona, J., Cortadella, J., Kishinevsky, M., 2008. A region-based algorithm for discovering petri nets from event logs. In: Dumas, M., Reichert, M., Shan, M. C. (Eds.), Proceedings of the 6th International Conference on Business Process Management, BPM. Vol. 5240 of Lecture Notes in Computer Science. Springer, Milan, Italy, pp. 358–373.

[9] de Medeiros, A. K. A., 2006. Genetic process mining. Ph.D. thesis, Technische Universiteit Eindhoven.

[10] de Medeiros, A. K. A., Guzzo, A., Greco, G., van der Aalst, W. M. P., Weijters, A. J. M. M., van Dongen, B. F., Saccà, D., 2007. Process mining based on clustering: A quest for precision. In: Hofstede, A. H. M. T., Benatallah, B., Paik, H. Y. (Eds.), Proceedings of the 5th International Workshops on Business Process Management, BPM. Vol. 4928 of Lecture Notes in Computer Science. Springer, Brisbane, Australia, pp. 17–29.

[11] de Medeiros, A. K. A., Weijters, A. J. M. M., van der Aalst, W. M. P., 2007. Genetic process mining: an experimental evaluation. Data Mining and Knowledge Discovery 14 (2), 245–304.

[12] Deb, K., Pratap, A., Agarwal, S., Meyarivan, T. A. M. T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE transactions on Evolutionary Computation 6 (2), 182–197.

[13] Goedertier, S., Martens, D., Vanthienen, J., Baesens, B., 2009. Robust process discovery with artificial negative events. The Journal of Machine Learning Research 10, 1305–1340.

[14] Herbst, J., 2000. A machine learning approach to workflow management. In: de Mántaras, R. L., Plaza, E. (Eds.), Proceedings of the 11th European Conference on Machine Learning, ECML. Vol. 1810 of Lecture Notes in Computer Science. Springer, Barcelona, Spain, pp. 183–194.

[15] Herbst, J., Karagiannis, D., 2004. Workflow mining with inwolve. Computers in Industry 53 (3), 245–264.

[16] Hodges, J. L., Lehmann, E. L., 1962. Rank methods for combination of independent experiments in analysis of variance. The Annals of Mathematical Statistics 33 (2), 482–497.

[17] Holm, S., 1979. A simple sequentially rejective multiple test procedure. Scandinavian journal of statistics, 65–70.

[18] Leemans, S. J. J., Fahland, D., van der Aalst, W. M. P., 2013. Discovering block-structured process models from event logs containing infrequent behaviour. In: Lohmann, N., Song, M., Wohed, P. (Eds.), Proceedings of the 12th International Workshops on Business Process Management, BPM. Vol. 171 of Lecture Notes in Business Information Processing. Springer, Beijing, China, pp. 66–78.

[19] Li, J., Liu, D., Yang, B., 2007. Process mining: Extending α-algorithm to mine duplicate tasks in process logs. In: Chang, K. C. C., Wang, W., Chen, L., Ellis, C. A., Hsu, C. H., Tsoi, A. C., Wang, H. (Eds.), Advances in Web and Network Technologies, and Information Management. Vol. 4537 of Lecture Notes in Computer Science. Springer, Huang Shan, China, pp. 396–407.

[20] Mans, R., van der Aalst, W., Vanwersch, R., 2015. Process Mining in Healthcare: Evaluating and Exploiting Operational Healthcare Processes, 1st Edition. Springer.

[21] Rodríguez-Fdez, I., Canosa, A., Mucientes, M., Bugarín, A., 2015. STAC: A web platform for the comparison of algorithms using statistical tests. In: Yazici, A., Pal, N. R., Kaymak, U., Martin, T., Ishibuchi, H., Lin, C., Sousa, J. M. C., Tütmez, B. (Eds.), Proceedings of the IEEE International Conference on Fuzzy Systems, FUZZ-IEEE. IEEE, Istanbul, Turkey, pp. 1–8.

[22] Rozinat, A., van der Aalst, W. M. P., 2008. Conformance checking of processes based on monitoring real behavior. Information Systems 33 (1), 64–95.

[23] Sánchez-González, L., García, F., Mendling, J., Ruiz, F., Piattini, M., 2010. Prediction of business process model quality based on structural metrics. In: Parsons, J., Saeki, M., Shoval, P., Woo, C. C., Wand, Y. (Eds.), Proceedings of the 29th International Conference on Conceptual Modeling - ER 2010. Vol. 6412 of Lecture Notes in Computer Science. Springer, pp. 458–463.

[24] Solé, M., Carmona, J., 2010. Process mining from a basis of state regions. In: Lilius, J., Penczek, W. (Eds.), Proceedings of the 31st International Conference on Applications and Theory of Petri Nets, PETRI NETS. Vol. 6128 of Lecture Notes in Computer Science. Springer, Braga, Portugal, pp. 226–245.

[25] van der Aalst, W., 2016. Process Mining: Data Science in Action, 2nd Edition. Springer Publishing Company, Incorporated.

[26] van der Aalst, W., Adriansyah, A., van Dongen, B., 2012. Replaying history on process models for conformance checking and performance analysis. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 2 (2), 182–192.

[27] van der Aalst, W. M. P., Adriansyah, A., van Dongen, B. F., 2011. Causal nets: a modeling language tailored towards process discovery. In: Katoen, J. P., König, B. (Eds.), Proceedings of the 22nd International Conference on Concurrency Theory CONCUR. Vol. 6901 of Lecture Notes in Computer Science. Springer, Aachen, Germany, pp. 28–42.

[28] van der Aalst, W. M. P., Ter Hofstede, A. H. M., Kiepuszewski, B., Barros, A. P., 2003. Workflow patterns. Distributed and parallel databases 14 (1), 5–51.

[29] van der Aalst, W. M. P., van Dongen, B. F., Herbst, J., Maruster, L., Schimm, G., Weijters, A. J. M. M., 2003. Workflow mining: a survey of issues and approaches. Data & Knowledge Engineering 47 (2), 237–267.

[30] van der Aalst, W. M. P., Weijters, A. J. M. M., 2004. Process mining: a research agenda. Computers in Industry 53 (3), 231–244.

[31] van der Aalst, W. M. P., Weijters, A. J. M. M., Maruster, L., 2004. Workflow mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering 16 (9), 1128–1142.

[32] van der Werf, J. M. E. M., van Dongen, B. F., Hurkens, C. A. J., Serebrenik, A., 2009. Process discovery using integer linear programming. Fundamenta Informaticae 94 (3-4), 387–412.

[33] van Dongen, B. F., de Medeiros, A. K. A., Verbeek, H. M. W., Weijters, van der Aalst, W. M. P., 2005. The ProM framework: A new era in process mining tool support. In: Applications and Theory of Petri Nets 2005. Springer, pp. 444–454.

[34] van Dongen, B. F., de Medeiros, A. K. A., Wen, L., 2009. Process mining: Overview and outlook of petrinet discovery algorithms. In: Jensen, K., van der Aalst, W. M. P. (Eds.), Transactions on Petri Nets and Other Models of Concurrency II. Vol. 5460 of Lecture Notes in Computer Science. Springer, pp. 225–242.

[35] Vázquez-Barreiros, B., Chapela, D., Mucientes, M., Lama, M., 2016. Process Mining in IT Service Management: A Case Study. In: van der Aalst, W., Bergenthum, R., J.Carmona (Eds.), Proceedings of the 2016 International Workshop on Algorithms & Theories for the Analysis of Event Data, ATAED. Toruń, Poland, pp. 16–30.

[36] Vázquez-Barreiros, B., Mucientes, M., Lama, M., 2015. ProDiGen: Mining complete, precise and minimal structure process models with a genetic algorithm. Information Sciences 294, 315–333.

[37] Weijters, A. J. M. M., Ribeiro, J. T. S., 2010. Flexible heuristics miner (FHM). BETA Working Paper Series WP 334, Eindhoven University of Technology.

[38] Weijters, A. J. M. M., van der Aalst, W. M. P., de Medeiros, A. K. A., 2006. Process mining with the heuristics miner-algorithm. BETA Working Paper Series WP 166, Eindhoven University of Technology.

[39] Wen, L., van der Aalst, W. M. P., Wang, J., Sun, J., 2007. Mining process models with non-free-choice constructs. Data Mining and Knowledge Discovery 15 (2), 145–180.

[40] Wen, L., Wang, J., van der Aalst, W. M. P., Huang, B., Sun, J., 2010. Mining process models with prime invisible tasks. Data & Knowledge Engineering 69 (10), 999–1021.

[41] Wilcoxon, F., 1945. Individual comparisons by ranking methods. Biometrics Bulletin 1, 80–83.