Revisiting the Alpha Algorithm To Enable Real-Life Process Discovery Applications

Aaron Küsters[1], Wil M.P. van der Aalst[1]

[1]*Chair of Process and Data Science (PADS), RWTH Aachen University, Germany*

Abstract
The Alpha algorithm was the first process discovery algorithm that was able to discover process models with concurrency based on incomplete event data while still providing formal guarantees. However, as was stated in the original paper, practical applicability is limited when dealing with exceptional behavior and processes that cannot be described as a structured workflow net without short loops. This paper presents the Alpha+++ algorithm that overcomes many of these limitations, making the algorithm competitive with more recent process mining approaches. The different steps provide insights into the practical challenges of learning process models with concurrency, choices, sequences, loops, and skipping from event data. The approach was implemented in ProM and tested on various publicly available, real-life event logs.

Keywords
Process Discovery, Process Mining, Process Models, Petri Nets

1. Introduction

The original *Alpha algorithm* was developed over twenty years ago [1, 2]. The goal of the algorithm was to show the challenges related to discovering process models with concurrency from example traces. It was formally proven that, a process modeled as a structured workflow net without short loops, can be rediscovered from an event log that is directly-follows complete [2]. Despite this remarkable theoretical result, the Alpha algorithm has limited practical relevance for two main reasons:

- The original algorithm did not attempt to filter out infrequent behavior. Since exceptional behavior is not separated from frequent behavior, it is generally impossible to uncover structure from real-life event logs.
- The original algorithm assumed that the process can be modeled as a free-choice Petri net with unique visible activity labels. Most real-life processes can *not* be modeled as a structured workflow net without short loops and unique visible labels.

These limitations were already acknowledged in the papers proposing the algorithm, e.g., the focus of [2] was on showing the theoretical limits of process discovery based on directly-follows complete event logs. Many of the later process discovery approaches use these insights. Various

Algorithms and Theories for the Analysis of Event Data (ATAED'23), 2023, Portugal
kuesters@pads.rwth-aachen.de (A. Küsters); wvdaalst@pads.rwth-aachen.de (W. M.P. van der Aalst)
0009-0006-9195-5380 (A. Küsters); 0000-0002-0955-6940 (W. M.P. van der Aalst)

extensions of the Alpha algorithm have been proposed, e.g., [3] extends the core algorithm to deal with long-term dependencies, and [4] extends the core algorithm to deal with invisible activities (e.g., skipping). Region-based process-discovery approaches provide formal guarantees. State-based regions were introduced by Ehrenfeucht and Rozenberg [5] in 1989 and generalized by Cortadella et al. [6]. In [7], it is shown how these state-based regions can be applied to process mining by first creating a log-based automaton using different abstractions. In [8, 9], refinements are proposed to tailor state-based regions toward process discovery. Language-based regions work directly on traces without creating an automaton first; see, for example, the approaches presented in [10, 11, 12].

Variants of the Alpha algorithm and the region-based approaches have problems dealing with infrequent behavior and are rarely used in practice. The region-based approaches are also infeasible for larger models and logs. Approaches such as the eST-Miner [13] and the different variants of the inductive miner [14, 15] aim to provide formal guarantees but can also handle infrequent behavior. Variants of the inductive miner have also been implemented in various commercial systems (e.g., Celonis). The so-called split-miner uses a combination of approaches to balance recall and precision [16].

The goal of this paper is to go back to the original ideas used by the Alpha algorithm and make the algorithm work in practical settings. The result is the *Alpha+++ algorithm*, which, not only extends the core algorithm, but also removes problematic noisy activities, adds invisible activities, repairs loops, and post-processes the resulting Petri net. The approach uses a broad combination of novel ideas, making the Alpha algorithm competitive when compared with the state-of-the-art. The ideas incorporated in the Alpha+++ algorithm may also be used in combination with other approaches (e.g., identifying problematic activities and introducing artificially created invisible activities).

The remainder of this paper is organized as follows. Section 2 introduces event logs, directly-follows graphs, and the original Alpha algorithm. Section 3 describes the Alpha+++ algorithm. The algorithm has been implemented in ProM (cf. Section 4) and evaluated using various event logs (cf. Section 5). Section 6 concludes the paper.

2. Preliminaries

2.1. Event Logs

Process mining starts from event data. An event may have many different attributes. However, here we focus on discovering the control flow and assume that each *event* has a *case* attribute, an *activity* attribute, and a *timestamp* attribute. We only use the timestamps to order events related to the same case. Therefore, each case can be described as a sequence of activities, also called *trace*. An *event log* is a multiset of traces, as different cases can exhibit the same trace.

Definition 1 (Event Log). *$\mathcal{U}_{act}$ is the universe of activity names. A trace $\sigma = \langle a_1, a_2, \dots, a_n \rangle \in {\mathcal{U}_{act}}^*$ is a sequence of activities. An event log $L \in \mathcal{B}({\mathcal{U}_{act}}^*)$ is a multiset of traces.*

For example, $L_1 = [\langle a, b, c, d \rangle^{400}, \langle a, b, d \rangle^{250}, \langle d, a, b, c \rangle^{4}, \langle d, a, b \rangle^{2}]$ is an event log containing 656 cases with 4 different variants. Variant $\langle a, b, c, d \rangle$ is the most frequent one, i.e., $L_1(\langle a, b, c, d \rangle) = 400$.

We write $\mathit{actMult}(L) = [\,\sigma(i) \mid \sigma \in L \wedge 1 \leq i \leq |\sigma|\,]$ for the multiset of activities in an event log L and $act(L) = \{a \mid a \in \mathit{actMult}(L)\}$ for the set of activities.

2.2. Directly-Follows Graphs

A *Directly-Follows Graph* (DFG) is a graph showing how often one activity is followed by another. A DFG consists of the activities as nodes and has an arc from an activity $a \in \mathcal{U}_{act}$ to an activity $b \in \mathcal{U}_{act}$ if a is directly followed by b. Two special nodes, corresponding to a start and an end node, are added additionally.

Definition 2 (Directly-Follows Graph). *A Directly-Follows Graph (DFG) is a pair $G = (A, \overset{G}{\Rightarrow})$, where $A \subseteq \mathcal{U}_{act}$ is a set of activities and $\overset{G}{\Rightarrow} \in \mathcal{B}((A \times A) \cup (\{\blacktriangleright\} \times A) \cup (A \times \{\blacksquare\}) \cup (\{\blacktriangleright\} \times \{\blacksquare\}))$ is a multiset of arcs. $\blacktriangleright$ is the start node and $\blacksquare$ is the end node.*

Note that a DFG has arc weights. Hence, $\overset{G}{\Rightarrow}$ is a multiset, where $\overset{G}{\Rightarrow}(a, b)$ denotes how often a is followed by b. We write $a \overset{G}{\Rightarrow} b$ if and only if $\overset{G}{\Rightarrow}(a, b) > 0$ holds. Similarly, we say that $a \underset{\geq t}{\overset{G}{\Rightarrow}} b$ holds if and only if $\overset{G}{\Rightarrow}(a, b) \geq t$.

The construction of a DFG from an event log is straightforward.

Definition 3 (Constructing DFGs from Event Logs). *Let $L \in \mathcal{B}({\mathcal{U}_{act}}^*)$ be an event log. We can construct a DFG $disc_{dfg}(L) = (A, \overset{L}{\Rightarrow})$ based on the directly-follows relations of event log L, with the set of activities $A = \{a \in \sigma \mid \sigma \in L\}$ and the multiset of arcs $\overset{L}{\Rightarrow} = [(\sigma_i, \sigma_{i+1}) \mid \sigma \in L' \wedge 1 \leq i < |\sigma|]$, where $L' = [\langle\blacktriangleright\rangle \cdot \sigma \cdot \langle\blacksquare\rangle \mid \sigma \in L]$ denotes the event log where artificial start and end activities have been added.*

Given an event log L, we can construct a DFG $disc_{dfg}(L) = (A, \overset{L}{\Rightarrow})$, and in the context of L refer to the directly-follows relations in L represented by $\overset{L}{\Rightarrow}$ directly.

2.3. Petri Nets

We would like to discover process models which can represent more complex control-flow structures, like choices, loops, and concurrency. Therefore, we use *labeled* Petri nets as a target format for process discovery. The reader is assumed to be familiar with the Petri net basics.

Definition 4 (Labeled Petri Net). *A labeled Petri net is a tuple $N = (P, T, F, l)$ with a set of places P, a set of transitions T (where $T \cap P = \emptyset$), a flow relation $F \subseteq (P \times T) \cup (T \times P)$, and a labeling function $l \in T \not\rightarrow \mathcal{U}_{act}$. We write $l(t) = \tau$ if $t \in T \setminus dom(l)$ (i.e., t is a silent transition that cannot be observed).*

A marking is represented by a multiset of places $M \in \mathcal{B}(P)$. For a node $x \in P \cup T$, we define the preset of x as $\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$ and the postset of x as $x\bullet = \{y \in P \cup T \mid (x, y) \in F\}$. We focus on so-called *accepting* Petri nets, i.e., Petri nets with a defined initial and final state.

Definition 5 (Accepting Petri Net). *An accepting Petri net is a triplet AN* $= (N, M_{\mathit{init}}, M_{\mathit{final}})$ *where* $N = (P, T, F, l)$ *is a labeled Petri net,* $M_{\mathit{init}} \in \mathcal{B}(P)$ *is the initial marking, and* $M_{\mathit{final}} \in \mathcal{B}(P)$ *is the final marking.* $\mathcal{U}_{AN}$ *is the set of accepting Petri nets.*

The language defined by an accepting Petri net is then simply given by the set of traces corresponding to all firing sequences that start in the initial marking M_{init} and end in the final marking M_{final}. A firing sequence leading from M_{init} to M_{final} is converted into a trace, i.e., a sequence of activities. Note that transitions that fire are mapped onto the corresponding activities. If a transition t is silent (i.e., $l(t) = \tau$), no corresponding activity is created when firing t. Hence, the language of an accepting Petri net is a set of traces.

2.4. Alpha Algorithm

A *process discovery algorithm* aims to discover a model from event data such that the language of the model best characterizes the example behavior seen in the event log.

Definition 6 (Process Discovery Algorithm). *A process discovery algorithm is a function* $\mathit{disc} \in \mathcal{B}(\mathcal{U}_{\mathit{act}}{}^*) \to \mathcal{U}_{AN}$*, i.e., based on a multiset of traces, an accepting Petri net is discovered.*

The classical Alpha process discovery algorithm was introduced in [2]. To be able to better explain the extensions presented in this paper, we split the description into three main parts. From an input event log L, place candidates are constructed based on the directly-follows relations of the log. The resulting set of place candidates is pruned to remove dominated candidates. Finally, the discovered Petri net is constructed.

Candidate Building

$$\begin{aligned} \mathit{Cnd} = \{(A, B) \mid \emptyset \subsetneq A, B \subseteq \mathit{act}(L) \wedge \forall_{a \in A} \forall_{b \in B} (a \overset{L}{\Rightarrow} b) \\ \wedge \forall_{a, a' \in A} (a \overset{L}{\not\Rightarrow} a') \wedge \forall_{b, b' \in B} (b \overset{L}{\not\Rightarrow} b')\} \end{aligned}$$

Candidate Pruning

$$\begin{aligned} \mathit{Sel} = \{(A_1, A_2) \in \mathit{Cnd} \mid \forall_{(A_1', A_2') \in \mathit{Cnd}} ((A_1 \subseteq A_1' \wedge A_2 \subseteq A_2') \\ \Rightarrow (A_1, A_2) = (A_1', A_2'))\} \end{aligned}$$

Petri Net Construction Let $PN = ((P, T, F, l), M_{init}, M_{final})$, where:

- $P = \{p_{(A,B)} \mid (A, B) \in \mathit{Sel}\} \cup \{i_W, o_W\}$
- $T = \{t_a \mid a \in \mathit{act}(L)\}$
- $F = \{(t_a, p_{(A,B)}) \mid (A, B) \in \mathit{Sel} \wedge a \in A\} \cup \{(p_{(A,B)}, t_b) \mid (A, B) \in \mathit{Sel} \wedge b \in B\} \cup \{(i_W, t_s) \mid \exists_\sigma \langle s \rangle \cdot \sigma \in L\} \cup \{(t_e, o_W) \mid \exists_\sigma \sigma \cdot \langle e \rangle \in L\}$
- $l = \{(t_a, a) \mid a \in \mathit{act}(L)\}$
- $M_{init} = [i_W]$
- $M_{final} = [o_W]$

3. Alpha+++

In this section, we introduce the Alpha+++ process discovery algorithm based on the classical Alpha algorithm. Through certain pre-processing steps on the event log and a corresponding DFG, as well as fitness-based place filtering, this algorithm is especially well suited for real-life event logs.

The input for this process discovery algorithm is an event log L. In particular, only ordered traces of activities with corresponding frequencies are required. For the main steps of the algorithm, a DFG based on the event log L is used exclusively. Traces of the event log are only used for replay to remove unfitting place candidates. For simplicity, we assume that the traces of L already include artificial start and end activities, in particular, we assume $\{\blacktriangleright, \blacksquare\} \subseteq act(L)$.

We introduce the steps of the algorithm in the following order:

1. *Determine Activities*, where the set of activities used throughout the algorithm is determined. Problematic activities are removed from the event log and artificial activities are added, resulting in a repaired event log $\hat{L}$.

2. *Create an Advising DFG*, where an *advising DFG* is constructed based on the DFG corresponding to the repaired log $\hat{L}$, retaining only some of the original DFG edges.

3. *Candidate Building*, where a set of place candidates is built based on the directly-follows relation of the activities.

4. *Candidate Pruning*, where through efficient multistep filtering unfit or undesirable place candidates are discarded.

5. *Petri Net Construction*, where a Petri net is constructed based on the activities of the event log, the added artificial activities and the remaining place candidates.

6. *Post-Processing Petri Net*, where the repaired event log is replayed on the Petri net to remove problematic places.

3.1. Determine Activities

First, we determine the set of activities used in the later steps. Initially, starting with the set of activities occurring in the log, we first remove problematic activities that can cause issues with discovering place candidates later on. Next, we also add artificial activities to allow discovery of place candidates for certain loop and skip constructs.

3.1.1. Removing Problematic Activities

Problematic activities can significantly alter the directly-follows relations of an event log, which are used in the later steps to identify place candidates. In the most extreme case, if a problematic activity randomly occurs between any other two activities in all traces, all the directly-follows information between two other activities would be lost.

We select a subset $\mathcal{A}_L \subseteq act(L)$ of activities to keep and remove the other problematic activities $act(L) \setminus \mathcal{A}_L$. There are many possible approaches to identifying problematic activities,

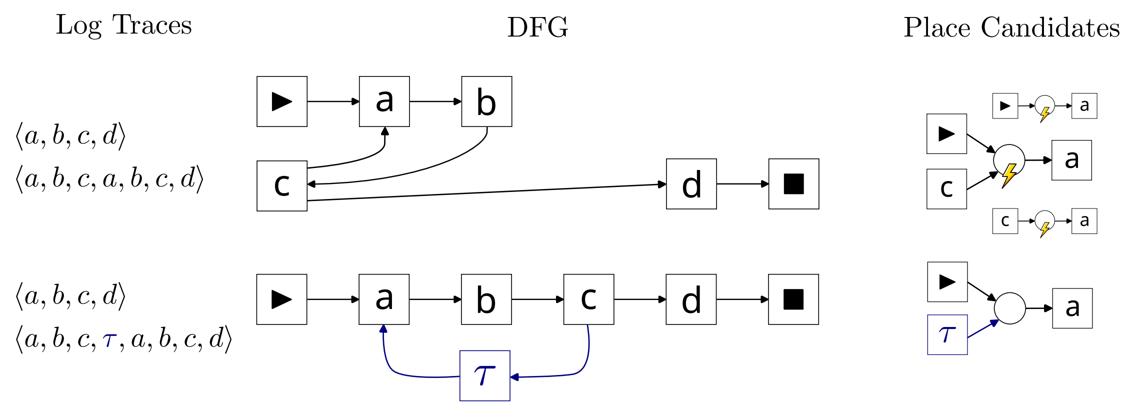

Figure 1: Two event logs (traces shown on the left) and their DFGs. In the first event log ($L_{\circlearrowleft}$) the directly-follows relation between ▶ and a is the same as between c and a. This causes issues, as the corresponding place candidates all have very low fitness. The added artificial activity τ inserted between the looped sequence $\langle a, b, c\rangle$ solves this problem, as the problematic directly-follows relation between c and a is replaced.

such as calculating a problem-score per activity and considering all values above a certain threshold as problematic. For instance, for a very simple problem-score, the fraction of directly-follows relation involving an activity which are parallel, i.e., also occur in the opposite direction, could be considered. This would, for example, allow to correctly identify the problem in the aforementioned extreme case.

3.1.2. Adding Artificial Activities

Discovering Petri net constructs involving silent transitions is a non-trivial task for a DFG-based algorithm. Additionally, in later steps, we want to use traces from the log to assess the fitness of place candidates. Silent activities make calculating fitness scores significantly harder, as then token-based replay is no longer sufficient and computationally expensive alignments have to be computed. As a solution, we propose adding *artificial activities* to traces. They are not part of the activity set of the event log and are only used to find and evaluate place candidates. In the final discovered Petri net, these artificial activities are then translated as silent transitions. This allows discovering Petri nets with silent transitions, while still retaining the advantages of token-based replay fitness evaluation during the algorithm steps. We add artificial activities for two types of constructs: *Loops* and *Skips*.

Adding artificial activities for loops is necessary, as the directly-follows relation between an end activity and a start activity of a loop can cause the discovery of problematic places. For example, consider the event log $L_{\circlearrowleft} = [\langle a, b, c, d\rangle, \langle a, b, c, a, b, c, d\rangle]$. Clearly, this event log can be nicely expressed by a Petri net containing a loop construct, which allows repeating the activities a, b, c. However, the directly-follows relation $c \xRightarrow{L_{\circlearrowleft}} a$ prevents discovering this loop accurately, as shown in Figure 1.

We detect loop constructs based on the directly-follows relations of the input event log L. For a given threshold $d \in \mathbb{R}^+$, we can define the set of detected loops:

Definition 7 (Detected Loops). *Let loops be the function that maps an event log to the set of detected loop start and end activities.*

$$\begin{aligned} \textit{loops}(L) = \{(b,a) \in \textit{act}(L) \times \textit{act}(L) \mid \; & \exists_{(x_1,\dots,x_k)\in act(L)^*, i\in\{1,\dots,k\}}(x_i = a \\ & \wedge \forall_{i\in\{1,\dots,k-1\}}(x_i \xRightarrow[\geq d]{L} x_{i+1}) \\ & \wedge x_k \xRightarrow[\geq d]{L} b \wedge b \xRightarrow[\geq d]{L} a)\} \end{aligned}$$

The parameter d determines the minimal DFG edge weight to consider when looking for loops. For example, with a threshold d=1 and the event log $L_{\circlearrowleft}$, we can calculate $\textit{loops}(L_{\circlearrowleft}) = \{(c,a)\}$. As loop constructs can make a process model very imprecise, we do not want to falsely detect loop behavior from rather infrequent directly-follows relations. For convenience, we can also consider threshold values d relative to the mean directly-follows weight.

For each detected loop endpoint pair $(b,a) \in \textit{loops}(L)$, we want to add an artificial activity $\text{loop}_{b,a} \notin act(L)$. We write $\mathcal{A}_{loop} = \{\text{loop}_{b,a} \mid (b,a) \in \textit{loops}(L)\}$ to denote the set of added artificial loop activities. Additionally, we define a transformation function which transforms a trace $\sigma \in L$ to a trace $\sigma' \in (\mathcal{A}_L \cup \mathcal{A}_{loop})^*$.

Definition 8 (Loop Repair Function). *Let $\textit{repair}_{\circlearrowleft}$ be the function that transforms a trace σ into a repaired trace with added artificial loop activities.*

$$\textit{repair}_{\circlearrowleft}(\sigma, L) = \begin{cases} \langle b, \text{loop}_{b,a}, a\rangle \cdot \textit{repair}_{\circlearrowleft}(\sigma', L) & \text{if } \exists_{(b,a)\in \textit{loops}(L)} \sigma = \langle b,a\rangle \cdot \sigma' \\ \langle\rangle & \text{if } \sigma = \langle\rangle \\ \langle x\rangle \cdot \textit{repair}_{\circlearrowleft}(\sigma', L) & \text{otherwise, with } \sigma = \langle x\rangle \cdot \sigma' \end{cases}$$

This function will be later used to transform the input event log L into a repaired event log, in which artificial activities have been added to relevant traces.

Next, we describe how artificial activities can assist in correctly discovering activity *Skips*, as shown in Figure 2.

For a directly-follows-weight threshold $d \in \mathbb{R}^+$, the detected skips for event log L are defined by the following function, which provides the set of activities that have been detected as being "skippable" after an activity $a \in (\mathcal{A}_L \cup \mathcal{A}_{loop})$.

$$\begin{aligned} \textit{skips}(a, L) = \{b \in \textit{act}(L) \mid \; & a \overset{L}{\Rightarrow} b \wedge a \overset{L}{\not\Rightarrow} a \wedge b \underset{\geq d}{\overset{L}{\not\Rightarrow}} a \wedge b \underset{\geq d}{\overset{L}{\not\Rightarrow}} b \wedge a, b \notin \{\blacktriangleright, \blacksquare\} \\ & \wedge \emptyset \subsetneq \{x \in \textit{act}(L) \mid b \xRightarrow[\geq d]{L} x\} \subseteq \{x \in \textit{act}(L) \mid a \xRightarrow[\geq d]{L} x\}\} \end{aligned}$$

If $B \in \textit{skips}(a, L)$ we assume that all activities $b \in B$ are optional steps after a. To allow appropriate model discovery in the rest of the algorithm, the log is repaired using a new artificial activity $skip_{a,B} \notin act(L)$. The set of all artificial skip activities is denoted by $\mathcal{A}_{skip}$. This artificial skip activity is inserted everywhere in a trace σ of L, where activity a is not directly followed by an activity $b \in B$ in σ (i.e., b was skipped). For that, we define the following

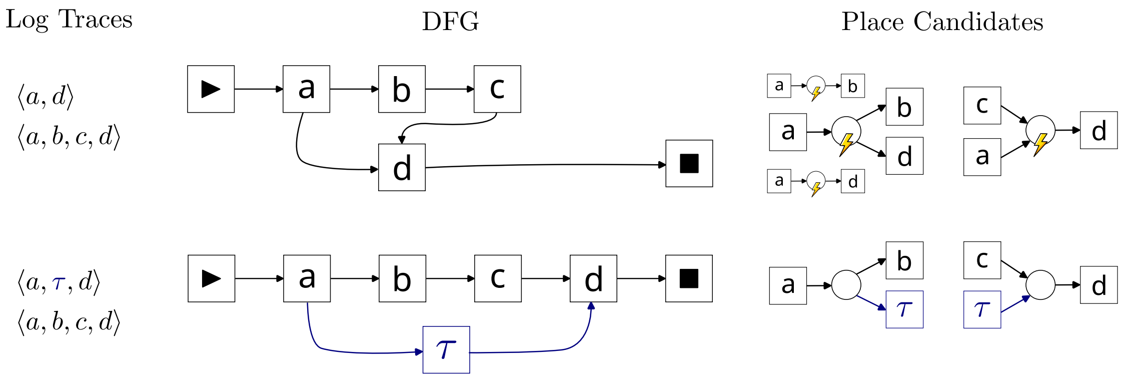

Figure 2: Two event logs and their DFGs showcasing the motivation for repairing implicit skips. The directly-follows relation between a and d would suggest considering place candidates with poor fitness. The second log, where an artificial activity τ is inserted where b and c are skipped, mitigates this problem by replacing the directly-follows relation between a and d.

transformation function:

$$\mathit{repair}_{\tau}(\sigma, L) = \begin{cases} \langle\rangle & \text{if } \sigma = \langle\rangle \\ \langle x\rangle \cdot \mathit{repair}_{\tau}(\sigma', L) & \text{if } \sigma = \langle x\rangle \cdot \sigma' \wedge \mathit{skips}(x, L) = \emptyset \\ \langle a, \mathrm{skip}_{\mathrm{a,B}}\rangle \cdot \mathit{repair}_{\tau}(\langle x\rangle \cdot \sigma', L) & \text{if } \sigma = \langle a, x\rangle \cdot \sigma' \wedge x \notin \mathit{skips}(a, L) \\ \langle a, x\rangle \cdot \mathit{repair}_{\tau}(\sigma', L) & \text{if } \sigma = \langle a, x\rangle \cdot \sigma' \wedge x \in \mathit{skips}(a, L) \end{cases}$$

We can now construct a *repaired* event log $\hat{L}$ from the input event log L based on the previously identified set of detected loops $\mathit{loops}(L)$ and skips $\mathit{skips}(L)$. For that, we use their corresponding artificial activity set $\mathcal{A}_{loop}$ and $\mathcal{A}_{skip}$ as well as their corresponding trace transformation functions $\mathit{repair}_{\circlearrowleft}$ and repair_{τ} to transform the input event log L into a repaired event log $\hat{L}$. Note that $act(\hat{L}) = \left(\mathcal{A}_L \uplus \mathcal{A}_{loop} \uplus \mathcal{A}_{skip}\right)$.

$$\hat{L} = [\mathit{repair}_{\tau}\left(\mathit{repair}_{\circlearrowleft}(\sigma, L{\upharpoonright}\mathcal{A}_L), L{\upharpoonright}\mathcal{A}_L\right) \mid \sigma \in L{\upharpoonright}\mathcal{A}_L]$$

3.2. Create an Advising DFG

Next, we extract a pruned DFG from the repaired event log $\hat{L}$, which ignores infrequent directly-follows relations. This DFG is used as guidance using the following algorithm steps. Note that this step does not modify the repaired event log: The output of this step is a pruned DFG containing the activities $act(\hat{L})$ as nodes. Edges between activities a and b are retained if their weight corresponds to at least 1% of the sum of the weights of all incoming edges to b or 1% of the sum of all outgoing edges from a. The value of 1% was determined as a good cutoff through experimentation. In addition, edges with weights below an absolute threshold value $n \in \mathbb{N}_0$ are also removed.

For the repaired event log $\hat{L}$ and a given DFG-weight threshold $n \in \mathbb{N}_0$, we define the *advising DFG* (abbreviated as aDFG) as follows:

$$\mathit{minW}(a,b) = 0.01 \cdot \min\left\{ \sum_{c \in act(\hat{L})} \overset{\hat{L}}{\Rightarrow}(c,b), \sum_{c \in act(\hat{L})} \overset{\hat{L}}{\Rightarrow}(a,c) \right\}$$

$$\text{aDFG} = \left(act(\hat{L}), \left[(a,b) \in act(\hat{L})^2 \middle| \overset{\hat{L}}{\Rightarrow}(a,b) \geq \max\{n, \mathit{minW}(a,b)\} \right] \right)$$

3.3. Candidate Building

With the repaired event log and the aDFG, we can continue with building place candidates. Place candidates are composed of two sets of activities: The first set corresponds to the transitions that should add a token to this place in a Petri net. The second set corresponds to transitions that should remove a token from this place.

The set of all place candidates is given by:

$$\begin{aligned} \mathit{Cnd}_0 = \{(A_1, A_2) \mid A_1, A_2 \subseteq act(\hat{L}) & \land \forall_{a_1 \in A_1} \forall_{a_2 \in A_2} (a_1 \overset{\text{aDFG}}{\Longrightarrow} a_2) \\ & \land \forall_{a_1 \in A_1} \forall_{a_2 \in A_1 \setminus A_2} (a_1 \overset{\text{aDFG}}{\not\Rightarrow} a_2) \\ & \land \forall_{a_1 \in A_2 \setminus A_1} \forall_{a_2 \in A_2} (a_1 \overset{\text{aDFG}}{\not\Rightarrow} a_2) \\ & \land \exists_{a_1 \in A_1 \setminus A_2} \exists_{a_2 \in A_2 \setminus A_1} (a_2 \overset{\text{aDFG}}{\not\Rightarrow} a_1)\} \end{aligned}$$

3.4. Candidate Pruning

The set of place candidates Cnd_0 includes many unfit places, which would produce process models with very low fitness. Furthermore, some place candidates might be dominated by others (e.g., the place candidate $(\{a\}, \{f\})$ is dominated by the candidate $(\{a, b\}, \{e, f\})$). Pruning the set of place candidates requires an efficient approach, as the number of place candidates can easily grow huge. We propose a three-step pruning approach. First, place candidates are filtered purely based on activity counts. If the difference in frequency of the input and output activity set is relatively large, the place candidate is rather unfit. This condition can be checked very efficiently. Next, the local fitness of the place candidate is calculated based on local trace replay. Local trace replay takes the order of the activities in the traces into account, and thus can detect even more unfit place candidates. Finally, to remove dominated place candidates, we retain only maximal place candidates.

3.4.1. Balance-based Pruning:

For the balance-based pruning, we consider the number of activity occurrences in the log $\hat{L}$ using $\mathit{actMult}(\hat{L})$. For a set of activities, $A \subseteq act(\hat{L})$ we can then sum the frequencies together as $\mathit{count}(\hat{L}, A) = \sum_{a \in A} \mathit{actMult}(\hat{L})(a)$. Based on that, we define the *balance* of a candidate (A_1, A_2):

$$\mathit{balance}(\hat{L}, A_1, A_2) = \frac{|\mathit{count}(\hat{L}, A_1) - \mathit{count}(\hat{L}, A_2)|}{\max\{\mathit{count}(\hat{L}, A_1), \mathit{count}(\hat{L}, A_2)\}}$$

The balance of a candidate is between 0 and 1. Higher values are an indication that the place candidate is unfit. Based on a balance threshold $b \in [0,1]$, candidates with a higher balance value than b can be filtered out:

$$\mathit{Cnd}_1 = \{(A_1, A_2) \in \mathit{Cnd}_0 \mid \mathit{balance}(\hat{L}, A_1, A_2) \leq b\}$$

3.4.2. Fitness-based Pruning:

Let $\mathit{fit}(\sigma, (A_1, A_2), k)$ be defined as follows:

$$\mathit{fit}(\sigma, (A_1, A_2), k) = \begin{cases} 1 & \text{if } \sigma = \langle\rangle, k = 0 \\ 0 & \text{if } \sigma = \langle\rangle, k \neq 0 \\ 0 & \text{if } \sigma = \langle a\rangle \cdot \sigma', k = 0, a \notin A_1, a \in A_2 \\ \mathit{fit}(\sigma', (A_1, A_2), k+1) & \text{if } \sigma = \langle a\rangle \cdot \sigma', a \in A_1, a \notin A_2 \\ \mathit{fit}(\sigma', (A_1, A_2), k-1) & \text{if } \sigma = \langle a\rangle \cdot \sigma', k \geq 1, a \notin A_1, a \in A_2 \\ \mathit{fit}(\sigma', (A_1, A_2), k) & \text{if } \sigma = \langle a\rangle \cdot \sigma', (a \in A_1 \cap A_2 \vee a \notin A_1 \cup A_2) \end{cases}$$

Note that $\mathit{fit}(\sigma, (A_1, A_2), 0) = 1$ if the place candidate (A_1, A_2) fits the trace; otherwise it takes the value 0.

The traces relevant for a place candidate (A_1, A_2) are defined by the following function:

$$\mathit{rel}(A_1, A_2) = \left[\sigma = \langle a_1, \ldots, a_n\rangle \in \hat{L} \mid \exists_{i \in \{1,\ldots,n\}} (a_i \in A_1 \vee a_i \in A_2)\right]$$

We consider traces relevant for a place candidate, if they contain at least one activity that is in the set of outgoing or ingoing activities of that place candidate. For a single activity, we use the notation $\mathit{rel}(a) := \mathit{rel}(\{a\}, \emptyset)$ to denote the traces containing that activity.

We write $\mathit{fit}(\sigma, (A_1, A_2)) := \mathit{fit}(\sigma, (A_1, A_2), 0)$ and

$$\mathit{fit}(\hat{L}, (A_1, A_2)) := \sum_{\sigma \in \mathit{rel}(A_1, A_2)} \mathit{fit}(\sigma, (A_1, A_2))$$

for ease of notation.

For a given local candidate fitness threshold $t \in [0,1]$, the candidates remaining after the local fitness replay pruning are then given as:

$$\mathit{mfit}(A_1, A_2) = \min \left\{ \frac{\sum_{\sigma \in \mathit{rel}(a)} \mathit{fit}(\sigma, (A_1, A_2))}{|\mathit{rel}(a)|} \,\middle|\, a \in A_1 \cup A_2 \right\}$$

$$\mathit{Cnd}_2 = \left\{ (A_1, A_2) \in \mathit{Cnd}_1 \,\middle|\, \frac{\mathit{fit}(\hat{L}, (A_1, A_2))}{|\mathit{rel}(A_1, A_2)|} \geq t \wedge \mathit{mfit}(A_1, A_2) \geq t \right\}$$

3.4.3. Maximal Candidate Selection:

Finally, as the last candidate pruning step, all dominated place candidates are removed, just like in the original Alpha algorithm.

$$\begin{aligned} \mathit{Sel} = \{(A_1, A_2) \in \mathit{Cnd}_2 \mid \forall_{(A_1', A_2') \in \mathit{Cnd}_2} ((A_1 \subseteq A_1' \wedge A_2 \subseteq A_2') \\ \Rightarrow (A_1, A_2) = (A_1', A_2'))\} \end{aligned}$$

3.5. Petri Net Construction

Based on the remaining place candidates, an accepting Petri net is constructed as the tuple $((P, T, F, l), M_{init}, M_{final})$, where

- $P = \{p_{(A_1,A_2)} \mid (A_1, A_2) \in \mathit{Sel}\}$
- $T = \{t_a \mid a \in act(\hat{L}) \setminus \{▶, ■\}\}$
- $F = \{(t_a, p_{(A_1,A_2)}) \mid (A_1, A_2) \in \mathit{Sel} \wedge a \in A_1 \setminus \{▶, ■\}\} \cup \{(p_{(A_1,A_2)}, t_a) \mid (A_1, A_2) \in \mathit{Sel} \wedge a \in A_2 \setminus \{▶, ■\}\}$
- $l = \{(t_a, a) \mid a \in \mathcal{A}_L\} \cup \{(t_a, \tau) \mid a \in (\mathcal{A}_{loop} \cup \mathcal{A}_{skip}\}$
- $M_{init} = \left[p_{(A_1,A_2)} \in P \mid ▶ \in A_1\right]$
- $M_{final} = \left[p_{(A_1,A_2)} \in P \mid ■ \in A_2\right]$

are the components defined using the results of the previous steps.

3.6. Post-Processing Petri Net

Let $\mathit{replay}(p, PN, \sigma)$ be the replay function, which takes the value 1 exactly when the place p of the Petri net PN can replay trace σ (i.e., there is no missing or remaining token in p at any time when replaying σ on PN).

For a given local place replay fitness threshold $r \in [0, 1]$, we can then define the result of the post-process replay as $((P', T, F', l), M'_{init}, M'_{final})$, where the set of updated places P' is given by:

$$P' = \left\{ p_{(A_1,A_2)} \mid (A_1, A_2) \in \mathit{Sel} \wedge \frac{\sum_{\sigma \in rel(A_1,A_2)} \mathit{replay}(p, PN, \sigma)}{|rel(A_1, A_2)|} \geq r \right\}$$

The flow relation and initial and final markings are also updated correspondingly:

- $F' = \{(i, o) \in F \mid i \in P' \wedge o \in P'\}$
- $M'_{init} = [p \in M_{init} \mid p \in P']$
- $M'_{final} = [p \in M_{final} \mid p \in P']$

The final accepting Petri net discovered is $((P', T, F', l), M'_{init}, M'_{final})$.

4. Implementation

We implemented the Alpha+++ algorithm as a ProM[1] plugin (Java) and also created a Python implementation[2] for large-scale evaluation on a variety of real-life event logs. The ProM plugin

[1]https://promtools.org/
[2]https://github.com/aarkue/alpha-revisit-python

Table 1
Overview of the event logs used for evaluation. We used a random sample of 3000 cases from the BPI Challenge 2019 log for computational reasons, as it allowed for alignment-based evaluation of the discovered models.

Event Log	#Events	#Activities	#Traces	#Variants	Reference
RTFM	561,470	11	150,370	231	[17]
Sepsis	15,214	16	1,050	846	[18]
BPI Challenge 2019 (Sample of 3000 Cases)	18,972	34	3,000	470	[19]
BPI Challenge 2020 (Request for Payment)	36,796	19	6,886	89	[20]
BPI Challenge 2020 (Domestic Declaration)	56,437	17	10,500	99	[21]

(AlphaRevisitExperiments) can be installed in ProM Nightly versions and can be used in standard mode to simply discover a Petri net or in interactive mode to experiment with different algorithm step options and view additional information (e.g., how many place candidates were pruned in which step). In both versions, the Alpha+++ preset can be selected out of the preset list on the top. The parameters used throughout the algorithm steps can then be changed. Additionally, the different algorithm steps can be swapped with alternatives or skipped, allowing for further experimentation.

5. Evaluation

To evaluate the proposed Alpha+++ algorithm (α+++), we discovered Petri nets for five real-life event logs, shown in Table 1. For comparison, we also discovered models using the Inductive Miner Infrequent (IMf) and the standard Alpha algorithm (α). We subsequently calculated alignment-based fitness, precision and F1-scores using PM4Py[3].

For IMf, we evaluated four models per event log using noise thresholds of 0.1, 0.2, 0.3 and 0.4. For α, we used four variant filtering approaches upfront: Either only selecting the 10 most common variants or the n most common variants to cover at least 10%, 50% or 80% of traces. For α+++, we chose artificial activity thresholds of 2 and 4 (relative to the mean directly-follows weight) for the log repair steps. Here, a lower threshold value causes more artificial activities to be added. For each artificial activity threshold, we selected five combinations of the balance b, local candidate fitness t and local place replay fitness r thresholds. Note, that for t and r a value closer to 1 and for b a value closer to 0 is more restrictive. We did not apply problematic activities filtering.

The evaluation results are shown in Table 2. Overall, the fitness and F1-scores of α+++ are competitive compared to the IMf. 8 of the 20 models discovered with α are not easy sound (i.e., no final marking is reachable), and thus no alignment scores could be computed. The remaining 12 models exhibit rather low fitness for some logs but very high precision across the board, significantly boosting the corresponding F1-values. Although our approach does not guarantee

[3]https://pm4py.fit.fraunhofer.de/ (Version 2.6.1)

easy soundness, all 50 Petri nets discovered with α+++ are easy sound and allow computation of alignments. There are notable differences across the different event logs: α+++ performs significantly worse compared to the IMf on the Sepsis log in terms of F1-score, caused by lower precision scores, as the models discovered with α+++ seem to be underfitting. On the two BPI Challenge 2020 logs, α+++ outperforms the IMf in most configurations, often also exhibiting better fitness and precision scores simultaneously.

The influence of the parameters of α+++ is mostly as expected: More restrictive b, t, r values improve the fitness of the models while decreasing the precision.

Manual inspection of the discovered models reveals that the models discovered with α+++ are mostly rather simple and often consist of several disconnected model fragments. Furthermore, multiple models exhibit redundant structures involving silent transitions (e.g., a place with one labeled transition as preset and one silent transition as postset). Such constructs could be removed by further post-processing of the Petri net. For further details, a comprehensive list of the discovered models, as well as simplicity and generalization results, we refer interested readers to the extended report version of this paper at [22].

6. Conclusion

In this paper, we revisited the Alpha algorithm to overcome its limitations, focusing on real-life event logs. For that, we presented the Alpha+++ algorithm which, like the Alpha algorithm, primarily uses directly-follows relations to discover Petri nets. Alpha+++ pre-processes event logs by adding artificial activities for potential loop or skip constructs. This allows discovering silent transitions while still assessing the fitness of places by easily computable token-based replay instead of expensive alignment computations. Subsequently, place candidates are generated based on a pruned DFG. A multistep candidate filtering approach efficiently removes place candidates with low fitness, configurable through parameters. We implemented the Alpha+++ algorithm both as a ProM plugin and in Python. The ProM plugin is available in ProM nightly builds and also features an interactive mode to allow experimenting with different algorithm steps and parameters. We evaluated the Alpha+++ on five real-life event logs and compared the results to the classical Alpha algorithm and the widely adapted Inductive Miner Infrequent. Overall, the results indicate that the Alpha+++ algorithm is competitive in terms of fitness and precision. In general, the different step parameter configurations tested reliably determine the trade-off between fitness and precision.

Further research should include further evaluation of the algorithm. For that, additional performance metrics like simplicity or generality could be included and also compared to other process discovery algorithms. It is particularly interesting to see if there are any patterns regarding algorithm parameters, event log properties, and model performance. Such observations could enable automatic parameter selection based on the log, and thus simplify Alpha+++ to a well-performing one-in-all algorithm. Additionally, a more comprehensive qualitative analysis of the discovered models is needed. Furthermore, the assumptions and effects of the algorithm steps should be studied, e.g., using artificial event logs with relevant control structures. Further research could also explore if any theoretical guarantees, such as easy-soundness, are attainable, e.g., using more sophisticated post-processing of the discovered Petri net.

Table 2: Evaluation Results

	Inductive Miner Infrequent				**Alpha Algorithm**				**Alpha+++ Algorithm**									
	Noise Threshold				Variant Filtering				*Artificial Activity Threshold of 2.0*					*Artificial Activity Threshold of 4.0*				
	0.1	0.2	0.3	0.4	Top10	10%	50%	80%	b=0.5 t=0.5 r=0.5	b=0.3 t=0.7 r=0.6	b=0.2 t=0.8 r=0.7	b=0.2 t=0.8 r=0.8	b=0.1 t=0.9 r=0.9	b=0.5 t=0.5 r=0.5	b=0.3 t=0.7 r=0.6	b=0.2 t=0.8 r=0.7	b=0.2 t=0.8 r=0.8	b=0.1 t=0.9 r=0.9
RTFM																		
Fitness	0.9871	0.9094	0.9091	0.7657	0.6711	0.6731	0.8769	0.8769	0.7880	0.9412	0.9935	0.9935	0.9998	0.9160	0.9925	0.9935	0.9935	0.9998
Precision	0.6218	0.6705	0.7959	0.9929	0.6797	1.0000	1.0000	1.0000	0.5223	0.3677	0.3082	0.3082	0.3086	0.4056	0.3127	0.3082	0.3082	0.3086
F1 Score	0.7630	0.7719	0.8487	0.8646	0.6754	0.8046	0.9344	0.9344	0.6282	0.5289	0.4705	0.4705	0.4716	0.5622	0.4756	0.4705	0.4705	0.4716
Sepsis Cases																		
Fitness	0.9382	0.9075	0.8421	0.8108	0.6378	–	–	–	0.9183	0.9362	0.9828	0.9965	0.9965	0.9275	0.9636	0.9948	0.9948	1.0000
Precision	0.6049	0.6158	0.6298	0.7285	0.9916	–	–	–	0.3758	0.2922	0.3152	0.2633	0.2633	0.2855	0.2923	0.2923	0.2923	0.2805
F1 Score	0.7356	0.7337	0.7206	0.7675	0.7763	–	–	–	0.5334	0.4454	0.4773	0.4166	0.4166	0.4365	0.4485	0.4518	0.4518	0.4381
BPI Challenge 2019 (Sample of 3000 Cases)																		
Fitness	0.9906	0.9938	0.9536	0.9177	0.6282	0.7462	–	0.3205	0.9422	0.9431	0.9506	0.9506	1.0000	0.9422	0.9433	0.9602	0.9602	1.0000
Precision	0.2086	0.2379	0.2383	0.2528	0.9986	1.0000	–	0.9964	0.3395	0.3180	0.2416	0.2416	0.1968	0.3748	0.3487	0.2501	0.2501	0.1968
F1 Score	0.3446	0.3839	0.3813	0.3964	0.7712	0.8547	–	0.4850	0.4992	0.4757	0.3852	0.3852	0.3288	0.5363	0.5092	0.3968	0.3968	0.3288
BPI Challenge 2020 (Requests for Payment)																		
Fitness	0.9476	0.9051	0.9051	0.9051	–	0.8678	0.8380	–	0.9179	0.9438	0.9438	0.9438	0.9595	0.9179	0.9438	0.9438	0.9438	0.9595
Precision	0.3173	0.2704	0.2704	0.2704	–	1.0000	1.0000	–	0.5415	0.4451	0.4451	0.4451	0.3500	0.5415	0.4451	0.4451	0.4451	0.3500
F1 Score	0.4754	0.4164	0.4164	0.4164	–	0.9292	0.9119	–	0.6812	0.6049	0.6049	0.6049	0.5129	0.6812	0.6049	0.6049	0.6049	0.5129
BPI Challenge 2020 (Domestic Declaration)																		
Fitness	0.9499	0.9302	0.9302	0.9302	–	0.8906	0.8549	–	0.9029	0.9265	0.9308	0.9308	0.9493	0.9143	0.9143	0.9461	0.9461	0.9477
Precision	0.4056	0.2469	0.2469	0.2469	–	1.0000	1.0000	–	0.9094	0.7206	0.7192	0.7192	0.4897	0.6795	0.6795	0.4780	0.4780	0.4780
F1 Score	0.5685	0.3902	0.3902	0.3902	–	0.9421	0.9218	–	0.9061	0.8107	0.8114	0.8114	0.6461	0.7796	0.7796	0.6351	0.6351	0.6354

References

[1] W.M.P. van der Aalst, B.F. van Dongen, Discovering Workflow Performance Models from Timed Logs, in: Y. Han, S. Tai, D. Wikarski (Eds.), International Conference on Engineering and Deployment of Cooperative Information Systems (EDCIS 2002), Vol. 2480 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2002, pp. 45–63.

[2] W.M.P. van der Aalst, A.J.M.M. Weijters, L. Maruster, Workflow Mining: Discovering Process Models from Event Logs, IEEE Transactions on Knowledge and Data Engineering 16 (9) (2004) 1128–1142.

[3] L. Wen, W.M.P. van der Aalst, J. Wang, J. Sun, Mining Process Models with Non-Free-Choice Constructs, Data Mining and Knowledge Discovery 15 (2) (2007) 145–180.

[4] L. Wen, J. Wang, W.M.P. van der Aalst, B. Huang, J. Sun, Mining Process Models with Prime Invisible Tasks, Data and Knowledge Engineering 69 (10) (2010) 999–1021.

[5] A. Ehrenfeucht, G. Rozenberg, Partial (Set) 2-Structures - Part 1 and Part 2, Acta Informatica 27 (4) (1989) 315–368.

[6] J. Cortadella, M. Kishinevsky, L. Lavagno, A. Yakovlev, Deriving Petri Nets from Finite Transition Systems, IEEE Transactions on Computers 47 (8) (1998) 859–882.

[7] W.M.P. van der Aalst, V. Rubin, H.M.W. Verbeek, B.F. van Dongen, E. Kindler, C.W. Günther, Process Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting, Software and Systems Modeling 9 (1) (2010) 87–111.

[8] J. Carmona, J. Cortadella, M. Kishinevsky, A Region-Based Algorithm for Discovering Petri Nets from Event Logs, in: Business Process Management (BPM 2008), 2008, pp. 358–373.

[9] M. Solé, J. Carmona, Process Mining from a Basis of State Regions, in: J. Lilius, W. Penczek (Eds.), Applications and Theory of Petri Nets 2010, Vol. 6128 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2010, pp. 226–245.

[10] R. Bergenthum, J. Desel, R. Lorenz, S. Mauser, Process Mining Based on Regions of Languages, in: G. Alonso, P. Dadam, M. Rosemann (Eds.), International Conference on Business Process Management (BPM 2007), Vol. 4714 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2007, pp. 375–383.

[11] J.M.E.M. van der Werf, B.F. van Dongen, C.A.J. Hurkens, A. Serebrenik, Process Discovery using Integer Linear Programming, Fundamenta Informaticae 94 (2010) 387–412.

[12] S.J. van Zelst, B.F. van Dongen, W.M.P. van der Aalst, H.M.W Verbeek, Discovering Workflow Nets Using Integer Linear Programming, Computing 100 (5) (2018) 529–556.

[13] L.L. Mannel, W.M.P. van der Aalst, Discovering Process Models with Long-Term Dependencies While Providing Guarantees and Handling Infrequent Behavior, in: L. Bernardinello, L. Petrucci (Eds.), Application and Theory of Petri Nets and Concurrency (Petri Nets 2022), Vol. 13288 of Lecture Notes in Computer Science, 2022, pp. 303–324.

[14] S.J.J. Leemans, D. Fahland, W.M.P. van der Aalst, Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour, in: N. Lohmann, M. Song, P. Wohed (Eds.), Business Process Management Workshops, International Workshop on Business Process Intelligence (BPI 2013), Vol. 171 of Lecture Notes in Business Information Processing, Springer-Verlag, Berlin, 2014, pp. 66–78.

[15] S.J.J. Leemans, D. Fahland, W.M.P. van der Aalst, Scalable Process Discovery and Conformance Checking, Software and Systems Modeling 17 (2) (2018) 599–631. doi:10.1007/

s10270-016-0545-x.

[16] A. Augusto, R. Conforti, M. Marlon, M. La Rosa, A. Polyvyanyy, Split Miner: Automated Discovery of Accurate and Simple Business Process Models from Event Logs, Knowledge Information Systems 59 (2) (2019) 251–284.

[17] M. de Leoni, F.Mannhardt, Road Traffic Fine Management Process, https://data.4tu.nl/articles/dataset/Road_Traffic_Fine_Management_Process/12683249 (2015). doi:10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5.

[18] F. Mannhardt, Sepsis Cases - Event Log, https://data.4tu.nl/articles/dataset/Sepsis_Cases_-_Event_Log/12707639 (2016). doi:10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.

[19] B. van Dongen, BPI Challenge 2019, https://data.4tu.nl/articles/dataset/BPI_Challenge_2019/12715853 (2019). doi:10.4121/uuid:d06aff4b-79f0-45e6-8ec8-e19730c248f1.

[20] B. van Dongen, BPI Challenge 2020: Request For Payment, https://data.4tu.nl/articles/dataset/BPI_Challenge_2020_Request_For_Payment/12706886 (2020). doi:10.4121/uuid:895b26fb-6f25-46eb-9e48-0dca26fcd030.

[21] B. van Dongen, BPI Challenge 2020: Domestic Declarations, https://data.4tu.nl/articles/dataset/BPI_Challenge_2020_Domestic_Declarations/12692543 (2020). doi:10.4121/uuid:3f422315-ed9d-4882-891f-e180b5b4feb5.

[22] A. Küsters, W.M.P. van der Aalst, Revisiting the Alpha Algorithm To Enable Real-Life Process Discovery Applications – Extended Report (2023). arXiv:2305.17767.