I-PALIA: Discovering BPMN Processes with Duplicated Activities for Healthcare Domains

Carlos Fernandez-Llatas[1,3] and Andrea Burattin[2]

[1] Universitat Politecnica de Valencia, Spain
[2] Technical University of Denmark, Denmark
[3] Karolinska Institutet, Sweden

Abstract. Process mining encompasses a range of methods designed to analyze event logs. Among these methods, control-flow discovery algorithms are particularly significant, as they enable the identification of real-world process models, known as *in-vivo* processes, in contrast to anticipated models. An obstacle faced by control-flow discovery algorithms is their limited ability to recognize duplicated activities, which are activities that occur in multiple locations within a process. This issue is particularly relevant in the healthcare sector, where numerous instances of duplicated activities exist in processes but remain undetected by conventional algorithms. This article introduces a novel concept for a control-flow discovery algorithm capable of effectively revealing duplicated activities. The effectiveness of this technique is demonstrated through experimentation on a synthetic dataset. Moreover, the algorithm has been implemented and its source code is available as open-source software, accessible both as a ProM plugin and a Java Maven dependency.

Keywords: Process mining · Control-flow discovery · BPMN · Duplicated activities.

1 Introduction

Process mining is the scientific discipline aiming at connecting process models and recordings of activity executions [17]. In particular, *control-flow discovery* techniques pertain to the synthesis of models which are capable of explaining in a compact way all (or most of) the executions reported in an event log.

The ultimate goal of process mining techniques is to improve processes and how these are actually deployed in the real world. Since these models are supposed to improve actual processes it is essential that the models identified are as reliable and *good* as possible.

When considering the control-flow discovery techniques, the main challenge they need to face consists of extracting a model which is the most suitable representation possible. Defining "most suitable representation" is a challenge in itself and, in the literature, numerical approaches to quantify this dimension have been proposed, in particular *fitness*, *precision*, *generalisation* and *simplicity* [17]. Fitness indicates that a model should be able to replicate the log it has been generated from; precision quantifies how much more behavior (w.r.t. the starting log) the mined model permits; generalization tries to capture to what extent behavior not observed in the log is present in the model; and, finally, simplicity verifies that the model should be as simple as possible, to foster understandability. All these metrics should be maximized in order to obtain good results.

While many algorithms for control-flow discovery have put a lot of focus on optimizing fitness and precision [11,19,1], less emphasis has been put on the simplicity dimension, in particular regarding the type of supported behavior. Specifically, as mentioned in the Process Mining Manifesto [10] (as guiding principle GP3), control-flow discovery algorithms should be able to identify basic control-flow constructs [15,18], such as concurrency and choice. In many situations, a limiting factor towards better simplicity is the problem of *duplicated activities*. This problem stems from the fact that most control-flow discovery algorithms are not able to produce process models where the same activity occurs more than once.

Within the healthcare sector, it's common to encounter activities that occur repeatedly. Instances such as revisiting medical appointments, undergoing assessment procedures like laboratory tests and medical imaging, or undergoing cyclic treatments like dialysis or chemotherapy are recurrent events throughout a patient's journey within the care process. These activities carry significant importance in representing the overall process. To illustrate, the sequencing of treatments can vary significantly based on the timing of assessments. For instance, in the context of cancer treatments, administering chemotherapy before and/or after surgery can yield distinct results. The initial treatment aims to shrink the tumor size and streamline the surgical procedure, while the subsequent treatment focuses on preventing the proliferation of harmful cells, introducing differing objectives and complexities that impact process delineation.

When applying conventional process discovery algorithms to this scenario, the ability to differentiate between these distinct behaviors becomes compromised. In this example, as well as in numerous other cases within the healthcare sphere, the utilization of duplicated nodes becomes imperative. Not only do they contribute to a lucid depiction of the process, but they also facilitate comprehension of the preparatory and follow-up stages surrounding these recurrent activities.

In this paper we present a new algorithm capable of extracting process models, represented as BPMN [12]. The algorithm can identify all the basic workflow patterns (i.e., sequence, parallel split, synchronization, exclusive choice, and simple merge) with the addition of duplicated activities.

The rest of the paper is structured as follows: Section 2 introduces the background and the state of the art of the proposed technique; Section 3 presents the actual algorithm proposed in the paper; Section 4 describes the implementation of the technique; Section 5 presents some performance results of the algorithm against some extreme scenarios; and finally Section 6 concludes the paper.

2 Background and State of the Art

In the literature on control-flow discovery [1] many algorithms have been developed able to identify all the basic workflow patterns. Among these, the Alpha miner is typically recalled as one of the first algorithms explicitly tackling the control-flow discovery problem. More advanced algorithms, such as the Heuristics Miner, the Fuzzy Miner, the Split Miner, and the Inductive Miner, have gained a lot of popularity due to the quality of the output they produce and their performance. However, none of these is actually capable of producing duplicated activities. The algorithms which are able to achieve this are very few, including the $\alpha *$-algorithm [5] which nonetheless have very restrictive assumptions on the event log. Fodina [5] can discover duplicated activities by pre-processing the event log only based on some heuristics. Genetic Process Mining [6], ETM [2], AGNES [9] are evolutionary algorithms that, in principle, can discover duplicated activities at the expense of extreme computational complexity. Other algorithms that exploit region theory are also capable of mining duplicated activities which however SLAD [20] is another approach that post-processes the mined model for duplicated activities. Also in this case, however, the algorithm exploits some heuristics for simplifying the model by duplicating some behavior.

As a basis for the paper we should provide some definitions. Classic PALIA algorithm [8,14] uses as a representation model, a Timed Parallel Automaton [7]. This model has an expressiveness equivalent to a safe Petri Net [13] and has a Regular complexity [7] based on the concept of Parallel Finite Automaton [16]. For this paper a TPA is defined as follows:

Definition 1. *A Timed Parallel Automaton(TPA) [7] is a tuple $\mathcal{T}$ ={N, Q, γ, δ, S, F} where:*

- *N is a finite set of Nodes, where a Node n is a graphical representation of the action a,*
- *Q is a finite set of states where $q \subseteq N^+ \forall q \in Q$,*
- *$\gamma : N^+ \rightarrow N^+$ is the Node Transition function, where $\gamma = \{(n_0^s, .., n_i^s), (n_0^e, .., n_j^e)\}$. Γ is the set of γ functions;*
- *$\delta : Q \rightarrow Q$ is the State transition function where $\delta = \{q^s, q^e\}$. Δ is the set of δ functions where $(\forall \delta \in \Delta \; \exists \; \gamma \in \Gamma \; / \; (n_0^s, \ldots, n_i^s) = q^s \wedge (n_0^e, \ldots, n_i^e) = q^e)$ where γ is the associated transition of δ;*
- *$S \subseteq N$ is the set of Starting Nodes;*
- *$F \subseteq N$ is the set of Final Nodes.*

A TPA has a double transition function, defining Nodes (N) and States (Q). The transitions between nodes can be n to n multiple for defining parallelism. For example, a transition $\gamma_1 = \{(n_1), (n_2, n_3)\}$ represents a parallel split from the node n_1 to the nodes n_2 and n_3. In the same way, a transition $\gamma_2 = \{(n_1, n_2), (n_3)\}$ represent a parallel synchronization from the nodes n_1 and n_2 to the node n_3. Exclusive Choice can be represented using several γ transitions. For example, having $\gamma_1 = \{(n_1), (n_2)\}$ and $\gamma_2 = \{(n_1), (n_3)\}$ we can represent that, from n_1 is possible to go to n_2 or n_3.

The view of States provides a regular language view over the interpretation of the automaton. In the states view ($\varDelta$ set), the parallelism is represented at the state level, A State q is a set of Nodes that represents the actions that are active in the state q in an analogous way to a Petri-Net marking. $\varDelta$ and $\varGamma$ are complementary. The state transitions (δ) keep the regular complexity due to their simplicity (N to N) where the Node Transitions ($\varGamma$) represent parallel situations (N^+ to N^+). In practice, there is only one state active in a moment in time, which supposes that it could be several nodes active. Node transitions represent the single pass from one node to another. This double transition function allows representing complex workflow patterns, like milestones or interleaved parallel routines [7]. There is an algorithm that given a $\varGamma$ set it is possible to construct a basic $\varDelta$ set of states [7]. The objective of the discovery algorithm of this paper is to construct a TPA with a $\varGamma$ set assuming that $\varDelta$ can be automatically inferred.

Following, we present the definitions of Event and Event Log:

Definition 2. *An Event is a pair $e = \{a, \pi\}$ where $a \in \mathcal{A}$ is the set of possible actions in a process, π is the timestamp execution of the action.*

Definition 3. *An Event Log ($\mathcal{L}$) is a set of traces $\mathcal{L}$=$\{t_0, \ldots, t_i\}$, where a trace is a sequence of events t=$\{e_0, \ldots, e_j\}$*

An *Event* is the digital representation of an action in a moment in time. A set of events referenced to the same user or the same execution instance is called *Trace.*

3 Inductive PALIA (I-PALIA): The Algorithm

Within this document, we introduce an upgraded iteration of the PALIA algorithm, specifically devised to uncover activity logs through the application of Grammar Inference methods. Subsequently, the algorithm undertakes the task of detecting parallel configurations within the inferred model. To facilitate this process, we have established certain foundational definitions:

Definition 4. *Given events $e_0 = \{a_0, \pi_0\}, e_1 = \{a_1, \pi_1\}$, e_0 is equivalent to e_1 ($e_0 \equiv e_1$) where $a_0 = a_1$.*

Given event $e = \{a_0, \pi\}$ and a Node n representing the action a_1, e_0 is equivalent to n ($e_0 \equiv n$) where $a_0 = a_1$.

Given two Nodes n_0, n_1, they are equivalents ($n_0 \equiv n_1$) if they have the same representing action.

Definition 5. *Given two sets of nodes N_0, N_1 they are equivalent ($N_0 \equiv N_1$) if $|N_0| = |N_1|$ and $\exists n_i \equiv n_j | \forall n_i \in N_0, n_j \in N_1$*

Colloquially, Nodes and Events are equivalent when they refer to the same action, and two sets of nodes are equivalent if each one of the nodes of each set is equivalent to a node of the other set.

Definition 6. *Given two Node Transitions $\gamma_0 = \{D_0, R_0\}$, $\gamma_1 = \{D_1, R_1\}$, they are equivalents ($\gamma_0 \equiv \gamma_1$) if their Domains (D_0,D_1) and Ranges (R_0,R_1) are equivalent*

In an analogous way, two Node Transitions are equivalent if their domains and ranges are equivalent.

Definition 7. *Given a TPA, and $n_0, n_1 \in N$, n_0 is directly followed by n_1 ($n_0 \rightarrow n_1$) if $\exists$ a node transition $\gamma_0 \in \gamma | n_0 \rightarrow n_1$.*

γ_0 is directly followed by γ_1 ($\gamma_0 \rightarrow \gamma_1$) if $n^e \in \gamma_0 = n^s \gamma_1$

Definition 8. *Given a TPA, and $n_0, n_1 \in N$, n_0 is eventually followed by n_1, ($n_0 \Rightarrow n_1$) where $\exists$ a sequence of node transition $\{\gamma_0..\gamma_j\}$ $\forall_{0<i<j} | \gamma_i \rightarrow \gamma_{i+1} \wedge n^s \in \gamma_0 = n_0 \wedge n^e \in \gamma_j = n_1$*

γ_0 is eventually followed by γ_1, ($\gamma_0 \Rightarrow \gamma_1$) if $n^e \in \gamma_0 \Rightarrow n^s \in \gamma_1$

The next definitions define when two node transitions are directly followed ($\rightarrow$) when the second transition can be immediately accessed from the first one and eventually followed ($\Rightarrow$) when the second transition can be eventually accessed from the first one.

Definition 9. *A Prefix Acceptor Tree (PAT) is a tree-like TPA built from the learning Log by taking all the prefixes in the sample as states and constructing the smallest TPA which is a tree, strongly consistent with the Log.*

The Prefix Acceptor Tree creates a TPA that represents a tree from left to right with the events of the samples. The head of the tree is formed by the nodes representing the starting events of the traces, and the leaves represent the last events of the traces. Figure 1 shows an example of how this tree is created. Should be noticed that although TPA is able to represent parallel situations, the Prefix Acceptor Tree creates only non-parallel δ functions so this algorithm is not able to represent parallelism.

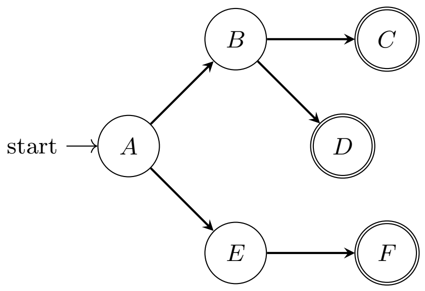


Fig. 1: Prefix Acceptor Tree of Log =$\{$ABC,ABD,AEF$\}$

Algorithm 1 shows the pseudocode of the presented algorithm. The first action is to compute the Prefix Acceptor Tree that represents the Log. Using

Algorithm 1 Inductive PALIA Algorithm

```
Require: $\mathcal{L}$
Ensure: $\mathcal{T}$
 1: $\mathcal{T} \leftarrow PrefixAceptorTree(\mathcal{L})$
 2: for all $\gamma : (n^s, n^e) \in \Gamma)$ do
 3:     if $n^s \equiv n^e$ then Merge($n^s$, $n^e$)
 4:     end if
 5: end for
 6: for all $n_0, n_1 \in F | n_0 \equiv n_1$  do $Merge(n_0, n_1)$
 7: end for
 8: while $\mathcal{T}$ has changes do
 9:     for all $\gamma_x \Rightarrow \gamma_y \mid \gamma_x = \{d_x, r_x\}$, $\gamma_y = \{d_y, r_y\} \in \Gamma$  do
10:         if $\gamma_x \equiv \gamma_y$ then
11:             $Merge(d_x, d_y)$
12:             $Merge(r_x, r_y)$
13:         end if
14:     end for
15: end while
16: $\mathcal{T} \leftarrow ParallelMerge(\mathcal{T})$                 ▷ See Algorithm 2
```

the Prefix Acceptor Tree as a basis, Inductive PALIA will perform some generalizations over the algorithm using Grammar Inference techniques. First, the algorithm merges equivalent consecutive nodes, generalizing all the transitions from one node to itself. Second the algorithm merges all the final nodes that are equivalent. Finally, the algorithm Merge the domains (d_x, d_y) and ranges (r_x, r_y) of each couple of γ in Γ that are eventually followed $(\gamma_x \Rightarrow \gamma_y)$ and are equivalent $(\gamma_x \equiv \gamma_y)$. This action is performed until the TPA $\mathcal{T}$ has no new merges. This algorithm allows an ordered merging of the nodes that prevent their massive merging like in other basic algorithms such as Directly Follows Graphs (DFG).

Until this moment, the algorithm has discovered a process structure Discovering with sequences, splits, and loops, assuming no parallelism, but differentiating repeated non-consecutive nodes. The next step is to identify the parallel sequences with the Algorithm 2: Parallel Merge. In this algorithm, the parallelism has been defined as:

Definition 10. *Given $n_0, n_1 \in N$, n_0 and n_1 are parallel $(n_0 \| n_1)$ where:*

$$n_0 \Rightarrow n_1 \wedge n_1 \Rightarrow n_0 \tag{1}$$

$$!n_0 \Leftrightarrow n_1 \wedge !n_1 \Leftrightarrow n_0 \tag{2}$$

Colloquially, two nodes are parallel $(n_0 \| n_1)$ if both are eventually followed between themselves and these nodes are not acting as a loop, that means, not exist a path of transitions that contains $n_0 \Rightarrow n_1$ and $n_1 \Rightarrow n_0$ in the same path $(n_0 \Leftrightarrow n_1)$, and vice-versa. With that definition, the Parallel Merge (Algorithm 2) defines the Parallel Regions for discovering the parallel situations:

Definition 11. *Given a TPA $\mathcal{T}$ and node n_{Split}, A Parallel Region is a set of nodes R where $n_{Split} \Rightarrow n_x \wedge \exists n_y : n_x \| n_y | \forall n_x, n_y \in R$*

A Parallel Region is a set of nodes that occurs after a split node n_{Split} and have parallelisms between themselves.

Definition 12. *Given a TPA $\mathcal{T}$, a node n_{Split}, and a Parallel Region R, a Syncronization Node is a node n_{Syncro} where $n_i \Rightarrow n_{Syncro} | \forall n_i \in R$ and $\forall n_x | n_{Split} \Rightarrow n_x \wedge n_x \Rightarrow n_{Syncro} | n_x \in R$*

A Synchronization Node n_{Syncro} is a node that occurs immediately after the Parallel region. n_{Split} and n_{Syncro} delimits the Parallel Region and are the nodes that will define the start and the end of parallelism. According to that, the Algorithm 2 Discover and create the Parallel structures.

Algorithm 2 Parallel Merge

```
Require: $\mathcal{T}$
Ensure: $\mathcal{T}$
1: for all $n_{Split} \in N$ do
2:     $R \leftarrow GetParallelRegions(\mathcal{T}, n_{Split})$
3:     for all $p \in R$ do
4:         $n_{Syncro} \leftarrow GetSyncroNode(\mathcal{T}, n_{Split}, R)$
5:         $seq \leftarrow$IdentifyParallelSequences($\mathcal{T}, R$)        ▷ See Algorithm 3
6:         $\mathcal{T} \leftarrow$CreateParallelTransitions($\mathcal{T}, n_{Split}, n_{Syncro}, R, seq$)
7:     end for
8: end for
```

The *Parallel Merge* Algorithm tries to discover parallel regions after each node on the TPA (representing this node the n_{split} of the Parallel Region). Once a Parallel Region is detected, the subsequent task involves pinpointing the synchronization node that marks the boundaries of said Parallel Region. Subsequently, the algorithm proceeds to determine the parallel sequences within the identified Parallel Region. This specific process is detailed in Algorithm 3.

Algorithm 3 Identify Parallel Sequences

```
Require: $\mathcal{T}, R$
Ensure: $SeqMap$
1: for all $r \in R$ do
2:     $SeqMap(r) \leftarrow (r, S)$ where $S \subset R \wedge s_i \nparallel r | \forall s_i \in S$
3: end for
```

The algorithm *Identify Parallel Sequences* segregates the nodes within the Parallel Region into distinct groups, classifying those that do not exhibit parallelism in relation to the rest. This methodology serves to discern the parallel sequences contained within the defined Parallel Region.

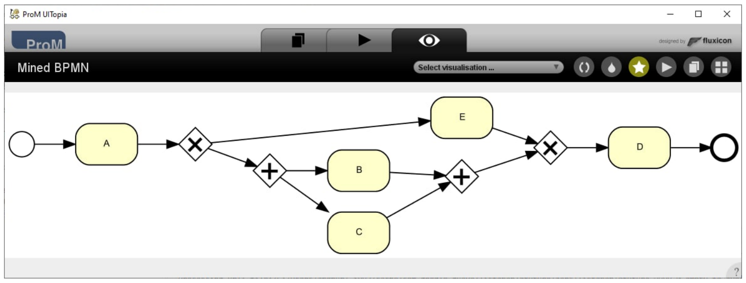


Fig. 2: The PaliaProM plugin showing the result of a mining session.

Upon successful identification of the parallel sequences, the Parallel Merge Algorithm (Algorithm 2) allows the creation of the ascertained parallel structure. This involves establishing connections between each parallel sequence and the Split node at the beginning, as well as the Synchronization node at the conclusion of the respective sequences.

4 Implementation

I-PALIA has been implemented in Java and as a Maven project as well as a ProM plugin[4]. Having the code available as stand-alone Java project simplifies its embedding into new projects (the code can easily be imported into any Maven/Gradle/Ivy project[5]). The PaliaProM package, which imports the Maven dependency, allows us to easily benefit from the algorithm leveraging the infrastructure made available by ProM. The package contains two plugins, one called "Palia Miner" which takes an XES log object as input and produces a standard BPMN object as result. The second plugin made available in the PaliaProM package is a visualizer for BPMN called "Graphviz BPMN visualisation" which exploits Graphviz to display any BPMN process model. A graphical representation of the latter is reported in Fig. 2.

5 Evaluation and Discussion

In order to evaluate the effectiveness of the algorithm, we decided to compare the models resulting from the mining of the process using the state of the art algorithms and tools for control-flow discovery. We designed a process (available in Fig. 3) specifically expressing the challenges of duplicated activities, in the case of the process it is activity A. In addition to that, we incorporated behavior coming from the most common workflow patterns [18]: sequences, parallel split, synchronization, exclusive choice, and simple merge.

[4] The Maven project is available at `https://github.com/delas/palia`. The ProM package, called PaliaProM, is available at `https://github.com/delas/PaliaProM`.
[5] See `https://jitpack.io/#delas/palia`.

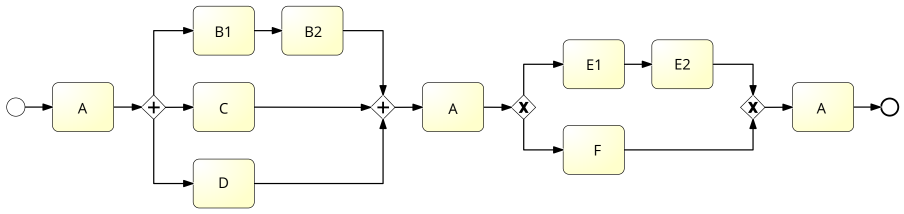


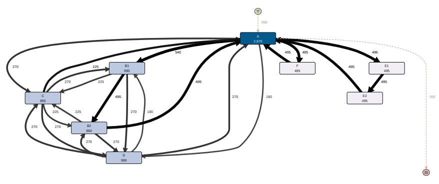
Fig. 3: The reference process we used for our simulation

We simulated the process using the tool Purple [4][6], configuring the tool for the rediscoverability purpose, which led to an event log with 990 traces and 8415 events. We mined this log with I-PALIA as well as with Fluxicon Disco[7], Celonis[8], Apromore[9] and the Inductive Miner [11]. The results are available in Fig. 4. While the model mined with I-PALIA matches perfectly the expected one, all other mining tools and algorithms fail at extracting a model that resembles the original one. The main issue, as expected, is the duplication of activity A which is observed at the beginning, in the middle, and at the end of the process. This causes all models to show unstructured behavior, quite similar for each mining algorithm: the process can start with activity A and finish immediately, or can have repetitions of the combinations of the other activities (the part before and after the A in the middle).

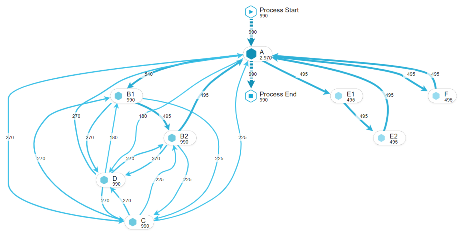
In another battery of tests we aimed at learning something about the computational performance of the I-PALIA implementation. For this purpose, we generated a random process model using PLG2 [3] (cf. Fig. 5). With this model, we generated 6 event logs with 100, 500, 1000, 2000, 5000, and 10000 traces. These logs were mined with I-PALIA and we monitored the execution time. The tests were performed on a standard laptop, equipped with Java 1.15(TM) SE Runtime Environment on Windows 10 Enterprise 64bit, an Intel Core i7-7500U 2.70GHz CPU and 16GB of RAM. Results are reported in Fig. 6. As the plot shows, the time required to process is not negligible and could grow quite quickly. Considering the biggest log we had 10000 traces, 58713 events, and our implementation of I-PALIA took almost 7 seconds for the actual mining.

While the algorithm shows very promising results, it still suffers from important issues. The most important ones are the current lack of robustness to noise and its computational complexity. Regarding the lack of robustness to noise, this is certainly a problem that makes the algorithm not mature enough for many industrial settings, however, we believe this is an issue that can easily be ad-

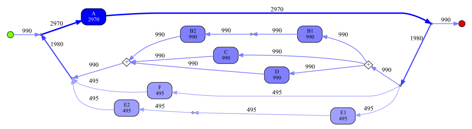
[6] See `http://pros.unicam.it:4300/`.
[7] See `https://fluxicon.com/disco/`.
[8] See `https://www.celonis.com/`.
[9] See `https://apromore.com/`.



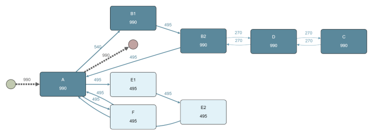
(a) Model discovered with Disco



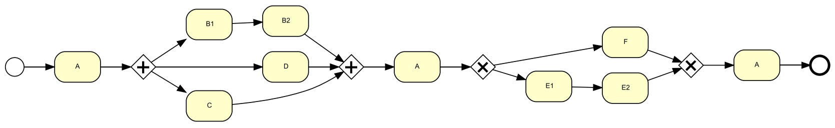
(b) Model discovered with Celonis



(c) Model discovered with Inductive Miner



(d) Model discovered with Apromore



(e) Model discovered with I-PALIA

Fig. 4: Results of the mining of the log using different techniques

dressed by considering frequencies of the direct following relations and applying some threshold on those before applying the rest of the computation. A more fundamental challenge regards the complexity of the algorithm. Right now, several steps and needed in order to reach the goal, and each of these contribute substantially to the complexity. For small logs the mining time is acceptable, but it grows quickly as the numbers of traces and events grow. Optimizations can be employed also in this case, both in terms of implementation (e.g., multithreading) as well as more conceptual ones.

6 Conclusion and Future Work

In this paper we presented I-PALIA, a process mining algorithm for control-flow discovery. The algorithm is capable of synthesising BPMN process models containing all basic workflow patterns (i.e., sequence, parallel split, synchronization, exclusive choice, and simple merge) as well as duplicated activities. The algorithm, which has a publicly available implementation as both ProM package as well as a Java Maven dependency has been tested and evaluated both qualitatively (against state of the art tools) as well as quantitatively on logs of different sizes.

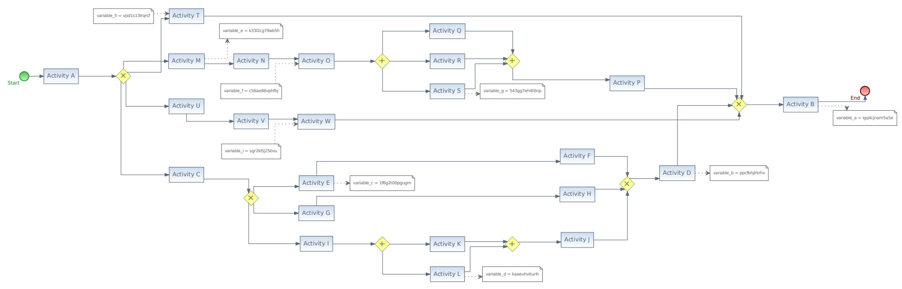


Fig. 5: Random model generated for the stress test

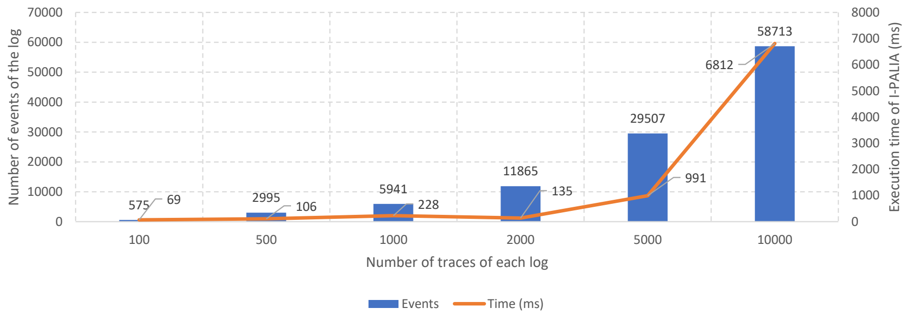


Fig. 6: Stress test on the algorithm against log with different sizes

Future work for I-PALIA certainly includes the extension of the algorithm to become noise tolerant, for example employing frequencies. Additionally improving the performance and the computational complexity of the approach certainly represents a fundamental step towards wider adoption.

Acknowledgements This research has been supported through projects MINE-GUIDE(PID2020-113723RB-C21) funded by MCIN/AEI/10.13039/501100011033 and LIFECHAMPS(Grant Agreement No 875329) under European Union's Horizon 2020 research and innovation program.

References

1. Adriano Augusto, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, Fabrizio Maria Maggi, Andrea Marrella, Massimo Mecella, and Allar Soo. Automated Discovery of Process Models from Event Logs: Review and Benchmark. *IEEE Transactions on Knowledge and Data Engineering*, 31(4):686–705, 2019.
2. J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. Quality Dimensions in Process Discovery: The Importance of Fitness, Precision, Generalization and Simplicity. *International Journal of Cooperative Information Systems*, 23(01):1440001, 3 2014.

3. Andrea Burattin. PLG2 : Multiperspective Process Randomization with Online and Offline Simulations. In *Online Proceedings of the BPM Demo Track 2016*. CEUR-WS.org, 2016.
4. Andrea Burattin, Barbara Re, Lorenzo Rossi, and Francesco Tiezzi. A purpose-guided log generation framework. In *Proceedings of BPM (To Appear)*, 2022.
5. T. Calders, Christian W. Günther, Mykola Pechenizkiy, and Anne Rozinat. Using minimum description length for process mining. In *Proceedings of the 2009 ACM symposium on Applied Computing - SAC '09*, pages 1451–1455, New York, New York, USA, 2009. ACM Press.
6. Ana Karla Alves de Medeiros. *Genetic Process Mining*. PhD thesis, Technische Universiteit Eindhoven, 2006.
7. Carlos Fernandez-Llatas, Salvatore F. Pileggi, Vicente Traver, and Jose M. Benedi. Timed parallel automaton: A mathematical tool for defining highly expressive formal workflows. In *2011 Fifth Asia Modelling Symposium*, pages 56–61, 2011.
8. Carlos Fernández-Llatas, Teresa Meneu, Jose Miguel Benedí, and Vicente Traver. Activity-based process mining for clinical pathways computer aided design. In *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology*, pages 6178–6181, 2010.
9. Stijn Goedertier, David Martens, Jan Vanthienen, and Bart Baesens. Robust Process Discovery with Artificial Negative Events. *The Journal of Machine Learning Research*, 10:1305–1340, 2009.
10. IEEE Task Force on Process Mining. Process Mining Manifesto. In Florian Daniel, Kamel Barkaoui, and Schahram Dustdar, editors, *Business Process Management Workshops*, pages 169–194. Springer-Verlag, 2011.
11. Sander J. J. Leemans, Dirk Fahland, and Wil M.P. van der Aalst. Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour. In *Business Process Management Workshops*, pages 66–78, 2014.
12. OMG. *Business Process Model and Notation (BPMN) - Version 2.0, Beta 1*. 2009.
13. James L. Peterson. Petri Nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, 1977.
14. Eric Rojas, Carlos Fernández-Llatas, Vicente Traver, Jorge Munoz-Gama, Marcos Sepúlveda, Valeria Herskovic, and Daniel Capurro. Palia-er: Bringing question-driven process mining closer to the emergency room. In *15th International Conference on Business Process Management (BPM 2017)*, 2017.
15. Nick Russell, Arthur H.M. ter Hofstede, Wil M.P. van der Aalst, and Nataliya Mulyar. Workflow Control-flow Patterns: A Revised View. *BPM Center Report BPM-06-22, BPMcenter. org*, 2006.
16. P David Stotts and William Pugh. Parallel finite automata for modeling concurrent software systems. *Journal of Systems and Software*, 27(1):27–43, 1994.
17. Wil M.P. van der Aalst. *Process Mining*. Springer Berlin Heidelberg, second edition, 2016.
18. Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
19. Wil M.P. van der Aalst, Ton A. J. M. M. Weijters, and Ana Karla Alves de Medeiros. Process Mining with the Heuristics Miner-algorithm. BETA Working Paper Series, WP 166, 2006.
20. Borja Vázquez-Barreiros, Manuel Mucientes, and Manuel Lama. Enhancing discovered processes with duplicate tasks. *Information Sciences*, 373:369–387, 2016.