Alignment Approximation for Process Trees

Daniel Schuster[1], Sebastiaan van Zelst[1,2], and Wil M. P. van der Aalst[1,2]

[1] Fraunhofer Institute for Applied Information Technology FIT, Germany
{daniel.schuster,sebastiaan.van.zelst}@fit.fraunhofer.de
[2] RWTH Aachen University, Germany
wvdaalst@pads.rwth-aachen.de

Abstract. Comparing observed behavior (event data generated during process executions) with modeled behavior (process models), is an essential step in process mining analyses. Alignments are the de-facto standard technique for calculating conformance checking statistics. However, the calculation of alignments is computationally complex since a shortest path problem must be solved on a state space which grows non-linearly with the size of the model and the observed behavior, leading to the well-known *state space explosion problem*. In this paper, we present a novel framework to approximate alignments on process trees by exploiting their hierarchical structure. Process trees are an important process model formalism used by state-of-the-art process mining techniques such as the inductive mining approaches. Our approach exploits structural properties of a given process tree and splits the alignment computation problem into smaller sub-problems. Finally, sub-results are composed to obtain an alignment. Our experiments show that our approach provides a good balance between accuracy and computation time.

Keywords: Process mining · Conformance checking · Approximation.

1 Introduction

Conformance checking is a key research area within process mining [1]. The comparison of observed process behavior with reference process models is of crucial importance in process mining use cases. Nowadays, *alignments* [2] are the de-facto standard technique to compute conformance checking statistics. However, the computation of alignments is complex since a shortest path problem must be solved on a non-linear state space composed of the reference model and the observed process behavior. This is known as the *state space explosion problem* [3]. Hence, various approximation techniques have been introduced. Most techniques focus on decomposing Petri nets or reducing the number of alignments to be calculated when several need to be calculated for the same process model [4–8].

In this paper, we focus on a specific class of process models, namely process trees (also called *block-structured* process models), which are an important process model formalism that represent a subclass of sound *Workflow nets* [9]. For instance, various state-of-the-art process discovery algorithms return process

trees [9–11]. In this paper, we introduce an alignment approximation approach for process trees that consists of two main phases. First, our approach splits the problem of alignments into smaller sub-problems along the tree hierarchy. Thereby, we exploit the hierarchical structure of process trees and their semantics. Moreover, the definition of sub-problems is based on a *gray-box view* on the corresponding subtrees since we use a simplified/abstract view on the subtrees to recursively define the sub-problems along the tree hierarchy. Such sub-problems can then be solved individually and in parallel. Secondly, we recursively compose an alignment from the sub-results for the given process tree and observed process behavior. Our experiments show that our approach provides a good balance between accuracy and computation effort.

The remainder is structured as follows. In Section 2, we present related work. In Section 3, we present preliminaries. In Section 4, we present the formal framework of our approach. In Section 5, we introduce our alignment approximation approach. In Section 6, we present an evaluation. Section 7 concludes the paper.

2 Related Work

In this section, we present related work regarding alignment computation and approximation. For a general overview of conformance checking, we refer to [3].

Alignments have been introduced in [2]. In [12] it was shown that the computation is reducible to a shortest path problem and the solution of the problem using the A* algorithm is presented. In [13], the authors present an improved heuristic that is used in the shortest path search. In [14], an alignment approximation approach based on approximating the shortest path is presented.

A generic approach to decompose Petri nets into multiple sub-nets is introduced in [15]. Further, the application of such decomposition to alignment computation is presented. In contrast to our approach, the technique does not return an alignment. Instead, only partial alignments are calculated, which are used, for example, to approximate an overall fitness value. In [4], an approach to calculate alignments based on Petri net decomposition [15] is presented that additionally guarantees optimal fitness values and optionally returns an alignment. Comparing both decomposition techniques with our approach, we do not calculate sub-nets because we simply use the given hierarchical structure of a process tree. Moreover, our approach always returns a valid alignment.

In [5], an approach is presented that approximates alignments for an event log by reducing the number of alignments being calculated based on event log sampling. Another technique based on event log sampling is presented in [8] where the authors explicitly approximate conformance results, e.g., fitness, rather than alignments. In contrast to our proposed approach, alignments are not returned. In [6] the authors present an approximation approach that explicitly focuses on approximating multiple optimal alignments. Finally, in [7], the authors present a technique to reduce a given process model and an event log s.t. the original behavior of both is preserved as much as possible. In contrast, the proposed approach in this paper does not modify the given process model and event log.

Table 1: Example of an event log from an order process

Event-id	Case-id	Activity name	Timestamp	···
···	···	···	···	···
200	13	create order (c)	2020-01-02 15:29	···
201	27	receive payment (r)	2020-01-02 15:44	···
202	43	dispatch order (d)	2020-01-02 16:29	···
203	13	pack order (p)	2020-01-02 19:12	···
···	···	···	···	···

3 Preliminaries

We denote the power set of a given set X by $\mathcal{P}(X)$. A multi-set over a set X allows multiple appearances of the same element. We denote the universe of multi-sets for a set X by $\mathcal{B}(X)$ and the set of all sequences over X as X^*, e.g., $\langle a,b,b\rangle \in \{a,b,c\}^*$. For a given sequence σ, we denote its length by $|\sigma|$. We denote the empty sequence by $\langle\rangle$. We denote the set of all possible permutations for given $\sigma \in X^*$ by $\mathbb{P}(\sigma) \subseteq X^*$. Given two sequences σ and σ', we denote the concatenation of these two sequences by $\sigma \cdot \sigma'$. We extend the $\cdot$ operator to sets of sequences, i.e., let $S_1, S_2 \subseteq X^*$ then $S_1 \cdot S_2 = \{\sigma_1 \cdot \sigma_2 \mid \sigma_1 \in S_1 \wedge \sigma_2 \in S_2\}$. For traces σ, σ', the set of all interleaved sequences is denoted by $\sigma \diamond \sigma'$, e.g., $\langle a,b\rangle \diamond \langle c\rangle = \{\langle a,b,c\rangle, \langle a,c,b\rangle, \langle c,a,b\rangle\}$. We extend the $\diamond$ operator to sets of sequences. Let $S_1, S_2 \subseteq X^*$, $S_1 \diamond S_2$ denotes the set of interleaved sequences, i.e., $S_1 \diamond S_2 = \bigcup_{\sigma_1 \in S_1, \sigma_2 \in S_2} \sigma_1 \diamond \sigma_2$.

For $\sigma \in X^*$ and $X' \subseteq X$, we recursively define the projection function $\sigma_{\downarrow_{X'}} : X^* \to (X')^*$ with: $\langle\rangle_{\downarrow_{X'}} = \langle\rangle, \big(\langle x\rangle \cdot \sigma\big)_{\downarrow_{X'}} = \langle x\rangle \cdot \sigma_{\downarrow_{X'}}$ if $x \in X'$ and $(\langle x\rangle \cdot \sigma)_{\downarrow_{X'}} = \sigma_{\downarrow_{X'}}$ else.

Let $t = (x_1, \ldots, x_n) \in X_1 \times \ldots \times X_n$ be an n-tuple over n sets. We define projection functions that extract a specific element of t, i.e., $\pi_1(t) = x_1, \ldots, \pi_n(t) = x_n$, e.g., $\pi_2((a,b,c)) = b$. Analogously, given a sequence of length m with n-tuples $\sigma = \langle (x_1^1, \ldots, x_n^1), \ldots, (x_1^m, \ldots, x_n^m)\rangle$, we define $\pi_1^*(\sigma) = \langle x_1^1, \ldots, x_1^m\rangle, \ldots, \pi_n^*(\sigma) = \langle x_n^1, \ldots, x_n^m\rangle$. For instance, $\pi_2^*\big(\langle (a,b),(a,c),(b,a)\rangle\big) = \langle b,c,a\rangle$.

3.1 Event Logs

Process executions leave *event data* in information systems. An *event* describes the execution of an activity for a particular *case*/process instance. Consider Table 1 for an example of an *event log* where each event contains the executed activity, a timestamp, a case-id and potentially further attributes. Since, in this paper, we are only interested in the sequence of activities executed, we define an event log as a multi-set of sequences. Such sequence is also referred to as a *trace*.

Definition 1 (Event log). *Let $\mathcal{A}$ be the universe of activities. $L \in \mathcal{B}(\mathcal{A}^*)$ is an event log.*

3.2 Process Trees

Next, we define the syntax and semantics of process trees.

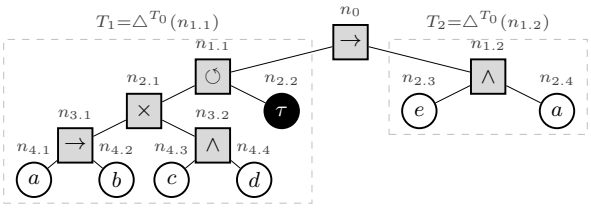


Fig. 1: Process tree $T_0{=}\big(\{n_o,\ldots,n_{4.4}\},\big\{(n_0,n_{1.1}),\ldots,(n_{3.2},n_{4.4})\big\},\lambda,n_0\big)$ with $\lambda(n_0){=}\rightarrow,\ldots,\lambda(n_{4.4}){=}d$

Definition 2 (Process Tree Syntax). *Let $\mathcal{A}$ be the universe of activities and $\tau{\notin}\mathcal{A}$. Let $\bigoplus{=}\{\rightarrow,\times,\wedge,\circlearrowleft\}$ be the set of process tree operators. We define a process tree $T{=}(V,E,\lambda,r)$ consisting of a totally ordered set of nodes V, a set of edges E, a labeling function $\lambda{:}V{\rightarrow}\mathcal{A}{\cup}\{\tau\}{\cup}\bigoplus$ and a root node $r{\in}V$.*

- $\big(\{n\},\{\},\lambda,n\big)$ *with* $\lambda(n){\in}\mathcal{A}{\cup}\{\tau\}$ *is a process tree*
- *given* $k{>}1$ *process trees* $T_1{=}(V_1,E_1,\lambda_1,r_1),\ldots,T_k{=}(V_k,E_k,\lambda_k,r_k)$, $T{=}(V,E,\lambda,r)$ *is a process tree s.t.:*
  - $V{=}V_1{\cup}\ldots{\cup}V_k{\cup}\{r\}$ *(assume* $r{\notin}V_1{\cup}\ldots{\cup}V_k$*)*
  - $E{=}E_1{\cup}\ldots{\cup}E_k{\cup}\big\{(r,r_1),\ldots,(r,r_k)\big\}$
  - $\lambda(x){=}\lambda_j(x)\ \forall j{\in}\{1,\ldots,k\}\forall x{\in}V_j,\lambda(r){\in}\{\rightarrow,\wedge,\times\}$
- *given two process trees* $T_1{=}(V_1,E_1,\lambda_1,r_1)$ *and* $T_2{=}(V_2,E_2,\lambda_2,r_2)$, $T{=}(V,E,\lambda,r)$ *is a process tree s.t.:*
  - $V{=}V_1{\cup}V_2{\cup}\{r\}$ *(assume* $r{\notin}V_1{\cup}V_2$*)*
  - $E{=}E_1{\cup}E_2{\cup}\big\{(r,r_1),(r,r_2)\big\}$
  - $\lambda(x){=}\lambda_1(x)$ *if* $x{\in}V_1,\lambda(x){=}\lambda_2(x)$ *if* $x{\in}V_2,\lambda(r){=}\circlearrowleft$

In Figure 1, we depict an example process tree T_0 that can alternatively be represented textually due to the totally ordered node set, i.e., $T_0{\widehat{=}}{\rightarrow}({\circlearrowleft}({\times}({\rightarrow}(a,b),{\wedge}(c,d)),\tau),{\wedge}(e,a))$. We denote the universe of process trees by $\mathcal{T}$. The degree d indicates the number of edges connected to a node. We distinguish between incoming d^+ and outgoing edges d^-, e.g., $d^+(n_{2.1}){=}1$ and $d^-(n_{2.1}){=}2$. For a tree $T{=}(V,E,\lambda,r)$, we denote its *leaf nodes* by $T^L{=}\{v{\in}V|d^-(v){=}0\}$. The child function $c^T{:}V{\rightarrow}V^*$ returns a sequence of child nodes according to the order of V, i.e., $c^T(v){=}\langle v_1,\ldots,v_j\rangle$ s.t. $(v,v_1),\ldots,(v,v_j){\in}E$. For instance, $c^T(n_{1.1}){=}\langle n_{2.1},n_{2.2}\rangle$. For $T{=}(V,E,\lambda,r)$ and a node $v{\in}V$, $\triangle^T(v)$ returns the corresponding tree T' s.t. v is the root node, i.e., $T'{=}(V',E',\lambda',v)$. Consider T_0, $\triangle^{T_0}(n_{1.1}){=}T_1$ as highlighted in Figure 1. For process tree $T{\in}\mathcal{T}$, we denote its height by $h(T){\in}\mathbb{N}$.

Definition 3 (Process Tree Semantics). *For given $T{=}(V,E,\lambda,r){\in}\mathcal{T}$, we define its language $\mathcal{L}(T){\subseteq}\mathcal{A}^*$.*

- *if* $\lambda(r){=}a{\in}\mathcal{A}$, $\mathcal{L}(T){=}\{\langle a\rangle\}$
- *if* $\lambda(r){=}\tau$, $\mathcal{L}(T){=}\{\langle\rangle\}$
- *if* $\lambda(r){\in}\{\rightarrow,\times,\wedge\}$ *with* $c^T(r){=}\langle v_1,\ldots,v_k\rangle$
  - *with* $\lambda(r){=}\rightarrow$, $\mathcal{L}(T){=}\mathcal{L}(\triangle^T(v_1))\cdot\ldots\cdot\mathcal{L}(\triangle^T(v_k))$
  - *with* $\lambda(r){=}\wedge$, $\mathcal{L}(T){=}\mathcal{L}(\triangle^T(v_1))\diamond\ldots\diamond\mathcal{L}(\triangle^T(v_k))$

trace part	a	b	$\gg$	$\gg$	c	f	$\gg$	$\gg$
model part	$n_{4.1}$ $\lambda(n_{4.1})=a$	$n_{4.2}$ $\lambda(n_{4.2})=b$	$n_{2.2}$ $\lambda(n_{2.2})=\tau$	$n_{4.4}$ $\lambda(n_{4.4})=d$	$n_{4.3}$ $\lambda(n_{4.3})=c$	$\gg$	$n_{2.4}$ $\lambda(n_{2.4})=a$	$n_{2.3}$ $\lambda(n_{2.3})=e$

Fig. 2: Optimal alignment $\gamma=\big\langle(a,n_{4.1}),\ldots,(\gg,n_{2.3})\big\rangle$ for $\langle a,b,c,f\rangle$ and T_0

  - *with* $\lambda(r)=\times$, $\mathcal{L}(T)=\mathcal{L}(\triangle^T(v_1))\cup\ldots\cup\mathcal{L}(\triangle^T(v_k))$
- *if* $\lambda(r)=\circlearrowleft$ *with* $c^T(r)=\langle v_1,v_2\rangle$, $\mathcal{L}(T)=\{\sigma_1\cdot\sigma_1'\cdot\sigma_2\cdot\sigma_2'\cdot\ldots\cdot\sigma_m \mid m\geq 1\wedge\forall 1\leq i\leq m\big(\sigma_i\in\mathcal{L}(\triangle^T(v_1))\big)\wedge\forall 1\leq i\leq m-1\big(\sigma_i'\in\mathcal{L}(\triangle^T(v_2))\big)\}$

In this paper, we assume binary process trees as input for our approach, i.e, every node has two or none child nodes, e.g., T_0. Note that every process tree can be easily converted into a language equivalent binary process tree [9].

3.3 Alignments

Alignments [12] map observed behavior onto modeled behavior specified by process models. Figure 2 visualizes an alignment for the trace $\langle a,b,c,f\rangle$ and T_0 (Figure 1). The first row corresponds to the given trace ignoring the skip symbol $\gg$. The second row (ignoring $\gg$) corresponds to a sequence of leaf nodes s.t. the corresponding sequence of labels (ignoring τ) is in the language of the process tree, i.e., $\langle a,b,d,c,a,e\rangle\in\mathcal{L}(T_0)$. Each column represents an alignment move. The first two are *synchronous moves* since the activity and the leaf node label are equal. The third and fourth are *model moves* because $\gg$ is in the log part. Moreover, the third is an *invisible* model move since the leaf node label is τ and the fourth is a *visible* model move since the label represents an activity. Visible model moves indicate that an activity should have taken place w.r.t. the model. The sixth is a log move since the trace part contains $\gg$. Log moves indicate observed behavior that should not occur w.r.t. the model. Note that we alternatively write $\gamma\widehat{=}\big\langle(a,a),\ldots,(\gg,e)\big\rangle$ using their labels instead of leaf nodes.

Definition 4 (Alignment). *Let $\mathcal{A}$ be the universe of activities, $\sigma\in\mathcal{A}^*$ be a trace and $T=(V,E,\lambda,r)\in\mathcal{T}$ be a process tree with leaf nodes T^L. Note that $\gg,\tau\notin\mathcal{A}$. A sequence $\gamma\in\big((\mathcal{A}\cup\{\gg\})\times(T^L\cup\{\gg\})\big)^*$ with length $n=|\gamma|$ is an alignment iff:*

1. $\sigma=\pi_1^*(\gamma)_{\downarrow_{\mathcal{A}}}$
2. $\Big\langle\lambda\Big(\pi_2\big(\gamma(1)\big)\Big),\ldots,\lambda\Big(\pi_2\big(\gamma(n)\big)\Big)\Big\rangle_{\downarrow_{\mathcal{A}}}\in\mathcal{L}(T)$
3. $(\gg,\gg)\notin\gamma$ *and* $(a,v)\notin\gamma\ \forall a\in\mathcal{A}\ \forall v\in T^L\big(a\neq\lambda(v)\big)$

For a given process tree and a trace, many alignments exist. Thus, costs are assigned to alignment moves. In this paper, we assume the *standard cost function*. Synchronous and invisible model moves are assigned cost 0, other moves are assigned cost 1. An alignment with minimal costs is called *optimal*. For a process tree T and a trace σ, we denote the set of all possible alignments by $\Gamma(\sigma,T)$. In this paper, we assume a function α that returns for given $T\in\mathcal{T}$ and $\sigma\in\mathcal{A}^*$ an optimal alignment, i.e., $\alpha(\sigma,T)\in\Gamma(\sigma,T)$. Since process trees can be easily converted into Petri nets [1] and the computation of alignments for a Petri net was shown to be reducible to a shortest path problem [12], such function exists.

4 Formal Framework

In this section, we present a general framework that serves as the basis for the proposed approach. The core idea is to recursively divide the problem of alignment calculation into multiple sub-problems along the tree hierarchy. Subsequently, we recursively compose partial sub-results to an alignment.

Given a trace and tree, we recursively split the trace into sub-traces and assign these to subtrees along the tree hierarchy. During splitting/assigning, we regard the semantics of the current root node's operator. We recursively split until we can no longer split, e.g., we hit a leaf node. Once we stop splitting, we calculate optimal alignments for the defined sub-traces on the assigned subtrees, i.e., we obtain sub-alignments. Next, we recursively compose the sub-alignments to a single alignment for the parent subtree. Thereby, we consider the semantics of the current root process tree operator. Finally, we obtain a *valid*, but not necessarily optimal, alignment for the initial given tree and trace since we regard the semantics of the process tree during splitting/assigning and composing.

Formally, we can express the splitting/assigning as a function. Given a trace $\sigma \in \mathcal{A}^*$ and $T=(V, E, \lambda, r) \in \mathcal{T}$ with subtrees T_1 and T_2, ψ splits the trace σ into k sub-traces $\sigma_1, \dots, \sigma_k$ and assigns each sub-trace to either T_1 or T_2.

$$\psi(\sigma, T) \in \Big\{ \big\langle (\sigma_1, T_{i_1}), \dots, (\sigma_k, T_{i_k}) \big\rangle \mid i_1, \dots, i_k \in \{1,2\} \wedge \sigma_1 \cdot \ldots \cdot \sigma_k \in \mathbb{P}(\sigma) \Big\} \quad (1)$$

We call a splitting/assignment *valid* if the following additional conditions are satisfied depending on the process tree operator:

- if $\lambda(r)=\times$: $k=1$
- if $\lambda(r)=\rightarrow$: $k=2 \wedge \sigma_1 \cdot \sigma_2 = \sigma$
- if $\lambda(r)=\wedge$: $k=2$
- if $\lambda(r)=\circlearrowleft$: $k \in \{1,3,5,\dots\} \wedge \sigma_1 \cdot \ldots \cdot \sigma_k = \sigma \wedge i_1 = 1 \wedge \forall j \in \{1, \dots, k-1\} \big((i_j = 1 \Rightarrow i_{j+1} = 2) \wedge (i_j = 2 \Rightarrow i_{j+1} = 1) \big)$

Secondly, the calculated sub-alignments are recursively composed to an alignment for the respective parent tree. Assume a tree $T \in \mathcal{T}$ with sub-trees T_1 and T_2, a trace $\sigma \in \mathcal{A}^*$, a valid splitting/assignment $\psi(\sigma, T)$, and a sequence of k sub-alignments $\langle \gamma_1, \dots, \gamma_k \rangle$ s.t. $\gamma_j \in \Gamma(\sigma_j, T_{i_j})$ with $(\sigma_j, T_{i_j}) = \psi(\sigma, T)(j) \forall j \in \{1, \dots, k\}$. The function ω composes an alignment for T and σ from the given sub-alignments.

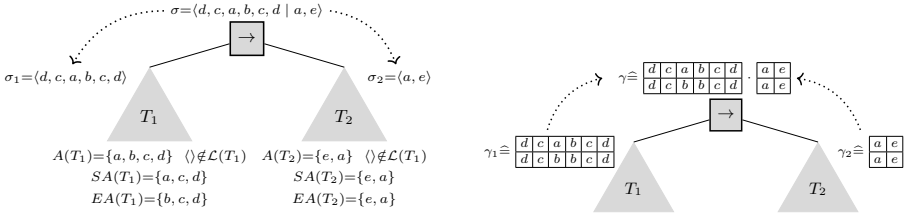
$$\omega(\sigma, T, \langle \gamma_1, \dots, \gamma_k \rangle) \in \{ \gamma \mid \gamma \in \Gamma(\sigma, T) \wedge \gamma_1 \cdot \ldots \cdot \gamma_k \in \mathbb{P}(\gamma) \} \quad (2)$$

By utilizing the definition of process tree semantics, it is easy to show that, given a valid splitting/assignment, such alignment γ returned by ω always exists.

The overall, recursive approach is sketched in Algorithm 1. For a given tree T and trace σ, we create a valid splitting/assignment (line 4). Next, we recursively call the algorithm on the determined sub-traces and subtrees (line 6). If given thresholds for trace length (TL) or tree height (TH) are reached, we stop splitting and return an optimal alignment (line 2). Hence, for the sub-traces created, we eventually obtain optimal sub-alignments, which we recursively compose to an alignment for the parent tree (line 7). Finally, we obtain a valid, but not necessarily optimal, alignment for T and σ.

Algorithm 1: Approximate alignment

input: $T{=}(V, E, \lambda, r){\in}\mathcal{T}, \sigma{\in}\mathcal{A}^*, TL{\geq}1, TH{\geq}1$
begin
1 **if** $|\sigma|{\leq}TL \vee h(T){\leq}TH$ **then**
2 **return** $\alpha(\sigma, T)$; // optimal alignment
3 **else**
4 $\psi(\sigma, T){=}\langle(\sigma_1, T_{i_1}), \ldots, (\sigma_k, T_{i_k})\rangle$; // valid splitting
5 **for** $(\sigma_j, T_{i_j}){\in}\langle(\sigma_1, T_{i_1}), \ldots, (\sigma_k, T_{i_k})\rangle$ **do**
6 $\gamma_j \leftarrow$approx. alignment for σ_j and T_{i_j}; // recursion
7 $\gamma \leftarrow \omega(\sigma, T, \langle\gamma_1, \ldots, \gamma_k\rangle)$; // composing
8 **return** γ;



(a) Trace splitting and assignment (b) Alignment composition

Fig. 3: Overview of the two main actions of the approximation approach

5 Alignment Approximation Approach

Here, we describe our proposed approach, which is based on the formal framework introduced. First, we present an overview. Subsequently, we present specific strategies for splitting/assigning and composing for each process tree operator.

5.1 Overview

For splitting a trace and assigning sub-traces to subtrees many options exist. Moreover, it is inefficient to try out all possible options. Hence, we use a *heuristic* that guides the splitting/assigning. For each subtree, we calculate four characteristics: the activity labels A, if the empty trace is in the subtree's language, possible start-activities SA and end-activities EA of traces in the subtree's language. Thus, each subtree is a *gray-box* since only limited information is available.

Consider the trace to be aligned $\sigma{=}\langle d, c, a, b, c, d, a, e\rangle$ and the two subtrees of T_0 with corresponding characteristics depicted in Figure 3a. Since T_0's root node is a sequence operator, we need to split σ once to obtain two subtraces according to the semantics. Thus, we have 9 potential splittings positions: $\langle|_1\ d\ |_2\ c\ |_3\ a\ |_4\ b\ |_5\ c\ |_6\ d\ |_7\ a\ |_8\ e\ |_9\rangle$. If we split at position 1, we assign $\sigma_1{=}\langle\rangle$ to the first subtree T_1 and the remaining trace $\sigma_2{=}\sigma$ to T_2. Certainly, this is not a good decision since we know that $\langle\rangle{\notin}\mathcal{L}(T_1)$, the first activity of σ_2 is not a start activity of T_2 and the activities b, c, d occurring in σ_2 are not in T_2.

Assume we split at position 7 (Figure 3a). Then we assign $\sigma_1{=}\langle d, c, a, b, c, d\rangle$ to T_1. All activities in σ_1 are contained in T_1, σ_1 starts with $d{\in}SA(T_1)$ and ends with $d{\in}EA(T_1)$. Further, we obtain $\sigma_2{=}\langle a, e\rangle$ whose activities can be replayed in T_2, and start- and end-activities match, too. Hence, according to the gray-box-view, splitting at position 7 is a good choice. Next, assume we receive two alignments γ_1 for T_1, σ_1 and γ_2 for T_2, σ_2 (Figure 3b). Since T_1 is executed before T_2, we concatenate the sub-alignments $\gamma{=}\gamma_1{\cdot}\gamma_2$ and obtain an alignment for T_0.

5.2 Calculation of Process Tree Characteristics

In this section, we formally define the computation of the four tree characteristics for a given process tree $T{=}(V, E, \lambda, r)$. We define the activity set A as a function, i.e., $A{:}\mathcal{T}{\to}\mathcal{P}(\mathcal{A})$, with $A(T){=}\{\lambda(n) \mid n{\in}T^L, \lambda(n){\neq}\tau\}$. We recursively define the possible start- and end-activities as a function, i.e., $SA{:}\mathcal{T}{\to}\mathcal{P}(\mathcal{A})$ and $EA{:}\mathcal{T}{\to}\mathcal{P}(\mathcal{A})$. If T is not a leaf node, we refer to its two subtrees as T_1 and T_2.

$$SA(T){=}\begin{cases}\{\lambda(r)\} & \text{if } \lambda(r){\in}\mathcal{A}\\ \emptyset & \text{if } \lambda(r){=}\tau\\ SA(T_1) & \text{if } \lambda(r){=}{\to}\wedge\langle\rangle{\notin}\mathcal{L}(T_1)\\ SA(T_1){\cup}SA(T_2) & \text{if } \lambda(r){=}{\to}\wedge\langle\rangle{\in}\mathcal{L}(T_1)\\ SA(T_1){\cup}SA(T_2) & \text{if } \lambda(r){\in}\{\wedge,\times\}\\ SA(T_1) & \text{if } \lambda(r){=}{\circlearrowleft}\wedge\langle\rangle{\notin}\mathcal{L}(T_1)\\ SA(T_1){\cup}SA(T_2) & \text{if } \lambda(r){=}{\circlearrowleft}\wedge\langle\rangle{\in}\mathcal{L}(T_1)\end{cases}\quad EA(T){=}\begin{cases}\{\lambda(n)\} & \text{if } \lambda(r){\in}\mathcal{A}\\ \emptyset & \text{if } \lambda(r){=}\tau\\ EA(T_2) & \text{if } \lambda(r){=}{\to}\wedge\langle\rangle{\notin}\mathcal{L}(T_2)\\ EA(T_1){\cup}EA(T_2) & \text{if } \lambda(r){=}{\to}\wedge\langle\rangle{\in}\mathcal{L}(T_2)\\ EA(T_1){\cup}EA(T_2) & \text{if } \lambda(r){\in}\{\wedge,\times\}\\ EA(T_1) & \text{if } \lambda(r){=}{\circlearrowleft}\wedge\langle\rangle{\notin}\mathcal{L}(T_1)\\ EA(T_1){\cup}EA(T_2) & \text{if } \lambda(r){=}{\circlearrowleft}\wedge\langle\rangle{\in}\mathcal{L}(T_1)\end{cases}$$

The calculation whether the empty trace is accepted can also be done recursively.

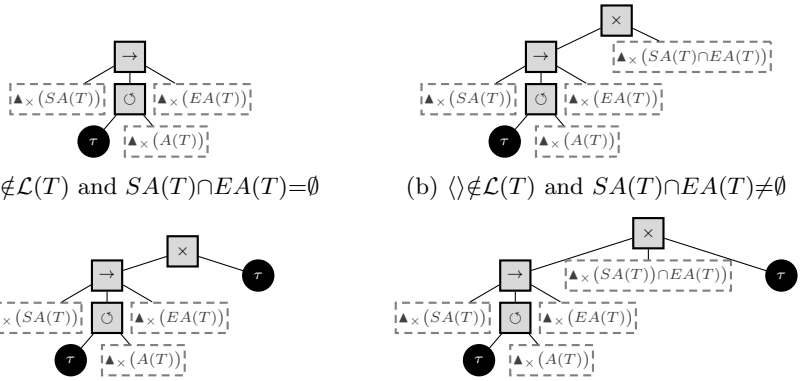
- $\lambda(r){=}\tau \Rightarrow \langle\rangle{\in}\mathcal{L}(T)$ and $\lambda(r){\in}\mathcal{A} \Rightarrow \langle\rangle{\notin}\mathcal{L}(T)$
- $\lambda(r){\in}\{{\to},\wedge\} \Rightarrow \langle\rangle{\in}\mathcal{L}(T_1) \wedge \langle\rangle{\in}\mathcal{L}(T_2) \Leftrightarrow \langle\rangle{\in}\mathcal{L}(T)$
- $\lambda(r){\in}\times \Rightarrow \langle\rangle{\in}\mathcal{L}(T_1) \vee \langle\rangle{\in}\mathcal{L}(T_2) \Leftrightarrow \langle\rangle{\in}\mathcal{L}(T)$
- $\lambda(r){=}{\circlearrowleft} \Rightarrow \langle\rangle{\in}\mathcal{L}(T_1) \Leftrightarrow \langle\rangle{\in}\mathcal{L}(T)$

5.3 Interpretation of Process Tree Characteristics

The decision where to split a trace and the assignment of sub-traces to subtrees is based on the four characteristics per subtree and the process tree operator. Thus, each subtree is a gray-box for the approximation approach since only limited information is available. Subsequently, we explain how we interpret the subtree's characteristics and how we utilize them in the splitting/assigning decision.

Consider Figure 4 showing how the approximation approach assumes a given subtree T behaves based on its four characteristics, i.e., $A(T), SA(T), EA(T)$, $\langle\rangle{\in}\mathcal{L}(T)$. The most liberal *interpretation* $\mathcal{I}(T)$ of a subtree T can be considered as a heuristic that guides the splitting/assigning. The interpretation $\mathcal{I}(T)$ depends on two conditions, i.e., if $\langle\rangle{\in}\mathcal{L}(T)$ and whether there is an activity that is both, a start- and end-activity, i.e., $SA(T){\cap}EA(T){\neq}\emptyset$. Note that $\mathcal{L}(T){\subseteq}\mathcal{L}(\mathcal{I}(T))$ holds. Thus, the interpretation is an approximated view on the actual subtree.

In the next sections, we present for each tree operator a splitting/assigning and composing strategy based on the presented subtree interpretation. All strategies return a splitting per recursive call that minimizes the overall edit distance between the sub-traces and the closest trace in the language of the interpretation



(a) $\langle\rangle \notin \mathcal{L}(T)$ and $SA(T) \cap EA(T) = \emptyset$ (b) $\langle\rangle \notin \mathcal{L}(T)$ and $SA(T) \cap EA(T) \neq \emptyset$

(c) $\langle\rangle \in \mathcal{L}(T)$ and $SA(T) \cap EA(T) = \emptyset$ (d) $\langle\rangle \in \mathcal{L}(T)$ and $SA(T) \cap EA(T) \neq \emptyset$

Fig. 4: Most liberal interpretation $\mathcal{I}(T)$ of the four characteristics of a process tree $T \in \mathcal{T}$. For a set $X = \{x_1, \ldots, x_n\}$, $\blacktriangle_\times(X)$ represents the tree $\times(x_1, \ldots, x_n)$

of the assigned subtrees. For $\sigma_1, \sigma_2 \in \mathcal{A}^*$, let $\updownarrow(\sigma_1, \sigma_2) \in \mathbb{N} \cup \{0\}$ be the Levenshtein distance [16]. For given $\sigma \in \mathcal{A}^*$ and $T \in \mathcal{T}$, we calculate a valid splitting $\psi(\sigma, T) = \big\langle (\sigma_1, T_{i_1}), \ldots, (\sigma_j, T_{i_k}) \big\rangle$ w.r.t. Eq. (1) s.t. the sum depicted below is minimal.

$$\sum_{j \in \{1,\ldots,k\}} \big(\min_{\sigma' \in \mathcal{I}(T_{i_j})} \updownarrow(\sigma_j, \sigma') \big) \tag{3}$$

In the upcoming sections, we assume a given trace $\sigma = \langle a_1, \ldots, a_n \rangle$ and a process tree $T = (V, E, \lambda, r)$ with subtrees referred to as T_1 and T_2.

5.4 Approximating on Choice Operator

The choice operator is the most simple one since we just need to assign σ to one of the subtrees according to the semantics, i.e., assigning σ either to T_1 or T_2. We compute the edit distance of σ to the closest trace in $\mathcal{I}(T_1)$ and in $\mathcal{I}(T_2)$ and assign σ to the subtree with smallest edit distance according to Eq. (3).

Composing an alignment for the choice operator is trivial. Assume we eventually get an alignment γ for the chosen subtree, we just return γ for T.

5.5 Approximating on Sequence Operator

When splitting on a sequence operator, we must assign a sub-trace to each subtree according to the semantics. Hence, we calculate two sub-traces: $\langle (\sigma_1, T_1), (\sigma_2, T_2) \rangle$ s.t. $\sigma_1 \cdot \sigma_2 = \sigma$ according to Eq. (3). The optimal splitting/assigning can be defined as an optimization problem, i.e., Integer Linear Programming (ILP).

In general, for a trace with length n, $n+1$ possible splitting-positions exist: $\langle |_1\ a_1\ |_2\ a_2\ |_3\ \ldots\ |_n\ a_n\ |_{n+1} \rangle$. Assume we split at position 1, this results in $\big\langle (\langle\rangle, T_1), (\sigma, T_2) \big\rangle$, i.e., we assign $\langle\rangle$ to T_1 and the original trace σ to T_2.

Composing the alignment from sub-alignments is straightforward. In general, we eventually obtain two alignments, i.e, $\langle \gamma_1, \gamma_2 \rangle$, for T_1 and T_2. We compose the alignment γ for T by concatenating the sub-alignments, i.e., $\gamma{=}\gamma_1{\cdot}\gamma_2$.

5.6 Approximating on Parallel Operator

According to the semantics, we must define a sub-trace for each subtree, i.e., $\langle (T_1, \sigma_1), (T_2, \sigma_2) \rangle$. In contrast to the sequence operator, $\sigma_1{\cdot}\sigma_2{=}\sigma$ does *not* generally hold. The splitting/assignment w.r.t. Eq. (3) can be defined as an ILP. In general, each activity can be assigned to one of the subtrees independently.

For example, assume $\sigma{=}\langle c, a, d, c, b \rangle$ and $T \mathrel{\widehat{=}} \wedge\big(\rightarrow(a, b), \circlearrowleft(c, d)\big)$ with subtree $T_1 \mathrel{\widehat{=}} \rightarrow(a, b)$ and $T_2 \mathrel{\widehat{=}} \circlearrowleft(c, d)$. Below we assign the activities to subtrees.

$$\begin{array}{ccccccc} \langle & c, & a, & d, & c, & b & \rangle \\ & T_2 & T_1 & T_2 & T_2 & T_1 & \end{array}$$

Based on the assignment, we create two sub-traces: $\sigma_1{=}\langle a, b \rangle$ and $\sigma_2{=}\langle c, d, c \rangle$. Assume that $\gamma_1 \mathrel{\widehat{=}} \langle (a, a), (b, b) \rangle$ and $\gamma_2 \mathrel{\widehat{=}} \langle (c, c), (d, d), (c, c) \rangle$ are the two alignments eventually obtained. To compose an alignment for T, we have to consider the assignment. Since the first activity c is assigned to T_2, we extract the corresponding alignment steps from γ_1 until we have explained c. The next activity in σ is an a assigned to T_1. We extract the alignment moves from γ_1 until we explained the a. We iteratively continue until all activities in σ are covered. Finally, we obtain an alignment for T and σ, i.e., $\gamma \mathrel{\widehat{=}} \big\langle (c, c), (a, a), (d, d), (c, c), (b, b) \big\rangle$.

5.7 Approximating on Loop Operator

We calculate $m{\in}\{1, 3, 5, \dots\}$ sub-traces that are assigned alternately to the two subtrees: $\langle (\sigma_1, T_1), (\sigma_2, T_2), (\sigma_3, T_1), \dots, (\sigma_{m-1}, T_2), (\sigma_m, T_1) \rangle$ s.t. $\sigma{=}\sigma_1{\cdot}\dots{\cdot}\sigma_m$. Thereby, σ_1 and σ_m are always assigned to T_1. Next, we visualize all possible splitting positions for the given trace: $\langle |_1\, a_1\, |_2\, |_3\, a_2\, |_4 \dots |_{2n-1}\, a_n\, |_{2n} \rangle$. If we split at each position, we obtain $\big\langle \big(\langle\rangle, T_1\big), \big(\langle a_1 \rangle, T_2\big), \big(\langle\rangle, T_1\big), \dots, \big(\langle a_n \rangle, T_2\big), \big(\langle\rangle, T_1\big) \big\rangle$. The optimal splitting/assignment w.r.t Eq. (3) can be defined as an ILP.

Composing an alignment is similar to the sequence operator. In general, we obtain m sub-alignments $\langle \gamma_1, \dots, \gamma_m \rangle$, which we concatenate, i.e., $\gamma{=}\gamma_1{\cdot}\dots{\cdot}\gamma_m$.

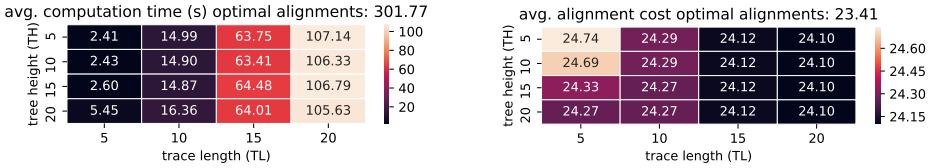
6 Evaluation

This section presents an experimental evaluation of the proposed approach.

We implemented the proposed approach in PM4Py[3], an open-source process mining library. We conducted experiments on real event logs [17,18]. For each log, we discovered a process tree with the Inductive Miner infrequent algorithm [10].

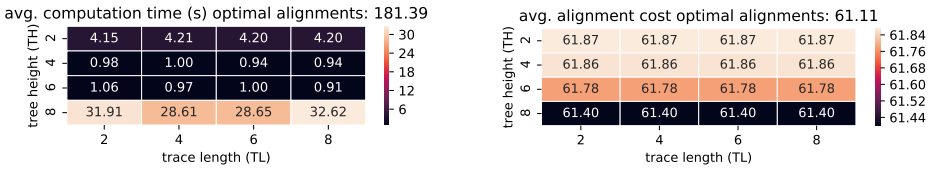
In Figures 5 and 6, we present the results. We observe that our approach is on average always faster than the optimal alignment algorithm for all tested parameter settings. Moreover, we observe that our approach never underestimates

[3] https://pm4py.fit.fraunhofer.de/



(a) Avg. computation time (s) (b) Avg. alignment costs

Fig. 5: Results for [17], sample: 100 variants, tree height 24, avg. trace length 28



(a) Avg. computation time (s) (b) Avg. alignment costs

Fig. 6: Results for [18], sample: 100 variants, tree height 10, avg. trace length 65

the optimal alignment costs, as our approach returns a valid alignment. W.r.t. optimization problems for optimal splittings/assignments, consider parameter setting TH:5 and TL:5 in Figure 5. This parameter setting results in the highest splitting along the tree hierarchy and the computation time is the lowest compared to the other settings. Thus, we conclude that solving optimization problems for finding splittings/assignments is appropriate. In general, we observe a good balance between accuracy and computation time. We additionally conducted experiments with a decomposition approach [15] (available in ProM[4]) and compared the calculation time with the standard alignment implementation (LP-based) [12] in ProM. Consider Table 2. We observe that the decomposition approach does not yield a speed-up for [17] but for [18] we observe that the decomposition approach is about 5 times faster. In comparison to Figure 6a, however, our approach yields a much higher speed-up.

7 Conclusion

We introduced a novel approach to approximate alignments for process trees. First, we recursively split a trace into sub-traces along the tree hierarchy based

[4] http://www.promtools.org/

Table 2: Results for decomposition based alignments

Approach	[17] (sample: 100 variants)	[18] (sample: 100 variants)
decomposition [15]	25.22 s	20.96 s
standard [12]	1.51 s	103.22 s

on a gray-box view on the respective subtrees. After splitting, we compute optimal sub-alignments. Finally, we recursively compose a valid alignment from sub-alignments. Our experiments show that the approach provides a good balance between accuracy and calculation time. Apart from the specific approach proposed, the contribution of this paper is the formal framework describing how alignments can be approximated for process trees. Thus, many other strategies besides the one presented are conceivable.

References

1. W. M. P. van der Aalst, *Process Mining - Data Science in Action*. Springer, 2016.
2. W. M. P. van der Aalst, A. Adriansyah, and B. F. van Dongen, "Replaying history on process models for conformance checking and performance analysis," *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.*, vol. 2, no. 2, 2012.
3. J. Carmona, B. F. van Dongen, A. Solti, and M. Weidlich, *Conformance Checking - Relating Processes and Models*. Springer, 2018.
4. W. L. J. Lee, H. M. W. Verbeek, J. Munoz-Gama, W. M. P. van der Aalst, and M. Sepúlveda, "Recomposing conformance: Closing the circle on decomposed alignment-based conformance checking in process mining," *Inf. Sci.*, vol. 466, 2018.
5. M. F. Sani, S. J. van Zelst, and W. M. P. van der Aalst, "Conformance checking approximation using subset selection and edit distance," in *CAiSE 2020*, ser. LNCS, vol. 12127. Springer, 2020.
6. F. Taymouri and J. Carmona, "An evolutionary technique to approximate multiple optimal alignments," in *BPM 2018*, ser. LNCS, vol. 11080. Springer, 2018.
7. ——, "Model and event log reductions to boost the computation of alignments," in *SIMPDA 2016*, vol. 1757. CEUR-WS.org, 2016.
8. M. Bauer, H. van der Aa, and M. Weidlich, "Estimating process conformance by trace sampling and result approximation," in *BPM 2019*, ser. LNCS, vol. 11675. Springer, 2019.
9. S. J. J. Leemans, "Robust process mining with guarantees," Ph.D. dissertation, Department of Mathematics and Computer Science, 2017.
10. S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour," in *BPM Workshops 2013*, ser. LNBIP, vol. 171. Springer, 2013.
11. D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst, "Incremental discovery of hierarchical process models," in *RCIS 2020*, ser. LNBIP, vol. 385. Springer, 2020.
12. A. Adriansyah, "Aligning Observed and Modeled Behavior," Ph.D. dissertation, Eindhoven University of Technology, 2014.
13. B. F. van Dongen, "Efficiently computing alignments - using the extended marking equation," in *BPM 2018*, ser. LNCS, vol. 11080. Springer, 2018.
14. B. F. van Dongen, J. Carmona, T. Chatain, and F. Taymouri, "Aligning modeled and observed behavior: A compromise between computation complexity and quality," in *CAiSE 2017*, ser. LNCS, vol. 10253. Springer, 2017.
15. W. M. P. van der Aalst, "Decomposing Petri Nets for Process Mining: A Generic Approach," *Distributed and Parallel Databases*, no. 4, 2013.
16. V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966.
17. B. F. van Dongen, "BPI Challenge 2019. Dataset," 2019.
18. B. F. van Dongen and F. Borchert, "BPI Challenge 2018. Dataset," 2018.

Stochastic Process Discovery By Weight Estimation

Adam Burke (✉), Sander J. J. Leemans, and Moe Thandar Wynn

Queensland University of Technology, Brisbane, Australia,
at.burke@qut.edu.au,s.leemans@qut.edu.au,m.wynn@qut.edu.au

Abstract. Many algorithms now exist for discovering process models from event logs. These models usually describe a control flow and are intended for use by people in analysing and improving real-world organizational processes. The relative likelihood of choices made while following a process (i.e., its stochastic behaviour) is highly relevant information which few existing algorithms make available in their automatically discovered models. This can be addressed by automatically discovered stochastic process models.
We introduce a framework for automatic discovery of stochastic process models, given a control-flow model and an event log. The framework introduces an estimator which takes a Petri net model and an event log as input, and outputs a Generalized Stochastic Petri net. We apply the framework, adding six new weight estimators, and a method for their evaluation. The algorithms have been implemented in the open-source process mining framework ProM. Using stochastic conformance measures, the resulting models have comparable conformance to existing approaches and are shown to be calculated more efficiently.

Key words: Stochastic Petri nets, process mining, stochastic process mining, stochastic process discovery

1 Introduction

The world abounds in information systems, generating data about the processes they mediate, execute, or observe. Using this data to compute and analyze process models is the concern of process mining [3], within the field of Business Process Management (BPM). BPM studies the impact and improvement of processes in organizations. Automatic process discovery is one aspect of process mining concerned with finding a formal process model computationally from an input event log.

To understand a process, we often want to know how likely an event is. If we travel to work, a journey where our train reliably arrives on time is different from one where the train sometimes breaks down, is sometimes replaced by a bus, or is often so crowded that it's quicker to ride a bike. A highly contagious disease with rare side effects differs importantly from one difficult to transmit but with severe side effects, even if observable symptoms are similar. Detecting fraud in financial transactions depends on recognizing certain client actions happening more frequently than usual. Existing process mining techniques already recognize

this: where noise or probability is considered in creating control flows (e.g. [30, 19]), they acknowledge the importance of likelihood in process modeling. Better stochastic representations and stochastic-aware techniques have been flagged as a key research challenge for process mining [2].

Process discovery techniques have become quite sophisticated at determining causal relationships between activities from event logs, and representing that in process models. There are far fewer techniques for discovering relative probabilities (discussed in Section 5). We introduce a framework in Section 3 which leverages this by allowing transformation of models with only control flows into stochastic process models. This extends an existing stochastic process discovery technique by Rogge-Solti et al (RSD) [25, 26], in two ways. Firstly, it generalizes one estimation algorithm to a general class of *weight estimators*. Secondly, it specializes the possible outputs from general probability distributions to Generalized Stochastic Petri Nets (GSPNs) [4]. The framework does not prescribe whether the estimation calculation is deterministic, uses stochastic simulation, or other techniques, and our introduced estimators include both deterministic and non-deterministic types.

We describe our approach as a form of Stochastic Process Discovery, as it takes an event log input and produces a GSPN output. In decoupling weight estimation from control flow discovery, the technique also shares some features with process model enhancement for time and probability [3, p290]. Unlike enhancement techniques, estimators can potentially change control flows when producing a stochastic process model. Stochastic process models have a corresponding, emerging, set of stochastic process conformance measures [20, 21, 16]. Consequently, the algorithms and models presented here are evaluated, in Section 4, as stochastic process discovery algorithms, using stochastic process conformance measures. Evaluation, which also includes performance, is against real-life event logs, multiple control flow discovery algorithms, and RSD [25].

In the next section, we introduce existing concepts. In Section 3, we describe the weight estimation framework and instantiate it by introducing novel estimators. In Section 4, the results of using the estimators on real-world event logs are presented. Related work is reviewed in Section 5, and Section 6 concludes the paper.

2 Preliminaries

Petri nets and Generalized Stochastic Petri Nets are well-established formalisms for modelling processes and a number of good overviews exist [4, 8]. We use notations from the process mining literature [3, 21].

A Petri net is a tuple $PN = (P, T, F, M_0)$, where P is a finite set of places, T is a finite set of transitions, and $F : (P \times T) \rightarrow (T \times P)$ is a flow relation. A *marking* is a multiset of places $\subseteq P$ that indicate a state of the Petri net, with M_0 the initial marking. A transition is enabled if every incoming place contains a token. A transition fires by changing the marking of the net to consume incoming tokens and producing tokens for its outgoing transitions. For a node $n \in P \cup T$, we define $\bullet n = \{y \mid (y, x) \in F\}$ and $n\bullet = \{y \mid (x, y) \in F\}$.

A *Generalized Stochastic Petri Net (GSPN)* is a tuple $(P, T, F, M_0, W, T_i, T_t)$ such that $T_i \subseteq T$, $T_t \subseteq T$ and $T_i \cap T_t = \emptyset$. Weight function $W : T \to \mathbb{R}^+$ assigns each transition a weight. T_i is a set of immediate transitions. If multiple transitions $T_e \subseteq T_i$ are enabled in a particular marking, the probability of a transition $t \in T_i$ firing is given by $\frac{W(t)}{\Sigma_{t' \in T_e} W(t')}$. T_t is a set of timed transitions. Immediate transitions take priority over timed transitions. A timed transition, if enabled, fires according to an exponentially distributed wait time. Given a set of enabled timed transitions $T_e \subseteq T_t$, a particular transition t fires first with probability $\frac{W(t)}{\Sigma_{t' \in T_e} W(t')}$ [4].

Event logs. A process consists of activities from the set $\mathcal{A}$. A trace is a non-empty sequence of activities, and an event log L is a finite multiset of traces observing the underlying process. Partial function $\lambda : T \to \mathcal{A}$ designates labels for Petri net transitions that represent log activities. The number of traces in a log L is denoted with $|L|$, while the the number of events is denoted with $||L||$.

Control Flow Process Discovery. A process discovery algorithm for Petri Nets is then defined by $cfd : L \to (P, T, F, M_0)$.

Sequence operations. A finite sequence over $\mathcal{A}$ of length n is a mapping $\sigma \in \{1..n\} \to \mathcal{A}$ and denoted by $\sigma = \langle a_1, a_2, ..., a_n \rangle$ where $\forall_i a_i = \sigma(i)$. Concatenation operator $+$ appends one sequence to another such that $\langle a_1, ..., a_n \rangle + \langle b_1, ..., b_m \rangle = \langle a_1, ...a_n, b_1, ..., b_m \rangle$. The tail function is then $tail(\langle a \rangle + \sigma) = \sigma$.

Subsequence. Function ct returns the number of times a subsequence is present in a sequence: $\mathrm{ct}(\varsigma, \sigma) = \begin{cases} 0 & \text{if } \sigma = \langle\rangle \\ 1 + \mathrm{ct}(\varsigma, tail(\sigma)) & \text{if } \sigma = \varsigma + x \\ \mathrm{ct}(\varsigma, tail(\sigma)) & \text{if } \sigma \neq \varsigma + x \end{cases}$

Alignments. An alignment [1] represents paired paths between a log and a model. That is, a move is a tuple where (a, t) represents a synchronous move on activity a in a trace and a transition t in the model (with the same label: $\lambda(t) = a$), $(a, \perp)$ represents a log move, and $(\perp, t)$ represents a model move. For our purposes, we assume that a function γ is available taking a Petri net, a set of final markings and an event log, and that γ returns a sequence of move tuples that represent all moves necessary to align every trace in the log.

3 Stochastic Process Model Weight Estimation

In this section, we first introduce our framework to transform a Petri net into a GSPN using an event log. Then, we introduce six estimators using the framework, which we will illustrate using the running example shown in Figure 2. Estimators are a large solution space with many potential algorithms. Our six estimators are chosen to emphasize broad applicability of inputs, computational tractability, using the implicit causal information in control flow models, and reapplying established process mining concepts.

3.1 A Framework for GSPN Discovery

The framework defines functions which together transform an event log into a GSPN, as shown in Figure 1.

A stochastic process discovery algorithm for GSPNs (*mine_spn*) is a function $mine\_spn : L \rightarrow (P, T, F, M_0, W, T_i, T_t)$. Our framework considers functions of the form $mine\_spn = est(cfd(L), L)$. Functions $est : L \times (P, T, F, M_0) \rightarrow (P, T, F, M_0, W, T_i, T_t)$ are termed *estimators*.

Functions $se : L \times (P, T, F, M_0) \rightarrow T \times \mathbb{R}^+$ are *simple weight estimators* and use the control flow of the input Petri net intact in the output Petri net, such that for discovered control flow model $cfd(L) = (P_d, T_d, F_d, Md_0)$,

$$\exists_{pe \in est} pe = (P_d, T_d, F_d, Md_0, se(L, (P_d, T_d, F_d, Md_0)), T_d, \emptyset)$$

The estimators discussed next are of this simpler form.

Specific estimators may have further restrictions on their inputs, or provide guarantees on their outputs. For example, estimators discussed below do not distinguish transitions with duplicate labels. A challenge common to several estimators is treatment of silent transitions, as those transitions in a discovered model serve a structural role and do not directly represent an activity in the log. Assigning such a transition a weight of zero in a stochastic net is equivalent to deleting the transition, and all subsequent model paths. To avoid this impact, default values are assigned to silent transitions where the calculation would otherwise result in zero weights. In general, estimators make no distinction between silent transitions and transitions without a corresponding activity in the log. In the remainder of this section, we introduce several examples of estimators that instantiate this framework.

3.2 Frequency Estimator

The first estimator, w_{freq}, straightforwardly uses how often each transition t appeared in the event log L:

$$w_{freq}(L, t) = \max(1, \Sigma_{\sigma \in L} \, \mathrm{ct}(\langle \lambda(t) \rangle, \sigma))$$

Silent transitions are assigned the arbitrary weight of 1, equivalent to a single observation in the log. The complexity of this estimator is linear in the number of events in the log. Figure 2c shows the results of this estimator on our running example, e.g. $w_{freq}(EL, b) = 15$.

3.3 Activity-Pair Frequency Estimators

An Activity-Pair Estimator uses the frequency of pairs of successor activities to better reflect the constraints of more general Petri nets. These are *edge-structured estimators*, in that Petri net edges inform the weighting.

We first introduce some frequency definitions. The functions q_I and q_F capture how often an activity appears as the first/last in a trace. The function q_P captures the frequency of activity pairs in the log, that is, where the two given activities follow one another directly in the log:

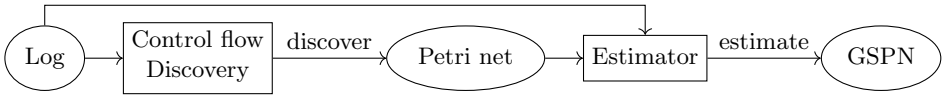
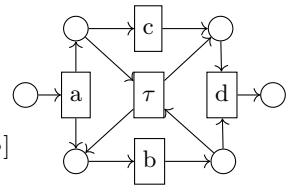


Fig. 1: Our framework for GSPN Discovery.

$[\langle a,b,c,d\rangle^5,$
$\langle a,c,b,d\rangle^4,$
$\langle a,b,b,d\rangle^2,$
$\langle a,b,c,b,d\rangle]$



(a) Log EL. (b) Petri net EPN.

	w_{freq}	w_{lhpair}	w_{rhpair}	$w_{pairscale}$	w_{fork}	w_{align}
a	12	12	12	$1\frac{11}{49}$	12	12
b	15	8	7	$\frac{35}{49}$	8	14
c	10	4	5	$1\frac{1}{49}$	$4\frac{10}{11}$	9
d	12	12	12	$1\frac{11}{49}$	$11\frac{6}{13}$	12
τ	1	1	1	1	$\frac{129}{143}$	0

(c) Six example estimators.

Fig. 2: Running example of an event log and a Petri net, and the estimators.

$$q_I(L,t) = |[\langle \lambda(t), \ldots\rangle \in L]|$$
$$q_F(L,t) = |[\langle \ldots, \lambda(t)\rangle \in L]|$$
$$q_P(L,s,t) = \Sigma_{\sigma \in L}\, \mathrm{ct}(\langle \lambda(s), \lambda(t)\rangle, \sigma)$$

There are both left-handed and right-handed variants of the Activity-Pair estimator, depending on whether weights are informed by successor or predecessor transitions, defined as:

$$w_{lhpair}(L,t) = \max(1, q_I(L,t) + q_F(L,t) + \sum_{s \in \bullet(\bullet t)} q_P(L,s,t))$$

$$w_{rhpair}(L,t) = \max(1, q_I(L,t) + q_F(L,t) + \sum_{s \in (t\bullet)\bullet} q_P(L,t,s))$$

There are no restrictions on input Petri nets and they can be calculated in time $O(||L|||F|)$, that is, the number of events times the number of model edges.

When using activity pair frequency data, two important types of path through the model are neglected for any given trace: paths from the initial place to the first transition, and the paths from the last transition to the final place. Traces of length one are also invisible from this perspective. To account for this, how often an activity appears as the initial or final activity in a trace is also included in the weight estimation. Note that not all activity pairs occurring in the log are used to calculate the resulting transition weights. For instance, where a given Petri net represents two transitions a and b as concurrent, the frequency of $\langle a,b\rangle$ will not be used. In our running example (see Figure 2c), $w_{lhpair}(EL,c) = 4$ and $w_{rhpair}(EL,c) = 5$.

3.4 Mean-Scaled Activity-Pair Frequency Estimator

The previous estimators depend on the size of the log. Two logs with the same traces in the same ratios will result in two models with two distinct sets of weights, which challenges human analysis. Though comparison and comprehensibility of stochastic process models appears not to have been directly addressed in the literature, it is consistent with research that finds "small variations between models can lead to significant differences in their comprehensibility" [24] and the usability principle of minimizing user memory load. The mean-scaled activity-pair estimator $w_{pairscale}$ mitigates this effect by scaling weights by average transition frequency ($\frac{||L||}{|T|}$) in the log L:

$$pairscale(L,T,t) = \frac{q_I(L,t) + q_F(L,t) + \sum_{s \in (t^\bullet)^\bullet} q_P(L,t,s)}{\frac{||L||}{|T|}}$$

$$w_{pairscale}(L,(P,T,F,M_0),t) = \begin{cases} pairscale(L,T,t) & \text{if } pairscale(t) \neq 0 \\ 1 & \text{otherwise} \end{cases}$$

One effect of defaulting after scaling is that silent or unrepresented transitions are weighted more heavily, that is, the same as an activity of mean-frequency, rather than the equivalent of an activity occurring once in the log. In our running example of Figure 2c, $||L|| = 49$, $|T| = 5$ and the numerator of *pairscale* is equal to w_{rhpair} for a, b, c and d. Then, for instance $w_{pairscale}$ of c is $\frac{10}{\frac{49}{5}} = 1\frac{1}{49}$.

3.5 Fork Distribution Estimator

The Fork Distribution Estimator w_{fork} uses a two-stage approach: it first assigns weights to each place in a Petri net using activity-pair frequencies. Second, it distributes those weights to transitions according to the activity frequency in the event log.

$$pw(L,p) = \begin{cases} |L| & \text{if } p \in M_0 \\ \Sigma_{s \in \bullet p} \Sigma_{t \in p^\bullet} q_P(L,s,t) & \text{otherwise} \end{cases}$$

$$placeWeights(L,p) = \max(1, pw(L,p))$$

$$w_{fork}(L,(P,T,F,M_0),t) = \Sigma_{p \in \bullet t} placeWeights(L,p) \frac{w_{freq}(t)}{\Sigma_{t'}^{p^\bullet} w_{freq}(t')}$$

This estimator only applies to Petri nets which have at least one place without incoming edges, such as workflow nets [3, p81]. This is an edge-structured estimator informed by the structure of the input net. The complexity is $O(||L||\ |F|)$. The w_{fork} estimator shares similarities with the Alpha algorithm [3, p167], in that it treats a place as defining a neighbourhood of related activities represented as transitions. In our example (Figure 2), let p_1 be the top-right place and p_2 the bottom-right place. Then, $pw(EL,p_1) = q_P(c,d) + q_P(\tau,d) = 5$, $pw(EL,p_2) = q_P(\tau,d) + q_P(b,d) = 7$, $placeWeights(EL,p_1) = 5$, $placeWeights(EL,p_2) = 7$ and w_{fork} of $d = 5\frac{12}{12} + 7\frac{12}{13} = 11\frac{6}{13}$.

3.6 Alignment Estimator

The estimator w_{align} applies alignments [1] to estimate weights. To this end, it counts the number of times a transition t appears either as a model move or as a synchronous move in the alignments:

$$w_{align}(L,PN,M_F,t) = |[(x,t) \in \gamma(PN,M_F,L)]|$$

This algorithm only applies to Petri nets with at least one final marking. The time complexity is $O(|T|\ |\gamma|)$ plus the time to compute γ. The alignment estimator has similarities with RSD [25], which fits duration distributions to aligned logs. In our example of Figure 2, the last trace of log EL does not fit the model EPN, as b is executed a second time and c is executed. Thus,

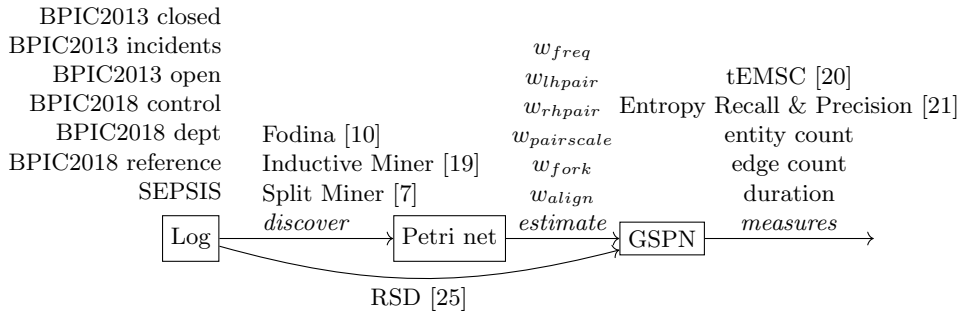


Fig. 3: Set-up of the evaluation.

alignments will (based on a cost function, or if that does not discriminate the options an arbitrary choice) include a log move on either b or a log move on c. If the alignments choose a b for a log move, then $w_{align}(EL, EPN, M_F, b) = 14$ and $w_{align}(EL, EPN, M_F, \tau) = 0$. Alignments are not always deterministic, and consequently neither is w_{align}.

4 Implementation and Evaluation

4.1 Evaluation Design

The six estimators introduced in Section 3 were implemented in the ProM framework [13][1]. For our evaluation, a discovery algorithm was applied to an event log. Where necessary, the result was converted to a Petri net. Each estimator was invoked on the resulting Petri net, resulting in a GSPN. Finally, the conformance of the resulting GSPN was measured against the original log. For comparison, an existing stochastic discovery algorithm by Rogge-Solti et al [25] (RSD) was also applied to the log. This direct discovery algorithm also outputs GSPNs, and the same conformance measures were applied. The implementation of this plugin in ProM 6.9 uses the Inductive Miner internally as an initial control flow discovery step, which has been updated from the gradient-descent procedure described in [25]. Algorithms, reference event logs and conformance measures are summarized as Figure 3.

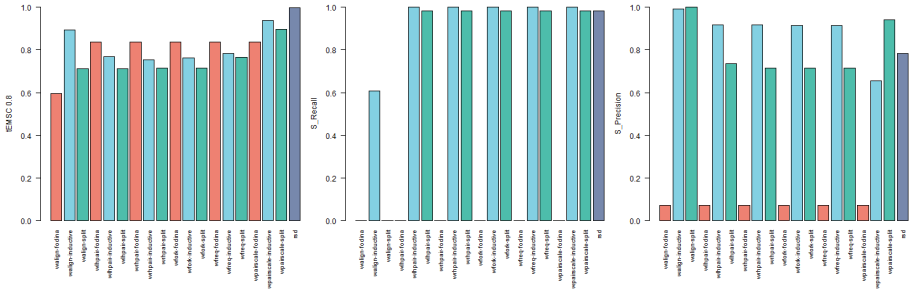
Measures include (1) Truncated Earth Movers' Distance (tEMSC) [20] provides a measure expressing the cost of transforming the distribution of activity traces from one stochastic language into another. We use a minimum probability mass parameter setting of 0.8 for feasibility. (2) Entropy Precision and Recall [21], are stochastic conformance measures based on the entropy of equivalent automata constructed from a given log or model. (3) Petri net entity count (places and transitions) and (4) edge count are used as structural simplicity measures, ensuring that conformance quality has not been achieved by sacrificing model simplicity and comprehensibility. Entity and arc counts have existing uses in process model evaluation [14, 17], and were preferred here over behavioural simplicity measures [16], though these measures also have limitations, including specificity to Petri nets, and insensitivity to the stochastic perspective of GSPNs.

[1] Source code is accessible via `https://github.com/adamburkegh/spd_we`

The duration of a discovery process was also captured, and direct discovery times are compared with combined runtimes for discovery and estimation.

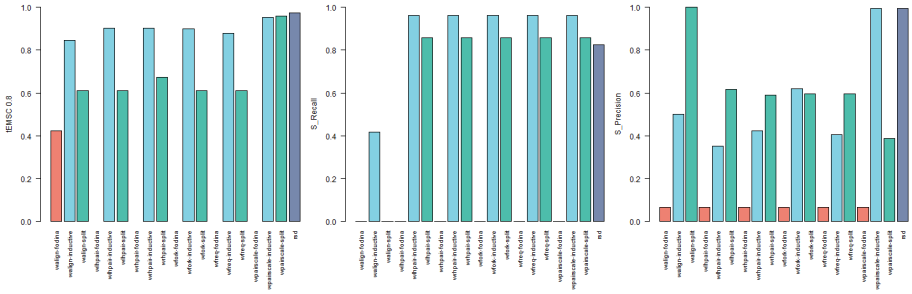
The experiments were run on a Windows 10 machine with 2.3GHz CPU and 50 Gb of memory allocated to each process on JDK 1.8.0_222. All logs are publicly available at `https://data.4tu.nl/`. The full results for these experiments are available in an accompanying technical report [11].

4.2 Results and Discussion



(a) tEMSC (b) entropy-recall (c) entropy-precision

Fig. 4: Results on BPIC 2018 Control log categorized by {*estimator*}-{*control flow algorithm*}, plus RSD.



(a) tEMSC (b) entropy-recall (c) entropy-precision

Fig. 5: Results on BPIC 2018 Reference log.

The estimators produced different, relevant, stochastic models when applied to a range of real-life logs. As seen in Figures 4 and 5, stochastic conformance for these models was comparable, but not uniformly better, than existing techniques, and was highly dependent on the discovery algorithm, and log.

The estimators combined well with the Inductive Miner and Split Miner control discovery algorithms. Frequency-based estimators combined poorly with

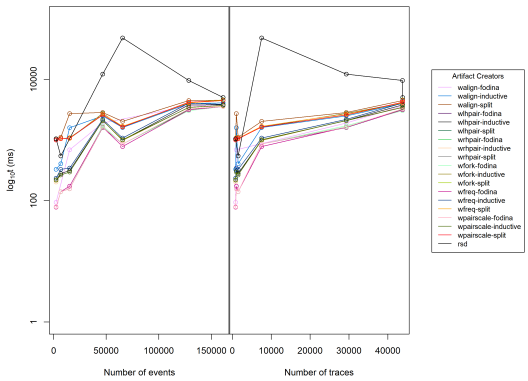


Fig. 6: Run times for control flow discovery and weight estimation by event and trace count. 12 hour time out for RSD [25] on sepsis log is excluded.

the Fodina discovery algorithm for some logs. This is at least partly due to Petri net representational bias in the presented framework. Fodina outputs a causal net, which was converted to a Petri net. The resulting Petri net includes a large number of silent transitions, often intermediating between transitions corresponding to activity pairs in the log. This can be seen distinctly in results for BPIC 2018 reference log in Figure 5, where w_{align} produces a stochastically relevant model on the output of a Fodina input, but no other estimator does. For Split Miner and Inductive Miner, though they use other representations internally, the Petri net model produced used fewer silent transitions and were less impacted by this property.

For the BPIC 2013 closed and incidents logs, Fodina returned a model without an initial place, to which w_{fork}, w_{align}, tEMSC and Entropy-Recall and Entropy-Precision conformance measures do not apply. For some algorithm-estimator combinations, these conformance measures could not be calculated due to soundness, time or memory constraints. Nevertheless, in these results it is clear that tEMSC 0.8 is more sensitive to the stochastic perspective produced by estimators than the Entropy Precision and Recall measures. Where RSD [25] produced a model on which measures could be calculated, the resulting models often conformed well to the logs, but not consistently better than the estimator-produced models. There were a number of event logs where RSD returned no model within the constraints of time (12 hour timeout) and machine memory, or where conformance measures were unable to be calculated within time (5 hour timeout) and memory constraints.

The run time of the estimators, which took never more than 10 seconds, was always comparable or better than RSD, orders of magnitude better in some cases, as shown in Figure 6. In the future, we aim to extend these experiments with larger logs containing more traces, events, and activities. However, even though our estimators returned results for each model and log combination quickly, the conformance measures were the limiting factors in these experiments in terms

of time and memory, which indicates that future research should be directed towards more efficient stochastic conformance checking techniques.

In summary, our new estimators, even the alignment-based w_{align}, are able to handle real-life event logs and outputs from existing discovery techniques much faster than existing approaches. Depending on the applied discovery technique, they can also achieve higher stochastic quality, providing alternatives to the existing RSD discovery technique when analyzing control flow and stochastic perspectives.

5 Related Work

Significant work exists on performance analysis using process mining and Stochastic Petri Nets (SPNs) with pre-existing normative models. This includes improving parameters from an input SPN [22, 29, 26], from models in UML [9], and industrial case studies [26, 9]. These and other applications can benefit directly from automatic discovery of stochastic models.

RSD [25] is a technique, with publicly available implementation, for discovering Generally Distributed Transition Stochastic Petri Nets (GDT_SPNs), with some high level descriptions of techniques and algorithms preceding it [18, 15, 6]. RSD first discovers a control flow model in the form of a Petri net, then performs a fitness calculation, and attempts to repair the model if fitness is low. An alignment and replay calculation then informs the production of an output GDT_SPN. The distinction between control flow discovery and stochastic perspectives is extended by our proposed framework to many possible weight estimators. The post-control flow discovery steps in RSD are a *weight estimator*, but not a *simple estimator*, in our terminology.

In [27, 28], queues are discovered in stochastic process mining using two formalisms, Process Trees [28] and Queue-Enabling Colored Stochastic Petri Nets [27]. The Process Tree approach is informed by statistics theory and uses both Bayesian and Markov-Chain Monte-Carlo fitting.

Hidden Markov Models (HMMs) have seen some applications to stochastic process discovery [12, 5]. For instance, [12] constructs HMMs for resource usage using a variant of the Alpha algorithm [3, p167], an early process mining algorithm with known weaknesses on real-world event data. [5] uses event log data to prune unlikely paths from a HMM process model in the context of a semi-automated stochastic process discovery procedure.

Declarative process models describe a process in terms of constraints on behaviour. This contrasts with control-flow based process models, such as Petri nets used in our framework, which describe permitted behaviour. Techniques for automatic process discovery of probabilistic declarative models also exist [23]. Transforming the significant differences between the forms of control-flow and declarative models, and evaluating the result for stochastic conformance, put rigorous comparison beyond the scope of this paper.

6 Conclusion

The likelihood of an event is important information in understanding many real-world processes. Automatically discovered stochastic process models may

then help analyze and improve organizations. In this paper we presented a framework for discovery of General Stochastic Petri Nets (GSPNs) from logs. The framework leverages existing control flow discovery algorithms, and introduces *estimators* which transform discovered Petri nets into GSPNs. We introduced six estimators; their implementation is publicly available, and evaluated against real-life logs using multiple stochastic conformance measures. The evaluation used three existing flow discovery algorithms, and an existing stochastic discovery technique, finding models of comparable quality, across a broader range of logs, in a generally shorter time.

The estimators presented here are not exhaustive, and we look forward to future research on novel, improved estimators. The estimator framework also implies the possibility of "direct stochastic discovery" algorithms which do not use a separate control flow algorithm, but produce a control flow model as a side-effect of a stochastic one. A simplicity measure sensitive to both structural representation and stochastic information in a process model would be a useful evaluation tool for work in this area, and is an avenue of future research.

Acknowledgement. Computational resources used included those provided by the eResearch Office at QUT.

References

[1] Wil M. P van der Aalst, Arya Adriansyah, and Boudewijn van Dongen. "Replaying history on process models for conformance checking and performance analysis". In: *DMKD* 2.2 (2012), pp. 182–192.

[2] Wil M. P. van der Aalst. "Academic View: Development of the Process Mining Discipline". In: Springer, 2020, pp. 181–196.

[3] Wil van der Aalst. *Process Mining: Data Science in Action*. 2nd ed. Berlin Heidelberg: Springer-Verlag, 2016.

[4] M. Ajmone Marsan et al. "The effect of execution policies on the semantics and analysis of stochastic Petri nets". In: *TSE* (1989).

[5] Amirah Mohammed Alharbi. "Unsupervised Abstraction for Reducing the Complexity of Healthcare Process Models". PhD thesis. University of Leeds, July 2019.

[6] Nikolas Anastasiou and William Knottenbelt. "Deriving coloured generalised stochastic petri net performance models from high-precision location tracking data". In: *PE*. 2013, pp. 375–386.

[7] Adriano Augusto et al. "Split miner: automated discovery of accurate and simple business process models from event logs". In: *KaIS* (2019).

[8] F. Bause and P.S. Kritzinger. *Stochastic Petri Nets: An Introduction to the Theory*. Vieweg+Teubner Verlag, 2002.

[9] Simona Bernardi et al. "A systematic approach for performance evaluation using process mining: the POSIDONIA operations case study". In: QUDOS. 2016, pp. 24–29.

[10] Seppe K. L. M. vanden Broucke et al. "Fodina: A robust and flexible heuristic process discovery technique". In: *DSS* (2017), pp. 109–118.

[11] Adam Burke et al. *Report On Stochastic Process Discovery By Weight Estimation Experimental Results*. Tech. rep. `https://eprints.qut.edu.au/204662/`. Sept. 2020.
[12] Berny Carrera et al. "Constructing probabilistic process models based on hidden Markov models for resource allocation". In: *BPM*. 2014.
[13] Boudewijn F. van Dongen et al. "The ProM Framework: A New Era in Process Mining Tool Support". In: *Petri Nets*. 2005, pp. 444–454.
[14] Volker Gruhn and Ralf Laue. "Adopting the Cognitive Complexity Measure for Business Process Models". In: *CI*. 2006, pp. 236–241.
[15] Haiyang Hu, Jianen Xie, and Hua Hu. "A novel approach for mining stochastic process model from workflow logs". In: *JCIS* (2011).
[16] Anna Kalenkova et al. "A Framework for Estimating Simplicity of Automatically Discovered Process Models Based on Structural and Behavioral Characteristics". In: *ICPM*. 2020.
[17] Krzysztof Kluza et al. "Square Complexity Metrics for Business Process Models". In: *ABICT*. Springer, 2014, pp. 89–107.
[18] Edouard Leclercq et al. "Identification of timed stochastic Petri net models with normal distributions of firing periods". In: *IFAC* (2009).
[19] Sander J. J. Leemans et al. "Discovering block-structured process models from event logs-a constructive approach". In: *Petri nets*. 2013.
[20] Sander J. J. Leemans et al. "Earth movers' stochastic conformance checking". In: *BPM forum*. Springer, 2019, pp. 127–143.
[21] Sander J. J. Leemans et al. "Stochastic-Aware Conformance Checking: An Entropy-Based Approach". In: *CAiSE*. 2020, pp. 217–233.
[22] Chuang Lin et al. "Performance equivalent analysis of workflow systems based on stochastic petri net models". In: *CoopIS*. 2002, pp. 64–79.
[23] Fabrizio Maria Maggi, Marco Montali, and Rafael Peñaloza. "Probabilistic Conformance Checking Based on Declarative Process Models". en. In: *CAiSE*. 2020, pp. 86–99.
[24] Jan Mendling, Hajo A. Reijers, and Jorge Cardoso. "What Makes Process Models Understandable?" In: *BPM*. 2007, pp. 48–63.
[25] Andreas Rogge-Solti et al. "Discovering Stochastic Petri Nets with Arbitrary Delay Distributions from Event Logs". In: *BPM workshops*. 2014, pp. 15–27.
[26] Andreas Rogge-Solti et al. "Prediction of business process durations using non-Markovian stochastic Petri nets". In: *IS* (2015).
[27] Arik Senderovich et al. "Data-driven performance analysis of scheduled processes". In: *BPM*. 2016, pp. 35–52.
[28] Arik Senderovich et al. "Discovering Queues from Event Logs with Varying Levels of Information". In: *BPM workshops*. 2016, pp. 154–166.
[29] Loukas C. Tsironis et al. "Fuzzy Performance Evaluation of Workflow Stochastic Petri Nets by Means of Block Reduction". In: *ToS* (2010).
[30] A.J.M.M. Weijters and J.T.S. Ribeiro. "Flexible Heuristics Miner (FHM)". In: *CIDM*. 2011, pp. 310–317.

Graph-based process mining

Amin Jalali

Department of Computer and Systems Sciences
Stockholm University, Sweden
aj@dsv.su.se

Abstract. Process mining is an area of research that supports discovering information about business processes from their execution event logs. One of the challenges in process mining is to deal with the increasing amount of event logs and the interconnected nature of events in organizations. This issue limits the organizations to apply process mining on a large scale. Therefore, this paper introduces and formalizes a new approach to store and retrieve event logs into/from graph databases. It defines an algorithm to compute Directly Follows Graph (DFG) inside the graph database, which shifts the heavy computation parts of process mining into the graph database. Calculating DFG in graph databases enables leveraging the graph databases' horizontal and vertical scaling capabilities to apply process mining on a large scale. We implemented this approach in Neo4j and evaluated its performance compared with some current techniques using a real log file. The result shows the possibility of using a graph database for doing process mining in organizations, and it shows the pros and cons of using this approach in practice.

Keywords: Process mining, graph database, Big Data, Neo4j

1 Introduction

Business Process Management (BPM) is a research area that aims to enable organizations to narrow the gap between business goals and information technology support [21]. Business process evaluation is a key support in narrowing down this gap. There are two evaluation techniques to analyze business processes, a.k.a., model-based analysis, and data-based analysis [17]. While model-based analysis deals with analyzing business process models, the data-based analysis mostly focuses on analyzing business processes based on their execution event logs.

Process Mining is a discipline in the BPM area that enables data-based analysis for business processes in organizations [18]. It allows analysts not only to evaluate the business processes but also to perform process discovery, compliance checking, and process enhancement based on the execution result, a.k.a., event logs. As the volume of logs increases, new opportunities and challenges also appear. The large volume of logs enables the discovery of more information about business processes; while also raises some challenges, such as feasibility, performance, and data management.

The large volume of data is a challenge to perform process mining in organizations. There are different approaches to deal with this problem. This paper proposes and formalizes a new approach to store and retrieve event logs in graph

databases to do process mining on a large volume of data. It also defines an algorithm to compute Directly Follows Graph (DFG) inside the graph database. As a result, it enables i) removing the requirement to move data into analysts' computer, and ii) scaling the DFG computation vertically and horizontally.

The approach is implemented in Neo4j, and its performance is evaluated in comparison with some current techniques based on a real log file. The result shows the feasibility of this approach in discovering process models when the data is much bigger than the computational memory. It also shows better performance when dicing data into small chunks.

The remainder of this paper is organized as follows. Section 2 gives a short background on process mining and graph database. Section 3 introduces the graph-based process mining approach, and Section 4 elaborates on the implementation of the approach in Neo4j. Section 5 reports the evaluation results. Section 6 discuss alternative approaches and related works, and finally, Section 7 concludes the paper and introduces future research.

2 Background

2.1 Process Mining

Process Mining is a research area that supports business process data-based analysis [18]. Process discovery is a sort of process mining technique that enables identifying process models from event logs automatically. There are different sorts of perspectives that can be discovered from event logs. Control-flow, which describes the flow of activities that happened in a business process, is one of the most important ones. Directly-Follows Graphs (DFGs) is a simple notation widely used and considered a de-facto standard for commercial process mining tools [19].

Fig. 1 shows an overview of how a process model can be discovered from event logs using DFG graphs. The process discovery starts by loading a log file that stores business process execution results, a.k.a., log files. Each log contains a set of traces representing different cases that are performed in the business process. Each trace contains a set of events representing the execution result of

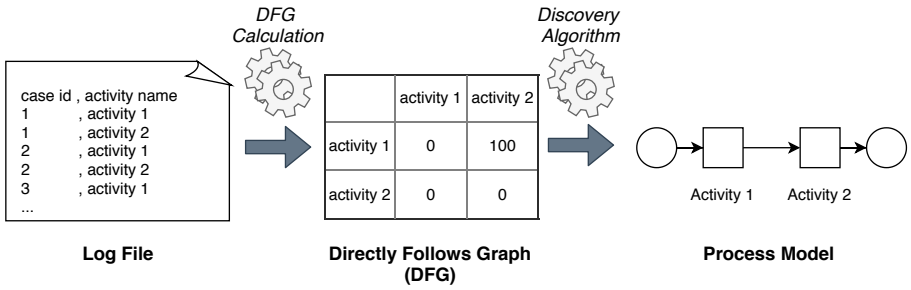


Fig. 1. Steps in a process discovery algorithm

activities in the business process. Thus, a log file shall contain information about traces and events at a minimum. Note that the events should be stored according to the execution order with this basic setup, unless we have information about execution time. It is usual to have more information like the execution time and the resource who has done the activity in the log file.

The next step is calculating the Directly Follows Graph (DFG). This graph shows the frequency of direct relations between activities that are captured in the log file. The result can be considered as a square matrix with the activity names as the index for rows and columns. Let's consider the cell with the index of *activity 1* for the row and *activity 2* for the column (see Fig. 1). The value of the cell shows the number of times that the *activity 2* happened after *activity 1*. Although the calculation of DFG comes back to alpha miner, which was introduced around 20 years ago, it is still the backbone for many process mining algorithms and tools [20]. There are different variations of DFG that store more information, but the basic idea is the same.

The last step is to infer the process model from DFG matrix based on rules that are specified by a process discovery algorithm. This step usually does not take much time since the computation is performed on top of DFG.

2.2 Graph Database

Graph databases are Database Management Systems (DBMS) that support creating, storing, retrieving, and managing graph database models. Graph database models are defined as the data structure where schema and instances are modeled as graphs, and the operation on graphs are graph-oriented [2]. The idea is not new, and it comes back to the late eighties when the object-oriented models were also introduced [2]. However, it recently got much attention in both research and industry due to its ability to handle the huge amount of data and networks. It enables leveraging parallel computing capabilities to analyze massive graphs. As a result, a new discipline is emerged in research, called Parallel Graph Analytics [15].

There are different sorts of graph databases with different features. For example, Neo4j is a graph DBMS that supports both vertical and horizontal scaling, meaning that not only the hardware of the system that runs the DBMS can be scaled out, but the number of physical nodes that run the DBMS as a network can be increased. These features enable having a considerable performance at runtime.

3 Approach

This paper proposes a new approach to store event logs and retrieve a DFG using a graph database. In this way, the scalability capabilities in graph databases can be used in favor of applying process mining. The aim is to introduce an alternative approach to enable discovering process models from large event logs.

Thus, the formal definitions of event repository in graph form are introduced. Then, the soundness property of such a repository log is defined. Finally, an algorithm to discover DFG is introduced.

Note that the formal definition is simplified by limiting the set of attributes to hold information about activities. In practice, the definition of attributes can be extended to store all information about the data perspective.

3.1 Definitions

Definition 1 (Event Repository). *An event repository is a tuple* $G = (N = L \cup T \cup E \cup A, R)$*, where:*

- N *is the superset of* L*,* T*,* E*, and* A *subsets which are pairwise disjoint, where:*
  - L *represents the set of logs,*
  - T *represents the set of traces,*
  - E *represents the set of events,*
  - A *represents the set of attributes, representing activities, where:*
  - $L \cap T \cap E \cap A = \emptyset$*.*
- $R = L \times T \cup T \times E \cup E \times E \cup E \times A$ *is the set of relations connecting:*
  - *logs to traces, i.e.,* $L \times T$
  - *traces to events, i.e.,* $T \times E$*,*
  - *events to events, i.e.,* $E \times E$*,*
  - *events to attributes, i.e.,* $E \times A$*, where:*
  - $N \cap R = \emptyset$

Let's also define two operators on the graph's nodes as:

- $\bullet n$ *represents the operator that retrieves the set of nodes from which there are relations to node* n*, i.e.,* $\bullet n = \{\forall e \in N | (e, n) \in R\}$*.*
  - *This operator enables retrieving incoming nodes for a given node, e.g., retrieving the set of events that occurred for an activity.*
- $n\bullet$ *represents the operator that retrieves the set of nodes to which there are relations from node* n*, i.e.,* $n\bullet = \{\forall e \in N | (n, e) \in R\}$*.*
  - *This operator enables retrieving outcoming coming nodes for a given node, e.g., retrieving the set of events that occurred for a trace.*

Note that the relations among logs, traces, events, and attributes are adopted from the eXtensible Event Stream (XES) standard [1]. The information is stored in attributes like XES standard which states: "Information on any component (log, trace, or event) is stored in attribute components" [1]. This is the reason why the activities are represented as attributes in this work. Note that we limit attributes to represent activities only in this work for making formalization simple for the sake of presentation. In practice, the attributes can have types to represent different properties. For example, they can be used to store different data properties of an event, e.g., who has performed it, what data it generates, etc. The usage of attributes in practice can also be extended to hold case id properties for traces and metadata information for the log node. Despite it is good to have the case id as an attribute, we kept the formalization simple by ignoring that as traces represent cases in this structure. Note that you need to know the case id to create such a structure, which is needed in the ETL process.

Definition 2 (Soundness). *An event repository $G = (N = L \cup T \cup E \cup A, R)$, where N, L, T, E, A, R, represent the set of Nodes, Logs, Traces, Events, Attributes, Relations respectively, is sound iff:*

- $\forall t \in T, |\bullet t| = 1$, *meaning that a trace must belong to 1 and only 1 log.*
- $\forall e \in E, |\bullet e \cap T| = 1$, *meaning that an event must belong to 1 and only 1 trace.*
- $\forall e \in E, |\bullet e \cap E| <= 1$, *meaning that an event can only have at most 1 input flow from another event.*
- $\forall e \in E, |e \bullet \cap E| <= 1$, *meaning that an event can only have at most 1 output flow to another event.*
- $\forall e \in E, |e \bullet \cap A| = 1$, *meaning that an event must be related to 1 and only 1 attribute.*

Note that the soundness is a property of event repository and shall not be mistaken by the soundness property of a modeling notation like Petri nets. It is worth mentioning that this formalization can be extended to enable several types of sequences among event logs. To calculate DFG, we need to count the number of direct relations among events for each activity pairs. Algorithm 1 defines how the DFG for a given sound event repository can be calculated.

Algorithm 1: Algorithm for calculating dfg

```
Algorithm dfgcalculator(G = (N = L ∪ T ∪ E ∪ A, R))
    Ψ ← ∅;
    foreach two attributes a, b ∈ A do
        c ← 0;
        foreach e ∈ •a, e' ∈ •b do
            if (e, e') ∈ R then
                c ← c + 1;
        Ψ ← Ψ ∪ {(a, b, c)};
return Ψ;
```

3.2 Example

This section elaborates on the definitions through an example.

Fig.2 shows an example of a sound event repository graph. The set of nodes for Log, Trace, Event, and Attribute are colored as green, red, white, and yellow, respectively. This repository includes one log file, called *l1*, which has two traces, i.e., *t1* and *t2*. *t1* has three events that occurred in this order $e1 \rightarrow e2 \rightarrow e3$. *t2* also has three events that occurred in this order $e4 \rightarrow e5 \rightarrow e6$.

As it can be seen, each event is related to one activity, e.g., *e1* is the execution of activity *a1*. To get the list of events that happened for an activity *a1*, we can use $\bullet a1$ operator, which returns $\{e1\}$. For some activities, there might be more than one event, e.g., $\bullet a2$ returns $\{e2, e4\}$. Applying Algorithm 1 on this event repository will return the DFG. The DFG calculation is described as below:

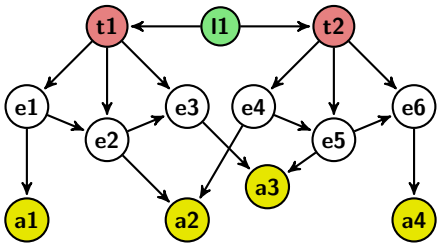


Fig. 2. An example of a sound event repository graph

- for each pair of activities, the algorithm will calculate the frequency. We show the calculation for one pair example, i.e., $a2, a3$:
  - $\bullet a2$ retreives $\{e2, e4\}$
  - $\bullet a3$ retreives $\{e3, e5\}$
  - $c = \sum_{\forall e \in \bullet a2, e' \in \bullet a3} |(e, e') \in R| = \sum_{\forall e \in \{e2, e4\}, e' \in \{e3, e5\}} |(e, e') \in R|$
  $= |\{(e2, e3), (e4, e5)\}| = 2$

If we calculate the frequencies for all pairs of activities, the result will be like Table1.

	a1	a2	a3	a4
a1	0	1	0	0
a2	0	0	2	0
a3	0	0	0	1
a4	0	0	0	0

Table 1. DFG calculation for the sample event repository graph

4 Implementation

The approach presented in this paper is implemented using the Neo4j, which was chosen because it supports i) storing graphs and doing graph operations, ii) both vertical and horizontal scaling, iii) querying the graph using Cypher, iv) containerizing the database, which allows controlling the computational CPU and memory.

We implemented a data-aware version of the approach. The main differences with the formalization are:

- Attributes store activity names and other attributes that might be associated with an event like resource id, case id, etc. To comply with PM4Py, we stored the log, case, and activity name by 'log_concept_name', 'case_concept_name', and 'concept_name' respectively.

– events have timestamps to enable dicing information based on time. Note that the timestamp cannot be defined as an attribute with its own key since we will end up with many extra nodes due to many timestamps that exist for each event. Thus, they are kept as an attribute of Event class, following the same practice to deal with times in data warehousing [14].

The calculation of DFG is implemented using a Cypher query as below:

```
match
(a1:Attribute {key:'concept_name'})<--(:Event)-[n]->(:Event)
    -->(a2:Attribute {key:'concept_name'})
return
a1.val as dfg_from, a2.val as dfg_to, count(n) as dfg_freq
```

The match clause in the query identifies all patterns in sub-graphs that match the expression. This expression selects two attributes *a1* and *a2* with the type of *concept_name*, which indicates that they are activities' names. Then, it selects all incoming events to those attributes where there is a direct relationship between those two events. The return clause retrieves all combinations of attributes in addition to the number of total direct relations between their events, which is the calculation that we formalized in Algorithm 1.

To limit the number of events based on their timestamp, we can easily add a where clause to the cypher query to limit the timestamp. For other attributes, the associated attribute node can be filtered.

5 Evaluation

This section reports the evaluation result of the approach, which is presented in this paper[1]. To evaluate the approach, we calculated DFG for a real public log file [6] using Process Mining for Python (PM4Py) library [3]. This dataset [6] is selected because it is published openly, which makes the experiment repeatable. It is also the biggest log file that we could find in the BPI challenges, which can help us to evaluate the performance.

To evaluate the performance, we need to control the resources that are available for performing process mining. Thus, we decided to containerize the experiments and run them with Docker. Docker is a Platform as a Service (PaaS) product that enables creating, running, and managing containers. It also enables the control of the resources that are available for each container, such as RAM and CPU.

Among different process mining tools, we chose PM4Py [3], because i) it is open-source; ii) the DFG calculation step and discovery step can be separated easily, and iii) it can easily be encapsulated in a container. The separation of DFG calculation and discovery step in this library also enables reusing all discovery algorithms along using our approach, which makes our approach very reusable.

We designed two experiments to evaluate our approach. In *Experiment 1*, we loaded the whole log file into both containers running neo4j and PM4Py, so we

[1] The data, code and isntructions can be found at `https://github.com/neo4pm/supporting_materials/tree/master/papers/Graph-based%20process%20mining`.

kept the number of event logs constant. We calculated DFG several times by changing the RAM and CPU, so we defined the computational resources as a variable. In *Experiment 2*, we kept RAM and CPU constant for both containers, and we calculated DFG by dicing the data. The dicing is done based on a time constraint, and we added more days in an accumulative way to increase the number of events. We ran the experiments for each container separately to make sure that the assigned resources are free and available.

	Constant	Variable
Experiment 1	Events in the Log (9 million events)	CPU & RAM
Experiment 2	CPU & RAM	Events in the Log

Table 2. Evaluation setting

5.1 Experiment 1

To simulate the situation where the computational memory is less than the log size, we started by assigning 512 megabytes of ram to each container. We added the same amount of RAM in each experiment round until we reached 4 gigabytes. We also changed the CPU starting from half of a CPU (0.5), by adding the same amount at each round until we reached 4.0.

Fig.3 shows the execution result for both containers, where the x, y and z axes refer to the available memory (RAM) (in megabytes), DFG calculation time (in seconds), and available CPU quotes, respectively. The experiment related to neo4j and PM4Py containers is plotted in red and blue, respectively. As can be

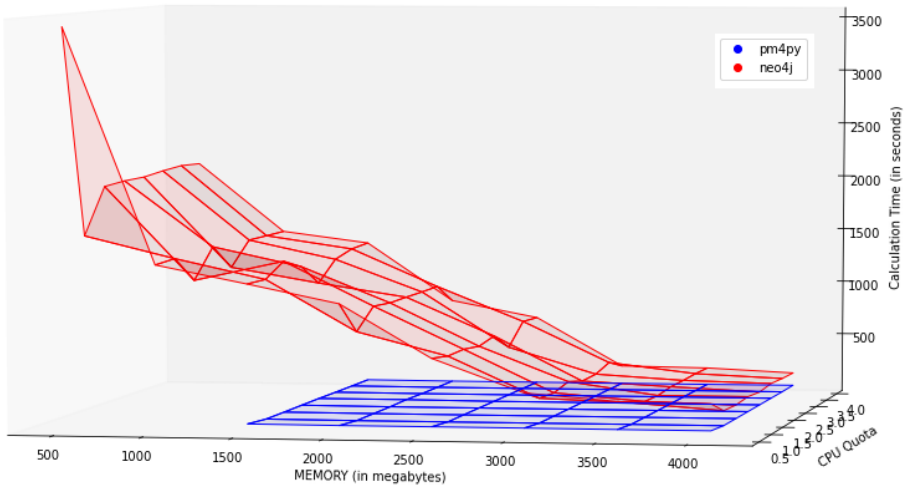


Fig. 3. Evaluating DFG calculation time by scaling resources

seen, PM4Py could not compute DFG when the memory was less than the size of the log, i.e., around 1.5 gigabytes, while neo4j could calculate DFG in that setting. This shows that the graph database can compute DFG when computational memory is less than the log size, which is an enabler when applying process mining on a very large volume of data.

As it can be seen in the figure, the increasing amount of memory reduced the time that neo4j computed the DFG, while it has very little effect on PM4Py. This is no surprise for in-memory calculation since if the log fits the memory, then the performance will not be increased much by adding more memory. It is also visible that assigning more CPU does not affect the performance of either of these approaches.

It should also be mentioned that despite increasing memory can reduce the DFG calculation time for neo4j significantly; it cannot be faster than PM4Py when calculating the DFG on the complete log file. The reason can be that graph databases shall process metadata, which adds more computation than in-memory calculation approaches. Thus, for small log files that can fit the computer's memory, the in-memory approach can be better if the security and access control are not necessary.

5.2 Experiment 2

Event logs usually contain different variations that exist in the enactment of business processes [4]. These variations make process mining challenging because discovering the process based on the whole event log usually produces the so-called spaghetti models, which usually cannot be comprehended by humans, so they have very little value. Thus, analysts need to filter data to produce a meaningful model, which is a common practice in applying process mining [4,11]. Therefore, we designed this experiment to compare our approach and PM4Py when calculating DFG on a filtered subset of data without scaling the infrastructure.

To evaluate this scenario, we kept the resources (RAM and CPU) constant for both containers, but we changed the condition for filtering the data. The condition is set based on the dates in which the event occurred. We started by filtering events for a day range, and we calculated DFG for the filtered data. Then, we expanded the filter range by including events that occurred the day after, and we calculated DFG again. We repeated the process for 30 days. As we expanded the filter range by including events that occurred on more days, we increased the number of events. This means that we kept the number of events in the log as a variable. We assigned 14 Gb for RAM and 4 CPU for each container, which was run separately. We diced the data in both settings by filtering events that happened during the first day; then, we added one more day to the filter condition to increase the events in an accumulative way. We repeated this step for almost four months. In this way, we could compare the performance by considering how the size of the filtered events affects the performance of calculating DFG.

Fig.4 shows the evaluation result, where the x and y axes refer to the number of events (in millions) and DFG calculation time (in seconds). As can be seen,

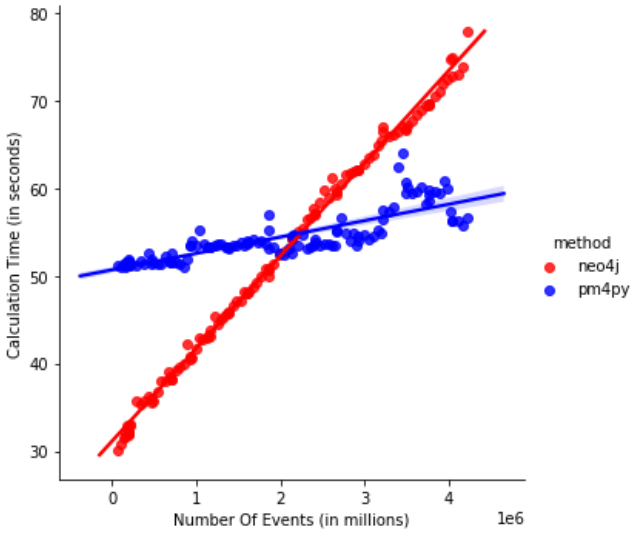


Fig. 4. Evaluating DFG calculation time by dicing the log

our approach performed better when the number of events is less than 2 million. Note that this is still a very big sub-log to analyze for process mining, so this shows that our approach can improve the performance of process mining when dealing with sub-sets of the log. However, PM4Py performed better when the number of events exceeded 2 million. This is no surprise since PM4Py loaded logs into memory first, so increasing the size will have less effect on its performance. Indeed, the difference is only related to filtering the log and retrieving the biggest chunk of data in each iteration.

6 Related Work and Discussion

The related work can be divided into two categories: those related to scalability and those using graph databases.

6.1 Scalability

The scalability issue in process mining is a big concern for applying the techniques on a large volume of data. Thus, different researchers investigated this problem through different techniques.

Hernández, S. et al. computed intermediate DFG and other matrixes through the MapReduce technique over a Hadoop cluster [10]. The evaluation of their approach shows a similar trend for a performance like what we presented in Fig.4. The performance cannot be compared precisely due to different setup and resources. This is the closest approach to ours.

MapReduce has been used by other researchers for the aim of process mining, e.g., [9,16]. As discussed by [10], MapReduce has been used to support only event

correlation discovery in [16], and it is used to discover process models using Alpha Miner and the Flexible Heuristics Miner in [9].

6.2 Graph database

There are different attempts to use graph databases with process mining.

Esser S. and Fahland D. used the graph database to query multi-dimensional aspects from event logs. This is one important use case that has been introduced by a graph database, i.e., adding more features to the data [7]. They have used Neo4j as the graph database and used Cypher to query the logs. The approach uses a graph database as a log repository to store data without any predefined structure, which is quite different from the topic of this paper. In this regard, the approach is similar to [5], where a relational database is used to store the data. The main difference is that [7] demonstrates that the graph database has more capability to add more features to data, which is a very important topic in any machine learning related approach in general.

Joishi J. and Sureka A. also used a graph database for storing non-structured event logs [12, 13]. They also demonstrated that Actor-activity matrix could be calculated using Cypher. However, the approach is context-dependent since the logs are not standardized like our approach. Also, the approach cannot be used with other process discovery algorithms since it does not shift and separate the computation of DFG to a graph database.

Parallel to this work, we realized that Esser S. and Fahland D. [8] extended their approach [7] to discover different perspectives from events which are stored in neo4j. They also introduced an approach to discover DFG from their repository. The approach is similar, yet its focus is more on creating the repository, while our focus is mostly on measuring the performance and scaling. This study also confirms the benefits of using a graph database for process mining, which can extend the application of process mining in practice.

7 Conclusion

This paper introduced and formalized a new approach to support process mining using graph databases. The approach defines how log files shall be stored in a graph database, and it also defines how Directly Follows Graphs (DFG) can be calculated in the graph database. The approach is evaluated in comparison with PM4Py by applying it to a real log file. The evaluation result shows that the approach supports mining processes when the event log is bigger than computational memory. It also shows that it is scalable, and the performance is better when dicing the event log in a small chunk.

Graph databases can bring more benefits to process mining than what we have presented in this paper. They are useful to support complex analysis, which requires taking the interconnected nature of data into account. Thus, they can enable more advanced analysis by incorporating data relations while applying different process mining techniques. As future work, we aim to extend the formalization to represent the data-aware event repository. It is also interesting to

compare this approach with process discovery approaches that can be implemented in Apache Spark. We also intend to develop a new library to support the use of a graph database for process mining for practitioners and researchers.

References

1. IEEE standard for extensible event stream (xes) for achieving interoperability in event logs and event streams. *IEEE Std 1849-2016*, pages 1–50, 2016.
2. R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1).
3. A. Berti, S. van Zelst, and W. van der Aalst. Process Mining for Python (PM4Py): Bridging the Gap Between Process-and Data Science. page 13–16, 2019.
4. A. Bolt, M. De Leoni, W. van der Aalst, and P. Gorissen. Exploiting process cubes, analytic workflows and process mining for business process reporting: A case study in education. In *SIMPDA*, pages 33–47, 2015.
5. E. De Murillas, H. Reijers, and W. van der Aalst. Connecting databases with process mining: a meta model and toolset.
6. M. Dees and B. van Dongen. Bpi challenge 2016: Clicks not logged in. 2016.
7. S. Esser and D. Fahland. Storing and querying multi-dimensional process event logs using graph databases. In *BPM Conference*, pages 632–644. Springer, 2019.
8. S. Esser and D. Fahland. Multi-dimensional event data in graph databases. *arXiv preprint arXiv:2005.14552*, 2020.
9. J. Evermann. Scalable process discovery using map-reduce. *IEEE Transactions on Services Computing*, 9(3):469–481, 2014.
10. S. Hernández, J. Ezpeleta, S. van Zelst, and W. van der Aalst. Assessing process discovery scalability in data intensive environments. In *Big Data Computing (BDC)*, pages 99–104. IEEE, 2015.
11. A. Jalali. Exploring different aspects of users behaviours in the dutch autonomous administrative authority through process cubes. *Business Process Intelligence (BPI) Challenge*, 2016.
12. J. Joishi and A. Sureka. Vishleshan: performance comparison and programming process mining algorithms in graph-oriented and relational database query languages. In *International Database Engineering & Applications Symposium*, pages 192–197, 2015.
13. J. Joishi and A. Sureka. Graph or relational databases: A speed comparison for process mining algorithm. *arXiv preprint arXiv:1701.00072*, 2016.
14. R. Kimball and M. Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.
15. A. Lenharth, D. Nguyen, and K. Pingali. Parallel graph analytics. *Communications of the ACM*, 59(5):78–87, 2016.
16. H. Reguieg, F. Toumani, H. Motahari-Nezhad, and B. Benatallah. Using mapreduce to scale events correlation discovery for business processes mining. In *BPM Conference*, pages 279–284. Springer, 2012.
17. W. van der Aalst. Business process management: a comprehensive survey. *ISRN Software Engineering*, 2013, 2013.
18. W. van der Aalst. *Process Mining: Data Science in Action*. Springer, 2016.
19. W. van der Aalst. A practitioner's guide to process mining: Limitations of the directly-follows graph, 2019.
20. W. van der Aalst. Academic view: Development of the process mining discipline. In *Process Mining in Action: Principles, Use Cases and Outlook*. 2020.
21. M. Weske. *Business process management: concepts, languages, architectures*. Springer, 2019.