Process Tree Alignments

Christopher T. Schwanen[1(✉)], Wied Pakusa[2], and Wil M. P. van der Aalst[1]

[1] Chair of Process and Data Science (PADS), RWTH Aachen University, Aachen, Germany
{schwanen,wvdaalst}@pads.rwth-aachen.de
[2] Federal University of Applied Administrative Sciences, Brühl, Germany
Wied.Pakusa@hsbund.de

Abstract. The state-of-the-art approach for computing alignments is to apply an exhaustive state-space search with a well-tailored A*-heuristic function. If the heuristic fails to provide good estimates, the alignment computation quickly becomes infeasible even for small event logs given the exponential search space. Since intractability is unavoidable for general process models, we here consider the restricted class of process trees which provides a good balance between expressiveness and algorithmic feasibility. As our main result, we prove that alignments on process trees can be expressed as solutions of Mixed Integer Linear Programs (MILP). Our novel approach does not only position the problem inside the class of NP, but also paves the way for applying a host of new optimization techniques from the field of mathematical programming to alignments on process trees. We further show that for process trees without parallel executions, our MILP formulation becomes a Linear Program (LP) which can be solved efficiently. This result gives fresh insights into the structure of the alignment problem and the role of concurrency as a key factor for intractability. Finally, we implement our new algorithmic approach in PM4Py and evaluate the performance against the standard algorithms.

Keywords: Process Mining · Conformance Checking · Alignments · Process Trees · Mixed Integer Linear Programming

1 Introduction

Constructing optimal alignments between a trace and a process model is a key task in conformance checking. The standard approach is to formulate the alignment computation as a reachability problem on the product of the model and the trace. In general, this product is of exponential size which leads to high computational costs and hinders scalability of alignment computations. This, in turn, poses a major obstacle for practical applications where massive event logs and business models have to be analyzed.

Unfortunately, computing alignments on sound workflow nets (the standard modeling notation in process mining) is a PSPACE-complete problem. While

J. Borbinha et al. (Eds.): EDOC 2024, LNCS 15409, pp. 300–317, 2025.
https://doi.org/10.1007/978-3-031-78338-8_16

this might seem discouraging, we rather see it as a call for a more intensive root-cause analysis of the algorithmic complexity of alignments. In fact, we are going to show that one can derive much better bounds if certain syntactic restrictions are imposed on the process models. This goes in line with the observation that, in practice, process models often provide such extra structure that we can exploit to speed up alignment computations.

In this paper, we focus on the important example of *process trees*, a class of models that can be decomposed into subprocesses which are interconnected in a tree-like fashion. Process trees are highly relevant in practice and form the basis for one of the most popular family of mining algorithms, the so-called *Inductive Miner* [14]. Our key observation is that process trees have optimal alignments of *linear* length (linear in the size of the trace plus the size of the process tree). This allows us to solve the alignment problem on process trees in NP (rather than PSPACE). As our central contribution, we give the first *Mixed Integer Linear Programming* (MILP) formulation of the alignment problem on process trees.

More specifically, we encode the alignment problem as a minimum-cost network flow problem. This network flow construction can then, in a second step, be readily expressed as a MILP instance. For the former translation, the difficult part is to express the *parallel operator* in a process tree as a flow gateway in a network graph. The parallel operator models independent concurrent computations and is the root cause for the exponential state explosion that we get from a translation of process trees to standard transition systems (aka. finite automata). It should be noted that the parallel operator can be expressed succinctly using Petri nets, but it is unclear, how we could get an efficient MILP formulation from this presentation. Intuitively, we model the parallel operator by splitting up a flow into equal parts and send the subflows through different parts of the network. When all subflows have completed their subcomputations, we merge the subflows again, and get back the original flow. It turns out that for this synchronized split and merge we need to use *integer variables*. In fact, this is the only part where discrete variables are required. As a consequence, for process trees without the parallel operator, our MILP instance becomes a *Linear Program* (LP) which can be solved in polynomial time.

To complement our MILP encoding, we provide a proof-of-concept implementation based on the *PM4Py* ecosystem [3] and the *Gurobi Optimizer* [13]. We further evaluate the performance of our MILP approach on a set of synthetic and real-life benchmark logs in comparison with the state-of-the-art alignment algorithms based on A* and the reachability approach. Our experiments show that the MILP-based approach is extremely promising and outperforms the two other alignment algorithms for process trees which are available in PM4Py.

2 Related Work

Alignments were introduced by [1] and are now the state-of-the-art technique for conformance checking [8,9]. In particular, they have surpassed token-based replay [18] in terms of accuracy and flexibility. Due to the high computational

costs of the textbook algorithm based on A^*, several techniques have been studied to improve scalability, see, e.g., [4,5]. This also includes techniques from mathematical optimization. Most notably, [12] uses Linear Programming (LP) to improve A^*-heuristics and to reduce the runtime significantly. Furthermore, approximative algorithms have been proposed, see, e.g., [20] for a scheme based on Mixed Integer Linear Programming (MILP). However, until today, no full MILP encoding of the alignment problem has been studied. The only work that somewhat goes into this direction is [5,6]. There, the authors showed that computing alignments is in NP if the length of optimal alignments is polynomially bounded. Since it can be shown that this holds for process trees and since NP problems can be encoded into MILPs, this result implies the existence of a MILP encoding. However, the authors did neither provide a MILP formulation nor any concrete example of an interesting model class with this property.

The notion of process trees is inspired by the observation that many real-life process models can be decomposed into distinct blocks that are interconnected in a tree-like fashion. This allows divide-and-conquer strategies for solving algorithmic problems. Process trees were first applied by [7,21] in the context of genetic process discovery. Since then, process trees have proven to be a modeling language with a great balance between expressiveness and algorithmic simplicity. In particular, they form the basis of one of the most popular process discovery algorithms, the so-called *Inductive Miner* [14]. Thus, it comes at no surprise that also optimized algorithms for alignment computations on process trees have been studied. Most notably, [19] proposed an approximation algorithm which performs well on many process trees, but which does not guarantee to compute optimal alignments in all cases. This is in stark contrast with our MILP encoding approach which always yields optimal solutions.

Finally, we like to mention work on the *error correction problem* for regular languages. Here, the goal is to compute edit distances between an input string and a regular expression. This is very intimately related to computing optimal alignments. In essence, process trees correspond to regular expressions extended by the shuffle operator and for those languages it was shown a long time ago that deciding membership (i.e., edit distance 0) is NP-complete, see, e.g., [17]. Since the membership problem is a special case of the alignment problem (where the costs of an optimal alignment are 0), our MILP encoding in this work can also be used as an error-correction algorithm for regular expressions with shuffle.

3 Preliminaries

Let $\mathbb{N}$ be the set of natural numbers excluding 0. For any tuple a, $\pi_i(a)$ denotes the *projection* on its ith element, i.e., $\pi_i \colon A_1 \times \cdots \times A_n \to A_i, (a_1, \ldots, a_n) \mapsto a_i$. For any node v in a graph, let $\delta^-(v)$ ($\delta^+(v)$) denote the set of incoming (outgoing) arcs at node v.

Definition 1 (Alphabet)**.** An *alphabet* Σ is a finite, non-empty set of *labels* (also referred to as *activities*).

Definition 2 (Sequence)**.** *Sequences* with index set I over a set A are denoted by $\sigma = \langle a_i \rangle_{i \in I} \in A^I$. The *length* of a sequence σ is written as $|\sigma|$ and the set of all finite sequences over A is denoted by A^*. For a sequence $\sigma = \langle a_i \rangle_{i \in I} \in A^I$, $\sum \sigma$ is a shorthand for $\sum_{i \in I} a_i$. The restriction of a sequence $\sigma \in A^*$ to a set $B \subseteq A$ is the subsequence $\sigma|_B$ of σ consisting of all elements in B. A function $f \colon A \to B$ can be applied to a sequence $\sigma \in A^*$ given the recursive definition $f(\langle\rangle) := \langle\rangle$ and $f(\langle a \rangle \cdot \sigma) := \langle f(a) \rangle \cdot f(\sigma)$. For a sequence of tuples $\sigma \in (A^n)^*$, $\pi_i^*(\sigma)$ denotes the sequence of every ith element of its tuples, i.e., $\pi_i^*(\langle\rangle) := \langle\rangle$ and $\pi_i^*(\langle (a_1, \ldots, a_n) \rangle \cdot \sigma) := \langle \pi_i(a_1, \ldots, a_n) \rangle \cdot \pi_i^*(\sigma) = \langle a_i \rangle \cdot \pi_i^*(\sigma)$.

Definition 3 (Shuffle $\sqcup\!\sqcup$)**.** For two sequences $x, y \in \Sigma^*$, the *shuffle* $x \sqcup\!\sqcup y$ of x and y is defined as

$$x \sqcup\!\sqcup y := \{v_1 w_1 \cdots v_k w_k \,|\, x = v_1 \cdots v_k, y = w_1 \cdots w_k, v_i, w_i \in \Sigma^*, 1 \leq i \leq k\}.$$

Let $\mathcal{L}_1, \mathcal{L}_2 \subseteq \Sigma^*$ be two languages. Then the shuffle of the two languages is defined as

$$\mathcal{L}_1 \sqcup\!\sqcup \mathcal{L}_2 := \bigcup \{w_1 \sqcup\!\sqcup w_2 \,|\, w_1 \in \mathcal{L}_1, w_2 \in \mathcal{L}_2\}.$$

Definition 4 (Transition System)**.** A *transition system* TS is a tuple $TS = (S, \Sigma, T, s_{init}, s_{final})$ where S is the set of *states*, Σ is the set of *activities*, $T \subseteq S \times \Sigma \times S$ is the set of *transitions*, and $s_{init}, s_{final} \in S$ are two distinguished states, namely the *initial state* s_{init} and the *final state* s_{final}.

Definition 5 (Process Tree)**.** Let Σ be an alphabet of activities and let $\tau \notin \Sigma$ be the silent activity. A *process tree* is defined recursively where

- each activity $a \in \Sigma$ and the silent activity τ is a process tree,
- $\rightarrow (PT_1, \ldots, PT_n)$, $\times(PT_1, \ldots, PT_n)$, $\circlearrowleft (PT_1, PT_2)$, and $\wedge(PT_1, \ldots, PT_n)$ are process trees with $PT_1, \ldots, PT_n$, $n \in \mathbb{N}$ being process trees as well.

The symbols $\rightarrow$ (sequence), $\times$ (exclusive choice), $\circlearrowleft$ (loop), and $\wedge$ (parallel) are *process tree operators*. The *language* of a process tree PT is denoted by $\mathcal{L}(PT)$ and is also recursively defined where

- $\mathcal{L}(\tau) = \{\langle\rangle\}$ and $\mathcal{L}(a) = \{\langle a \rangle\}$,
- $\mathcal{L}(\rightarrow (PT_1, \ldots, PT_n)) = \mathcal{L}(PT_1) \cdot \ldots \cdot \mathcal{L}(PT_n)$,
- $\mathcal{L}(\times(PT_1, \ldots, PT_n)) = \mathcal{L}(PT_1) \cup \cdots \cup \mathcal{L}(PT_n)$,
- $\mathcal{L}(\circlearrowleft (PT_1, PT_2)) = \mathcal{L}(PT_1) \cdot (\mathcal{L}(PT_2) \cdot \mathcal{L}(PT_1))^*$, and
- $\mathcal{L}(\wedge(PT_1, \ldots, PT_n)) = \mathcal{L}(PT_1) \sqcup\!\sqcup \cdots \sqcup\!\sqcup \mathcal{L}(PT_n)$.

The τ-language $\mathcal{L}^\tau(PT)$ of a process tree PT preserves silent activities and is defined accordingly, but with $\mathcal{L}^\tau \tau = \{\langle \tau \rangle\}$ instead. A sequence $x \in \mathcal{L}^\tau PT$ is also referred to as an *execution* of the process tree PT.

4 Computing Alignments on Process Trees

Alignments [1] juxtapose observed and modeled behavior. Thereby, activities in the observed trace are compared in pairs with activities from an execution of

the process tree. These pairs are called moves and they are considered legal if the observed activity matches the activity from the process tree execution or the pair consists of just one activity, either from the observation or the model, while its counterpart is considered to have not yet proceeded, indicated by a special "no move" symbol $\gg$.

$$\gamma_1 = \frac{b \quad a \quad c \quad \gg}{\gg \quad a \quad c \quad b} \qquad \gamma_2 = \frac{\gg \quad b \quad a \quad c}{\tau \quad b \quad \gg \quad c}$$

Let γ_1 and γ_2 be two exemplary alignments between an observed trace $\langle b, a, c\rangle$ and a process tree $\rightarrow (\times(a, \tau), \wedge(b, c))$. The top row indicates the progress in the trace, while the bottom row contains the labels executed in the process tree. In γ_1, the second and third move show that the observed activities could be synchronized with the execution of the process tree; hence, they are called *synchronous moves*. While the first move $(b, \gg)$ indicates that the observed activity b was not performed in the model, the fourth move $(\gg, b)$ indicates the reverse, namely that activity b executed by the process tree could not be matched with an activity in the trace. A move only proceeding on the trace is called *log move* and a move only proceeding on the model is called *model move*. The model move $(\gg, \tau)$ in γ_2 is special as the silent activity τ cannot be observed. Such moves are therefore not considered as deviations and also called *silent moves*.

Definition 6 (Legal Move, Alignment)**.** Let Σ be an alphabet of activities, let $\tau \notin \Sigma$ be the silent activity, let $\sigma \in \Sigma^*$ be a trace, let PT be a process tree, and let $\gg \notin \Sigma$ be a distinguished "no move" symbol. Without loss of generality, we assume the trace σ and the process tree PT being defined over the same alphabet Σ. A *move* is an ordered pair $(a, t) \in (\Sigma \cup \{\gg\}) \times (\Sigma \cup \{\tau, \gg\})$ and we distinguish three types of *legal moves*: The move (a, t) is a

- *synchronous move* if $a, t \in \Sigma$ and $a = t$,
- *log move* if $a \in \Sigma$ and $t = \gg$,
- *model move* if $a = \gg$ and $t \in \Sigma \cup \{\tau\}$.

A model move $(\gg, \tau)$ is also called *silent move*. All other moves are considered to be illegal. The set LM denotes all legal moves between alphabet Σ and process tree PT, i.e., $LM := \{(a, a) \mid a \in \Sigma\} \cup (\Sigma \times \{\gg\}) \cup (\{\gg\} \times (\Sigma \cup \{\tau\}))$. A sequence of legal moves $\gamma \in LM^*$ is an *alignment* between trace σ and process tree PT if and only if $\sigma = \pi_1^*(\gamma)|_\Sigma$ and $\pi_2^*(\gamma)|_{\Sigma \cup \{\tau\}} \in \mathcal{L}^\tau PT$. The set Γ_σ denotes all alignments between a trace $\sigma \in \Sigma^*$ and process tree PT.

Looking at the two alignments γ_1 and γ_2 from above, we see that there are multiple ways to align observed and modeled behavior. In general, we are interested in an *optimal* alignment, i.e., an alignment that fits a trace to the closest execution of the process model and only consists of inevitable deviations. Therefore, deviations are associated with costs so that minimizing the costs leads to an alignment where the synchronization between trace and model is maximal. Formally, this is achieved via a cost function that assigns costs to moves and then finding an alignment with minimal costs.

Definition 7 (Optimal Alignment)**.** Let LM be the set of all legal moves and Γ_σ be the set of all alignments between a trace $\sigma \in \Sigma^*$ and a process tree PT and let $c\colon LM \to \mathbb{Q}_{\geq 0}$ be a cost function. An alignment $\gamma_{opt} \in \Gamma_\sigma$ is *optimal* if and only if no other alignment between σ and PT has lower costs, i.e., $\sum c(\gamma_{opt}) = \min_{\gamma \in \Gamma_\sigma}\{\sum c(\gamma)\}$.

Note that, in principle, for the approach presented in this paper, any function $c\colon LM \to \mathbb{Q}_{\geq 0}$ can be chosen as a cost function. For better comprehensibility, however, the *standard cost function* is assumed in the following where synchronous or silent moves have no costs and log or non-silent model moves are associated with costs of 1.

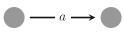
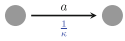
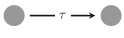
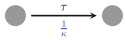
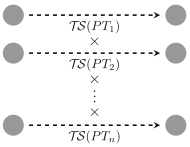
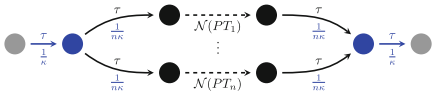
4.1 Alignments Based on Transition Systems

The standard approach to find optimal alignments is to solve a shortest path problem in the *synchronous product* between the trace and the model. We now give a translation of process trees into equivalent transition systems (where *equivalent* means, that the traces generated by the process tree and the transition system are the same). Our approach is a relatively straightforward textbook translation of a process tree into a finite automaton except for the *parallel* operator (note that process trees without concurrency correspond to regular expressions). Unfortunately, a transition system has no means to express concurrency. Typically, this problem is circumvented by first using Petri nets and, in a second step, by transforming the resulting Petri nets into equivalent transition systems. In this paper, we skip the detour through Petri nets and directly translate process trees into equivalent transitions systems.

Definition 8 (Transition System of a Process Tree)**.** Let PT be a process tree. The *transition system* of PT is denoted by $\mathcal{TS}(PT) := (S, \Sigma \cup \{\tau\}, T, s_{init}, s_{final})$ and can be constructed recursively by starting with the initial and final state $s_{init}, s_{final} \in S$ and

- if $PT = \tau$, adding a transition $(s_{init}, \tau, s_{final})$,
- for each activity $a \in \Sigma$, if $PT = a$, adding a transition (s_{init}, a, s_{final}),
- for process trees $PT_1, \ldots, PT_n$ (with pairwise disjoint state spaces), $n \in \mathbb{N}$,
  - if $PT = \rightarrow(PT_1, \ldots, PT_n)$, adding new states $s_1, \ldots, s_{n-1}$ and inserting $\mathcal{TS}PT_i$ with initial state s_{i-1} and final state s_i for $1 \leq i \leq n$ where $s_0 = s_{init}$ and $s_n = s_{final}$,
  - if $PT = \times(PT_1, \ldots, PT_n)$, we take all $\mathcal{TS}PT_i$, for $1 \leq i \leq n$, as independent subsystems and then merge all initial states to the initial state s_{init} and all final states to the final state s_{final},
  - if $PT = \circlearrowleft(PT_1, PT_2)$, adding new states s_1 and s_2, transitions (s_{init}, τ, s_1) and (s_2, τ, s_{final}), and inserting $\mathcal{TS}PT_1$ with initial state s_1 and final state s_2 and inserting $\mathcal{TS}PT_2$ with initial state s_2 and final state s_1,
  - if $PT = \wedge(PT_1, \ldots, PT_n)$, we take $\mathcal{TS}PT$ to be the direct product of the transition systems $\mathcal{TS}PT_i$, $1 \leq i \leq n$, where we declare the state $(s_1, \ldots, s_n)$, where s_i is the initial state of $\mathcal{TS}PT_i$, to be the initial state s_{init} of $\mathcal{TS}PT$ and, analogously, $(s'_1, \ldots, s'_n)$, where s'_i is the final state of $\mathcal{TS}PT_i$, to be the final state s_{final} of $\mathcal{TS}PT$.

Table 1. Comparison between the construction of a transition system $\mathcal{TS}(PT)$ and a process tree network $\mathcal{N}(PT)$ based on a process tree PT.

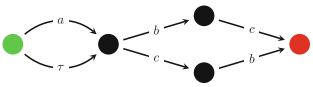
Process Tree (PT)	Construction of $\mathcal{TS}(PT)$	Construction of $\mathcal{N}(PT)$
a		
τ		
$\rightarrow(PT_1, \ldots, PT_n)$		
$\times(PT_1, \ldots, PT_n)$		
$\circlearrowleft(PT_1, PT_2)$		
$\wedge(PT_1, \ldots, PT_n)$		

The construction of a transition system from a process tree is also illustrated in Table 1. For any process tree PT it can be easily verified that $\mathcal{L}(PT) = \mathcal{L}(\mathcal{TS}PT)$. Here, the language $\mathcal{L}(TS)$ of a transition system TS consists of all label sequences of paths from the initial to the final state. Figure 1a shows the transition system of the exemplary process tree $\rightarrow(\times(a, \tau), \wedge(b, c))$. For better recognizability, the initial state is marked in green and the final state in red. A trace $\sigma \in \Sigma^*$ is expressed by a directed path TS_σ, which consists of $|\sigma| + 1$ states and where each event in the trace is associated with a transition, i.e., $TS_\sigma = (\{s_i \mid 0 \leq i \leq |\sigma|\}, \Sigma, \{(s_{i-1}, \pi_i(\sigma), s_i) \mid 1 \leq i \leq |\sigma|\}, s_0, s_{|\sigma|})$.

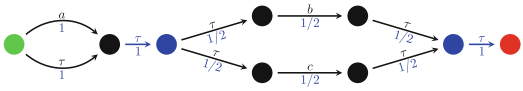
Now, to obtain an optimal alignment between a trace and a process tree, we construct the synchronous product of their transition systems [cf. 1]. We start off from a standard direct product of two transition systems, i.e., we take as state space all pairs consisting of states of the trace and states of the model (the trace can easily be encoded as a labeled directed path). We then extend both components by a new *idle* transition $\gg$ which is always active and does not alter the current state (i.e., a self-loop on every state with label $\gg$). For the product, we allow transition pairs where either both (original) transitions have the same activity label or where precisely one of them is the idle transition. Note that the resulting transitions in the synchronous product are the *legal moves* which we defined above (for details see [1,22]).

Definition 9 (Synchronous Product)**.** Let TS_1 and TS_2 be two transition systems. Their *synchronous product* is denoted by $TS_1 \otimes TS_2$ and defined according

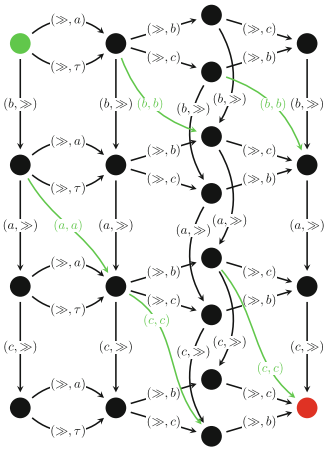
to [22, Definition 8.6] where we restrict the resulting transitions to those where either both original transitions have the same activity or one of them is idle (denoted by $\gg$). The initial and final state of the synchronous product are the states that are compositions of the original initial or final states, respectively.



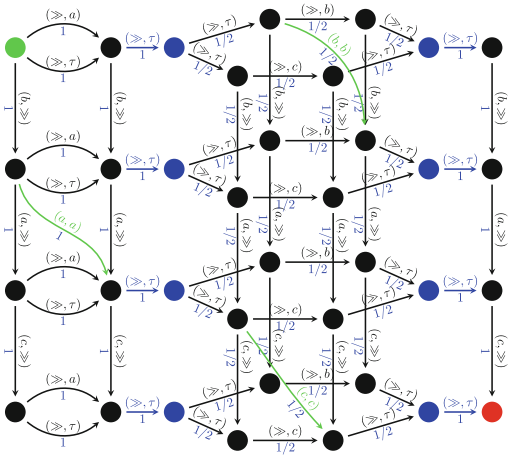
(a) Transition system $\mathcal{TS}(PT)$.



(b) Process tree network $\mathcal{N}(PT)$.



(c) Synchronous product of transition systems TS_σ and $\mathcal{TS}(PT)$.



(d) Synchronous network $\mathcal{N}(\sigma, PT)$.

Fig. 1. Comparison between the transition system and the process tree network of $PT = \rightarrow(\times(a,\tau), \wedge(b,c))$ and between the transition system of the synchronous product with trace $\sigma = \langle b, a, c\rangle$ and the corresponding synchronous network. Transitions representing synchronous moves and synchronous arcs, respectively, are highlighted in green, synchronization nodes and arcs in blue.

Figure 1c shows the synchronous product of the transition system of the trace $\langle b, a, c\rangle$ and that of the process tree $\rightarrow(\times(a,\tau), \wedge(b,c))$. It can be seen that the activities of the resulting transitions correspond to legal moves where synchronous moves are highlighted in green. Thus, each path from the initial to the final state of the synchronous product corresponds to an alignment. If each arc is weighted with the cost function according to the move it represents, an optimal alignment is found via a shortest path. Alignment γ_2 from above represents the shortest path in the synchronous product and is therefore optimal.

We can also solve an optimization problem to find a shortest path from the initial state to the final state. Let $x_t \in \{0, 1\}$ be a binary decision variable

indicating whether a transition $t \in T$ is part of the shortest path ($x_t = 1$) or not ($x_t = 0$). To obtain a valid path from the initial state to the final state, we do not only have to ensure that it originates in the initial state and terminates in the final state, but also that it is not interrupted in any other state. This can be achieved by requiring the number of incoming transitions $\delta^-(s)$ and outgoing transitions $\delta^+(s)$ used in the path to be equal in any state $s \in S$ except for the initial and final state. According to the standard cost function c, using a transition $(s, a, s') \in T$ costs $c(a)$ (note that a is a legal move). Hence, we aim to minimize $\sum_{t \in T} c_{\pi_2(t)} x_t$ which results in the following ILP formulation.

$$\min \quad \sum_{t \in T} c_{\pi_2(t)} x_t \tag{1}$$

$$\text{s.t.} \quad \sum_{t \in \delta^-(s)} x_t - \sum_{t \in \delta^+(s)} x_t = \begin{cases} -1 & s = s_{init} \\ 1 & s = s_{final} \\ 0 & \text{otherwise} \end{cases} \qquad \forall s \in S \tag{2}$$

$$x_t \in \{0, 1\} \qquad \forall t \in T \tag{3}$$

The shortest path problem is a special case of the minimum-cost flow problem which is known to be solvable by linear programming because here an integer minimum-cost flow always exists [2]. Hence, Eq. (3) can be relaxed to obtain the LP formulation given by Eqs. (1), (2) and (4).

$$x_t \geq 0 \qquad \forall t \in T \tag{4}$$

4.2 A Network Representation of Process Trees

When we transform process trees into equivalent transition systems, we see that concurrency causes the state space to grow exponentially. Of course, when we solve the shortest path problem on the resulting systems via an (I)LP as above, the exponential number of states results in an exponential number of variables and constraints. In the synchronous product, however, this state explosion problem in essence confines itself to transitions representing model moves while the ordering of both, log moves and synchronous moves is already widely determined by the sequence of activities defined in the trace.

Formally, we introduce a network representation of a process tree which we use as a basis for the alignment problem. This *process tree network* is largely similar to the transition system of a process tree except for the representation of the parallel operator. Apart from that, the most important change is the introduction of arc capacities, which allow us to split the flow for parallel subtrees. The idea is for the resulting subflows to capture (independent) computation sequences in the parallel subprocesses. Intuitively, the reader might think of the subflows as tokens that move through a Petri net.

Definition 10 (Process Tree Network)**.** Let PT be a process tree. Its *process tree network* $\mathcal{N}PT = (V, \Sigma \cup \{\tau\}, A, V', A', u, v_{init}, v_{final})$ is a tuple where V is

the set of nodes, $A \subseteq V \times (\Sigma \cup \{\tau\}) \times V$ is the set of arcs, $V' \subset V$ and $A' \subset A$ are the sets of synchronization nodes and arcs, respectively, $u\colon A \to [0,1]$ is a capacity function, $v_{init} \in V$ is the source node, and $v_{final} \in V$ is the target node. It is constructed recursively by starting with the source and target node $v_{init}, v_{final} \in V$ and a constant $\kappa = 1$ as follows:

- if $PT = \tau$, adding an arc $(v_{init}, \tau, v_{final})$ with capacity $1/\kappa$,
- for each activity $a \in \Sigma$, if $PT = a$, adding an arc (v_{init}, a, v_{final}) with capacity $1/\kappa$,
- for process trees $PT_1, \ldots, PT_n$ (with pairwise disjoint state spaces), $n \in \mathbb{N}$,
  - if $PT = \rightarrow(PT_1, \ldots, PT_n)$, adding new nodes $v_1, \ldots, v_{n-1}$ and inserting $\mathcal{N}PT_i$ with constant κ, source node v_{i-1}, and target node v_i for $1 \leq i \leq n$ where $v_0 = v_{init}$ and $v_n = v_{final}$,
  - if $PT = \times(PT_1, \ldots, PT_n)$, we take all $\mathcal{N}PT_i$ with constant κ, for $1 \leq i \leq n$, as independent subsystems and then merge all source nodes to source node v_{init} and all target nodes to target node v_{final},
  - if $PT = \circlearrowleft(PT_1, PT_2)$, adding new nodes v_1 and v_2, arcs (v_{init}, τ, v_1) and (v_2, τ, v_{final}) with capacity $1/\kappa$, and inserting $\mathcal{N}PT_1$ with constant κ, source node v_1, and target node v_2 and $\mathcal{N}PT_2$ with constant κ, source node v_2, and target node v_1, and
  - if $PT = \wedge(PT_1, \ldots, PT_n)$, adding new synchronization nodes v_S and v'_S, synchronization arcs (v_{init}, τ, v_S) and (v'_S, τ, v_{final}) with capacity $1/\kappa$, taking all $\mathcal{N}PT_i$ with constant $n\kappa$, for $1 \leq i \leq n$, as independent subsystems and then adding arcs (v_S, τ, v_i) and (v'_i, τ, v'_S) with capacity $1/(n\kappa)$ where v_i is the source and v'_i the target node of $\mathcal{N}PT_i$.

Let us give some intuition on the construction of a process tree network, also illustrated in Table 1. Given the process tree $\rightarrow(\times(a, \tau), \wedge(b, c))$ of our running example, the resulting process tree network shown in Fig. 1b is constructed recursively and in a similar fashion as a transition system, except for the arc capacities and the modeling of the parallel operator. The inverse of κ, i.e., $1/\kappa$, represents the intended intensity of the flow propagating through the particular network (from v_{init} to v_{final}). Initially, κ is set to 1 and therefore, all arc capacities outside the parallel construct are 1. We now take a closer look at the subtree $\wedge(b, c)$. Every parallel construct begins and ends with a synchronization arc and node (highlighted in blue) which connect the parallel subprocesses to the outer network construct. First, κ is set to 2 because there are two parallel subtrees b and c. Then, each subtree is constructed with $\kappa = 2$. Finally, every parallel subtree is connected with τ-arcs of capacity $1/\kappa = 1/2$ (so that the network flow is split) to the synchronization nodes leading to the final result.

A trace $\sigma \in \Sigma^*$ can also be expressed by a process tree network N_σ, which consists of $|\sigma| + 1$ nodes and where each event in the trace is associated with an arc of unit capacity, i.e., $N_\sigma = (\{v_i \,|\, 0 \leq i \leq |\sigma|\}, \Sigma, \{(v_{i-1}, \pi_i(\sigma), v_i) \,|\, 1 \leq i \leq |\sigma|\}, \emptyset, \emptyset, 1, v_0, v_{|\sigma|})$. Analogously to transition systems, we can now form a *synchronous network* following the same idea, but based on process tree networks, to provide the basic structure for the MILP formulation.

Definition 11 (Synchronous Network)**.** Let $\sigma \in \Sigma^*$ be a trace and let PT be a process tree. Given the trace network $N_\sigma = (\{v_i \,|\, 0 \leq i \leq |\sigma|\}, \Sigma, \{(v_{i-1}, \pi_i(\sigma), v_i) \,|\, 1 \leq i \leq |\sigma|\}, \emptyset, \emptyset, 1, v_0, v_{|\sigma|})$ and the network of the process tree $\mathcal{N}(PT) := (V_{PT}, \Sigma \cup \{\tau\}, A_{PT}, V'_{PT}, A'_{PT}, u_{PT}, v_{PT}, v'_{PT})$, their *synchronous product* $N_\sigma \otimes \mathcal{N}(PT) := (V, LM, A, V', A', u, v_{init}, v_{final})$, also denoted as *synchronous network* $\mathcal{N}(\sigma, PT)$, can be constructed iteratively where

- $V := V_\sigma \times V_{PT}$ and $V' := V_\sigma \times V'_{PT}$,
- $A := A^M \cup A^L \cup A^S \cup A' \subseteq V \times LM \times V$ where
  - $A^M := \bigcup_{0 \leq i \leq |\sigma|} A_i^M$ and $A^\tau := \bigcup_{0 \leq i \leq |\sigma|} A_i^\tau$ are model arcs with $A_i^M := \{((v_i, \pi_1(a)), (\gg, \pi_2(a)), (v_i, \pi_3(a))) \,|\, a \in A_{PT} \setminus A'_{PT} \wedge \pi_2(a) \neq \tau\}$ and $A_i^\tau := \{((v_i, \pi_1(a)), (\gg, \tau), (v_i, \pi_3(a))) \,|\, a \in A_{PT} \setminus A'_{PT} \wedge \pi_2(a) = \tau\}$,
  - $A^L := \bigcup_{1 \leq i \leq |\sigma|} A_i^L$ are log arcs with $A_i^L := \{((v_{i-1}, v), (\pi_i(\sigma), \gg), (v_i, v)) \,|\, v \in V_{PT} \setminus V'_{PT}\}$,
  - $A^S := \bigcup_{1 \leq i \leq |\sigma|} A_i^S$ are synchronous arcs with $A_i^S := \{((v_{i-1}, \pi_1(a)), (\pi_i(\sigma), \pi_2(a)), (v_i, \pi_3(a))) \,|\, a \in A_{PT} \setminus A'_{PT} \wedge \pi_2(a) = \pi_i(\sigma)\}$,
  - $A' := \bigcup_{0 \leq i \leq |\sigma|} A'_i$ are synchronization arcs with $A'_i := \{((v_i, \pi_1(a)), (\gg, \pi_2(a)), (v_i, \pi_3(a))) \,|\, a \in A'_{PT}\}$,
- $\forall a \in A\colon\ u(a) := \min(\{1\} \cup \{u_{PT}(a') \,|\, a' \in A_{PT} \wedge \pi_3(a') = \pi_2(\pi_1(a))\}) \in [0, 1]$,
- $v_{init} := (v_0, v_{PT}) \in V$ and $v_{final} := (v_{|\sigma|}, v'_{PT}) \in V$.

Figure 1d shows the resulting synchronous network of the trace $\langle b, a, c \rangle$ and the process tree $\rightarrow (\times(a, \tau), \wedge(b, c))$. There are no loops in the example, but note that their arc capacity is bounded like all other arcs. Due to the capacity constraint, running through a loop repeatedly is not possible (when we assume flows with maximal intensity). However, this is not a contradiction, as it ultimately represents model moves and only the shortest firing sequence between two states is sought. Moreover, it should be emphasized that synchronization nodes are only incident with synchronization and model arcs.

In terms of the ILP formulation in Eqs. (1) to (3), we have to adapt to the new network structure with arc capacities and adjust the cost function accordingly. The binary decision variable $x_a \in \{0, 1\}$ still indicates whether an arc $a \in A$ is part of the shortest path ($x_a = 1$) or not ($x_a = 0$).

$$\min \quad \sum_{a \in A^M} x_a + \sum_{a \in A^L} u_a x_a - \sum_{a \in A^S} (1 - u_a) x_a \tag{5}$$

$$\text{s.t.} \quad \sum_{a \in \delta^-(v)} u_a x_a - \sum_{a \in \delta^+(v)} u_a x_a = \begin{cases} -1 & v = v_{init} \\ 1 & v = v_{final} \\ 0 & \text{otherwise} \end{cases} \qquad \forall v \in V \tag{6}$$

$$\sum_{a \in A_i^S} x_a \leq 1 \qquad \forall 1 \leq i \leq |\sigma| \colon |A_i^S| > 1 \tag{7}$$

$$x_a \in \{0, 1\} \qquad \forall a \in A \tag{8}$$

Obviously, we now have to account for the arc capacities u_a in the flow conservation constraint for each node in Eq. (6). Further, we have to adjust the objective such that the costs of moves accord with the standard cost function. Based on the network structure, we see that using a model arc $a \in A^M \cup A^\tau$ always corresponds to a model move; therefore, their cost remain the same. This does not necessarily apply to log moves because in case of a log move within a parallel construct, a log arc $a \in A^L$ must be used for each flow in parallel subtrees; therefore, their cost are weighted with the arc capacity u_a. For the same reason, the costs for using a synchronous arc $a \in A^S$ must also be adjusted as the other partial flows within a parallel construct must switch to log arcs, whose additional costs must be compensated for here; therefore, their cost are reduced based on the difference to their arc capacity. Note that due to the network structure, neither log nor synchronous moves can be part of a cycle. Thus, the solution space remains bounded even with negative costs for synchronous arcs. In case of duplicate labels in the process tree, the network might allow to use more than one synchronous move on the same trace activity; however, the newly introduced constraint in Eq. (7) ensures that at most one of the synchronous arcs is used per activity in the trace.

4.3 Relaxed MILP Formulation

Due to the construction of the network, each synchronization node $v' \in V'$ is incident with exactly one synchronization arc $a' \in A'$. Hence, let $\delta' \colon V' \to A'$ be the bijection which assigns that particular arc $a' \in A'$ to each node $v' \in V'$. The binary decision variable $y_{v'} \in \{0, 1\}$ therefore implicitly indicates whether the corresponding synchronization arc $\delta'(v') \in A'$ is used ($y_{v'} = 1$) or not ($y_{v'} = 0$). For better readability, we also introduce the function ρ^+ (ρ^-) which adapts the function δ^+ (δ^-) in such a way that synchronization arcs are resolved.

$$\rho^{\pm} \colon V \to \mathcal{P}(A \setminus A'), v \mapsto \rho^{\pm}(v) := \left(\delta^{\pm}(v) \setminus A'\right) \cup \bigcup_{a' \in \delta^{\pm}(v) \cap A'} \rho^{\pm}(\pi_{2\pm1}(a'))$$

This way, we are able to isolate the synchronization arcs and relax the decision variable for all remaining arcs to represent the flow on that arc. That is, the continuous variable $x_a \in [0, u_a]$ denotes the flow on arc $a \in A \setminus A'$. As a result, capacities no longer have to be taken into account separately.

The isolation of the synchronization arcs also permits that they can be ignored in the flow conservation constraint in Eq. (10) at any node $v \in V \setminus V'$ as the corresponding arcs before or after the synchronization arc are now considered here instead. For each synchronization node $v' \in V'$, Eq. (11) ensures flow conservation where the flow on a synchronization arc is still determined via its capacity. Due to the network structure, the synchronization arc at a node $v' \in V'$ is always oriented contrary to all remaining arcs; thus, we simply use $\delta(v') := \delta^+(v') \cup \delta^-(v')$ here. Finally, the objective is adjusted by factoring in

the flow via the arc capacities.

$$\min \quad \sum_{a \in A^M} \frac{1}{u_a} x_a + \sum_{a \in A^L} x_a - \sum_{a \in A^S} \left(\frac{1}{u_a} - 1 \right) x_a \tag{9}$$

$$\text{s.t.} \quad \sum_{a \in \rho^-(v)} x_a - \sum_{a \in \rho^+(v)} x_a = \begin{cases} -1 & v = v_{init} \\ 1 & v = v_{final} \\ 0 & \text{otherwise} \end{cases} \qquad \forall v \in V \setminus V' \tag{10}$$

$$\sum_{a \in \delta(v') \setminus A'} x_a = u_{\delta'(v')} y_{v'} \qquad \forall v' \in V' \tag{11}$$

$$\sum_{a \in A_i^S} \frac{1}{u_a} x_a \leq 1 \qquad \forall 1 \leq i \leq |\sigma| \colon |A_i^S| > 1 \tag{12}$$

$$x_a \leq u_a \qquad \forall a \in A \setminus A' \tag{13}$$

$$x_a \geq 0 \qquad \forall a \in A \setminus A' \tag{14}$$

$$y_{v'} \in \{0, 1\} \qquad \forall v' \in V' \tag{15}$$

It remains to show that the relaxation leads to the same optimal solution as the ILP formulation in Eqs. (5) to (8). The network structure outside of parallel constructs is identical to that of the transition system and the individual subtree representations within a parallel construct are structurally independent. Although the arc capacities are not necessarily integer, they are constant for each subcomponent and Eqs. (11) ensures that the flow within a subcomponent is exactly this constant. Therefore, there exists a common factor such that all arc capacities are integer and because of the structural independence of subcomponents an integer minimum-cost flow would always exist [2].

5 Evaluation

To analyze the performance of our MILP approach, we developed a proof-of-concept implementation and evaluation[1] in the *PM4Py* ecosystem [3] using the *Gurobi Optimizer* [13]. We compared the performance of our implementation (*MILP*) with the general PM4Py implementation based on the A* approach (*Standard*) and an optimized approximation algorithm for alignments on process trees (*Approximation*). For each algorithm and trace variant, we took the best out of 10 repetitions (meaning the minimum required time for computing the costs of an optimal alignment). To visualize the results, we computed the *performance factors* for each trace variant, that is, we took the best runtime and divided the runtime of all three algorithms by this optimal runtime (trace-variant-wise). For instance, a performance factor of 2 indicates, that the algorithm took twice as long as the best algorithm.

Not all algorithms finished computation in a reasonable amount of time, so we set a timeout of 65 s (incl. 5 s to compensate for overhead and give each algorithm

[1] https://github.com/christopher-schwanen/process-tree-alignments.

the safe chance to finish within one minute). Algorithms that hit this timeout in any run were considered to have failed (on this variant), and performance factors are not computed. In the charts below, we plotted the empirical CDF of the performance factors per algorithm. In cases where the frequencies do not sum up to 1, the algorithm ran into timeouts on a certain fraction of instances.

Real-world event logs: We used the well-known *Sepsis Cases event log* [16] and the *Inductive Miner* [14] to discover process trees with different noise thresholds (0 %, 10 %, 25 %, and 50 %) against which we aligned the log. The Inductive Miner produces process trees with *unique labels*. Since the alignment problem for such trees is much simpler (solvable in polynomial time), we further renamed duplicate labels in traces (adding a suffix, up to 5 repetitions) so that we could later (after discovery) merge the labels again (by removing the suffix). The results are depicted in Fig. 2. It can be seen that our MILP approach outperforms both other algorithms clearly on the Sepsis Cases event log. The picture is even clearer for lower noise thresholds. We obtain a similar picture on the BPI Challenge 2012 and 2017 event logs [10,11]. While on process trees with unique labels, the MILP and the approximation algorithm are usually close (and clearly superior to the standard algorithm), as soon as we drop the unique label property, our MILP approach dramatically outperforms the other two algorithms. Table 2 provides some benchmark results for these event logs and the process trees created using our duplicate label strategy (cf. above) with respect to the different noise thresholds (0 %, 10 %, 25 %, and 50 %). Here, we depict the median computation times of the three algorithms for the different runs together with the percentage of instances solved, i.e., the fraction of variants where the algorithm did not run into a timeout. To give one example, for the BPI Challenge 2017 (with configuration 0 % noise threshold and duplicate labels) the approximation algorithm could align *none* of the 1000 randomly chosen variants within the time bound, while the standard algorithm could only align about 9 % of the variants. At the same time, our MILP approach was successful on 99.8 % of all variants with a median computation time of about 14 s (time bound 65 s).

Artificial Event Log: Concurrency is the main driver for the complexity of the alignment problem, so we also took artificial examples which put concurrency into focus. For $m = 10$ and $n = 10$ we considered traces $w_m = \langle a \rangle^m \cdot \langle b \rangle \cdot \langle a \rangle^m$ with two activities a and b of which we grouped n copies together to get a process tree $\mathcal{L}(PT[m,n]) = w_m \sqcup\!\sqcup w_m \sqcup\!\sqcup \cdots \sqcup\!\sqcup w_m$. We then sampled several traces of the form $\langle a, \ldots, a, b, a, \ldots, a \rangle^n$ and aligned them against the tree $PT[m,n]$. Our MILP approach could always find a solution (within the time bound of 65 s). The Approximation algorithm was only able to solve 8.0 % of instances. After extending the time bound to 330 s (incl. 30 s to compensate for overhead and give each algorithm the safe chance to finish within five minutes), it could find a solution in most cases, but was far beyond the performance of our MILP approach. The Standard algorithm uniformly failed to solve these instances, so we excluded it from the analysis. Figure 3 demonstrates that in more than 70 % of instances it took more than twice as long.

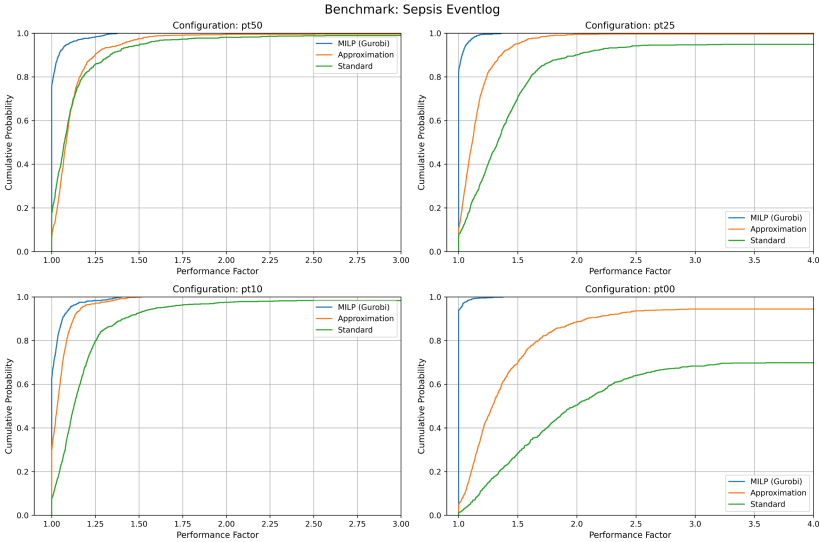


Fig. 2. Performance factors of our MILP approach, the Approximation approach, and the Standard approach on the runtimes when computing alignments for the *Sepsis Cases* event log [16].

Table 2. Comparison of median computation times ($\tilde{t}_{comp}$, in seconds) and percentages of instances solved with respect to the time bound of 65 s (% solved) for different event logs and process trees.

	MILP		**Approximation**		**Standard**	
	$\tilde{t}_{comp}$	% solved	$\tilde{t}_{comp}$	% solved	$\tilde{t}_{comp}$	% solved
Sepsis Cases [16]					(846 trace variants)	
50 %	**7.35**	**100.00**	7.54	**100.00**	7.55	99.76
25 %	**7.09**	**100.00**	8.11	94.68	8.16	98.23
10 %	**7.34**	**100.00**	7.41	**100.00**	7.49	99.88
0 %	**7.31**	**100.00**	8.30	98.70	19.48	98.58
BPI Challenge 2012 [10]			(1000 randomly chosen trace variants)			
50 %	**13.48**	99.20	15.97	**99.80**	50.80	52.60
25 %	**18.93**	**93.40**	—	45.40	—	16.40
10 %	**8.34**	**100.00**	11.13	99.90	11.61	90.70
0 %	**9.62**	**100.00**	30.34	85.10	—	12.70
BPI Challenge 2017 [11]			(1000 randomly chosen trace variants)			
50 %	**11.10**	**100.00**	17.10	85.20	—	27.50
25 %	**14.31**	**99.60**	40.11	59.10	—	28.90
10 %	**12.78**	**100.00**	21.84	60.40	—	5.20
0 %	**16.45**	**99.80**	—	0.00	—	8.30

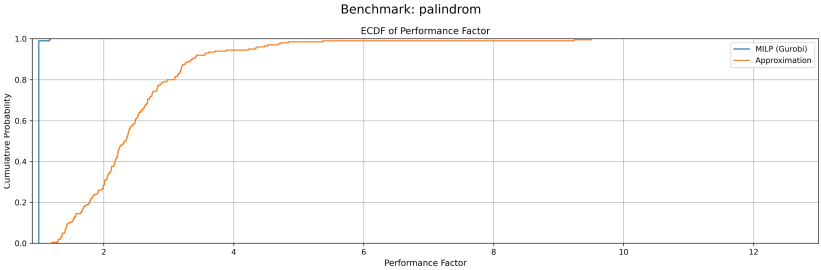


Fig. 3. Performance factors of our MILP approach and the Approximation approach on the runtimes when computing alignments for the artificial *Palindrome* event log.

6 Conclusion

We gave the first MILP formulation for alignments on process trees together with a proof-of-concept implementation and evaluation. Our experiments show that our new approach outperforms the existing algorithms in PM4Py. This sheds new light on the alignment problem which is known to be inherently difficult to solve in practice. In particular, our work demonstrates that the study of restricted model classes can lead to new algorithmic approaches and to specialized algorithms which perform more efficiently due to the utilization of additional structure. It is clear that our techniques generalize to larger classes of process models. It remains a key question for future research to see how far our MILP approach can be pushed. At the same time, there are many angles for deeper investigations of MILP encodings on process trees. For instance, are there other encodings for which common solvers perform even better or can we further improve the encoding given in this paper? Also, we can now access the huge toolbox of mathematical optimization and study questions such as how accurate LP relaxations of the alignment computation become. Specifically, it would be interesting to investigate the accuracy loss when we relax the MILP to become an efficiently solvable LP.

Acknowledgement. The research of the first author is funded by the IGF project 22485 N by the Federal Ministry for Economic Affairs and Climate Action (BMWK) on the basis of a decision of the German Bundestag. The first and third author thank the Alexander von Humboldt (AvH) Stiftung for supporting their research.

References

1. Adriansyah, A.: Aligning observed and modeled behavior, Ph.D. dissertation, Technische Universiteit Eindhoven (2014). ISBN: 978-90-386-3574-3. https://doi.org/10.6100/IR770080
2. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows, Theory, Algorithms, and Applications. Englewood Cliffs, Prentice Hall, NJ (1993). ISBN: 0-13-617549-X
3. Berti, A., van Zelst, S.J., Schuster, D.: PM4Py: a process mining library for Python. Softw. Impacts **17**, 100–556 (2023). https://doi.org/10.1016/j.simpa.2023.100556

4. Bloemen, V., van de Pol, J., van der Aalst, W.M.P.: Symbolically aligning observed and modelled behaviour. In: Application of Concurrency to System Design, IEEE Computer Society, pp. 50–59 (2018). ISBN: 978-1-5386-7013-2, https://doi.org/10.1109/ACSD.2018.00008
5. Boltenhagen, M., Chatain, T., Carmona, J.: Generalized alignment-based trace clustering of process behavior. In: Donatelli, S., Haar, S. (eds.) PETRI NETS 2019. LNCS, vol. 11522, pp. 237–257. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21571-2_14
6. Boltenhagen, M., Chatain, T., Carmona, J.: Optimized SAT encoding of conformance checking artefacts. Computing **103**(1), 29–50 (2020). https://doi.org/10.1007/s00607-020-00831-8
7. Buijs, J. C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: A genetic algorithm for discovering process trees. In: 2012 IEEE Congress on Evolutionary Computation, pp. 1–8 (2012). https://doi.org/10.1109/CEC.2012.6256458
8. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: conformance checking, relating processes and models. Springer, Cham (2018). 978-3-319-99413-0. https://doi.org/10.1007/978-3-319-99414-7
9. Carmona, J., van Dongen, B.F., Weidlich, M.: Conformance checking: foundations, milestones and challenges. In: van der Aalst, W.M.P., Carmona, J. (eds.) Process Mining Handbook, LNBIP, vol. 448. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-08848-3_5
10. van Dongen, B.F.: BPI challenge 2012. Eindhoven University of Technology (2012). https://doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f
11. van Dongen, B.F.: BPI challenge 2017. Eindhoven University of Technology (2017). https://doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b
12. van Dongen, B.F.: Efficiently computing alignments. In: Weske, M., Montali, M., Weber, I., vom Brocke, J. (eds.) BPM 2018. LNCS, vol. 11080, pp. 197–214. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98648-7_12
13. Gurobi Optimization, LLC, Gurobi Optimizer Reference Manual (2023). https://www.gurobi.com
14. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from incomplete event logs. In: Ciardo, G., Kindler, E. (eds.) PETRI NETS 2014. LNCS, vol. 8489, pp. 91–110. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07734-5_6
15. de Leoni, M., Marrella, A.: Aligning real process executions and prescriptive process models through automated planning. Expert Syst. Appl. **82**, 162–183 (2017). https://doi.org/10.1016/j.eswa.2017.03.047
16. Mannhardt, F.: Sepsis cases - event log. Eindhoven University of Technology (2016). https://doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460
17. Mayer, A.J., Stockmeyer, L.J.: The complexity of word problems - this time with interleaving. Inf. Comput. **115**(2), 293–311 (1994). https://doi.org/10.1006/inco.1994.1098
18. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. **33**(1), 64–95 (2008). https://doi.org/10.1016/j.is.2007.07.001
19. Schuster, D., van Zelst, S.J., van der Aalst, W.M.P.: Alignment approximation for process trees. In: Leemans, S., Leopold, H. (eds.) ICPM 2020. LNBIP, vol. 406, pp. 247–259. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72693-5_19
20. Taymouri, F., Carmona, J.: A recursive paradigm for aligning observed behavior of large structured process models. In: La Rosa, M., Loos, P., Pastor, O. (eds.) BPM

2016. LNCS, vol. 9850, pp. 197–214. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45348-4_12
21. van der Aalst, W.M.P., Buijs, J., van Dongen, B.F.: Towards improving the representational bias of process mining. In: Aberer, K., Damiani, E., Dillon, T. (eds.) SIMPDA 2011. LNBIP, vol. 116, pp. 39–54. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34044-4_3
22. Winskel, G.: Synchronization trees. Theoret. Comput. Sci. **34**(1–2), 33–82 (1984). https://doi.org/10.1016/0304-3975(84)90112-9