A Dynamic Programming Approach for Alignments on Process Trees

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. A fundamental task in conformance checking is to compute optimal alignments between a given event log and a process model. In general, it is known that this unavoidably incurs high computational costs which, in turn, leads to poor scalability in practice. One angle to attack the complexity is to develop alignment algorithms that exploit particular syntactic restrictions of the underlying process models. In this article, we study alignments for process trees with unique labels. These models are the output of the Inductive Miner, a family of state-of-the-art process discovery algorithms also used by the leading process mining tools. Our main contribution is a novel algorithm that constructs optimal alignments for process trees with unique labels efficiently, i.e., in polynomial time. This is in contrast with general process trees where the problem is NP-complete and general workflow nets where the problem is PSPACE-hard. We give a proof-of-concept implementation of our algorithm in PM4Py and evaluate it on a collection of real-life event logs.

Keywords: Process Mining · Conformance Checking · Alignments · Process Trees · Dynamic Programming

1 Introduction

Constructing optimal alignments between a trace and a process model is a key task in conformance checking. Unfortunately, the algorithmic complexity of alignments is a major bottleneck in practice. It can be shown that computing optimal alignments on safe and sound workflow nets is PSPACE-complete. One approach to overcome the intractability is to consider (syntactic) restrictions on the process models and to make use of the additional structure to speed up the alignment computation. Along these lines, we recently showed that computing optimal alignments on *process trees* is in NP and we gave a novel Mixed Integer Linear Programming (MILP) formulation which outperforms the state-of-the-art alignment algorithms in PM4Py [20]. In this work, we reconsider process trees,

A. Delgado and T. Slaats (Eds.): ICPM 2024 Workshops, LNBIP 533, pp. 84–97, 2025.
https://doi.org/10.1007/978-3-031-82225-4_7

but with the further restriction that each activity label occurs at most once in the process tree (i.e., *process trees with unique labels*).

In many real-life scenarios, process models have a tree-like structure, meaning that the full process decomposes into subprocesses that are interconnected in a tree-like fashion. In process mining, this kind of process models has been formalized as the concept of *process trees*. Process trees have gained quite some popularity in the process mining community, most importantly, since they form the basis for a widely used family of mining algorithms, the so-called *Inductive Miner* [15]. It is fair to say that process trees provide a good trade-off between expressiveness and computational efficiency.

But there is more to the story: the structure of process trees that come out of the Inductive Miner have *unique* activity labels, meaning that each activity label occurs at most once in the process tree. This clearly is a strong restriction, but it really is this assumption which makes the Inductive Miner tractable in practice. For us, it was reason enough to reconsider the alignment problem on process trees and ask if it is possible to exploit the *unique label property* to speed up the alignment computations even further, in particular: can alignments on process trees with unique labels be computed in polynomial time?

In this paper, we answer this question affirmatively. We give a new efficient (polynomial-time) dynamic programming algorithm to compute optimal alignments between a trace and a process tree with unique labels. This places the alignment problem for process trees with unique labels in P. Our key observation is that the unique label property allows us to handle the *parallel operator* in an efficient manner. The parallel operator models independent parallel computation and corresponds to a *parallel gateway* in BPMN or to the *shuffle operator* in formal languages. In fact, without the restriction to unique labels, the parallel operator requires the exploration of an exponential number of possible alignments which brings us to the realm of NP for general process trees. Besides the parallel operator, we further make use of the unique label property to speed-up computations of the sequence operator. We show how we can restrict the set of possible splits of a trace with respect to an optimal alignment. This saves a high number of recursive calls in the dynamic programming algorithm. We implemented our new algorithmic approach in form of a proof-of-concept based on the `PM4Py` ecosystem [4] and also evaluated it on a set of real-life benchmark logs. Our experiments show that the dynamic programming algorithm is competitive with the state-of-the-art alignment algorithms in `PM4Py` and even outperforms them in some cases. This underlines our belief that the structure of process models should be better taken into account when solving the alignment problem in practice.

2 Related Work

Alignments [3] are the state-of-the-art technique for conformance checking [7,8]. Besides the (textbook) algorithm based on A^*, several algorithmic approaches have been explored to compute alignments, e.g., see [5,11,16] for a technique

based on Linear Programming (LP) to improve the A^*-heuristics, or [21] for an approximative scheme based on Mixed Integer Linear Programming (MILP). Other approaches use decomposition techniques to tackle large process model instances, e.g., see [1].

Process trees were first applied by [2,6] 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* [13–15]. Thus, 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 optimality in all cases. We also like to point to our MILP formulation for the alignment problem on process trees [20].

Finally, alignments for process trees have been studied much earlier in the context of the *error correction problem* for regular languages with shuffle operator, e.g., see [18] (under a different term). Our new algorithmic approach can be transferred into this field as well where, by the best of our knowledge, the unique label property has not been studied before.

3 Preliminaries

Let $\mathbb{N}$ ($\mathbb{N}_0$) be the set of natural numbers excluding 0 (including 0). For an n-tuple $a \in A_1 \times \cdots \times A_n$, $\pi_i(a)$ denotes the *projection* on its ith element, i.e., $\pi_i \colon A_1 \times \cdots \times A_n \to A_i, (a_1, \ldots, a_n) \mapsto a_i$.

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 $f(\langle\rangle) \coloneqq \langle\rangle$ and $f(\langle a \rangle \cdot \sigma) \coloneqq \langle f(a) \rangle \cdot f(\sigma)$. For a sequence of tuples $\sigma \in (A^n)^*$, $\pi_i^*(\sigma)$ denotes the sequence of every ith element of its tuples, i.e., $\pi_i^*(\langle\rangle) \coloneqq \langle\rangle$ and $\pi_i^*(\langle (a_1, \ldots, a_n) \rangle \cdot \sigma) \coloneqq \langle \pi_i(a_1, \ldots, a_n) \rangle \cdot \pi_i^*(\sigma) = \langle a_i \rangle \cdot \pi_i^*(\sigma)$. As an important extension of π_i^* we write π_i^B for the composition of π_i^* with the restriction to B, i.e. $\pi_i^B \coloneqq \pi_i^*|_B$.

We identify languages of traces $\mathcal{L} \subseteq \Sigma^*$ with sets of (observed) behavior of a (business) process. Each trace corresponds to a single process execution (also known as a *case*). The symbols in the trace correspond to the *events* or *activities* that occurred. In this article, we study *process trees* as a modeling mechanism for business processes. Each process tree T defines a language $\mathcal{L}(T) \subseteq \Sigma^*$ of possible process behaviors. Before we give the definition, we recall a central operator which captures independent parallel computations.

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

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

Let $\mathcal{L}_1, \mathcal{L}_2 \subseteq \Sigma^*$. The shuffle of $\mathcal{L}_1$ and $\mathcal{L}_2$ is defined as

$$\mathcal{L}_1 \sqcup\!\sqcup \mathcal{L}_2 \coloneqq \bigcup \{w_1 \sqcup\!\sqcup w_2 \mid w_1 \in \mathcal{L}_1, w_2 \in \mathcal{L}_2\}.$$

Definition 4 (Process Trees)**.** Let Σ be an alphabet and let $\tau \notin \Sigma$ be the silent activity. The set of *process trees* (over Σ) is defined recursively:

- each activity $a \in \Sigma$ and the silent activity τ is a process tree,
- $\rightarrow(T_1, \dots, T_n)$, $\times(T_1, \dots, T_n)$, $\circlearrowleft(T_1, T_2)$, and $\wedge(T_1, \dots, T_n)$ are process trees with $T_1, \dots, T_n$, $n \in \mathbb{N}$ 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 T is denoted by $\mathcal{L}(T)$ and is also recursively defined where

- $\mathcal{L}(\tau) = \{\langle\rangle\}$ and $\mathcal{L}(a) = \{\langle a \rangle\}$,
- $\mathcal{L}(\rightarrow(T_1, \dots, T_n)) = \mathcal{L}(T_1) \cdot \ldots \cdot \mathcal{L}(T_n)$,
- $\mathcal{L}(\times(T_1, \dots, T_n)) = \mathcal{L}(T_1) \cup \ldots \cup \mathcal{L}(T_n)$,
- $\mathcal{L}(\circlearrowleft(T_1, T_2)) = \mathcal{L}(T_1) \cdot (\mathcal{L}(T_2) \cdot \mathcal{L}(T_1))^*$, and
- $\mathcal{L}(\wedge(T_1, \dots, T_n)) = \mathcal{L}(T_1) \sqcup\!\sqcup \ldots \sqcup\!\sqcup \mathcal{L}(T_n)$.

In order to simplify notation in this article, from now on, we consider the process tree operators $\{\rightarrow, \times, \circlearrowleft, \wedge\}$ in their binary form only. This also allows us to use infix notation, e.g., $T_1 \rightarrow T_2$ instead of $\rightarrow(T_1, T_2)$. This is no restriction, since the general n-ary version can easily be rewritten in form of binary operators (all operators are associative). For a process tree T, let $\mathit{Letters}(T) \subseteq \Sigma$ denote the set of all labels occurring in T. Inductively, $\mathit{Letters}(T)$ is defined as: $\mathit{Letters}(\tau) = \emptyset$, $\mathit{Letters}(a) = \{a\}$ for $a \in \Sigma$, and for all binary operators we have

$$\mathit{Letters}(T_1 \rightarrow T_2) = \mathit{Letters}(T_1 \times T_2) = \mathit{Letters}(T_1 \wedge T_2) = \mathit{Letters}(T_1 \circlearrowleft T_2) = \mathit{Letters}(T_1) \cup \mathit{Letters}(T_2).$$

A process tree T has *unique labels* if for all binary operators $\mathsf{op} \in \{\rightarrow, \times, \wedge, \circlearrowleft\}$ and all subtrees $(T_1\ \mathsf{op}\ T_2)$ that occur in T we have $\mathit{Letters}(T_1) \cap \mathit{Letters}(T_2) = \emptyset$. Note that the silent activity τ can occur multiple times in process trees T with unique labels.

Definition 5 (Moves, Alignments)**.** Let Σ be an alphabet and let $\gg$ be a fresh symbol not in Σ. We use $\gg$ to indicate a *skip* in the trace or model and define $\Sigma_{\gg} \coloneqq \Sigma \cup \{\gg\}$ as the alphabet extended by the skip-symbol $\gg$. We define $\mathit{Moves}(\Sigma) \subseteq \Sigma_{\gg} \times \Sigma_{\gg}$ as the set of all *moves* over Σ given by

$$\begin{aligned} Moves(\Sigma) \coloneqq \quad & \{(a,a) \mid a \in \Sigma\} && \textit{synchronous moves} \\ & \cup \{(a,\gg) \mid a \in \Sigma\} && \textit{model moves} \\ & \cup \{(\gg,a) \mid a \in \Sigma\} && \textit{log moves.} \end{aligned}$$

An *alignment* $\gamma \in Moves(\Sigma)^*$ between $w \in \Sigma^*$ and a process tree T is a sequence of moves $\gamma = \langle m_1, \ldots, m_n \rangle$ such that $\pi_1^{\Sigma}(\gamma) = w$ and $\pi_2^{\Sigma}(\gamma) \in \mathcal{L}(T)$.

In other words, γ forms an alignment if the first components of each move in γ yield the trace w (when we remove all skip symbols $\gg$) and the second components yield a trace in the language of the process tree T (again without skip symbols). Intuitively, we aim to modify the trace w (the first component) such that it becomes a trace in the language of the process tree T (the second component). From this point of view, a log move $(a, \gg)$ deletes the symbol a from w while a model move $(\gg, b)$ inserts the symbol b into the trace w.

We determine the *costs* $c(\gamma)$ of an alignment γ by summing up the costs $c(m)$ of the individual moves m in γ where synchronous moves have cost 0 and, with respect to the standard cost function, log and model moves have cost 1 (other cost functions are possible). The set of all alignments between a trace w and a process tree T is denoted by $\Gamma(w, T)$. An *optimal alignment* $\gamma_{opt} \in \Gamma(w, T)$ is an alignment with minimal costs $\sum c(\gamma_{opt})$ among all alignments in $\Gamma(w, T)$.

4 Structure of Process Tree Alignments

Process trees have an inductive definition which lends itself to recursive algorithms. We next show that this inductive structure carries over to the set of alignments as well.

For a trace $w \in \Sigma^*$ of length n, $w = \langle w_1, w_2, \ldots, w_n \rangle$, we call a mapping $\varphi \colon \{1, \ldots, n\} \to \{1, 2\}$ a *factorization* of w. For a factorization φ of w we define $\varphi_1 \in \Sigma^*$ as the trace that results by concatenating all symbols w_i with $\varphi(i) = 1$. Likewise, $\varphi_2 \in \Sigma^*$ denotes the trace that results by concatenating all symbols w_i with $\varphi(i) = 2$. For the special case where $n = 0$, we only have a single factorization $\varphi = \emptyset$ with $\varphi_1 = \varphi_2 = \langle\rangle$. We write $\Phi(w)$ to denote the set of all factorizations of w. Note the connection between factorizations and the shuffle operator: for $w, w_1, w_2 \in \Sigma^*$ we have $w \in w_1 \sqcup\!\sqcup w_2$ if and only if there exists a factorization φ of w such that $\varphi_1 = w_1$ and $\varphi_2 = w_2$. In this sense, the *factorization* can be seen as a kind of inverse of the *shuffle* operator.

Theorem 1 (Structure of Alignments over Process Trees)**.** *Let T_1 and T_2 be process trees and $w \in \Sigma^*$ be a trace. Then the following holds.*

$$\Gamma(w, T_1 \to T_2) = \bigcup_{w_1 \cdot w_2 = w} \Gamma(w_1, T_1) \cdot \Gamma(w_2, T_2) \tag{1}$$

$$\Gamma(w, T_1 \times T_2) = \Gamma(w, T_1) \cup \Gamma(w, T_2) \tag{2}$$

$$\Gamma(w, T_1 \wedge T_2) = \bigcup_{\varphi \in \Phi(w)} \{\gamma \in \Gamma(\varphi_1, T_1) \sqcup\!\sqcup \Gamma(\varphi_2, T_2) \mid \pi_1^{\Sigma}(\gamma) = w\} \tag{3}$$

$$\Gamma(w, T_1 \circlearrowleft T_2) = \bigcup_{k \in \mathbb{N}_0} \{\Gamma(w_0, T_1) \cdot \Gamma(y_1, T_2) \cdot \Gamma(z_1, T_1) \cdots \Gamma(y_k, T_2) \cdot \Gamma(z_k, T_1) \mid w = w_0 y_1 z_1 \dots y_k z_k, w_0, y_i, z_i \in \Sigma^*, 1 \leq i \leq k\} \tag{4}$$

Proof. Ad (1): Let $T = T_1 \to T_2$ and $w \in \Sigma^*$ be a trace. We show that $\Gamma(w, T) = \bigcup_{w_1 \cdot w_2 = w} \Gamma(w_1, T_1) \cdot \Gamma(w_2, T_2)$. The direction $\supseteq$ is obvious, so let us focus on the direction $\subseteq$. Let $\gamma \in \Gamma(w, T)$. Since $T = T_1 \to T_2$, we find $y_1 \in \mathcal{L}(T_1)$ and $y_2 \in \mathcal{L}(T_2)$ such that $\pi_2^{\Sigma}(\gamma) = y_1 \cdot y_2$. Hence, we can write $\gamma = \gamma_1 \cdot \gamma_2$ with $\pi_2^{\Sigma}(\gamma_1) = y_1$ and $\pi_2^{\Sigma}(\gamma_2) = y_2$. Define $w_1 = \pi_1^{\Sigma}(\gamma_1)$ and $w_2 = \pi_1^{\Sigma}(\gamma_2)$. Then we have $w = w_1 \cdot w_2$ and $\gamma_1 \in \Gamma(w_1, T_1)$ and $\gamma_2 \in \Gamma(w_2, T_2)$.
Ad (2): Straightforward.
Ad (3): For $\supseteq$, observe that for a projection operator π and sequences x, y we have $\pi(x \sqcup\!\sqcup y) = \pi(x) \sqcup\!\sqcup \pi(y)$. For the direction $\subseteq$, let $\gamma \in \Gamma(w, T)$. Let $y = \pi_2^{\Sigma}(\gamma)$. Since $T = T_1 \wedge T_2$, we find a factorization φ of y such that $y_1 \coloneqq \varphi_1 \in \mathcal{L}(T_1)$ and $y_2 \coloneqq \varphi_2 \in \mathcal{L}(T_2)$. We lift this factorization to a factorization of γ by assigning to each log move m in γ the value 2 (the choice of 2 is arbitrary and we could have chosen 1 as well). Call the resulting factorization ψ and let $\gamma_1 = \psi_1$ and $\gamma_2 = \psi_2$. Let $w_1 = \pi_1^{\Sigma}(\gamma_1)$ and $w_2 = \pi_1^{\Sigma}(\gamma_2)$. Then, $w \in w_1 \sqcup\!\sqcup w_2$ since $w = \pi_1^{\Sigma}(\gamma)$. Moreover, $\gamma_1 \in \Gamma(w_1, T_1)$ and $\gamma_2 \in \Gamma(w_2, T_2)$ since $\pi_2^{\Sigma}(\gamma_1) = y_1$ and $\pi_2^{\Sigma}(\gamma_2) = y_2$ (we have only assigned new log moves to the second component of the alignment). This concludes the argument for the parallel operator.
Ad (4): We can get a decomposition analogously as for the sequence operator (1) using the semantics of the loop operator $T_1 \circlearrowleft T_2$ as $\mathcal{L}(T_1) \cdot (\mathcal{L}(T_2) \cdot \mathcal{L}(T_1))^*$. □

From Theorem 1 we can derive a recursive algorithm for computing an optimal alignment between a trace w and a process tree T. Let $Cost(w, T)$ denote the minimal costs of an alignment in $\Gamma(w, T)$, i.e.,

$$Cost(w, T) = \min\{\textstyle\sum c(\gamma) \mid \gamma \in \Gamma(w, T)\}.$$

Then, we have the following recursive procedure for computing $Cost(w, T)$.

Theorem 2 (Recursive Computation of Alignment Costs)**.** *Let T_1 and T_2 be process trees and $w \in \Sigma^*$ a trace. Then the following holds.*

$$Cost(w, T_1 \to T_2) = \min_{w_1 \cdot w_2 = w} \{Cost(w_1, T_1) + Cost(w_2, T_2)\}$$

$$Cost(w, T_1 \times T_2) = \min\{Cost(w, T_1), Cost(w, T_2)\}$$

$$Cost(w, T_1 \wedge T_2) = \min_{\varphi \in \Phi(w)} \{Cost(\varphi_1, T_1) + Cost(\varphi_2, T_2)\}$$

$$Cost(w, T_1 \circlearrowleft T_2) = \min_{k \in \mathbb{N}_0} \left\{ Cost(w_0, T_1) + \sum_{i=1}^{k} Cost(y_i, T_2) + Cost(z_i, T_1) \;\middle|\; w = w_0 y_1 z_1 \dots y_k z_k, w_0, y_i, z_i \in \Sigma^*, 1 \le i \le k \right\}$$

Proof. This follows immediately from Theorem 1. Note that for the case of the parallel operator (3) we have dropped the condition $\pi_1^{\Sigma}(\gamma) = w$. This can be justified as follows. If $\varphi \in \Phi(w)$ and $\gamma \in \gamma_1 \sqcup\!\sqcup \gamma_2$ with $\gamma_1 \in \Gamma(\varphi_1, T_1)$ and $\gamma_2 \in \Gamma(\varphi_2, T_2)$, then the costs of *all* alignments in $\gamma_1 \sqcup\!\sqcup \gamma_2$ are the same (they consist of the same moves) *and* at least for one $\gamma \in \gamma_1 \sqcup\!\sqcup \gamma_2$ we have $\pi_1^{\Sigma}(\gamma) = w$. □

The missing base cases for $Cost(w, T)$ can be determined easily.

Theorem 3 (Alignment Costs for Base Cases)**.** *Let $w \in \Sigma^*$ be a trace and $a \in \Sigma$. Then $Cost(w, \tau) = |w|$. Moreover, $Cost(w, a) = |w| + 1$ if a does not occur in w and $Cost(w, a) = |w| - 1$ otherwise.*

5 A Dynamic Programming Algorithm

The recursive computation of the optimal alignment costs $Cost(w, T)$ can be turned into a dynamic programming algorithm. We use the formulae from Sect. 4 and avoid recomputation for identical subproblems by storing the results of $Cost(w, T)$ in a table which we denote by *CostTable*, see Algorithm 1.

As presented, Algorithm 1 has exponential runtime. First, the number of recursive calls required for the *loop operator* $T_1 \circlearrowleft T_2$ corresponds to the (exponential) number of decompositions of w into subtraces $w = w_0 y_1 z_1 \dots y_k z_k$. However, this blowup can be avoided. Consider a graph on the positions of w, $n := |w|$, with an edge from position $0 \le p \le n$ to position $n \ge q \ge p$ with costs

$$\min_{p \le r \le q} \{Cost(w[p, r], T_2) + Cost(w[r, q], T_1)\}.$$

These are the costs of aligning the subtrace $w[p, q]$ of w against the process tree $T_2 \to T_1$. In turn, a *path* from p to $n := |w|$ corresponds to a *partition* of the suffix $w[p, n]$ of w into segments (each edge yields one segment) where each segment is aligned against $T_2 \to T_1$. Specifically, the costs of a *cost-minimal* path from p to n are the costs of an optimal alignment of $w[p, n]$ against $(\mathcal{L}(T_2) \cdot \mathcal{L}(T_1))^*$.

Algorithm 1. Dynamic Programming Algorithm to Compute $Cost(w, T)$

$CostTable := \emptyset$
function $\textsc{cost}(w, T)$
  if $(w, T) \in$ CostTable **then return** CostTable(w, T)
  if $T = a$ **then** cost $\leftarrow |w| - 1$ if a occurs in w, else cost $\leftarrow |w| + 1$
  else if $T = \tau$ **then** cost $\leftarrow |w|$
  else if $T = T_1 \rightarrow T_2$ **then**
    for all $w_1 \cdot w_2 = w$ **do**
      cost $\leftarrow \min\{\text{cost}, \textsc{cost}(w_1, T_1) + \textsc{cost}(w_2, T_2)\}$
  else if $T = T_1 \times T_2$ **then** cost $\leftarrow \min\{\textsc{cost}(w, T_1), \textsc{cost}(w, T_2)\}$
  else if $T = T_1 \wedge T_2$ **then**
    for all $\varphi \in \Phi(w)$ **do**
      cost $\leftarrow \min\{\text{cost}, \textsc{cost}(\varphi_1, T_1) + \textsc{cost}(\varphi_2, T_2)\}$ (see improvements below)
  else if $T = T_1 \circlearrowleft T_2$ **then**
    for all $w_0 y_1 z_1 \dots y_k z_k = w$ with $w_0, y_i, z_i \in \Sigma^*, 1 \leq i \leq k, k \in \mathbb{N}_0$ **do**
      cost $\leftarrow \min\{\text{cost}, \textsc{cost}(w_0, T_1) + \textsc{cost}(y_1, T_2) + \textsc{cost}(z_1, T_1) + \dots + \textsc{cost}(y_k, T_2) + \textsc{cost}(z_k, T_1)\}$ (see improvements below)
  CostTable$(w, T) \leftarrow$ cost
  return cost

Hence, these costs can be determined efficiently with a shortest-path algorithm. This yields polynomial runtime for the loop operator.

The second problematic case is the *parallel operator* $T_1 \wedge T_2$. Here, we have to consider all factorizations of the trace w into two subtraces w_1 and w_2 which is an exponential number (in the length of w). In contrast to the loop operator, this exponential search cannot be avoided in general (unless $\mathsf{P} = \mathsf{NP}$). However, for process trees with *unique labels*, the situation is different.

Process Trees with Unique Labels. Let us reconsider the case $T = T_1 \wedge T_2$ where

$$Cost(w, T) = \min_{\varphi \in \Phi(w)} \{Cost(\varphi_1, T_1) + Cost(\varphi_2, T_2)\},$$

from Theorem 2 for process trees with *unique labels*. Let $L_1 = Letters(T_1)$ and $L_2 = Letters(T_2)$ be the sets of labels occurring in T_1 and T_2, respectively. We claim that we can restrict the set of factorizations to a singleton. Indeed, each letter w_i of w either belongs to L_1 or L_2 (or to none of them). For example, if $w_i = a$, and we know that $a \in L_1$, then it cannot reduce the alignment costs if we assign a to T_2. In fact, in T_2 we have to delete a anyway (log move $(a, \gg)$) and we could do exactly the same in T_1 (without increasing costs). In other words, without loss of generality we can assume that $\varphi(i) = 1$. We can argue analogously for letters in L_2. If the letter a at position i does neither belong to L_1 nor to L_2, then we can assign it to T_1 or T_2 without changing the alignment costs (in both cases, a deletion move $(a, \gg)$ is unavoidable). Arbitrarily, we assign such letters to T_2. In conclusion, for the *single* factorization $\varphi^\star$ with $\varphi^\star(i) = 1$ if $w_i \in L_1$

and $\varphi^*(i) = 2$ otherwise, we have:

$$Cost(w, T) = Cost(\varphi_1^\star, T_1) + Cost(\varphi_2^\star, T_2).$$

With these adaptations, Algorithm 1 becomes polynomial-time. To see this, let w be the input trace, and let T denote the input tree. Let $n = |w|$. A first observation is that the total number of entries in *CostTable* is bounded by $\mathcal{O}(|T| \cdot n^2)$. This is because each entry (v, T') in *CostTable* is determined by T' together with a segment $w[i, j]$, $1 \leq i \leq j \leq n$, of the original input trace w (indeed, v either is the segment $w[i, j]$ itself or the restriction of $w[i, j]$ to letters that occur in T'). This is in contrast to the case of process trees with *non-unique* labels where the shuffle operator would produce an exponential number of recursive calls for its subtrees (and the corresponding traces v could not easily be described as segements of the original input trace). With the same argument, $\mathcal{O}(|T| \cdot n^2)$ is a bound on the number of recursive calls of the function $\textsc{Cost}(w, T)$.

Secondly, we bound the runtime for each call of $\textsc{Cost}(w, T)$ (besides the recursive calls). By going through the different cases, it can be seen that the most expensive step is the shortest path computation for the loop operator. Here, we compute a cost-minimal path on a graph with $\mathcal{O}(|v|)$ nodes. Since $|v| \leq n$, and since shortest paths can be computed in quadratic time in the number of vertices (e.g., by using Dijkstra), the total runtime for a call of $\textsc{Cost}(w, T)$ is bounded by $\mathcal{O}(n^2)$ (not considering the runtime for the sparked recursive calls, of course). Altogether this yields a runtime bound of $\mathcal{O}(|T| \cdot n^4)$.

Theorem 4 (Dynamic Programming Algorithm for Process Trees with Unique Labels)**.** *The costs $Cost(w, T)$ of an optimal alignment between a process tree T with unique labels and a trace w can be computed efficiently in time $\mathcal{O}(|T| \cdot |w|^4)$.*

6 Evaluation

We implemented our novel alignment algorithm in Python[1] and compared its runtime against the available algorithms in `PM4Py` [4] on a set of real-life event logs. We like to discuss one further algorithmic idea which lead to a significant speed-up on the benchmarks. Consider the sequence operator $T_1 \rightarrow T_2$ and recall that

$$Cost(w, T) = \min_{w_1 \cdot w_2 = w} \{Cost(w_1, T_1) + Cost(w_2, T_2)\}.$$

Implemented naively, we need to check n splits of w into subtraces w_1 and w_2 where $n = |w|$. Let $L = Letters(T_1)$ and $R = Letters(T_2)$ be the sets of labels occurring in the (left) subtree T_1 and the (right) subtree T_2, respectively. Because of the unique label property, $L \cap R = \emptyset$. Let us label the letters in w with L if they belong to L and with R if they belong to R. Call the resulting $\{L, R\}$-trace $decomp(w)$. Of course, w could contain letters that neither belong to L nor to R. Such letters a can be removed in a preprocessing step (they incur deletion costs

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

anyway). Hence, we can assume that $decomp(w)$ and w have the same length. We claim that it is sufficient to check only the following split positions of the trace w: $seg = \{1, n\} \cup \{i \colon decomp(w)_i = L \wedge decomp(w)_{i+1} = R\}$.

To see why, let $i \in seg$ be a position with a flip from L- to R-labels. Then, by definition, this position is followed by R-labels. The alignment costs can only *increase* for a split within the upcoming R-segment. This is because to handle R-labels in the left subtree T_1 we need to delete them. Moreover, if the R-part is followed by an L-segment, then it makes sense to include as many L-labels for the next split as possible (since L-labels will necessarily incur deletion costs in the right subtree T_2). Hence, the next optimal split can only be after the last L-label (either at the end of the trace or right before the next R-label).

We compared our algorithm (*Dynamic*) against the standard (A*-based) algorithm for computing alignments in `PM4Py` (*Standard*) and an approximation algorithm in `PM4Py` tailored for process trees (*Approx*). For each algorithm and trace variant, we took the best out of 5 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, i.e., 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 for the given trace. We set a timeout of 65 s (instead of say 60 s) to compensate for overhead and to give each algorithm the safe chance to finish its computation in one minute. If a computation hits the timeout in one of the repetitions, the algorithm is considered to have failed on the trace/model pair. In the chart below, we plotted the empirical CDF of the performance factors for each of the three algorithms. The frequencies of performance factors of some algorithms do not sum up to 1; this indicates, that the algorithms ran into timeouts on a certain fraction of instances.

Log Data and Results. The general picture is that our algorithm (*Dynamic*) is *very* close to the approximation algorithm (*Approx*) and, in almost all cases, clearly outperforms the standard algorithm (*Standard*). Let us start with the BPI Challenge 2019 event log [12]. We used the *Inductive Miner* [15] to discover process trees with different noise thresholds (0 %, 10 %, 25 % and 50 %) and aligned 1000 randomly chosen trace variants from the log against the resulting process trees. The CDF of the performance factors is depicted in Fig. 1. The analogous graph for the Hospital Billing event log [17] can be found in Fig. 2.

On the BPI Challenge 2017 event log [10], our algorithm is slightly superior to Approx for noise thresholds of 10 % and 50 %, while it is slightly below the performance of Approx for thresholds of 0 % and 25 %. On the BPI Challenge 2012 event log [9], our algorithm is slightly superior to Approx for noise thresholds of 0 % and 10 %, while it is slightly below the performance of Approx for thresholds of 25 % and 50 %. This is with respect to the CDF of performance factors. To give some further results, Table 1 depicts the median computation times of the three algorithms for the runs on the BPI Challenge 2012 and 2017 event logs with respect to the different noise thresholds (0 %, 10 %, 25 %, 50 %).

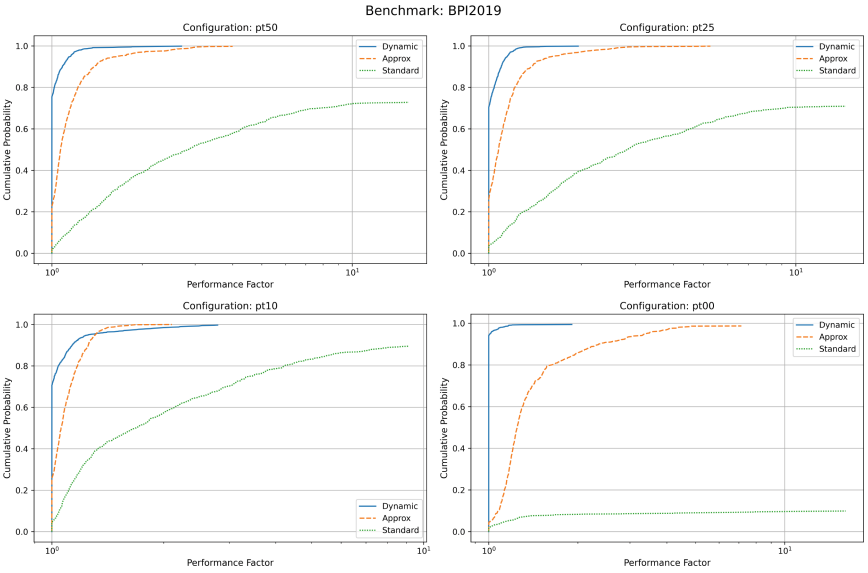

Fig. 1. CDF of the performance factors of *Dynamic*, *Approx*, *Standard* in `PM4Py` on the *BPI Challenge 2019* event log [12] with different noise thresholds.

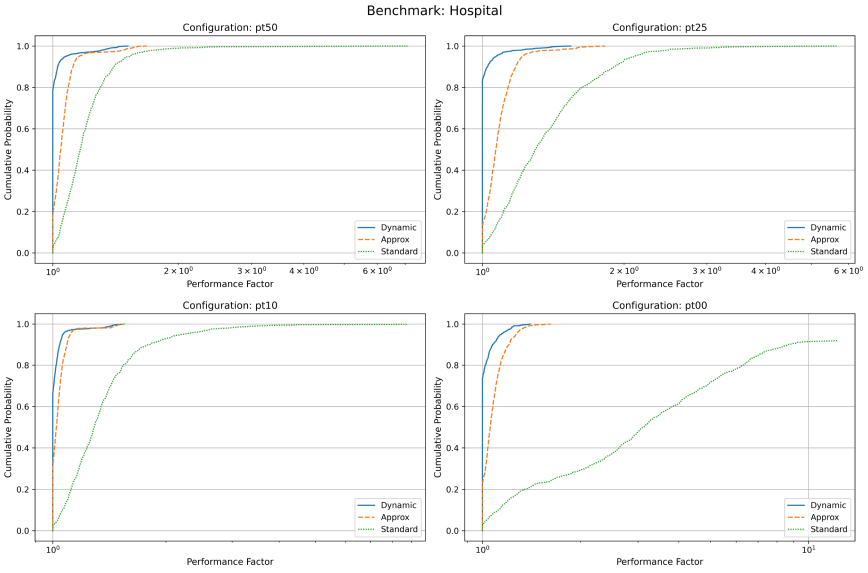

Fig. 2. CDF of the performance factors of *Dynamic*, *Approx*, *Standard* in `PM4Py` on the *Hospital Billing* event log [17] with different noise thresholds.

Table 1. Median computation times (in seconds) of *Dynamic*, *Approx*, and *Standard* in `PM4Py` for the BPI Challenge 2012 and 2017 event logs with different noise thresholds.

Threshold	BPI Challenge 2012 [9]				BPI Challenge 2017 [10]			
	50 %	*25 %*	*10 %*	*0 %*	*50 %*	*25 %*	*10 %*	*0 %*
Dynamic	5.69	**5.24**	**4.86**	**4.91**	**4.92**	8.24	**4.88**	6.00
Approx	**5.68**	5.25	5.48	5.51	5.37	**6.18**	5.29	**5.91**
Standard	21.40	22.49	7.91	8.48	17.44	40.07	9.87	37.56

7 Conclusion

We proved that the alignment problem for process trees with unique labels can be solved in polynomial time using dynamic programming. A proof-of-concept implementation in Python demonstrates that our algorithm is competitive with (and in some cases outperforms) the existing techniques of the `PM4Py` library. We discussed ideas how the algorithm can be further optimized in practice.

This article is part of a broader research agenda where we try to understand better the structure and algorithmic complexity of the alignment problem. We saw an interesting, and practically relevant, class of process models, where the alignment problem can be solved in polynomial time. This is rather the exception than the rule, since the alignment problem has high complexity in general (PSPACE-hard for sound workflow nets). Our work leads to many questions for future research. For example, it would be interesting to study relaxations of the *unique label* property and study the influence of these parameters on the complexity. Also, it would be interesting to see how restrictive the *unique label* property really is. Can we get a characterization of the event logs that can be defined using process trees with unique labels? And, as sound workflow nets with unique labels are more powerful than process trees with unique labels (in terms of modeling power), what is the complexity of the alignment problem for sound workflow nets with unique labels?

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. van der Aalst, W.M.P.: Decomposing petri nets for process mining: a generic approach. Distrib. Parallel Datab. **31**(4), 471–507 (2013). https://doi.org/10.1007/s10619-013-7127-5
2. van der Aalst, W.M.P., Buijs, J.C.A.M., 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

3. Adriansyah, A.: Aligning observed and modeled behavior. Ph.D. dissertation, Technische Universiteit Eindhoven (2014). https://doi.org/10.6100/IR770080
4. Berti, A., van Zelst, S.J., Schuster, D.: PM4Py: a process mining library for Python. Softw. Impacts **17**, 100556 (2023). https://doi.org/10.1016/j.simpa.2023.100556
5. 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, pp. 50–59. IEEE Computer Society (2018). https://doi.org/10.1109/ACSD.2018.00008
6. 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
7. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking, Relating Processes and Models. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99414-7
8. Carmona, J., van Dongen, B.F., Weidlich, M.: Conformance checking: foundations, milestones and challenges. In: Process Mining Handbook, LNBIP, vol. 448, pp. 155–190. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-08848-3_5.
9. van Dongen, B.F.: BPI challenge 2012. Eindhoven University of Technology (2012). https://doi.org/10.4121/UUID:3926DB30-F712-4394-AEBC-75976070E91F
10. van Dongen, B.F.: BPI challenge 2017. Eindhoven University of Technology (2017). https://doi.org/10.4121/UUID:5F3067DF-F10B-45DA-B98B-86AE4C7A310B
11. 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
12. van Dongen, B.F.: BPI challenge 2019, 4TU.Centre for Research Data (2019). https://doi.org/10.4121/uuid:d06aff4b-79f0-45e6-8ec8-e19730c248f1.
13. Leemans, S.J.J.: Robust Process Mining with Guarantees, Process Discovery, Conformance Checking and Enhancement (LNBIP), vol. 440. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-96655-3.
14. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs containing infrequent behaviour. In: Lohmann, N., Song, M., Wohed, P. (eds.) BPM 2013. LNBIP, vol. 171, pp. 66–78. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06257-0_6
15. 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
16. 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
17. Mannhardt, F.: Hospital billing - event log. Eindhoven University of Technology (2017). https://doi.org/10.4121/uuid:76c46b83-c930-4798-a1c9-4be94dfeb741
18. 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
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. Schwanen, C.T., Pakusa, W., van der Aalst, W.M.P.: Process tree alignments. In: Enterprise Design, Operations, and Computing. LNCS. Springer, Cham (2024), forthcoming. https://doi.org/10.1007/978-3-031-78338-8_16

21. 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