Alignments meet Linear Algebra

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*

[2]*Faculty of Mathematics, Informatics and Technology, Koblenz University of Applied Sciences, Koblenz, Germany*

Abstract

An optimal alignment corresponds to the minimal number of insertions and deletions of events to transform an observed trace into a trace of a process model. For the important class of process trees, we show how we can express the alignment problem as a system of matrices over a commutative semiring. This system of matrices only depends on the process tree, but not on the input trace, and the costs of alignments can be computed by matrix and vector multiplications. Our formulation in terms of linear algebra sheds new light on alignments and allows us to transfer methods from linear algebra into the field of conformance checking. As an application, we show that for a subclass of process trees with unique labels, we can efficiently decide alignment properties by using a symbolic representation of the matrices.

Keywords

Process Mining, Conformance Checking, Alignments, Linear Algebra, Process Trees

1. Introduction

Alignments [1] are the state-of-the-art technique in process mining to measure the deviation of an observed process execution to a normative process model. To this end, optimal alignments count the minimum number of insertions and deletions of events that is required to transform an observed trace into a trace of the reference model. Unfortunately, computing the alignment metric is computationally intractable. More precisely, it can be shown that for the standard class of process models used in process mining (so-called *sound workflow nets*) the alignment problem is PSPACE-complete [13]. At the same time, alignments play a pivotal role in a large number of applications in process mining. Hence, one of the most important research goals in conformance checking is to find algorithmic strategies that make alignment computations applicable on large real-world event logs and, in the best case, even allow for provable complexity bounds on interesting model classes. Along these lines, we were recently able to classify the algorithmic complexity of alignments on quite relevant classes of process models, such as *free-choice sound workflow nets*, *process trees*, and *process trees with unique labels* [13–15].

Here, we continue the quest towards a more thorough understanding of the complexity of alignments. In our main result Theorem 4.1 we establish a completely new perspective on the alignment problem for process trees that is rooted in linear algebra. More precisely, we are going to show that every process tree T can effectively be converted into a linear system of matrices $(S_a)_{a\in\Sigma}$ and vectors v_{in} and v_{fin} with entries in a commutative semiring R such that the optimal alignment costs for any given trace $w = w_1 w_2 \cdots w_n \in \Sigma^*$ (with respect to the process tree T) can be computed as a matrix multiplication problem as follows:

$$\text{Costs}(w, T) = v_{\text{in}}^t \cdot S_{w_1} S_{w_2} \ \cdots \ S_{w_n} \cdot v_{\text{fin}} \in R. \tag{1}$$

Our work is strongly inspired by the notion of *weighted automata* [5] known from the field of formal languages and automata theory. In a nutshell, a weighted automaton can (non-deterministically) process an input along different computation paths where each single path has certain costs. These costs are the product of the costs of transitions taken along the path. In the end, the sum over all computation paths is assigned as the costs of the read input. All arithmetic (sums and products) takes place in some

schwanen@pads.rwth-aachen.de (C. T. Schwanen); pakusa@hs-koblenz.de (W. Pakusa); wvdaalst@pads.rwth-aachen.de (W. M. P. van der Aalst)

0000-0002-3215-7251 (C. T. Schwanen); 0009-0004-6302-4445 (W. Pakusa); 0000-0002-0955-6940 (W. M. P. van der Aalst)

underlying commutative semiring R that can vary depending on the modeled situation. In this way, a weighted automaton does not only define a formal language (i.e., a Boolean function), but specifies a numeric function of the form $f\colon \Sigma^* \to R$ which assigns a certain cost to each input string. Somewhat the key idea of our whole construction is to show that the alignment cost function of every process tree is regular in the sense that it can be expressed by a weighted automaton.

To benefit from our new construction algorithmically, we have to take into account that process trees yield a *succinct* representation of an exponentially large state space. Indeed, while each process tree defines a regular language, the standard translation from process trees to finite automata results in an exponential increase in the number of states. Thus, in an explicit representation, the weighted automata (and matrices $(S_a)_{a\in\Sigma}$) become excessively large. The operator responsible for this blow-up is the *shuffle* operator, which models independent parallel computations. Hence, our second key idea is to preserve the compact, *symbolic* representation of process trees by expressing the translation using linear algebra and standard matrix/vector operations. Remarkably, we find that for *all* process tree operators, there exist corresponding standard linear-algebraic operators that accurately capture the alignment semantics of process trees. In particular, we show that the shuffle operator used in process trees corresponds to the well-known *Kronecker product* of matrices. This insight allows us to define our translation using only symbolic algebraic expressions for the matrices S_a and vectors v_{in} and v_{fin} with the properties described in Equation (1) and such that the sizes of the symbolic expressions coincide with the size of the process tree T itself (up to polynomial factors).

We also showcase first applications. In particular, we prove that for a certain subclass of process trees with unique labels we can use a symbolic representation of the matrices and vectors to obtain an efficient alignment algorithm. To this end, we show how to efficiently implement the required matrix and vector operations in a symbolic way. We also explain how our approach can enable the transfer of concepts from linear algebra to the field of conformance checking thereby providing new analysis techniques for the process mining community. These examples highlight the great potential of our new formulation for future research on the algorithmic structure of alignments.

2. Related Work

Alignments [1] are the state-of-the-art technique for conformance checking, an introduction to the field is provided in [2, 3]. When restricting the class of process models to *process trees with unique labels*, the resulting model of the well-known *Inductive Miner* family [7–9] and the de-facto standard in industrial process mining applications, the computation of alignments becomes tractable [14]. But even for process trees in their general form, alignments can still be computed more efficiently than on other models, e.g., sound workflow nets [13, 15].

Weighted automata can be traced back to the 1960s where formal power series were studied as a kind of generalization of formal languages, see e.g. [12]. While classic automata decide if a given word is accepted or not (a Boolean property), weighted automata define a numerical value for inputs in a commutative semiring. The semiring can model, e.g. resources, costs, time, access rights, or a probability for each input. They have manifold applications, for example in the area of model checking [4] or natural language processing [11]. We refer to [5, 6] for a survey and handbook for more details.

To algorithmically apply our linear-algebraic formulation, we have to represent the matrices and vectors in some compact, symbolic way (instead of working with the explicit, exponential-size representation). A similar approach has been taken, for example, in [10]. Here the authors show that computational problems for symbolic matrices are provably very hard. However, in our context, the matrices have a quite restricted form, so we are still able to implement certain operations efficiently (and, indeed, in Section 5 we prove that this is possible to a certain extent).

3. Preliminaries

Let $\mathbb{N} \coloneqq \{0, 1, 2, \dots\}$ be the set of natural numbers. For an n-tuple $a \in A_1 \times \dots \times A_n$, $\pi_i(a)$ denotes the *projection* on its ith element, i.e., $\pi_i \colon A_1 \times \dots \times A_n \to A_i, (a_1, \dots, a_n) \mapsto a_i$.

Definition 3.1 (Alphabet). An *alphabet* Σ is a finite, non-empty set of *labels* (also called *activities*).

Definition 3.2 (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 $\sum_{i \in I} a_i$. Given two sequences σ and σ', $\sigma \cdot \sigma'$ (or $\sigma\sigma'$ in short) denotes the concatenation of the two sequences. For $a_1, a_2, \dots, a_n \in A$, we also use $a_1 a_2 \cdots a_n$ to denote a sequence. 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 $\pi_i^*(\langle (a_1, \dots, a_n) \rangle \cdot \sigma) \coloneqq \langle \pi_i(a_1, \dots, a_n) \rangle \cdot \pi_i^*(\sigma) = \langle a_i \rangle \cdot \pi_i^*(\sigma)$. 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., $\pi_i^B \coloneqq \pi_i^*|_B$.

3.1. Process Trees

Languages of traces $\mathcal{L} \subseteq \Sigma^*$ correspond to sets of behaviors of a process. The symbols in one trace correspond to the *events* or *activities* that occurred. Here, 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. We recall the *shuffle operator* $\sqcup\!\sqcup$ which captures parallel execution within a process. Formally, this is defined by taking two traces $x, y \in \Sigma^*$ and by defining $x \sqcup\!\sqcup\, y$ to be the set of all traces obtained by interleaving the symbols of x and y while preserving their relative order. For example, if $x = ab$ and $y = cd$, then $x \sqcup\!\sqcup\, y = \{abcd, acbd, acdb, cabd, cadb, cdab\}$.

Definition 3.3 (Shuffle $\sqcup\!\sqcup$). For $x, y \in \Sigma^*$, the *shuffle* $x \sqcup\!\sqcup\, y$ of x and y is

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

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 3.4 (Process Tree). Let Σ be an alphabet and let $\tau \notin \Sigma$ be the silent activity. Both, the set of *process trees* (over Σ) and the *language* of a process tree T, denoted by $\mathcal{L}(T)$, are defined recursively:

- the silent activity τ is a process tree where $\mathcal{L}(\tau) = \{\langle\rangle\}$,
- each activity $a \in \Sigma$ is a process tree where $\mathcal{L}(a) = \{\langle a \rangle\}$,
- for process trees $T_1, \dots, T_n$, $n \in \mathbb{N} \setminus \{0\}$, the following are also process trees:
  - $\to(T_1, \dots, T_n)$ where $\mathcal{L}(\to(T_1, \dots, T_n)) = \mathcal{L}(T_1) \cdot \ldots \cdot \mathcal{L}(T_n)$,
  - $\times(T_1, \dots, T_n)$ where $\mathcal{L}(\times(T_1, \dots, T_n)) = \mathcal{L}(T_1) \cup \ldots \cup \mathcal{L}(T_n)$,
  - $\wedge(T_1, \dots, T_n)$ where $\mathcal{L}(\wedge(T_1, \dots, T_n)) = \mathcal{L}(T_1) \sqcup\!\sqcup \ldots \sqcup\!\sqcup\, \mathcal{L}(T_n)$, and
  - $\circlearrowleft(T_1, T_2)$ where $\mathcal{L}(\circlearrowleft(T_1, T_2)) = \mathcal{L}(T_1) \cdot (\mathcal{L}(T_2) \cdot \mathcal{L}(T_1))^*$.

The symbols $\to$ (sequence), $\times$ (exclusive choice), $\circlearrowleft$ (loop), and $\wedge$ (parallel) are *process tree operators.* A *process tree with unique labels* is a process tree where each activity occurs at most once.

We assume that *all operators* ($\to, \times, \wedge$) *are binary* (i.e. $n = 2$ in Definition 3.4). This is no restriction, since a general process tree can efficiently be transformed into an equivalent *binary* process tree.

3.2. Alignments

An *alignment* between a trace $w = w_1 \cdots w_n \in \Sigma^*$ and a process tree T is a trace $\gamma = \gamma_1 \cdots \gamma_m$ which consists of labels γ_i which are called *moves*. A move is either a pair of labels (a, a) for $a \in \Sigma$ (which we call a *synchronous move*), or a pair $(a, \gg)$ for $a \in \Sigma$ (which we call a *log move*), or a pair $(\gg, a)$ for $a \in \Sigma$ (which we call a *model move*). Here $\gg$ is a special symbol which indicates that an event is skipped in the trace or model. The first components of the moves in γ should yield the trace w (when we remove all skip symbols $\gg$) and the second components should yield a trace in the language of the process tree T (again without skip symbols). Intuitively, we aim to modify the trace w such that it becomes a trace in the language of the process tree T. From this point of view, a log move $(a, \gg)$ deletes the symbol a from w while a model move $(\gg, a)$ inserts the symbol a into the trace w.

Definition 3.5 (Move, Alignment). 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 $\Sigma_{\gg} := \Sigma \cup \{\gg\}$ as the alphabet extended by the skip-symbol $\gg$. We define $LM(\Sigma) \subseteq \Sigma_{\gg} \times \Sigma_{\gg}$ as the set of all *legal moves* over Σ given by

$$\begin{aligned} LM(\Sigma) :=\ & \{(a, a) \mid a \in \Sigma\} && \textit{synchronous moves} \\ & \cup \{(a, \gg) \mid a \in \Sigma\} && \textit{model moves} \\ & \cup \{(\gg, a) \mid a \in \Sigma\} && \textit{log moves.} \end{aligned}$$

An *alignment* $\gamma \in LM(\Sigma)^*$ 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 $\pi_2^\Sigma(\gamma) \in \mathcal{L}(T)$.

As explained, one perspective is that a log move $(a, \gg)$ *deletes* the symbol a from w while a model move $(\gg, a)$ *inserts* the symbol a into the trace w, so alignments resembles classic edit distance without swaps of characters. We determine the *costs* $c(\gamma)$ of an alignment γ by summing up the costs $c(m)$ of the individual moves m in γ. In this paper, we use the *standard cost function* where synchronous moves have cost 0 and where log and model moves have cost 1 (other cost functions are possible in principle). The set of all alignments between a trace $w \in \Sigma^*$ 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 $c(\gamma_{opt})$ among all alignments in $\Gamma(w, T)$. Let us denote the *optimal alignment costs* for a trace w and a process tree T by $\text{Costs}(w, T)$.

3.3. Matrices and Operations

A *commutative semiring* $R = (R, +, \cdot, \mathbf{0}, \mathbf{1})$ is a set R with two binary operations $+$ and $\cdot$, and two distinguished elements $\mathbf{0}$ and $\mathbf{1}$ such that $(R, +, \mathbf{0})$ is a commutative monoid, $(R, \cdot, \mathbf{1})$ is a commutative monoid and the multiplication $\cdot$ distributes over addition $+$, i.e., $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(a + b) \cdot c = a \cdot c + b \cdot c$ for all $a, b, c \in R$. In this paper, we *fix* R as the so-called *tropical semiring* $R = (\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$, where $\mathbf{0} = \infty$ is the neutral element for addition (min) and $\mathbf{1} = 0$ is the neutral element for multiplication $(+)$. We stress that addition is $\min$ and multiplication is $+$ in the tropical semiring. To avoid confusion between the symbolic expressions $\mathbf{0}$ and $\mathbf{1}$ for the neutral elements and $0, 1 \in \mathbb{N}$ for the natural numbers 0 and 1, we use the boldface notation $\mathbf{0}$ to denote the neutral element for the addition in the semiring, and $\mathbf{1}$ to denote the neutral element for the multiplication. For the particular semiring we use throughout this article, it holds that $\mathbf{0} = \infty$ and $\mathbf{1} = 0 \in \mathbb{N}$.

We consider matrices with row index set I and column index set J and entries in R as mappings $M \colon I \times J \to R$ and write $m_{i,j} = M(i, j) \in R$ for the entry in row i and column j. Here, and in what follows, we assume that I and J are non-empty finite sets. We denote by $\mathcal{M}_{I \times J}(R)$ the set of all matrices with row index set I and column index set J and entries in R. Matrix addition on $\mathcal{M}_{I \times J}(R)$ is defined component-wise, as usual. For matrices $A \in \mathcal{M}_{I \times J}(R)$ and $B \in \mathcal{M}_{J \times K}(R)$, the matrix product $A \cdot B \in \mathcal{M}_{I \times K}(R)$ is defined by $(A \cdot B)(i, k) = \sum_{j \in J} a_{i,j} \cdot b_{j,k}$. The transpose of a matrix $M \in \mathcal{M}_{I \times J}(R)$ is the matrix $M^t \in \mathcal{M}_{J \times I}(R)$ with entries $m^t_{j,i} = m_{i,j}$. We denote the $I \times I$-identity matrix over R by $\mathbf{1}_{I \times I} \in \mathcal{M}_{I \times I}(R)$. For this matrix we have $\mathbf{1}_{I \times I}(i, j) = \mathbf{1} = 0$ if $i = j$ and $\mathbf{1}_{I \times I}(i, j) = \mathbf{0} = \infty$ otherwise.

Vectors are special cases of matrices with a row index set I and the fixed column index set $\{\top\}$, and we write $\mathcal{M}_I(R)$ for the set of all vectors with entries in R. Hence, by default, vectors are considered as column vectors. If we want to consider a vector as a row vector, we write v^t for the transpose of the vector v, which is a mapping $v^t\colon \{\top\} \times I \to R$ with $v^t(\top, i) = v(i)$ for all $i \in I$. For the special case of $\{\top\} \times \{\top\}$ matrices M, we identify them with the element $M(\top,\top) \in R$. In particular, for two I-vectors v, w we have $\langle v, w\rangle = v^t \cdot w \in R$ (*standard inner product*).

Kronecker product. For matrices $A \in \mathcal{M}_{I\times J}(R)$ and $B \in \mathcal{M}_{K\times L}(R)$, the Kronecker product $A \otimes B \in \mathcal{M}_{(I\times K)\times(J\times L)}(R)$ is the matrix with row index set $(I \times K)$ and column index set $(J \times L)$ with entries given by $(A\otimes B)((i,k),(j,l)) = a_{i,j} \cdot b_{k,l}$. Note that the sizes of the two index sets become $|I \times K| = |I| \cdot |K|$ and $|J \times L| = |J| \cdot |L|$, respectively. For more details on the Kronkecker product, cf. [16]. Most importantly, we make use of the *mixed-product property* of the Kronecker product, which states that for matrices A, B, C, D such that the matrix products $A \cdot C$ and $B \cdot D$ exist, we have

$$(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D)$$

Direct sum. For matrices $A \in \mathcal{M}_{I\times J}(R)$ and $B \in \mathcal{M}_{K\times L}(R)$, the direct sum $A \oplus B \in \mathcal{M}_{(I\uplus K)\times(J\uplus L)}(R)$ is the matrix with row index set $(I \uplus K)$ and column index set $(J \uplus L)$ with entries given by $(A \oplus B)(i,j) = a_{i,j}$ if $(i,j) \in I \times J$ and $(A \oplus B)(k,l) = b_{k,l}$ if $(k,l) \in K \times L$, and $\mathbf{0}$ otherwise. Here, $\uplus$ denotes the disjoint union of sets. Analogously, we define the direct sum of vectors $v \in \mathcal{M}_I(R)$ and $w \in \mathcal{M}_J(R)$ as the vector $v \oplus w \in \mathcal{M}_{I\uplus J}(R)$ with entries given by $(v \oplus w)(i) = v(i)$ if $i \in I$ and $(v \oplus w)(j) = w(j)$ if $j \in J$.

Matrix composition. Let $A_1 \in \mathcal{M}_{I\times J}(R), A_2 \in \mathcal{M}_{I\times K}(R), A_3 \in \mathcal{M}_{L\times J}(R)$ and $A_4 \in \mathcal{M}_{L\times K}(R)$ be matrices over R. Then we define a new matrix A over the row index set $(I \uplus L)$ and the column index set $(J \uplus K)$ given as

$$A = \begin{array}{cc} & \begin{array}{cc} J & K \end{array} \\ \begin{array}{c} I \\ L \end{array} & \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \end{array},$$

with entries $A(i,j)$ defined according to the four submatrices. We sometimes use the same notation for vectors $v_1 \in \mathcal{M}_I(R)$ and $v_2 \in \mathcal{M}_J(R)$ to define the new $(I \uplus J)$-vector v given as:

$$v = \begin{array}{cc} & \top \\ \begin{array}{c} I \\ J \end{array} & \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \end{array}.$$

Note that v corresponds to the direct sum $v_1 \oplus v_2$. Moreover, note that, on the contrary, the matrix composition cannot be expressed as a direct sum if $A_2 \neq \mathbf{0}_{I\times K}$ or $A_3 \neq \mathbf{0}_{L\times J}$.

4. Construction of Matrix Systems for Process Trees

Next, we give our main construction to effectively translate a process tree T into a matrix system which captures the alignment costs for T. Let Σ be the alphabet. We construct a family of $I \times I$-matrices $(S_a)_{a\in\Sigma}$, an I-vector $v_{\text{in}} \in \mathcal{M}_I(R)$ and an I-vector $v_{\text{fin}} \in \mathcal{M}_I(R)$ such that the optimal alignment costs for a trace $w = w_1 w_2 w_3 \cdots w_n \in \Sigma^*$ and T can be expressed as a matrix multiplication problem as follows:

$$\text{Costs}(w, T) = v_{\text{in}}^t \cdot S_{w_1} S_{w_2} \ \cdots \ S_{w_n} \cdot v_{\text{fin}} \in R. \tag{2}$$

In particular, for the empty trace $w = \varepsilon$, the optimal alignment costs are given by $v_{\text{in}}^t v_{\text{fin}}$. We introduce some notation: for a sequence $w = w_1 w_2 \cdots w_n$ we write S_w to abbreviate the product $S_{w_1} S_{w_2} \ \cdots \ S_{w_n}$. In case $w = \varepsilon$, we have $S_\varepsilon = \mathbf{1}_{I\times I}$ (the index set $I \times I$ of the matrices will become clear from the context). With this notation, Equation (2) can be written as $\text{Costs}(w, T) = v_{\text{in}}^t \cdot S_w \cdot v_{\text{fin}}$. For the further algorithmic construction, we proceed by recursion on the structure of T:

Non-occurring activities. First, if an activity $a \in \Sigma$ does not occur in T, then we can always set $S_a = 1 \cdot \mathbf{1}_{I\times I}$ (no matter of the process tree operator we are considering), where $\mathbf{1}_{I\times I}$ is the $I \times I$-identity matrix (and where I is the appropriate index set). This step is sound since $1 \cdot \mathbf{1}_{I\times I}$ naturally commutes with all $I \times I$-matrices and does not affect the product of the remaining matrices. Also, this matrix ensures that we charge an extra cost of 1 for the necessary log move $(a, \gg)$ to delete the activity a from the trace (since this activity does not occur in the process tree, this is the only way to align it).

Labeled leaf. If $T = a \in \Sigma$ is a leaf with label a, we set $I = \{q_0, q_1\}$ and define:

$$\text{For } b \in \Sigma, b \neq a\text{:} \quad S_b = \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} = 1 \cdot \mathbf{1}_{I\times I}, \qquad \text{and for } a\text{:} \quad S_a = \begin{pmatrix} 1 & \mathbf{1} \\ \mathbf{0} & 1 \end{pmatrix}.$$

Moreover, we set $v_{\text{in}} = \begin{pmatrix} \mathbf{1} \\ \mathbf{0} \end{pmatrix}$ and $v_{\text{fin}} = \begin{pmatrix} 1 \\ \mathbf{1} \end{pmatrix}$ (recall: $\mathbf{0} = \infty \neq 0$ and $\mathbf{1} = 0 \neq 1$ and $1 \cdot \mathbf{1} = 1$).

To verify correctness, first note that $v_{\text{in}}^t v_{\text{fin}} = 1$ which indeed are the alignment costs for $w = \varepsilon$ in case $T = a$. Further, note that since $S_b = 1 \cdot \mathbf{1}_{I\times I}$ for $b \neq a$, the matrix S_b commutes with all $I \times I$ matrices. Also, it is readily verified that $S_a^k = (k-1) \cdot S_a$ for $k \geq 1$. Hence, if the trace $w = w_1 \cdots w_n$ contains no a, we have $S_w = n \cdot \mathbf{1}_{I\times I}$ and if w contains $k \geq 1$ many a, we have $S_w = (n-k) \cdot (k-1) \cdot S_a = (n-1) \cdot S_a$. Since $v_{\text{in}}^t \cdot v_{\text{fin}} = 1$ and $v_{\text{in}}^t \cdot S_a \cdot v_{\text{fin}} = 0$, we have $v_{\text{in}}^t S_w v_{\text{fin}} = n+1$ if w does not contain a and $n-1$ if w contains a which proves the correctness of the construction.

Silent leaf. For the case of a silent leaf $T = \tau$, we set $I = \{q_0\}$ and define $S_a = \begin{pmatrix} 1 \end{pmatrix}$ for all $a \in \Sigma$. Moreover, we let $v_{\text{in}} = \begin{pmatrix} \mathbf{1} \end{pmatrix}$ and $v_{\text{fin}} = \begin{pmatrix} \mathbf{1} \end{pmatrix}$. Then, we have $v_{\text{in}}^t \cdot S_{w_1} S_{w_2} \cdots S_{w_n} \cdot v_{\text{fin}} = n$. These are indeed the optimal alignment costs for the trace w in the case $T = \tau$.

Exlusive Choice. If $T = T_1 \times T_2$, let S_a^i, v_{in}^i, and v_{fin}^i for $i = 1, 2$ be such that

$$\text{Costs}(w, T_i) = (v_{\text{in}}^i)^t \cdot S_{w_1}^i S_{w_2}^i \cdots S_{w_n}^i \cdot v_{\text{fin}}^i.$$

Semantically, the exclusive choice operator allows us to choose between the two branches T_1 and T_2. Hence, we define the matrices S_a and vectors v_{in} and v_{fin} as follows:

$$S_a = S_a^1 \oplus S_a^2, \quad v_{\text{in}} = v_{\text{in}}^1 \oplus v_{\text{in}}^2, \quad v_{\text{fin}} = v_{\text{fin}}^1 \oplus v_{\text{fin}}^2. \tag{3}$$

Then, we have

$$\begin{aligned} v_{\text{in}}^t \cdot S_{w_1} S_{w_2} \cdots S_{w_n} \cdot v_{\text{fin}} &= (v_{\text{in}}^1 \oplus v_{\text{in}}^2)^t \cdot (S_{w_1}^1 \oplus S_{w_1}^2) \cdots (S_{w_n}^1 \oplus S_{w_n}^2) \cdot (v_{\text{fin}}^1 \oplus v_{\text{fin}}^2) \\ &= (v_{\text{in}}^1)^t \cdot S_{w_1}^1 S_{w_2}^1 \cdots S_{w_n}^1 \cdot v_{\text{fin}}^1 + (v_{\text{in}}^2)^t \cdot S_{w_1}^2 S_{w_2}^2 \cdots S_{w_n}^2 \cdot v_{\text{fin}}^2. \end{aligned}$$

Recalling that $+ = \min$, this expression indeed computes $\text{Costs}(w, T)$, since the addition operation in the semiring chooses the minimum of the two branches.

Sequence. For the sequence operator, i.e., $T = T_1 \rightarrow T_2$, let S_a^i, v_{in}^i, and v_{fin}^i for $i = 1, 2$ be such that

$$\text{Costs}(w, T_i) = (v_{\text{in}}^i)^t \cdot S_{w_1}^i S_{w_2}^i \cdots S_{w_n}^i \cdot v_{\text{fin}}^i.$$

Let I be the row and column index set of the matrices S_a^1 and J be the row and column index set of the matrices S_a^2. Then v_{in}^1 and v_{fin}^1 are vectors in $\mathcal{M}_I(R)$ and v_{in}^2 and v_{fin}^2 are vectors in $\mathcal{M}_J(R)$. We construct new matrices S_a, for $a \in \Sigma$, for the tree T with index sets $(I \uplus J) \times (I \uplus J)$ as follows:

$$S_a = \begin{array}{c|cc} & I & J \\ \hline I & S_a^1 & S_a^1 Q \\ J & \mathbf{0}_{J\times I} & S_a^2 \end{array} \quad \text{where} \quad Q = v_{\text{fin}}^1 \cdot (v_{\text{in}}^2)^t \in \mathcal{M}_{I\times J}(R). \tag{4}$$

Moreover, for $i = 1, 2$ we let

$$\lambda_i = (v_{\text{in}}^i)^t \cdot v_{\text{fin}}^i \in R, \quad \text{and}$$

$$v_{\text{in}} = v^{1}_{\text{in}} \oplus \lambda_1 \cdot v^{2}_{\text{in}} \in \mathcal{M}_{I \uplus J}(R) \quad \text{and} \quad v_{\text{fin}} = \lambda_2 \cdot v^{1}_{\text{fin}} \oplus v^{2}_{\text{fin}} \in \mathcal{M}_{I \uplus J}(R). \tag{5}$$

Then, we have for a trace $w = w_1 w_2 \cdots w_n$:

$$S_{w_1} \cdots\ S_{w_n} = \begin{pmatrix} S^{1}_{w_1} \cdots S^{1}_{w_n} & S^{1}_{w_1} \cdots S^{1}_{w_n} Q + S^{1}_{w_1} \cdots S^{1}_{w_{n-1}} Q S^{2}_{w_n} + \cdots + S^{1}_{w_1} Q S^{2}_{w_2} \cdots S^{2}_{w_n} \\ \mathbf{0}_{J\times I} & S^{2}_{w_1} \cdots S^{2}_{w_n} \end{pmatrix}$$

Let $D = S^{1}_{w_1} \cdots S^{1}_{w_n} Q + S^{1}_{w_1} \cdots S^{1}_{w_{n-1}} Q S^{2}_{w_n} + \cdots + S^{1}_{w_1} Q S^{2}_{w_2} \cdots S^{2}_{w_n}$. Note that the D corresponds to all possible decompositions of w into two segments which are aligned against the two subtrees T_1 and T_2, where the segment for T_1 must be non-empty. Then we have:

$$\begin{aligned} & v^{t}_{\text{in}} \cdot S_{w_1} \cdots\ S_{w_n} \cdot v_{\text{fin}} \\ &= v^{t}_{\text{in}} \cdot \begin{pmatrix} S^{1}_{w_1} \cdots S^{1}_{w_n} & D \\ \mathbf{0}_{J\times I} & S^{2}_{w_1} \cdots S^{2}_{w_n} \end{pmatrix} \cdot v_{\text{fin}} \\ &= \left((v^{1}_{\text{in}})^t S^{1}_{w_1} \cdots S^{1}_{w_n} \oplus (v^{1}_{\text{in}})^t D + \lambda_1 \cdot v^{2}_{\text{in}} \cdot S^{2}_{w_1} \cdots S^{2}_{w_n} \right) \cdot v_{\text{fin}} \\ &= (v^{1}_{\text{in}})^t S^{1}_{w_1} \cdots S^{1}_{w_n} \cdot Q \cdot v^{2}_{\text{fin}} + (v^{1}_{\text{in}})^t D \cdot v^{2}_{\text{fin}} + \lambda_1 \cdot v^{2}_{\text{in}} \cdot S^{2}_{w_1} \cdots S^{2}_{w_n} \cdot v^{2}_{\text{fin}}. \end{aligned}$$

This sum corresponds to all possible ways to decompose w into a sequence $w = w_1 w_2$ of two traces w_1, w_2, and to align the first subtrace w_1 against T_1 and the second subtrace w_2 against T_2. The optimal alignment costs are correctly taken as the minimum over all such decompositions.

Parallel Operator. For $T = T_1 \wedge T_2$, let S^{i}_{a}, v^{i}_{in}, and v^{i}_{fin} for $i = 1, 2$ be such that for all traces $w = w_1 w_2 \cdots w_n$:

$$\text{Costs}(w, T) = (v^{i}_{\text{in}})^t \cdot S^{i}_{w_1} S^{i}_{w_2} \ \cdots\ S^{i}_{w_n} \cdot v^{i}_{\text{fin}}.$$

Let I be the row and column index set of the matrices S^{1}_{a}, and J be the row and column index set of the matrices S^{2}_{a}. Then v^{1}_{in} and v^{1}_{fin} are vectors in $\mathcal{M}_I(R)$ and v^{2}_{in} and v^{2}_{fin} are vectors in $\mathcal{M}_J(R)$. We construct new matrices S_a, for $a \in \Sigma$ for T with row and column index sets $I \times J$ as follows:

$$S_a = (S^{1}_{a} \otimes \mathbf{1}_{J\times J}) + (\mathbf{1}_{I\times I} \otimes S^{2}_{a}).$$

Moreover, we set

$$v_{\text{in}} = v^{1}_{\text{in}} \otimes v^{2}_{\text{in}} \in \mathcal{M}_{I\times J}(R) \quad \text{and} \quad v_{\text{fin}} = v^{1}_{\text{fin}} \otimes v^{2}_{\text{fin}} \in \mathcal{M}_{I\times J}(R). \tag{6}$$

In anticipation of our treatment of process trees with unique labels, let us assume that the sets of labels in the two subtrees T_1 and T_2 are disjoint, i.e., $\Sigma_1 \cap \Sigma_2 = \emptyset$ (where Σ_i is the set of labels which occur in T_i). Assume that $a \notin \Sigma_1$. Then $S^{1}_{a} = 1 \cdot \mathbf{1}_{I\times I}$ and we get:

$$\begin{aligned} S_a &= (S^{1}_{a} \otimes \mathbf{1}_{J\times J}) + (\mathbf{1}_{I\times I} \otimes S^{2}_{a}) = (1 \cdot \mathbf{1}_{I\times I} \otimes \mathbf{1}_{J\times J}) + (\mathbf{1}_{I\times I} \otimes S^{2}_{a}) \\ &= (\mathbf{1}_{I\times I} \otimes 1 \cdot \mathbf{1}_{J\times J}) + (\mathbf{1}_{I\times I} \otimes S^{2}_{a}) = \mathbf{1}_{I\times I} \otimes (1 \cdot \mathbf{1}_{J\times J} + S^{2}_{a}) = \mathbf{1}_{I\times I} \otimes S^{2}_{a}. \end{aligned}$$

Here we used that $1 \cdot \mathbf{1}_{J\times J} + S^{2}_{a} = S^{2}_{a}$ (easy to verify by induction, since all matrices S_a have the elements $0, 1 \in \mathbb{N}$ on their diagonal.) This means that for process trees with unique labels, the definition can be simplified to:

$$S_a = \begin{cases} \mathbf{1}_{I\times I} \otimes S^{2}_{a} & \text{if } a \in \Sigma_2, \\ S^{1}_{a} \otimes \mathbf{1}_{J\times J} & \text{otherwise.} \end{cases} \tag{7}$$

Note that we derived this simplification purely algebraically. Let us verify the correctness of the construction. Using the mixed product property of the Kronecker product, we get:

$$\begin{aligned} S_w = S_{w_1} \cdots\ S_{w_n} &= ((S^{1}_{w_1} \otimes \mathbf{1}_{J\times J}) + (\mathbf{1}_{I\times I} \otimes S^{2}_{w_1})) \cdots ((S^{1}_{w_n} \otimes \mathbf{1}_{J\times J}) + (\mathbf{1}_{I\times I} \otimes S^{2}_{w_n})) \\ &= \sum \left(S^{1}_{x_1} S^{1}_{x_2} \cdots S^{1}_{x_k} \otimes S^{2}_{y_1} S^{2}_{y_2} \cdots S^{2}_{y_k} : x_1 y_1 x_2 y_2 \cdots x_k y_k = w, x_i, y_i \in \Sigma^* \right). \end{aligned}$$

Moreover, for each summand $S^1_{x_1}S^1_{x_2}\cdots S^1_{x_k}\otimes S^2_{y_1}S^2_{y_2}\cdots S^2_{y_k}$, we have

$$\begin{aligned}&v_{\text{in}}^t\cdot\left(S^1_{x_1}S^1_{x_2}\cdots S^1_{x_k}\otimes S^2_{y_1}S^2_{y_2}\cdots S^2_{y_k}\right)\cdot v_{\text{fin}}\\ &=(v^1_{\text{in}})^t\cdot S^1_{x_1}S^1_{x_2}\cdots S^1_{x_k}\cdot v^1_{\text{fin}}\cdot(v^2_{\text{in}})^t\cdot S^2_{y_1}S^2_{y_2}\cdots S^2_{y_k}\cdot v^2_{\text{fin}}.\end{aligned}$$

Hence, this term corresponds to the optimal alignment costs where we split the trace w into two subtraces $x_1x_2\cdots x_k$ and $y_1y_2\cdots y_k$, for which the shuffle yields w, and align them against the process trees T_1 and T_2, respectively. As the sum runs over all possible decompositions of w, this determines the optimal alignment costs for the trace w and the process tree T. Again, in anticipation of the treatment of process trees with unique labels, note that for the case $\Sigma_1\cap\Sigma_2=\emptyset$, the product simplifies to:

$$S_w=\prod_{w_i\notin\Sigma_2}S^1_{w_i}\otimes\prod_{w_i\in\Sigma_2}S^2_{w_i}. \tag{8}$$

Hence, instead of an exponential number of summands, we get a *single* term for the matrix S_w (this is the key for tractability of the alignment problem for process trees with unique labels).

Loop. Finally, we consider the case of a loop operator. Let $T=T_1\circlearrowleft T_2$, where, by recursion, we can assume to have already constructed the matrices S^1_a, v^1_{in}, and v^1_{fin} for the process tree T_1 and the matrices S^2_a, v^2_{in}, and v^2_{fin} for the process tree T_2. Then we set

$$S_a=\begin{pmatrix}S^1_a & Q_2S^2_a\\ Q_1S^1_a & S^2_a\end{pmatrix},\text{ where } Q_2=v^1_{\text{fin}}\cdot(v^2_{\text{in}})^t\in\mathcal{M}_{I\times J}(R)\text{ and }Q_1=v^2_{\text{fin}}\cdot(v^1_{\text{in}})^t\in\mathcal{M}_{J\times I}(R). \tag{9}$$

Moreover, for $\lambda=\langle v^1_{\text{in}},v^1_{\text{fin}}\rangle$, we set

$$v_{\text{in}}=(v^1_{\text{in}}\oplus\lambda v^2_{\text{in}})\text{ and }v_{\text{fin}}=(v^1_{\text{fin}}\oplus\lambda v^2_{\text{fin}}). \tag{10}$$

It can be verified that

$$S_w=\begin{pmatrix}D_{11} & D_{12}\\ D_{21} & D_{22}\end{pmatrix},\text{ with }D_{ij}=\sum\big(S^i_{x_0}Q_{3-i}S^{3-i}_{x_1}Q_iS^i_{x_2}\cdots Q_jS^j_{x_k}:$$
$$x_0x_1x_2\cdots x_k=w,\text{ where }k\geq 0,x_0\in\Sigma^*,x_1,\ldots,x_k\in\Sigma^+\big).$$

Each summand of D_{ij} corresponds to a decomposition of the trace w into segments $x_0x_1x_2\cdots x_k$ where the first segment is aligned against the process tree T_i and the last segment is aligned against the process tree T_j (in between we have strict alternation). Moreover, $v^t_{\text{in}}\cdot S_w\cdot v_{\text{fin}}=v^1_{\text{in}}\cdot D_{11}v^1_{\text{fin}}+\lambda v^2_{\text{in}}\cdot D_{21}v^1_{\text{fin}}+v^1_{\text{in}}\cdot D_{12}v^2_{\text{fin}}\lambda+\lambda v^2_{\text{in}}\cdot D_{22}v^2_{\text{fin}}\lambda$. Note that $\lambda=S^1_\varepsilon$, i.e. λ captures the costs to align the empty trace ε against the process tree T_1. Hence, the total sum consists of *all* possible ways to decompose the trace w into segments $x_0x_1x_2\cdots x_k$ which are aligned against the trees T_1 and T_2 in an alternating fashion. Depending on which tree we start/end with, λ takes care of the costs of aligning ε against T_1 (since the semantics of the loop operator demands that we have to start and end with T_1, recall: for $T=T_1\circlearrowleft T_2$ we have $\mathcal{L}(T)=\mathcal{L}(T_1)\cdot(\mathcal{L}(T_2)\cdot\mathcal{L}(T_1))^*$). Finally, the sum selects the optimal decomposition and hence the optimal alignment costs for the trace w and the process tree T.

We have established the main result of this work:

Theorem 4.1. *There exists an algorithm which takes as input a process tree T over some alphabet Σ and computes a family of $I\times I$-matrices $(S_a)_{a\in\Sigma}$, an I-vector $v_{in}\in\mathcal{M}_I(R)$ and an I-vector $v_{fin}\in\mathcal{M}_I(R)$, for some index set I, such that the optimal alignment costs for a trace $w=w_1w_2w_3\cdots w_n\in\Sigma^*$ and T can be expressed as a matrix multiplication problem as follows:*

$$\mathit{Costs}(w,T)=v^t_{in}\cdot S_{w_1}S_{w_2}\ \cdots\ S_{w_n}\cdot v_{fin}\in R.$$

Our construction is not efficient. Consider a process tree of the form $T=(((a_1\wedge a_2)\wedge\cdots)\wedge a_n)$ with n activities $a_1,a_2,\ldots,a_n$ that can occur in any order. For T, our construction yields matrices of size c^n (where c is the size of the base matrices for labels $a\in\Sigma$). This is because the size of the Kronecker product of two matrices is the product of their sizes. Hence, in general, the only size bound on S_a that we can get is exponential, i.e. $2^{\mathcal{O}(\|T\|)}$. In the following, we thus discuss how to avoid constructing the matrices S_a explicitly and how to work with their symbolic representations instead.

5. Symbolic Computations with Tree-Structured Matrices and Vectors

Our matrix/vector construction from Section 4 has one drawback: the Kronecker product leads to an exponential blow-up of the sizes of matrices and vectors. Since the alignment problem on process trees is NP-complete [13], it is clear that this state explosion cannot be avoided in the general case. On the other hand, the resulting matrices and vectors have a simple descriptions in form of algebraic expressions. This suggests that, instead of working with explicit matrices, we could compute with the symbolic representations of the matrices and vectors instead. In this section, we investigate this idea.

5.1. Tree-structured index sets

We start by defining *tree-structured sets (ts-sets, for short)* to index the matrices and vectors used by the construction in Section 4. We use *symbolic representations* for the sets to which we associate a semantics as (potentially exponential-sized) standard sets. In the following, we might get mixed-up with the symbolic representation of a ts-set I and its semantics as a standard set. Hence, to avoid confusion, we also write $\mathrm{rset}(I)$ to denote the standard set that is represented by I. For the case of base sets we have $I = \mathrm{rset}(I)$. Also, each ts-set has a *representation size* $\|I\| \in \mathbb{N}$ which quantifies the encoding length of the symbolic representation I.

Fix a countably infinite set of atoms $\mathrm{Atoms} = \{a_0, a_1, a_2, \ldots\}$ which are used as base elements. For simplicity, we assume that we can encode atoms with unit costs. A *tree-structured set (ts-set, for short)* is an expression according to the following inductive definition:

Base case. Each standard (finite, non-empty) set $I \subseteq \mathrm{Atoms}$ is a ts-set which represents the set $\mathrm{rset}(I) = I$. The representation size of the atomic set is $\|I\| = |I|$.

Disjoint union. For ts-sets I, J, we can construct the ts-set $I \uplus J$ which represents the (standard) set $\mathrm{rset}(I) \uplus \mathrm{rset}(J)$. The representation size of the set $I \uplus J$ is $\|I\| + \|J\| + 1$.

Cartesian product. For ts-sets I, J, we can construct the ts-set $I \times J$ which represents the (standard) set $\mathrm{rset}(I) \times \mathrm{rset}(J)$. The representation size of the set $I \times J$ is $\|I\| + \|J\| + 1$.

Note that the representation size $\|I\|$ of I is linear in the representation sizes of the ts-subsets for all operations and linear in the cardinality of the set I for the base case. This is in contrast to the size of the represented set, specifically for the case of the Cartesian product operation $I \times J$. While $\|I \times J\|$ is $\|I\| + \|J\| + 1$, the size of the represented standard set is $|\mathrm{rset}(I \times J)| = |\mathrm{rset}(I)| \cdot |\mathrm{rset}(J)|$.

5.2. Tree-structured vectors

We proceed by defining *tree-structured vectors (ts-vectors, for short)*. Let I be a ts-set. Then a ts-vector v with ts-index set I has a semantics in form of an $\mathrm{rset}(I)$-vector $v\colon \mathrm{rset}(I) \to R$. To emphasize the distinction between a ts-vector v (as a symbolic expression) and the represented standard vector (its semantics) we also denote the vector represented by v as $\mathrm{rvec}(v)$. Again, for simplicity, we assume that we can encode the ring elements in R with unit costs. As a consequence, the encoding length $\|v\| \in \mathbb{N}$ of a ts-vector v with ts-index set I corresponds to the representation size of the ts-set I, i.e., $\|v\| = \|I\|$.

5.2.1. Definition of ts-vectors

The definition of ts-vectors is recursively:

Base case. For a standard set I, each standard vector $v\colon I \to R$ is also a ts-vector v with index set I. The representation size of the vector is $\|v\| = \|I\|$.

Direct sum. Let v_1 be a ts-vector with ts-index set I and v_2 be a ts-vector with ts-index set J. Then we can construct a new ts-vector v with ts-index set $I \uplus J$ as $v = v_1 \oplus v_2$. The represented vector is $\mathrm{rvec}(v) = \mathrm{rvec}(v_1) \oplus \mathrm{rvec}(v_2)$. The representation size of the new vector is $\|v\| = \|v_1\| + \|v_2\| + 1$.

Kronecker product. Let v_1 be a ts-vector with ts-index set I and v_2 be a ts-vector with ts-index set J. Then we can construct a new ts-vector v with ts-index set $I \times J$ as $v = v_1 \otimes v_2$. The represented vector

$\operatorname{rvec}(v)$ is the $\operatorname{rset}(I) \times \operatorname{rset}(J)$-vector given as $\operatorname{rvec}(v) = \operatorname{rvec}(v_1) \otimes \operatorname{rvec}(v_2)$. The representation size of the new vector is $\|v\| = \|v_1\| + \|v_2\| + 1$.

5.2.2. Effective Computations with Ts-Vectors

We show that we can efficiently compute the *scalar product*, the *inner product* and the *vector addition (for non-Kronecker products)* of ts-vectors. This means the following: there are polynomial-time algorithms which take as input the *symbolic* representation of ts-vectors and output the *symbolic* representation of the result of a vector operation on the represented vectors.

Scalar multiplication. Given a scalar $\lambda \in R$ and a ts-vector v. To compute a representation of λv, it suffices to observe that $\lambda(v_1 \oplus v_2) = \lambda v_1 \oplus \lambda v_2$ and $\lambda(v_1 \otimes v_2) = (\lambda v_1) \otimes v_2 = v_1 \otimes (\lambda v_2)$. Using these identities, a recursive computation of a representation for λv can trivially be achieved in time $\mathcal{O}(\|v\|)$. Also note that scalar multiplication does not change the representation size of v, i.e., $\|\lambda v\| = \|v\|$.

Inner product. Let v, w be ts-vectors with the same ts-index set I. We explain how to efficiently compute the inner product $\langle v, w\rangle = v^t w = w^t v \in R$ via recursion on the structure of v, w (and I) where the case for base sets is trivial:

- If $v = v_1 \oplus v_2$ and $w = w_1 \oplus w_2$ are direct sum vectors, then we have: $\langle v, w\rangle = \langle v_1, w_1\rangle + \langle v_2, w_2\rangle$. Hence, we can recursively compute the inner products $\langle v_1, w_1\rangle$ and $\langle v_2, w_2\rangle$ and add the results.
- If $v = v_1 \otimes v_2$ and $w = w_1 \otimes w_2$ are Kronecker product vectors, then we have:

$$\langle v, w\rangle = v^t \cdot w = (v_1^t \otimes v_2^t) \cdot (w_1 \otimes w_2) = v_1^t \cdot w_1 \cdot v_2^t \cdot w_2 = \langle v_1, w_1\rangle \cdot \langle v_2, w_2\rangle.$$

  Hence, we recursively compute the inner products $\langle v_1, w_1\rangle$ and $\langle v_2, w_2\rangle$ and multiply the results.

In particular, it follows that the inner product of ts-vectors can be computed in linear time $\mathcal{O}(\|v\|)$.

Addition of ts-vectors without Kronecker products A final simple observation is that we can compute a representation for the sum $v + w$ of two ts-vectors v and w over a ts-index set I which are built *without the use of Kronecker products* in linear time $\mathcal{O}(\|v\| + \|w\|)$ as well. We only need to make use of the identity $(v_1 \oplus v_2) + (w_1 \oplus w_2) = (v_1 + w_1) \oplus (v_2 + w_2)$ for the case of direct sums.

5.3. Tree-structured matrices

We shift our attention to *tree-structured matrices (ts-matrices, for short).* Analogously, to ts-vectors, we consider ts-matrices M with ts-index sets $I \times J$ which have a semantics as standard $\operatorname{rset}(I) \times \operatorname{rset}(J)$-matrices $\operatorname{rmat}(M)$. Also analogously, each ts-matrix has a representation size $\|M\| \in \mathbb{N}$ which corresponds to its encoding complexity. In contrast to ts-vectors, however, we restrict the application of one operation (namely, forming matrix compositions) to certain types of input ts-matrices. This restriction may seem arbitrary, but is required later in order to show that ts-matrices can be efficiently multiplied. The inductive definition of tree-structures matrices is as follows:

Base case (standard matrices). For standard sets I and J, each mapping $M\colon \operatorname{rset}(I) \times \operatorname{rset}(J) \to R$ is a ts-matrix over R. Of course, we have $\operatorname{rmat}(M) = M$ for the base case. The representation size is $\|M\| = |\operatorname{rset}(I)| \cdot |\operatorname{rset}(J)|$.

Kronecker product. Let A be an $I \times J$ ts-matrix and B be a $K \times L$ ts-matrix (where I, J, K, L are ts-sets). Then we can construct a new ts-matrix $A \otimes B$ with ts-index set $(I \times K) \times (J \times L)$ as $A \otimes B$ with the represented matrix being the Kronecker product of the represented matrices, i.e., $\operatorname{rmat}(A \otimes B) = \operatorname{rmat}(A) \otimes \operatorname{rmat}(B)$. The representation size is $\|A \otimes B\| = \|A\| + \|B\| + 1$.

Direct sum. Let A be an $I \times I$ square ts-matrix and B be a $J \times J$ square ts-matrix (where I and J are ts-sets). Then we can construct a new square ts-matrix $A \oplus B$ with index set $(I \uplus J) \times (I \uplus J)$. The represented matrix is just $\operatorname{rmat}(A \oplus B) = \operatorname{rmat}(A) \oplus \operatorname{rmat}(B)$. The representation size of the new matrix is $\|A \oplus B\| = \|A\| + \|B\| + 1$.

Matrix composition As mentioned above, for this last operation we restrict the type of input matrices: we only allow ts-matrices as input that *do not contain the Kronecker product operator.* Let four ts-matrices A_1 with index set $I \times J$, A_2 with index set $I \times K$, A_3 with index set $L \times J$, and A_4 with index set $L \times K$ (all index sets I, J, K, L are ts-sets and so are $I \times J$, $I \times K$, $L \times J$, and $L \times K$) be given where all of the matrices A_1, A_2, A_3, and A_4 are *built without use of the Kronecker product.* Then we can construct a new ts-matrix A with index set $(I \uplus L) \times (J \uplus K)$ (note that this is a ts-set) as follows:

$$A = \begin{array}{c c} & \begin{array}{cc} J & K \end{array} \\ \begin{array}{c} I \\ L \end{array} & \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \end{array},$$

The representation size of this new matrix is $\|A\| = \|A_1\| + \|A_2\| + \|A_3\| + \|A_4\| + 1$. The represented matrix rmat(A) has index set rset$(I \uplus L) \times$ rset$(J \uplus K)$ and is obtained by the matrix composition of the four submatrices rmat(A_1), rmat(A_2), rmat(A_3), and rmat(A_4) as indicated.

5.3.1. Addition of Non-Kronecker-Product Ts-matrices

A representation of $A + B$ can be computed in time $\mathcal{O}(\|A\| + \|B\|)$ for two ts-matrices A and B with the same index set $I \times J$ if both matrices A, B are *built without the use of Kronecker products* (this is analogous to the case of ts-vectors that we considered above). That is clear for base case matrices, so let us only check the other cases:

Direct sum. For two direct sum matrices $A = A_1 \oplus A_2$ and $B = B_1 \oplus B_2$, we have $A + B = (A_1 + B_1) \oplus (A_2 + B_2)$, so we can reduce this case recursively to simpler types of matrices.

Matrix composition For matrices

$$A = \begin{array}{c c} & \begin{array}{cc} J & K \end{array} \\ \begin{array}{c} I \\ L \end{array} & \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \end{array} \quad \text{and} \quad B = \begin{array}{c c} & \begin{array}{cc} J & K \end{array} \\ \begin{array}{c} I \\ L \end{array} & \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} \end{array}, \quad \text{we have:} \quad A+B = \begin{array}{c c} & \begin{array}{cc} J & K \end{array} \\ \begin{array}{c} I \\ L \end{array} & \begin{pmatrix} A_1 + B_1 & A_2 + B_2 \\ A_3 + B_3 & A_4 + B_4 \end{pmatrix} \end{array},$$

so, this case can readily be reduced to simpler types of ts-matrices by recursively computing the sums of the corresponding submatrices $A_1 + B_1$, $A_2 + B_2$, $A_3 + B_3$, and $A_4 + B_4$.

5.3.2. Matrix-vector multiplication

It is further possible to compute a ts-vector representation of Av for a given $I \times J$-ts-matrix A and a ts-vector v with index set I in time $\mathcal{O}(\|A\|)$. To see this, we, again, recurse on the type of A.

Base case. If A is a base matrix, then J is a base set as and thus v is a base ts-vector. Hence, the standard matrix-vector product can be computed in time $\mathcal{O}(|I| \cdot |J|)$, where $|I|$ and $|J|$ are the sizes of the sets I and J, respectively. Since, $\|A\| = |I| \cdot |J|$, the claim follows.

Direct sum. If $A = A_1 \oplus A_2$ is a direct sum matrix, then $v = v_1 \oplus v_2$ is a direct sum as well (otherwise, the index sets would not match). Then we have $Av = A_1v_1 \oplus A_2v_2$ and we can reduce to simpler cases. The running time is $\mathcal{O}(\|A_1\| + \|A_2\|) = \mathcal{O}(\|A\|)$ as claimed.

Kronecker product. If $A = A_1 \otimes A_2$ is a Kronecker product matrix, then $v = v_1 \otimes v_2$ is a Kronecker product as well (again, otherwise the index sets would not match). Then $Av = A_1v_1 \otimes A_2v_2$ due to the mixed-product property, so we just need to compute a representation for Av_1 and Av_2 and obtain a representation for Av. The required computation time is $\mathcal{O}(\|A_1\| + \|A_2\|) = \mathcal{O}(\|A\|)$ as claimed.

Matrix composition. If $A = \begin{array}{c c} & \begin{array}{cc} J & K \end{array} \\ \begin{array}{c} I \\ L \end{array} & \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \end{array}$ is a matrix composition, then $v = v_1 \oplus v_2$ is a direct sum (with index set $J \uplus K$). By our (restricted) definition of the matrix composition operator for ts-matrices, no submatrix A_1, A_2, A_3, or A_4 contains the Kronecker product operator. We compute Av as follows:

$$Av = (A_1v_1 + A_2v_2) \oplus (A_3v_1 + A_4v_2).$$

Since A_1, A_2, A_3, and A_4 do not contain the Kronecker product operator, the recursive computation of A_1v_1, A_2v_2, A_3v_1, and A_4v_2 yields ts-vectors without the Kronecker product operator as well. We can recursively compute representations for these products in time $\mathcal{O}(\|A_1\| + \|A_2\| + \|A_3\| + \|A_4\|)$. Next, we compute the sums $A_1v_1 + A_2v_2$ and $A_3v_1 + A_4v_2$, which can also be done in time $\mathcal{O}(\|A_1\| + \|A_2\| + \|A_3\| + \|A_4\|)$ as shown earlier (since the vectors A_1v_1, A_2v_2, A_3v_1, and A_4v_2 do not contain the Kronecker product operator). In total, we obtain a representation for Av in time $\mathcal{O}(\|A\|)$, as claimed.

We remark that, although we have explicitly considered a matrix-vector product of the form Av, it should be clear that by a completely symmetric treatment, we can derive an efficient algorithm for a matrix-vector product of the form v^tA as well.

5.3.3. Rank-One Matrices

As a special case, we consider matrices which are formed by the multiplication of two ts-vectors v and w. More precisely, let v be a ts-vector with index set I and w be a ts-vector with index set J. Then we can efficiently construct a new ts-matrix A with index set $I \times J$ as A such that $\mathrm{rmat}(A) = \mathrm{rmat}(v) \cdot \mathrm{rmat}(w)^t$. Since the base case is trivial, we only need to consider the cases for direct sums and Kronecker products. Here, we can make use of the following identities:

$$(v_1 \oplus v_2) \cdot (w_1 \oplus w_2)^t = \begin{pmatrix} v_1 \cdot w_1^t & v_1 \cdot w_2^t \\ v_2 \cdot w_1^t & v_2 \cdot w_2^t \end{pmatrix}, \text{and}$$
$$(v_1 \otimes v_2) \cdot (w_1 \otimes w_2)^t = (v_1 \cdot w_1^t) \otimes (v_2 \cdot w_2^t).$$

In this way, we can recursively reduce to matrix operations and finally to base case matrices.

5.3.4. Matrix multiplication

Finally, we consider the matrix multiplication of ts-matrices A and B. To this end, let A be an $I \times J$ ts-matrix and B be a $J \times K$ ts-matrix. Then we show how we can efficiently compute a representation of the product AB as a ts-matrix with index set $I \times K$. The required computation time is $\mathcal{O}(\|I\| \cdot \|J\| \cdot \|K\|)$ which is polynomial in the representation sizes of A and B. We skip the base case (which is obvious), and focus on the remaining cases where we distinguish with respect to the structure of A:

Direct sum. If $A = A_1 \oplus A_2$, then $J = J_1 \uplus J_2$. One possible form of B is $B = B_1 \oplus B_2$. In this case, we can reduce the computation recursively to the case of A_1B_1 and A_2B_2 by using the identity:

$$(A_1 \oplus A_2)(B_1 \oplus B_2) = A_1B_1 \oplus A_2B_2.$$

The other possibility is that B is a matrix composition, i.e., $B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$. Then, we can use:

$$(A_1 \oplus A_2) \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} = \begin{pmatrix} A_1B_1 & A_1B_2 \\ A_2B_3 & A_2B_4 \end{pmatrix}$$

to reduce the computation to the simpler cases A_1B_1, A_1B_2, A_2B_3, and A_2B_4.

Kronecker product. If $A = A_1 \otimes A_2$, then $B = B_1 \otimes B_2$ is a Kronecker product as well. In this case, we can use the mixed-product property to reduce the computation to the simpler cases A_1B_1 and A_2B_2 via $AB = A_1B_1 \otimes A_2B_2$ and recursively obtain a representation of the product AB.

Matrix composition. As a final case, let $A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$. In this case, B can be a direct sum (but we covered the symmetric case already above) or a matrix composition $B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$ as well. In this case, we observe that:

$$AB = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} = \begin{pmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{pmatrix}.$$

At this point, we crucially make use of our assumption that the matrix composition operator is only applied to submatrices that do not contain the Kronecker product operator. First, we recursively compute the relevant products A_iB_j and then, since the resulting matrices do not contain the Kronecker product operator, we can compute a representation for the sums $A_1B_1 + A_2B_3$, $A_1B_2 + A_2B_4$, $A_3B_1 + A_4B_3$, and $A_3B_2 + A_4B_4$. If $I = I_1 \uplus I_2$, $J = J_1 \uplus J_2$ and $K = K_1 \uplus K_2$, then the subcomputations of the products A_iB_j can be done in time $\mathcal{O}(\|I_i\| \cdot \|J_i\| \cdot \|K_j\|)$ for $i, j = 1, 2$. The total computation time for the product subcomputations is thus bounded by $\mathcal{O}((\|I_1\| + \|I_2\|) \cdot (\|J_1\| + \|J_2\|) \cdot (\|K_1\| + \|K_2\|)) = \mathcal{O}(\|I\| \cdot \|J\| \cdot \|K\|)$. Also we saw above, that we can compute the sums of the matrices in linear time. This yields a total of bound of $\mathcal{O}(\|I\| \cdot \|J\| \cdot \|K\|)$ as claimed.

6. Applications of the Linear-Algebraic Formulation

The representation of the alignment problem for process trees in terms of matrix/vector products allows us to apply new algorithmic methods to compute alignments. The obvious ones include the application of highly optimized matrix multiplication algorithms as implemented in several linear algebra libraries. We could even adapt the computation to different hardware settings such as GPUs and, of course, parallelize the computation of the product $v_{\text{in}}^t \cdot S_{w_1} \cdots S_{w_n} \cdot v_{\text{fin}}$ for long traces w. Of course, we might also try to use linear-algebraic structure theory to make the product computation simpler in the first place. For example, since the matrices are structurally quite similar, it might be possible to find common transformations that yield (partial) representations with respect to eigenspaces. This could help to speed up the computation in some cases at least. Another advantage of our approach is that the precompilation of the ts-matrices S_a for all $a \in \Sigma$ is only required once, i.e., we can save time with preprocessing if we have a *single, fixed* process tree and want to align many different traces against it.

Besides such algorithmic considerations, well-established linear-algebraic concepts might yield novel insights for conformance checking applications. Let us briefly discuss one simple example. Assume we have constructed the matrix family $(S_a)_{a\in\Sigma}$ and the vectors $v_{\text{in}}, v_{\text{fin}}$ for a process tree T. Then we can define that two events $a, b \in \Sigma$ are *independent in all trace contexts* if the product S_aS_b is equal to S_bS_a. This actually means that, with respect to alignments of the given process trees, we can swap the order of a and b without changing the alignment costs no matter of what the context is in which the two events occur. In this way we can relate the notion of *commuting matrices* from linear algebra with a novel notion of causal independence. In fact, it would be interesting to study this independence notion on real-life event logs to see what it can tell us about the underlying process.

Unfortunately, for the algorithmic approaches, we still face the obstacle discussed in Section 5: the matrices S_a can become exponentially large (in the size of the underlying process tree) by the Kronecker product operator. Given the NP-hardness of the alignment problem for process trees this is not surprising, and the key question is: how far can we push the linear-algebraic technique from an algorithmic perspective without running into the exponential blow-up of the Kronecker product operator? Can we even find new classes of process trees which allow polynomial-time alignment computations via the linear-algebraic formulation? The symbolic representations that we discussed in Section 5 constitute a first step towards this very question.

A natural class to consider are process trees with unique labels: for those we recently proved that alignments can be computed in polynomial time via a dynamic programming algorithm [14]. Interestingly enough, we already saw in Section 4 that by a purely linear-algebraic argument we could strongly simplify the matrix expressions for the shuffle operator in cases of unique labels. This is clear indication that, indeed, the linear-algebraic approach mirrors the key argument of the polynomial-time algorithm for process trees with unique labels (cf. Equation (7)) which effectively eliminates choice for shuffle operators.

Unfortunately, the symbolic approach as given in Section 5 does not fully cover process trees with unique labels yet. Anyhow, in what follows we explain that it does capture a significant and interesting subclass. We are going to show that we can efficiently *construct* the matrices $(S_a)_{a\in\Sigma}$ and vectors $v_{\text{in}}, v_{\text{fin}}$ and also efficiently compute the product $v_{\text{in}}^t \cdot S_{w_1} \cdots S_{w_n} \cdot v_{\text{fin}}$ for a given trace $w = w_1 \cdots w_n$

for all process trees T with unique labels in which *the shuffle operator does not occur within the scope of a sequence or loop operator*. It is allowed, however, that the shuffle operator occurs within the scope of a choice operator and, of course, within the scope of other shuffle operators.

To show this, it suffices to argue that the matrices $(S_a)_{a\in\Sigma}$ and vectors $v_{\text{in}}, v_{\text{fin}}$ can efficiently be constructed as ts-matrices and ts-vectors as introduced in Section 5 since for those we proved that products can be computed in polynomial time. To be more precise, for our construction we guarantee that the representation size of the resulting matrices S_a for a process tree T is bounded by $\mathcal{O}(\|T\|^2)$ (and the same holds for the vectors v_{in} and v_{fin} whose representation size is $\mathcal{O}(\|T\|)$). Let us go through the different process tree operators and check that the matrices and vectors can be constructed as required. Since the base cases (labeled leaf and silent leaf) yield explicit, constant size matrices, we can trivially encode those as base case ts-matrices and ts-vectors. Let us check the other cases:

Exclusive choice. The matrices and vectors for the exclusive choice operator given in Equation (3) can obviously be constructed as ts-matrices and ts-vectors using the direct sum operator available for ts-matrices and ts-vectors.

Sequence. For the sequence operator, it is *not* as obvious that we can construct the required matrices (Equation (4)) and vectors (Equation (5)) as ts-matrices and ts-vectors. The reason is that matrix products, matrix-vector products, inner products, and scalar products occur in Equation (4) and Equation (5). Hence, for the construction to go through, we need to show that these operations can be computed efficiently. The results can then easily be assembled by using the matrix composition operator for matrices as in Equation (4) and the direct sum operator for the vectors as in Equation (5).

To show efficient computability, we need to use our assumption that, within the scope of the sequence operator, no shuffle operator occurs. With this assumption, we know that none of the involved submatrices and vectors contains the Kronecker product operator. Luckily, we saw in Section 5 that for such matrices and vectors *all* required operations are indeed computable in polynomial time on the symbolic level.

Loop. The argument for the loop operator is similar to the case of the sequence operator. Again, for the matrices and vectors in Equation (9) and Equation (10), it is not at all obvious that we can construct them as ts-matrices and ts-vectors as these involve matrix products and matrix addition. However, with the assumption that no shuffle operator occurs within the scope of a loop operator, the argument goes through as in the case of the sequence operator.

Parallel operator. For the parallel operator, we derived a simplified expression valid for process trees with unique labels in Equation (7). This expression is already in the form of a Kronecker product and can thus be constructed as a ts-matrix using the Kronecker product operator. The same holds for the associated vectors as given in Equation (6).

7. Discussion

We have shown that the alignment problem for process trees can be formulated in terms of matrix and vector products. This sheds new light on the problem, paves the way for new algorithmic techniques, and enables us to study alignments through the lens of linear algebra. There are several paths to follow in the future. One open question is if our symbolic approach can be extended to cover all process trees with unique labels. This would potentially lead to an alternative polynomial-time algorithm for the alignment problem for process trees with unique labels, a class highly relevant in practice. This algorithm might be even more efficient if we apply optimized linear algebra libraries to the matrix products involved or make use of parallelization techniques. Another angle would be to look into other semirings to verify, for example, that also more general cost functions are covered by our approach. It is also conceivable to define a semiring which does not only yield optimal alignment costs, but also the set of optimal alignments. Finally, another question is whether we can generalize the linear-algebraic approach to other classes of process models such as sound, free-choice workflow nets.

References

[1] A. Adriansyah. “Aligning observed and modeled behavior.” PhD thesis. Technische Universiteit Eindhoven, 2014. DOI: 10.6100/IR770080.

[2] J. Carmona, B. F. van Dongen, A. Solti, and M. Weidlich. *Conformance Checking. Relating Processes and Models.* Cham: Springer International Publishing, 2018. DOI: 10.1007/978-3-319-99414-7.

[3] J. Carmona, B. F. van Dongen, and M. Weidlich. “Conformance Checking: Foundations, Milestones and Challenges.” In: *Process Mining Handbook.* Vol. 448. LNBIP. Cham: Springer International Publishing, 2022. Chap. 5, pp. 155–190. DOI: 10.1007/978-3-031-08848-3_5.

[4] M. Droste and P. Gastin. “Weighted automata and weighted logics.” In: *Theor. Comput. Sci.* 380.1-2 (2007), pp. 69–86.

[5] M. Droste, W. Kuich, and H. Vogler. *Handbook of Weighted Automata.* Monographs in Theoretical Computer Science. An EATCS Series. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-01492-5.

[6] M. Droste and D. Kuske. “Weighted automata.” In: *Handbook of Automata Theory (I.)* European Mathematical Society Publishing House, Zürich, Switzerland, 2021, pp. 113–150.

[7] S. J. J. Leemans. *Robust Process Mining with Guarantees. Process Discovery, Conformance Checking and Enhancement.* Vol. 440. LNBIP. Cham: Springer International Publishing, 2022. DOI: 10.1007/978-3-030-96655-3.

[8] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst. “Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour.” In: *Business Process Management Workshops.* BPM Workshops 2013. Vol. 171. LNBIP. Cham: Springer International Publishing, 2014, pp. 66–78. DOI: 10.1007/978-3-319-06257-0_6.

[9] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst. “Discovering Block-Structured Process Models from Incomplete Event Logs.” In: *Application and Theory of Petri Nets and Concurrency.* PETRI NETS 2014. Vol. 8489. LNCS. Cham: Springer International Publishing, 2014, pp. 91–110. DOI: 10.1007/978-3-319-07734-5_6.

[10] M. Lohrey and M. Schmidt-Schauß. “Processing Succinct Matrices and Vectors.” In: *Theory Comput. Syst.* 61.2 (2017), pp. 322–351.

[11] J. Osterholzer. “Weighted Automata and Logics for Natural Language Processing.” In: *Joint Workshop of the German Research Training Groups in Computer Science.* Pro Business GmbH, 2014, p. 150.

[12] M. P. Schützenberger. “On the Definition of a Family of Automata.” In: *Inf. Control.* 4.2-3 (1961), pp. 245–270.

[13] C. T. Schwanen, W. Pakusa, and W. M. P. van der Aalst. “Complexity of Alignments on Sound Free-Choice Workflow Nets.” In: *Application and Theory of Petri Nets and Concurrency.* PETRI NETS 2025. 2025. In press.

[14] C. T. Schwanen, W. Pakusa, and W. M. P. van der Aalst. “A Dynamic Programming Approach for Alignments on Process Trees.” In: *Process Mining Workshops.* ICPM 2024. Vol. 533. LNBIP. Cham: Springer Nature Switzerland, 2025, pp. 84–97. DOI: 10.1007/978-3-031-82225-4_7.

[15] C. T. Schwanen, W. Pakusa, and W. M. P. van der Aalst. “Process Tree Alignments.” In: *Enterprise Design, Operations, and Computing.* EDOC 2024. Vol. 15409. LNCS. Cham: Springer International Publishing, 2025. DOI: 10.1007/978-3-031-78338-8_16.

[16] H. Zhang and F. Ding. “On the Kronecker Products and Their Applications.” In: *Journal of Applied Mathematics* 2013 (2013), pp. 1–8. DOI: 10.1155/2013/296185.