Process Mining through Tree Automata

Michal R. Przybylek
Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Warszawa, Poland

Keywords: Evolutionary Algorithms, Process Mining, Theory Discovery, Tree Automata.

Abstract: This paper introduces a new approach to mine business processes. We define bidirectional tree languages together with their finite models and show how they represent business processes. Then we propose an evolutionary heuristic based on skeletal algorithms to learn bidirectional tree automata. We show how the heuristic can be used in process mining.

1 INTRODUCTION

> ”Nowadays, there is no longer any question that the quality of a company’s business processes has a crucial impact on its sales and profits. The degree of innovation built into these business processes, as well as their flexibility and efficiency, are critically important for the success of the company. The importance of business processes is further revealed when their are considered as the link between business and IT; business applications only become business solutions when the processes are supported efficiently. The essential task of any standard business software is and always will be to provide efficient support of internal and external company processes.” — Torsten Scholz

In order to survive in today’s global economy more and more enterprises have to redesign their business processes. The competitive market creates the demand for high quality services at lower costs and with shorter cycle times. In such an environment business processes must be identified, described, understood and analysed to find inefficiencies which cause financial losses.

One way to achieve this is by modelling. Business modelling is the first step towards defining a software system. It enables the company to look afresh at how to improve organization and to discover the processes that can be solved automatically by software that will support the business. However, as it often happens, such a developed model corresponds more to how people think of the processes and how they wish the processes would look like, then to the real processes as they take place.

Another way is by extracting information from a set of events gathered during executions of a process. Process mining (van der Aalst, 2011; Valiant, 1984; Weijters and van der Aalst, 2001; de Medeiros et al., 2004; van der Aalst et al., 2000; van der Aalst et al., 2006b; Wynn et al., 2004; van der Aalst et al., 2006a; van der Aalst and M. Pesic, 2009; van der Aalst and van Dongen, 2002; Wen et al., 2006; Ren et al., 2007) is a growing technology in the context of business process analysis. It aims at extracting this information and using it to build a model. Process mining is also useful to check if the “a priori model” reflects the actual situation of executions of the processes. In either case, the extracted knowledge about business processes may be used to reorganize the processes to reduce they time and cost for the enterprise.

The aim of this paper is to extend methods for exploration of business processes developed in (Przybylek, 2013) to improve their effectiveness in a business environment. We generalise finite automata to bidirectional tree automata, which allow us to mine parallel processes. Then we express the process of learning bidirectional tree automata in terms of skeletal algorithms. We show sample applications of our algorithms in mining business processes.

2 SKELETAL ALGORITHMS

Skeletal algorithms (Przybylek, 2013) are a new branch of evolutionary metaheuristics (Bremermann, 1962; Friedberg, 1956; Friedberg et al., 1959; Rechenberg, 1971; Holland, 1975) focused on data and process mining. The basic idea behind the

R. Przybylek M..
Process Mining through Tree Automata.
DOI: 10.5220/0004555201520159
In *Proceedings of the 5th International Joint Conference on Computational Intelligence* (ECTA-2013), pages 152-159
ISBN: 978-989-8565-77-8

skeletal algorithm is to express a problem in terms of congruences on a structure, build an initial set of congruences, and improve it by taking limited unions/intersections, until a suitable condition is reached. Skeletal algorithms naturally arise in the context of data/process mining, where the skeleton is the "free" structure on initial data and a congruence corresponds to similarities in the data. In such a context, skeletal algorithms come equipped with fitness functions measuring the complexity of a model.

Skeletal algorithms, search for a solution of a problem in the set of quotients of a given structure called the skeleton of the problem. More formally, let S be a set, and denote by $Eq(S)$ the set of equivalence relations on S. If $i \in S$ is any element, and $A \in Eq(S)$ then by $[i]_A$ we shall denote the abstraction class of i in A — i.e. the set $\{j \in S\colon jAi\}$. We shall consider the following skeletal operations on $Eq(S)$:

1. Splitting
   The operation $split\colon \{0,1\}^S \times S \times Eq(S) \to Eq(S)$ takes a predicate $P\colon S \to \{0,1\}$, an element $i \in S$, an equivalence relation $A \in Eq(S)$ and gives the largest equivalence relation R contained in A and satisfying: $\forall_{j\in[i]_A} iRj \Rightarrow P(i) = P(j)$. That is — it splits the equivalence class $[i]_A$ on two classes: one for the elements that satisfy P and the other of the elements that do not.
2. Summing
   The operation $sum\colon S \times S \times Eq(S) \to Eq(S)$ takes two elements $i, j \in S$, an equivalence relation $A \in Eq(S)$ and gives the smallest equivalence relation R satisfying iRj and containing A. That is — it merges the equivalence class $[i]_A$ with $[j]_A$.
3. Union
   The operation $union\colon S \times Eq(S) \times Eq(S) \to Eq(S) \times Eq(S)$ takes one element $i \in S$, two equivalence relations $A, B \in Eq(S)$ and gives a pair $\langle R, Q\rangle$, where R is the smallest equivalence relation satisfying $\forall_{j\in[i]_B} iRj$ and containing A, and dually Q is the smallest equivalence relation satisfying $\forall_{j\in[i]_A} iQj$ and containing B. That is — it merges the equivalence class corresponding to an element in one relation, with all elements taken from the equivalence class corresponding to the same element in the other relation.
4. Intersection
   The operation $intersection\colon S \times Eq(S) \times Eq(S) \to Eq(S) \times Eq(S)$ takes one element $i \in S$, two equivalence relations $A, B \in Eq(S)$ and gives a pair $\langle R, Q\rangle$, where R is the largest equivalence relation satisfying $\forall_{x,y\in[i]_A} xRy \Rightarrow x, y \in [i]_B \vee x, y \notin [i]_A$ and contained in A, and dually Q is the largest equivalence relation satisfying $\forall_{x,y\in[i]_B} xQy \Rightarrow x, y \in [i]_A \vee x, y \notin [i]_A$ and contained in B. That is — it intersects the equivalence class corresponding to an element in one relation, with the equivalence class corresponding to the same element in the other relation.

Furthermore, we assume that there is also a fitness function. There are many things that can be implemented differently in various problems.

2.1 Construction of the Skeleton

As pointed out earlier, the skeleton of a problem should correspond to the "free model" build upon sample data. Observe, that it is really easy to plug in the skeleton some priori knowledge about the solution — we have to construct a congruence relation induced by the priori knowledge and divide by it the "free unrestricted model". Also, this suggests the following optimization strategy — if the skeleton of a problem is too big to efficiently apply the skeletal algorithm, we may divide the skeleton on a family of smaller skeletons, apply to each of them the skeletal algorithm to find quotients of the model, glue back the quotients and apply again the skeletal algorithm to the glued skeleton.

2.2 Construction of the Initial Population

Observe that any equivalence relation on a finite set S may be constructed by successively applying *sum* operations to the identity relation, and given any equivalence relation on S, we may reach the identity relation by successively applying *split* operations. Therefore, every equivalence relation is constructible from *any* equivalence relation with *sum* and *split* operations. If no priori knowledge is available, we may build the initial population by successively applying to the identity relation both *sum* and *split* operations.

2.3 Selection of Operations

For all operations we have to choose one or more elements from the skeleton S, and additionally for a split operation — a splitting predicate $P\colon S \to \{0,1\}$. In most cases these choices have to reflect the structure of the skeleton — i.e. if our models have an algebraic or coalgebraic structure, then to obtain a quotient model, we have to divide the skeleton by an equivalence relation *preserving* this structure, that is, by a congruence. The easiest way to obtain a congruence is to choose operations that map congruences to congruences. Another approach is to allow operations that move out congruences from they class, but then

"improve them" to congruences, or just punish them in the intermediate step by the fitness function.

2.4 Choosing appropriate Fitness Function

Data and process mining problems frequently come equipped with a natural fitness function measuring the total complexity of data given a particular model. One of the crucial conditions that such a function has to satisfy is the ability to easily adjust its value on a model obtained by applying skeletal operations.

2.5 Creation of Next Population

There is a room for various approaches. We have experimented most successful with the following strategy — append k-best congruences from the previous population to the result of operations applied in the former step of the algorithm.

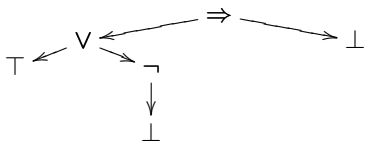
3 TREE LANGUAGES AND TREE AUTOMATA

Let us first recall the definition of an ordinary tree language and automaton (Comon et al., 2007). A ranked alphabet is a function $arity\colon \Sigma \to \mathcal{N}$ from a finite set of symbols Σ to the set of natural numbers $\mathcal{N}$ called arities of the symbols. We shall write σ/k to indicate that the arity of a symbol $\sigma \in \Sigma$ is $k \in \mathcal{N}$, that is $arity(\sigma) = k$. One may think of a ranked alphabet as of an algebraic signature — then a word over a ranked alphabet is a ground term over corresponding signature.

Example 3.1 (Propositional logic)**.** *A ranked alphabet of the propositional logic consists of symbols:*

$$\{\bot/0, \top/0, \vee/2, \wedge/2, \neg/1, \Rightarrow/2\}$$

Every propositional sentence like "$\top \vee \neg\bot \Rightarrow \bot$" corresponds to a word over the above alphabet — in this case to: "$\Rightarrow(\vee(\top,\neg(\bot)),\bot)$", or writing in a tree-like fashion:



Following (Comon et al., 2007) we define a finite top-down tree automaton over $arity\colon \Sigma \to \mathcal{N}$ as a tuple $A = \langle Q, q_s, \Delta\rangle$, where Q is a set of states, $q_s \in Q$ is the initial state, and Δ is the set of rewrite rules, or transitions, of the type:

$$q_0(f(x_1,\ldots,x_n)) \to f(q_1(x_1),\ldots,q_n(x_n))$$

where $f/n \in \Sigma$ and $q_i \in Q$ for $i = 0..n$. The rewrite rules are defined on the ranked alphabet $arity\colon \Sigma \to \mathcal{N}$ extended with $q/1$ for $q \in Q$. A word w is recognised by automaton A if $q_s(w) \xrightarrow{\Delta^*} w$, that is, if w may be obtained from $q_s(w)$ by successively applying finitely many rules from Δ.

We shall modify the definition of a tree automaton in two directions. First, it will be more convenient to associate symbols with states of an automaton, rather then with transitions. Second, we extend the definition of a ranked alphabet to allow terms return multiple results; moreover, to fit better the concept of business processes, we identify terms that are equal up to a permutation of their arguments and results.

Definition 3.1 (Ranked alphabet)**.** *A ranked alphabet is a function $biarity\colon \Sigma \to \mathcal{N} \times \mathcal{N}^+$. If the ranking function is known from the context, we shall write $\sigma/i/j \in \Sigma$ for a symbol $\sigma \in \Sigma$ having input arity i and output arity j; that is, if $biarity(\sigma) = \langle i, j\rangle$.*

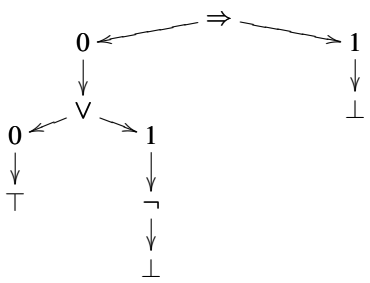
A definition of a term is more subtle, so let us first consider some special cases. By a multiset we shall understand a function $\overline{(-)}$ from a set X to the set of positive natural numbers $\mathcal{N}^+$ — it assigns to an element $x \in X$ its number of occurrences $\bar{x}$ in the multiset. If X is finite, then we shall write $\{\{x_1,\ldots,x_1,x_2,\ldots,x_2,\ldots x_k,\ldots\}\}$, where an element $x_k \in X$ occurs n-times when $\bar{x} = n$, and call the multiset finite. For multisets we use the usual set-theoretic operations $\cup, \cap, /$ defined pointwise — with possible extension or truncation of the domains.

A simple language over a ranked alphabet Σ is the smallest set of pairs, called simple terms, containing $\langle\sigma/0/j, \emptyset\rangle$ for each nullary symbol $\sigma/0/j \in \Sigma$ and closed under the following operation: if $\sigma/i/j \in \Sigma$ and $t_1 = \langle x_1/i_1/j_1, A_1\rangle, \ldots, t_k = \langle x_k/i_k/j_k, A_k\rangle$ are simple terms such that $\sum_{s=1}^{k} j_s = i$, then $\langle\sigma/i/j, \{\{t_k\colon 1 \le s \le k\}\}\rangle$ is a simple term. For convenience we write $\sigma\{\{t_1,\ldots,t_k\}\}$ for $\langle\sigma/i/j, \{\{t_k\colon 1 \le s \le k\}\}\rangle$ and call t_s a subterm of $\sigma\{\{t_1,\ldots,t_k\}\}$.

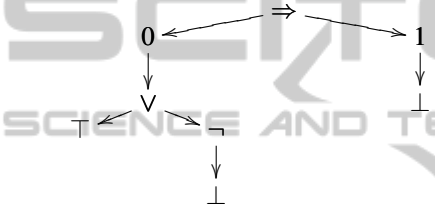
Example 3.2 (Ordinary language)**.** *A word over an ordinary alphabet Σ may be represented as a simple term over the ranked alphabet $biarity(\sigma) = (1,1)$ for $\sigma \in \Sigma$ and $biarity(\varepsilon) = (0,1)$.*

Example 3.3 (Ordinary tree language)**.** *A word over an ordinary ranked alphabet may be represented as a simple term over the ranked alphabet extended with unary symbols $n/1/1$ for natural numbers $n \in \mathcal{N}$ indicating a position of an argument. A tree-*

representation of sentence "$\top \vee \neg\bot \Rightarrow \bot$" from Example 3.1 have the following form:



Notice, that in every semantic of (any) propositional calculus $A \vee B \equiv B \vee A$, therefore we may use this knowledge on the syntax level and represent sentence "$\top \vee \neg\bot \Rightarrow \bot$" in a more compact form — carrying some extra information about possible models:



We extend the notion of a simple term to allow a single term to be a subterm of more than one term. Such extension would be trivial for ordinary terms, but here, thanks to the ability of returning more than one value, it gives us an extra power which is crucial for representing business processes.

Definition 3.2 (Term)**.** *Let Σ be a ranked alphabet. A term over Σ is a finite acyclic coalgebra $\langle S, s_0 \in S, subterm\colon S \to \mathcal{N}^{+S}, name\colon S \to \Sigma\rangle$ satisfying the following compatibility conditions:*

$$\forall_{x\in S} \sum_{y\in S} subterm(x)(y) = name(x)_1$$
$$\forall_{y\in S\setminus\{s_0\}} \sum_{x\in S} subterm(x)(y) = name(y)_2$$

where subscripts $_1$ and $_2$ indicates projections on first (i.e. input arity) and second (i.e. output arity) component respectively Two terms $\langle S, s_0, subterm, name\rangle$ and $\langle S', s'_0, subterm', name'\rangle$ are equivalent if there exists an isomorphism of the coalgebras, that is, if there exists a bijection $\sigma\colon S \to S'$ such that $\sigma(s_0) = s'_0$, $\mathcal{N}^{+\sigma} \circ subterm \circ \sigma = subterm'$ and $name \circ \sigma = name'$.

We shall not distinguish between equivalent terms.

Example 3.4 (Simple term)**.** *Consider a simple term t over a ranked alphabet Σ. It corresponds to the term $\langle S, s_0 \in S, subterm\colon S \to \mathcal{N}^{+S}, name\colon S \to \Sigma\rangle$, where S is the smallest multiset containing t and closed under subterms, $s_0 = t$, $name(\sigma\{\{t_1,\ldots,t_k\}\}) = \sigma$ and $subterm(\sigma\{\{t_1,\ldots,t_k\}\}) = \{\{t_1,\ldots,t_k\}\}$.*

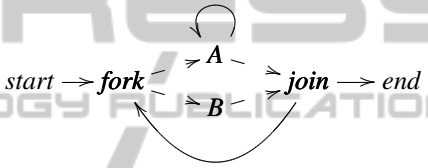
In line with the above example, we shall generally represent a term as a sequence of equations (add multiple variables, please):

$$\begin{aligned} & \sigma_0\{\{t_{0,1},\ldots,t_{0,k_0}\}\} \text{ in free variables } x_1,\ldots,x_n \\ x_1 \;=\; & \sigma_1\{\{t_{1,1},\ldots,t_{1,k_0}\}\} \text{ in free variables } x_2,\ldots,x_n \\ & \ldots \\ x_n \;=\; & \sigma_n\{\{t_{n,1},\ldots,t_{n,k_n}\}\} \text{ without free variables} \end{aligned}$$

where $t_{i,j}$ are simple terms and x_i are multisets of variables.

Corollary 3.1. *Terms are tantamount to finite sets of equations of the form $x = \sigma\{\{t_1,\ldots,t_k\}\}$ over simple terms without cyclic dependencies of free variables.*

Example 3.5 (Terms from a business process)**.** *Consider a business process:*



which starts in the "start" state and ends in the "end" state. The semantics of the process is that one have to preform simultaneously task B and at least one task A and then either finish or repeat the whole process. Some terms t_1, t_2, t_3 generated by this process are:

$$\begin{aligned} t_1 &= start\{\{fork\{\{A\{\{x\}\}, B\{\{x\}\}\}\}\}\} \\ x &= join\{\{end\}\} \\ t_2 &= start\{\{fork\{\{A\{\{A\{\{x\}\}\}, B\{\{x\}\}\}\}\}\} \\ x &= join\{\{end\}\} \\ t_3 &= start\{\{fork\{\{A\{\{A\{\{A\{\{x\}\}\}\}\}, B\{\{x\}\}\}\}\}\} \\ x &= join\{\{fork\{\{A\{\{A\{\{x\}\}\}\}, B\{\{y\}\}\}\}\}\} \\ y &= join\{\{end\}\} \end{aligned}$$

Generally, every term t generated by this process has to be of the following form:

$$\begin{aligned} t &= start\{\{fork\{\{A^{k_1}\{\{x_1\}\}, B\{\{x_1\}\}\}\}\}\} \\ x_1 &= join\{\{fork\{\{A^{k_2}\{\{x_2\}\}, B\{\{x_2\}\}\}\}\}\} \\ & \ldots \\ x_{n-1} &= join\{\{fork\{\{A^{k_n}\{\{x_m\}\}, B\{\{x_m\}\}\}\}\}\} \\ x_n &= join\{\{end\}\} \end{aligned}$$

The whole business process cannot be represented as a single term. One could write the following set of equations:

$$\begin{aligned} t &= start\{\{x\}\} \\ x &= fork\{\{A\{\{y\}\}, B\{\{z\}\}\}\} \\ y &= A\{\{y\}\} \vee y = z \\ z &= join\{\{x\}\} \vee z = join\{\{end\}\} \end{aligned}$$

However, there is no term corresponding to this set — there are cyclic dependencies between variables (for example y depends on y, also x depends on z, z depends on x), and there are disjunctions in the set of equations.

Definition 3.3 (Tree Automaton.)**.** *A tree automaton over a ranked alphabet Σ is a tuple $A = \langle Q, q_0, \Delta, name \rangle$, where:*

- *Q is the set of states of the automaton*
- *$q_0 \in Q$ is the initial state of the automaton*
- *name is a function from set of states Q to $\Sigma \sqcup \{\varepsilon/0/1\}$*
- *Δ is a set of rewrite rules (transitions) of the form:*

$$\{\{x_0, \ldots, x_k\}\} \xrightarrow{\delta} \{\{x'_0, \ldots, x'_l\}\}$$

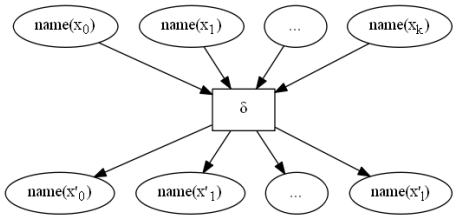
with:

$$\sum_{i=0}^{k} name(x_i)_1 = \sum_{i=0}^{l} name(x'_i)_0$$

where $x_0, \ldots, x_k, x'_0, \ldots, x'_l \in Q$.

Notice that in the above definition there is a single initial state, but there are no final states — an automaton finishes its run if it is in neither of the states.

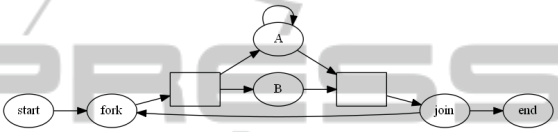
Example 3.6 (Business process as tree automaton)**.** *We shall use the following graphical representation of a tree automaton: every state is denoted by a circle with the letter associated to the state inside the circle, every rule $\{\{x_0, \ldots, x_k\}\} \xrightarrow{\delta} \{\{x'_0, \ldots, x'_l\}\}$ is denoted by a rectangle (optionally with letter δ inside); moreover this rectangle is connected by ingoing arrows from circles denoting states $\{\{x_0, \ldots, x_k\}\}$ and outgoing arrows to circles denoting states $\{\{x'_0, \ldots, x'_l\}\}$:*



For convenience we shell sometimes omit the intermediating box of a singleton rule $\{\{x\}\} \to \{\{x'\}\}$ and draw only a single arrow from the node representing x to the node representing x'. The business process from Example 3.5 defines over a signature $\Sigma = \{start/1/0, fork/2/1, A/1/1, B/1/1, join/1/2, end/0/1\}$ an automaton $\langle start, \Sigma, \Delta, id \rangle$ with rules Δ:

$$\begin{aligned}
\{\{start\}\} &\xrightarrow{\delta_1} \{\{fork\}\} \\
\{\{fork\}\} &\xrightarrow{\delta_2} \{\{A, B\}\} \\
\{\{A\}\} &\xrightarrow{\delta_3} \{\{A\}\} \\
\{\{A, B\}\} &\xrightarrow{\delta_4} \{\{join\}\} \\
\{\{join\}\} &\xrightarrow{\delta_5} \{\{fork\}\} \\
\{\{join\}\} &\xrightarrow{\delta_6} \{\{end\}\} \\
\{\{end\}\} &\xrightarrow{\delta_7} \{\{\}\}
\end{aligned}$$

which may be represented as:



Example 3.7 (Term as a skeletal tree automaton)**.** *The automaton corresponding to a term t is constructed in two steps. First we define the following automaton. For every $s \in S$ with $name(s) = \sigma/i/j$ define a multiset:*

$$E_s = \{\{\varepsilon_{s,1}, \varepsilon_{s,2}, \ldots, \varepsilon_{s,j}\}\}$$

and a rule:

$$\{\{s\}\} \to E_s$$

and for every $p \in S$ with $k = subterm(p)(s)$ choose any k-element subset X_p of E_p and put a rule:

$$\bigcup_{p \in S} X_p \to \{\{s\}\}$$

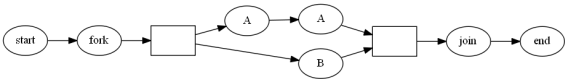
Then, for convenience, we simplify the automaton by cutting at ε-states. That is: every pair of rules

$$\begin{aligned}
X &\to \{\{Y, E\}\} \\
\{\{E\}\} &\to Z
\end{aligned}$$

where E consists only of ε-states, is replaced by a single rule:

$$X \to \{\{Y, Z\}\}$$

The next picture illustrates the skeletal automaton constructed from term t_2 from Example 3.5.



Given a finite multiset X, a rule $\{\{x_0, \ldots, x_k\}\} \xrightarrow{\delta} \{\{x'_0, \ldots, x'_l\}\}$ is applicable to X if $\{\{x_0, \ldots, x_k\}\}$ is a multisubset of X. In such a case we shall write $\delta[X]$ for the multiset

$(X \setminus \{\{x_0,\ldots,x_k\}\}) \cup \{\{x'_0,\ldots,x'_l\}\}$. We say that a term $t = \langle S, s_0, subterm_t, name_t\rangle$ is recognised by an automaton $A = \langle Q, q_0, \Delta_A, name_A\rangle$ if there is a finite sequence $\langle\{\{q_0\}\},\{\{q_0 \mapsto s_0\}\}\rangle = T_0, T_1, \ldots, T_n = \langle\{\{\}\},\{\{\}\}\rangle$ with $name(q_0) = name(s_0)$ satisfying for all $0 < m < n$ the induction laws:

- $T_{m+1} = \langle \delta[X_m], \pi_m[x_1 \not\mapsto, \ldots, x_k \not\mapsto][x'_1 \mapsto r'_1, \ldots, x'_l \mapsto r'_l]\rangle$
- $\langle X_m, \pi_m\rangle = T_m$
- a rule $\{\{x_0,\ldots,x_k\}\} \xrightarrow{\delta} \{\{x'_0,\ldots,x'_l\}\} \in \Delta_A$ is applicable to X_m and $subterm_t(\pi_m(x_0)) = subterm_t(\pi_m(x_1)) = \cdots = subterm_t(\pi_m(x_k)) = \{\{r_0,\ldots,r_l\}\}$
- if $name_A(x'_i) = \varepsilon$ then $r'_i = \varepsilon\{\{r_i\}\}$
- if $name_A(x'_i) \neq \varepsilon$ then $name_t(r_i) = name_A(x'_i)$ and $r'_i = r_i$

Notice that because $X_n = \{\{\}\}$, the last applied rule has to be of the form $\{\{x_0,\ldots,x_k\}\} \xrightarrow{\delta} \{\{\}\}$ and due to the compatibility condition on rules of a tree automaton:

$$\sum_{i=0}^{k} name_A(x_i)_1 = 0$$

which means that the states $x_0,\ldots,x_k$ generate only nullary letters. Therefore the corresponding subterms $\{\{\pi(x_0),\ldots,\pi(x_k)\}\}$ of t are nullary.

Example 3.8. *Let us show that term t_2 from Example 3.5 is recognised by automaton $\langle start, \Sigma, \Delta, id\rangle$ from Example 3.6. Since $name(t_2) = start = id(start)$ we may put $T_0 = \langle\{\{strat\}\}, strat \mapsto t\rangle$ and consider the following sequence:*

- $T_1 = \langle\{\{fork\}\}, fork \mapsto fork\{\{A\{\{A\{\{x\}\}\}\}, B\{\{x\}\}\}\}\rangle$ *by* δ_1
- $T_2 = \langle\{\{A,B\}\}, A \mapsto A\{\{A\{\{x\}\}\}\}, B \mapsto B\{\{x\}\}\rangle$ *by* δ_2
- $T_3 = \langle\{\{A,B\}\}, A \mapsto A\{\{A\{\{x\}\}\}\}, B \mapsto B\{\{x\}\}\rangle$ *by* δ_3
- $T_4 = \langle\{\{A,B\}\}, A \mapsto A\{\{x\}\}, B \mapsto B\{\{x\}\}\rangle$ *by* δ_3
- $T_5 = \langle\{\{join\}\}, join \mapsto join\{\{end\}\}\rangle$ *by* δ_4
- $T_6 = \langle\{\{end\}\}, end \mapsto end\rangle$ *by* δ_6
- $T_7 = \langle\{\{\}\},\{\{\}\}\rangle$ *by* δ_7

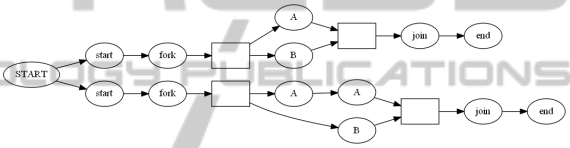
it is easy to verify that each T_m is constructed according to the induction laws.

4 SKELETAL ALGORITHMS IN TREE MINING

Given a finite list K of sample terms over a common alphabet Σ, we shall construct the skeletal automaton $skeleton(K) = \langle q_0, S, \Delta, name\rangle$ of K in the following way. For each term $K_i, 0 \leq i < length(K)$ let $skeleton(K_i) = \langle q^i_0, S^i, \Delta^i, name^i\rangle$ be the skeletal automaton of K_i constructed like in Example 3.7, then:

- $S = \{START\} \sqcup \bigcup_i S^i$
- $q_0 = START$
- $\Delta = \{\{\{START\}\} \to \{\{q^i_0\}\} : 0 \leq i < length(K)\} \sqcup \bigcup_i \Delta^i$
- $name(q) = \begin{cases} START & \text{if } q = \\ name^i(q) & \text{if } q \in S^i \end{cases}$

That is $skeleton(K) = \langle \Sigma, S, l, \delta\rangle$ constructed as a disjoint union of skeletal automatons for t_k enriched with two states *start* and *end*. So the skeleton of a sample is just an automaton corresponding to the disjoint union of skeletal automaton corresponding to each of the terms enriched with a single starting state. Such automaton describes the situation, where all actions are different. Our algorithm will try to glue some actions that give the same output (shall search for the best fitting automaton in the set of quotients of the skeletal automaton). The next figure shows the skeletal automaton of the sample t_1, t_2 from Example 3.7.



Given a finite list of sample data K, our search space $Eq(K)$ consists of all equivalence relations on the set of states S of the skeletal automaton for K.

4.1 Skeletal Operations

1. Splitting

   For a given congruence A, choose randomly a state $q \in skeleton(K)$ and make use of two types of predicates

   - split by output: $P(p) \Leftrightarrow \exists_{q' \in [q]_A} \exists_{X \xrightarrow{\delta} Y} p \in X \wedge q' \in Y$
   - split by input: $P(p) \Leftrightarrow \exists_{q' \in [q]_A} \exists_{X \xrightarrow{\delta} Y} q' \in X \wedge p \in Y$

2. Summing

   For a given congruence A, choose randomly two states p, q such that $name(p) = name(q)$.

3. Union/Intersection

   Given two skeletons A, B choose randomly a state $q \in skeleton(K)$.

Let us note that by choosing states and predicates according to the above description, all skeletal operations preserve congruences on $skeleton(K)$.

4.2 Fitness

The idea behind the fitness function for bidirectional tree automata is the same as for ordinary finite automata analysed in (Przybylek, 2013). The additional difficulty comes here from two reasons: a bidirectional tree automaton can be simultaneously in a multiset of states; moreover, two transitions may non-trivially depend on each other. Formally, let us say that two transitions $X \xrightarrow{\delta} Y$ and $X' \xrightarrow{\delta'} Y'$ are depended on each other if $X \cap X' \neq \{\{\}\}$, and are fully depended if $X = X'$. Unfortunately, extending the Bayesian interpretation to our framework yields a fitness function that is impractical from the computational point of view. For this reason we shall propose a fitness function that agrees with Bayesian interpretation only on some practical class of bidirectional tree automata — directed tree automata. A directed tree automaton is a bidirectional tree automaton whose each pair of rules is either fully depended or not depended. Now if δ is a sequence of rules of a directed tree automaton, then similarly to the Bayesian probability in (Przybylek, 2013), we may compute the probability of a multiset of states X:

$$p^{\delta}(X) = \frac{\Gamma(k)}{\Gamma(n+k)} \prod_{i=1}^{k} c_i^{c_i}$$

where:

- k is the number of rules $X \xrightarrow{\delta_i} Y$ for some Y of the automaton
- c_i is the total number of i-th rule $X \xrightarrow{\delta_i} Y$ used in δ
- $n = \sum_{i=1}^{k} c_i$ is the total number of rules of the form $X \to Y$ for some Y used in δ

and the total distribution as:

$$p(\delta) = \prod_{X \subseteq S} p^{\delta}(X)$$

which corresponds to the complexity:

$$p(\delta) = -\sum_{X \subseteq S} log(p^{\delta}(X))$$

This complexity does not include any information about the exact model of an automaton. Therefore, we have to adjust it by adding "the code" of a model. By using two-parts codes, we may write the fitness function in the following form:

$$fitness(A) = length(skeleton(K)/A) - \sum_{X \subseteq S} log(p^{\delta}(X))$$

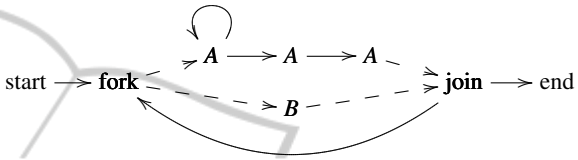
where $length(skeleton(K)/A)$ is the length of the quotient of the skeletal automaton $skeleton(K)$ by congruence A under any reasonable coding, and S is the set of states of the quotient automaton. For sample problems investigated in the next section, we chose this length to be:

$$clog(|S|)|\{\langle \delta, x \rangle : X \xrightarrow{\delta} Y \in \Delta, 0 \leq x < size(X) + size(Y)\}|$$

for constant $1 \leq c \leq 2$.

4.3 Bussiness Procces Mining

We shall start with a business process similar to one investigated in Example 3.5, but extended with multiple states generating the same action A:



This process starts in state *start* then performs simultaneously at least three tasks A and exactly one task B, and then finishes in *end* state. Figures 1, 2, 3 shows automata mined from 1, 2, and 8 random samples (with equal probabilities) for fitness function described in the previous section with $c = 2$.

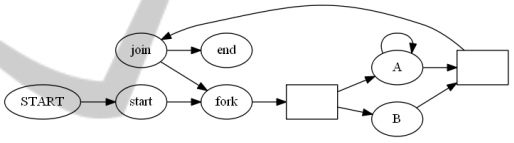


Figure 1: Model discovered after seeing 1 sample, $c = 2$.

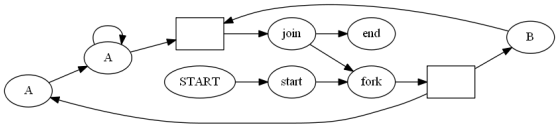


Figure 2: Model discovered after seeing 4 samples, $c = 2$.

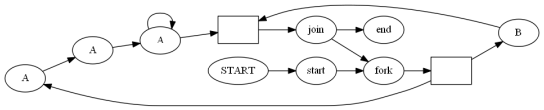


Figure 3: Model discovered after seeing 10 samples, $c = 2$.

Notice that the first mined automaton correspond to the minimal automaton recognizing any sample, and after seeing 8 samples the initial model is fully recovered. If we change our parameter c to 1, meaning that the fitness function should less prefer small models than we get automatons like in Figures 4, 5, 6.

Since the probability of generating $3 + n$ actions A is exponentially small, for large number of samples (in our case, 10), automata mined with $c = 2$ and $c = 1$ should be similar.

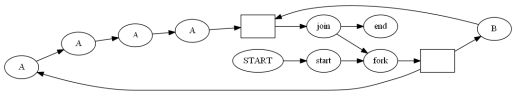


Figure 4: Model discovered after seeing 1 sample, $c = 1$.

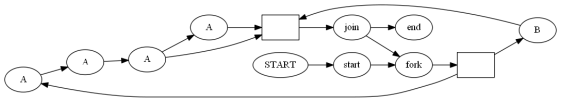


Figure 5: Model discovered after seeing 4 samples, $c = 1$.

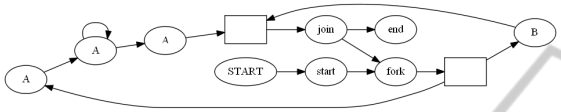


Figure 6: Model discovered after seeing 10 samples, $c = 1$.

5 CONCLUSIONS

In this paper we defined bidirectional tree automata, and showed how they can represent business process. We adapted skeletal algorithms introduced in (Przybylek, 2013) to mine bidirectional tree automata, resolving the problem of mining nodes that corresponds to parallel executions of a process (i.e. AND-nodes). In future works we will be mostly interested in validating the presented algorithms in industrial environment and apply them to real data.

REFERENCES

Bremermann, H. J. (1962). Optimization through evolution and recombination. In *Self-Organizing systems 1962, edited M.C. Yovitts et al.*, page 93106, Washington. Spartan Books.

Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (2007). Tree automata techniques and applications.

de Medeiros, A., van Dongen, B., van der Aalst, W., and Weijters, A. (2004). Process mining: Extending the alpha-algorithm to mine short loops. In *BETA Working Paper Series*, Eindhoven. Eindhoven University of Technology.

Friedberg, R. M. (1956). A learning machines part i. In *IBM Journal of Research and Development*, volume 2.

Friedberg, R. M., Dunham, B., and North, J. H. (1959). A learning machines part ii. In *IBM Journal of Research and Development*, volume 3.

Holland, J. H. (1975). Adaption in natural and artificial systems. Ann Arbor. The University of Michigan Press.

Przybylek, M. R. (2013). Skeletal algorithms in process mining. In *Studies in Computational Intelligence*, volume 465. Springer-Verlag.

Rechenberg, I. (1971). Evolutions strategie – optimierung technischer systeme nach prinzipien der biologischen evolution. In *PhD thesis*. Reprinted by Fromman-Holzboog (1973).

Ren, C., Wen, L., Dong, J., Ding, H., Wang, W., and Qiu, M. (2007). A novel approach for process mining based on event types. In *IEEE SCC 2007*, pages 721–722.

Valiant, L. (1984). A theory of the learnable. In *Communications of The ACM*, volume 27.

van der Aalst, W. (2011). Process mining: Discovery, conformance and enhancement of business processes. Springer Verlag.

van der Aalst, W., de Medeiros, A. A., and Weijters, A. (2006a). Process equivalence in the context of genetic mining. In *BPM Center Report BPM-06-15*, BPMcenter.org.

van der Aalst, W. and M. Pesic, M. S. (2009). Beyond process mining: From the past to present and future. In *BPM Center Report BPM-09-18*, BPMcenter.org.

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., and Barros, A. (2000). Workflow patterns. In *BPM Center Report BPM-00-02*, BPMcenter.org.

van der Aalst, W. and van Dongen, B. (2002). Discovering workflow performance models from timed logs. In *Engineering and Deployment of Cooperative Information Systems*, pages 107–110.

van der Aalst, W., Weijters, A., and Maruster, L. (2006b). Workflow mining: Discovering process models from event logs. In *BPM Center Report BPM-04-06*, BPMcenter.org.

Weijters, A. and van der Aalst, W. (2001). Process mining: Discovering workflow models from event-based data. In *Proceedings of the 13th Belgium-Netherlands Conference on Artificial Intelligence*, pages 283–290, Maastricht. Springer Verlag.

Wen, L., Wang, J., and Sun, J. (2006). Detecting implicit dependencies between tasks from event logs. In *Lecture Notes in Computer Science*, volume 3841, pages 591–603.

Wynn, M., Edmond, D., van der Aalst, W., and ter Hofstede, A. (2004). Achieving a general, formal and decidable approach to the or-join in workflow using reset nets. In *BPM Center Report BPM-04-05*, BPMcenter.org.