3rd International Workshop on Trends in Tree Automata and Tree Transducers

TTATT 2015

Informal Proceedings

ETAPS’15 Satellite Workshop    London, April 18
This volume contains proceedings of the 3rd International Workshop on Trends in Tree Automata and Tree Transducers (TTATT’15). The workshop was held on April 18, 2015 in London, and was a satellite event of European Joint Conferences on Theory and Practice of Software (ETAPS 2015).

The workshop provides an opportunity for researchers from different areas to exchange ideas on the theory and practice of tree automata and transducers. Tree automata and transducers have a long history within computer science, of more than 40 years, but over the last decade, they have experienced a huge revival of interest, due to their connections to other areas such as bioinformatics, computer networks, databases, linguistics, and verification.

I wish to express my gratitude to all authors, PC members, external reviewers and ETAPS satellite workshops organisers (special thank to Paulo Oliva), as well as to Mikołaj Bojańczyk, Christof Loeding and Sebastian Maneth who kindly accepted to give an invited talk.

– Emmanuel Filiot (Université Libre de Bruxelles, organizer) –

Invited presentations

Three invited presentations have been given at TTATT 2015.

• Synthesis of Transducers from Automatic Specifications. Christof Loeding (RTWH Aachen).
  Abstract: Given a specification as a binary relation that relates inputs to admissible outputs, the synthesis problem asks for a program that produces for each input an output that is admissible according to the specification. We consider this problem over the domain of words and trees. The specifications are given by automatic relations, that is, relations that can be defined by finite automata that read pairs of structures and accept those pairs that are in the relation. For such specifications, we study the synthesis problem of deterministic finite state transducers. In particular, we present some recent results and ongoing work on the synthesis of tree transducers.

• Transducers with origin information. Mikołaj Bojańczyk (Warsaw University).
  Abstract: Call a string-to-string function regular if it can be realised by one of the following equivalent models: mso transductions, two-way deterministic automata with output, and streaming transducers with registers. In the talk, based on an ICALP’14 paper, I propose to treat origin information as part of the semantics of a regular string-to-string function. With such semantics, the model admits a machine-independent characterisation, Angluin-style learning in polynomial time, as well as effective characterisations of natural subclasses such as one-way transducers or first-order definable transducers.


Regular contributions

For this edition, seven contributions have been submitted. Each of them have been reviewed by three reviewers. Five contributions have been accepted and presented at the workshop:


4. Younes Guellouma, Hadda Cherroun, Djelloul Ziadi and Bruce Watson. From tree automata minimization to string automata minimization.

5. Zoltán Fülöp and Andreas Maletti. Linking Theorems for Tree Transducers.
Program committee  The program committee of TTATT 2015 consisted of:

- Arnaud Carayol (Université Paris-Est)
- Olivier Carton (Université Paris-Diderot)
- Emmanuel Filiot (chair, Université Libre de Bruxelles)
- Andreas Maletti (Universität Stuttgart)
- Sebastian Maneth (University of Edinburgh)
- Keisuke Nakano (UEC Tokyo)
- Helmut Seidl (TU München)
- Hiroyuki Seki (Nagoya University)
- Jean-Marc Talbot (Université Aix-Marseille)
- Margus Veanes (Microsoft Research)

External reviewers  There were two external reviewers:

- Raffaella Gentilini (University of Perugia)
- Ismaël Jecker (Université Libre de Bruxelles)
Construction of Thompson’s Automaton from a Regular Tree Expression *

Ahlem Belabbaci  
LIM - University of Laghouat, Algeria  
ah.belabbaci@lagh-univ.dz

Hadda Cherroun  
LIM - University of Laghouat, Algeria  
h.cherroun@lagh-univ.dz

Djelloul Ziadi  
LITIS - University of Rouen, France  
Djelloul.Ziadi@univ-rouen.fr

Loek Cleophas  
Umeå University, Sweden and  
FASTAR Research Group, Stellenbosch University, South Africa  
loek@fastar.org

In this paper we deal with tree automata, particularly the construction of a tree automaton from a regular tree expression. In the late 1960s, Ken Thompson proposed an algorithm for the construction of an automaton from a regular expression over strings. In this work we try to generalize the former algorithm to trees. For this, we modify the general form of the tree automaton in order to facilitate the operations of union, closure and concatenation. This construction allows us to perform pattern matching over trees.

1 Introduction

Trees are widely used in applications nowadays, hence the importance of dealing with tree automata. Among the fields of handling of tree automata appears tree pattern matching, term rewriting, model checking, XML, ...

In the case of words, several algorithms were proposed in order to convert a regular expression into an automaton. The most common construction is the standard or position automaton [7, 12]. Brzozowski’s construction [2] of a deterministic finite automaton uses derivatives of regular expressions. This approach was modified by Antimirov [1] who defined partial derivatives to construct a non-deterministic automaton from a regular expression E. Another construction was proposed by Thompson [13] based on induction over the structure of a regular expression.

By analogy to words, some algorithms were proposed for trees. Among these works is the one of Sebti and Ziadi [9], who gave an algorithm to compute the position tree automaton. The work of Kuske and Meinecke [8] consists of the definition of partial derivatives for regular tree expressions and then building a non-deterministic finite tree automaton recognizing the language denoted by such an expression. They adapt and modify the approach of Champarnaud and Ziadi [3, 4] in the word case. Tree derivatives were introduced by Levine in [10, 11] and extend the concept of Brzozowski’s string derivatives. He used tree derivatives for minimizing and characterizing tree automata. Tree derivatives are used as a basis of inference of tree automata from finite samples of trees.

In this paper we present the construction of Thompson’s automaton for the case of trees. In the next section, preliminaries about trees, tree automata and regular tree expressions are presented. Afterwards we recall the process of the construction of the Thompson’s automaton for strings in Section 3. Our

*This work is supported by a South Africa-Algeria Cooperation Project funded by the South African National Research Foundation and the Algerian MESRS-DGRSDT.
A ranked alphabet is a set of symbols provided with an arity (rank) function: \( \# \Sigma : \rightarrow \mathbb{N} \). For \( a \in \Sigma \), \( \# a \) is called the arity (rank) of \( a \). A symbol of arity 0 is called a constant, terminal node or leaf. We denote by \( \Sigma_n \) the set of all symbols of arity \( n \) of \( \Sigma \). \( \Sigma_0 \) is the set of leaves. We consider \( \varepsilon \in \Sigma_1 \) as the empty node.

An ordered labeled tree \( t \) with arity over an alphabet \( \Sigma \) can be a constant or of the form \( g(t_1, t_2, \ldots, t_g) \), such that \( t_i, 1 < i < g \), is itself a tree.

A tree automaton (TA) \( \mathcal{A} \) is a tuple \( (Q, Q_f, Q_\sigma, \Delta) \) such that \( Q \) is a finite set of states, \( Q_\sigma \), is the set of initial states, \( Q_f \subset Q \) is the acceptance state and \( \Delta \) is the set of transition rules. A transition rule is a triplet \((q_1, \ldots, q_n), a, q)\) where \( q_1, \ldots, q_n \in Q \), \( a \in \Sigma \) and \( \# a = n \). We use \( a(q_1, \ldots, q_n) \rightarrow q \) and \( a(q) \rightarrow (q_1, \ldots, q_n) \) to denote respectively bottom-up or top-down automaton transition rules. For \( n = 0 \) leaves rules are represented by \( a \rightarrow q \) or \( q \rightarrow a \). The notation \( \sigma(e)(i) = Q_f \) is used to indicate that there is a sequence of transitions in the automaton recognizing the tree \( t \). The language recognized by an automaton is the set of all trees that can be recognized using this automaton.

The graphical representation of the tree automaton is similar to the one of strings. The states are represented by circles with double circles for the final states, and transitions between states by edges labeled with a symbol of the alphabet \( \Sigma \) or \( \varepsilon \) [5]. For tree automata, some changes are made in the representation of transitions. A transition from state \( q_0 \) to states \( q_1, q_2, \ldots, q_n \) with a symbol or an \( e \)-transition is represented by i) an edge connecting the state \( q_0 \) to a small circle unlabeled with a symbol or \( \varepsilon \), ii) \( n \) edges connecting the small circle to the state \( q_i \) labeled by \( i \) where \( i : 1 \ldots n \). In the case of directed automata (top-down or bottom-up) edges are directed.

A regular tree expression over a ranked alphabet \( \Sigma \) is inductively defined by \( E = \varepsilon, E \in \Sigma_0, E = g(E_1, \ldots, E_n), E = (E_1 + E_2), E = (E_1 \cdot E_2), E = (E_1^{\ast}) \), where \( c \in \Sigma_0, n \in \mathbb{N}, g \in \Sigma_n \) and \( E_1, E_2, \ldots, E_n \) are any regular tree expressions over \( \Sigma \). \( \|E\| \) is the number of occurrences of symbols from the ranked alphabet \( \Sigma \) in a regular tree expression \( E \).

### 3 Thompson’s Automaton for Strings

In his 1968 article [13], Ken Thompson describes a technique for the construction of a deterministic finite automaton from a given regular expression.

According to this technique, the regular expression must first be converted into post-fixed notation. Then the automaton is built by successive composition of automata. These automata are assembled according to the basic operations of regular expressions, namely concatenation, union and closure. A Thompson’s automaton has the following properties:

- only one initial state and one final state; these two states are distinct;
- there are neither incoming transitions to the initial state nor outgoing transitions to the final one;
- any state is the origin of at most one transition labeled by a letter and of at most two \( \varepsilon \)-transitions.

Formally, there are five sorts of regular expression: the empty string \( \varepsilon \), any character \( a \), the union \( E + F \), the concatenation \( E \cdot F \) and the closure \( E^{\ast} \). Figure 3 illustrates the different constructions.
Given the difference between strings and trees concerning the concatenation and closure operations, we should take care when constructing the tree automaton. Indeed, we have designed a special form of tree automaton that allows us to inductively construct tree automata in a straightforward way from a regular tree expression.

The basic idea of our construction is to build from a given regular tree expression $E$, a finite bottom-up tree automaton which has the form illustrated by Figure 2. The main characteristic of this automaton is that it contains one initial state for each element of $\Sigma_0$ (the frame $Q_{\Sigma_0}$). This condition makes more sense in the operation of concatenation, since we have to perform concatenation just in one state. In order to keep this form, some $\varepsilon$-transitions are added during the inductive constructions.

Formally, a (bottom-up) TA $Th_E$ for a regular expression $E$ is a tuple $(Q^E, q_f^E, Q^E_{\Sigma_0}, \Delta^E)$, where:

- $Q^E$: set of all states of $Th_E$
- $q_f^E$: final state of $Th_E$
- $Q^E_{\Sigma_0}$: set of initial states of $Th_E$, $Q^E_{\Sigma_0} = \{q_a | a \in \Sigma_0\}$
- $\Delta^E$: set of transition rules of $Th_E$, $\Delta^E : (Q^E)^* \rightarrow Q^E$.

### 4.1 Construction

In this section, the different constructions of Thompson’s tree automata are presented with a formal definition and graphical representation of each automaton.
Elementary Automaton, Leaf Tree Automaton: It is a one-state automaton that consists of an initial state which is itself the accepting state (Figure 3). $Q^E = q^E_f = q^E_0$ and $\Delta^E = \{a \rightarrow q^E_f\}$

Automaton of Arity $E = g(E_1, E_2, \ldots, E_n)$: The automaton is built from $n$ other automata $E_i$ by adding a new final state $Q^E_f$ (Figure 4). To keep the basic structure of Thompson’s tree automata, we merge sets $Q^E_{\Sigma_0}$. This is done by:

- adding $k$ initial states $q^E_{i0}$ as $k = |\Sigma_0|$ and $a \in \Sigma_0$,
- adding the transition $a \rightarrow q^E_{i0}$ for all $a \in \Sigma_0$,
- removing transitions $a \rightarrow q^E_{i0}$, such that $a \in \Sigma_0$ and $i: 1 \ldots n$,
- adding $\varepsilon$-transitions between $q^E_{i0}$ and $q^E_{i0}$.

This way of merging sets $Q^E_{\Sigma_0}$ is also used for the union and concatenation operations. So:

\[ Q^E = \bigcup_{i=1}^{n} Q^E_i \cup \{q^E_{i0}|a \in \Sigma_0\} \]
\[ \Delta^E = \{a \rightarrow q^E_{i0}, a \in \Sigma_0, i = 1 \ldots n\} \]
\[ \cup \{a \rightarrow q^E_{i0}|a \in \Sigma_0 \} \cup \{g(q^E_{i1} \cdot q^E_{i2} \ldots q^E_{in}) = q^E_{ij}\} \cup \{e(q^E_{i1} \cdot q^E_{i2} \ldots q^E_{in}) = q^E_{ij}\} \]

Automaton of Union $E = F + G$: The union of two Thompson’s automata $F$ and $G$ is built by adding a new final state connecting the final states of the two automata $q^E_f$ and $q^G_f$ with $\varepsilon$-transitions and merging $Q^E_{\Sigma_0}$ sets. See graphical representation for $Th_E$ in Figure 5.

Formally:

\[ Q^E = Q^F \cup Q^G \cup \{q^F_f\} \cup \{q^G_f|a \in \Sigma_0\} \]
\[ \Delta^E = \{a \rightarrow q^F_f, a \in \Sigma_0\} \cup \{\Delta^G\} \cup \{a \rightarrow q^G_f, a \in \Sigma_0\} \]
\[ \cup \{e(q^F_f) = q^G_f|a \in \Sigma_0\} \cup \{e(q^G_f) = q^F_f|a \in \Sigma_0\} \cup \{e(q^F_f) \rightarrow q^F_f\} \]
\[ \cup \{e(q^G_f) \rightarrow q^G_f\} \cup \{a \rightarrow q^G_f|a \in \Sigma_0\} \]

Automaton of Concatenation $E = F \cdot G$: The construction of the concatenation automaton (Figure 6) is a little bit different. We remove the transition $c \rightarrow q_c$ (for $c$ the concatenation symbol) from $\Delta^F$, then we add a $\varepsilon$-transition from the final state of $G$ to $q_c$. Finally, we merge sets $Q^E_{\Sigma_0}$ of $F$ and $G$.

Then:

\[ Q^E = Q^F \cup Q^G \cup \{q^F_f\} \cup \{q^G_f|a \in \Sigma_0\} \]
\[ \Delta^E = \{\Delta^F\} \cup \{a \rightarrow q^F_f, a \in \Sigma_0\} \cup \{\Delta^G\} \cup \{a \rightarrow q^G_f, a \in \Sigma_0\} \]
\[ \cup \{a \rightarrow q^F_f|a \in \Sigma_0\} \cup \{e(q^F_f) \rightarrow q^G_f|a \in \Sigma_0\} \cup \{e(q^F_f) \rightarrow q^F_f\} \]
\[ \cup \{e(q^G_f) \rightarrow q^G_f\} \cup \{e(q^F_f) \rightarrow q^F_f\} \]

Automaton of Closure $E = F^*$: For the construction of Thompson’s automaton of the closure operation ($*$), we extend the basic automaton with three $\varepsilon$-transitions like in Figure 7:

- a transition from the final state of $F$ to the state representing the concatenation symbol in $F$,
- a transition from the state representing the concatenation symbol in $E$ to the final state of $E$,
- and another transition from the final state of $F$ to the final state of $E$.

Formally:

\[ Q^E = Q^F \cup \{q^F_f\} \cup \{q^F_f|a \in \Sigma_0\} \]
\[ \Delta^E = \{\Delta^F\} \cup \{a \rightarrow q^F_f, a \in \Sigma_0\} \]
\[ \cup \{a \rightarrow q^F_f|a \in \Sigma_0\} \cup \{e(q^F_f) \rightarrow q^F_f\} \cup \{e(q^F_f) \rightarrow q^F_f\} \cup \{e(q^F_f) \rightarrow q^F_f\} \]

Thompson’s Automaton for Trees
4.2 Validity of the Construction

We should prove the equivalence between the regular language of the constructed Thompson’s automaton and the regular language recognized by $E$.

**Definition 1.** Let $t$ be a tree, we define the function $\varepsilon$-closure$(t)$ that removes $\varepsilon$-nodes from $t$ as follows:

- $\varepsilon$-closure$(a) = a$ for each $a \in \Sigma_0$
- $\varepsilon$-closure$(t) = t$
- $\varepsilon$-closure$(g(t_1,t_2,...,t_n)) = g(\varepsilon$-closure$(t_1), \varepsilon$-closure$(t_2), ..., \varepsilon$-closure$(t_n))$

From the definition of $\varepsilon$-closure$(t)$ we can deduce the following properties:

**Property 1.** For each trees two $t_1, t_2$, we have $\varepsilon$-closure$(t_1 \cdot t_2) = \{ \varepsilon$-closure$(t_1) \cdot \varepsilon$-closure$(t_2) \}$.

This property can be extended to sets of trees.

**Property 2.** $\varepsilon$-closure$(t, \{ t_1, t_2, ..., t_k \}) = \{ \varepsilon$-closure$(t) \} \cdot \{ \varepsilon$-closure$(t_1), \varepsilon$-closure$(t_2), ..., \varepsilon$-closure$(t_k) \}$

**Theorem 1.** Let $E$ be a regular tree expression, $Th_E$ is Thompson’s automaton created for $E$. $\mathcal{L}(Th_E)$ is the regular language recognizing $E$ and $\mathcal{L}(Th_E)$ is the regular language recognized by $Th_E$.

Then, we have $\varepsilon$-closure$(\mathcal{L}(Th_E)) \equiv \mathcal{L}(Th_E)$.

In order to prove this theorem, we prove the two next lemmas (in Appendices A and B).

**Lemma 1.** For each $t \in \mathcal{L}(Th_E)$, there exists a tree $t' \in \mathcal{L}(Th_E)$ so that $\varepsilon$-closure$(t') = t$.

**Lemma 2.** If $t \in \mathcal{L}(Th_E)$, then $\varepsilon$-closure$(t) \in \mathcal{L}(Th_E)$.
We now consider features and size of our construction of a tree automaton from a regular tree expression. The number of states and transitions generated for each construction in section 4.1 is around \(|E|\), since we can consider each regular tree expression as a tree where leaves represent symbols of the alphabet and internal nodes represent regular tree operators (arity, union, concatenation and closure).

5 Conclusion

In this paper, we have presented a method for the construction of Thompson’s automaton from a regular tree expression: we have proposed a general form of the automaton to facilitate the operations of closure and concatenation. Proofs of the equivalence between the automaton generated and the original regular expression are provided for the different operations on the automaton. Future work will be focused on tree pattern matching using this automaton.

References

Appendix A: Proof of Lemma 1

The proof is accomplished by induction on the construction of the tree automaton.

Leaf Tree

\( E = a \). This case is obvious because: \( \mathcal{L}(\text{Th}_E) = \{ a \} \), \( [E] = \{ a \} \), so \( \mathcal{L}(\text{Th}_E) \equiv [E] \), so for all \( t \in [E] \), there exists \( t' \in \mathcal{L}(\text{Th}_E) \) such as \( \varepsilon\text{-closure}(t') = t \).

Arity

\( E = g(E_1, E_2, ..., E_n) \), where \( n \) is the arity of the function \( f \).

Let \( t \in [E] \), so \( t = g(e_1, e_2, ..., e_n) \), such as \( e_1 \in [E_1], e_2 \in [E_2], ..., e_n \in [E_n] \).

According to the induction hypothesis, there exists \( t'_i \in \mathcal{L}(\text{Th}_E) \) such as
\[
\varepsilon\text{-closure}(t'_i) = t_i \text{ with } i = 1...n.
\]
Let us construct a term \( t' = g(t'_1, t'_2, ..., t'_n) \) and replacing each symbol \( x \) of arity 0 by \( \varepsilon(x) \). This means that
\[
t' = (g(t'_1, t'_2, ..., t'_n)) \sigma(\varepsilon(a), ..., \varepsilon(c))...
\]
We now show that \( t' \in \mathcal{L}(\text{Th}_E) \) and that \( \varepsilon\text{-closure}(t') = t \).

According to the construction of Thompson’s automaton for the arity, transition \( a \rightarrow q^E_0 \) in the path of recognizing \( t'_i \) in \( \text{Th}_E \) is replaced by \( a \rightarrow q^E_0 \) and \( \varepsilon(q^E_0) \rightarrow q^E_i \) for each \( i = 1...n \) and \( a \in \Sigma_0 \).

As we have \( \Delta^E \subset \Delta^F \) and \( t'_i \in \mathcal{L}(\text{Th}_E) \), it means \( \sigma^E(t'_i) = q^E_i \).

So: \( \sigma^E(t'_i) = q^E_i, i = 1...n \).

Moreover we have \( g(q^E_1, q^E_2, ..., q^E_n) \rightarrow q^F_i \in \Delta^F \), so: \( \sigma^F(t') = q^F_i \).

Therefore \( t' \) is recognized by \( \mathcal{L}(\text{Th}_E) \).

On the other hand we have:
\[
\varepsilon\text{-closure}(t') = \varepsilon\text{-closure}(g(t'_1, t'_2, ..., t'_n), \varepsilon(a), ..., \varepsilon(c))...
\]
\[
= \varepsilon\text{-closure}(g(t'_1, t'_2, ..., t'_n)) \sigma(\varepsilon(\varepsilon(a))), ..., \varepsilon(\varepsilon(\varepsilon(c)))...
\]
\[
= \varepsilon\text{-closure}(g(t'_1, t'_2, ..., t'_n)) \sigma(a, ..., c).
\]
\[
= g(\varepsilon\text{-closure}(t'_1), \varepsilon\text{-closure}(t'_2), ..., \varepsilon\text{-closure}(t'_n))
\]
\[
= g(t'_1, t'_2, ..., t'_n) = t.
\]
Thus there exists \( t' \in \mathcal{L}(\text{Th}_E) \) such that \( \varepsilon\text{-closure}(t') = t \).

Union

\( E = F + G \). We have \( [E] = [F] \cup [G] \).

Let \( t \in [E] \). Without loss of generality, we suppose that \( t \in [F] \).

According to the induction hypothesis, there exists \( t'_E \in \mathcal{L}(\text{Th}_E) \): \( \varepsilon\text{-closure}(t'_E) = t \).

Let us construct a term \( t' = \varepsilon(t'_E) \). Then we replace each symbol \( x \) of arity 0 by \( \varepsilon(x) \), that is to say:
\[
t' = (t'_E) \sigma(\varepsilon(a), ..., \varepsilon(c))...
\]
Let us show that \( t' \in \mathcal{L}(\text{Th}_E) \) and that \( \varepsilon\text{-closure}(t') = t \).
According to the construction of Thompson for the union, we replace each transition \( a \to q_a^F \) by \( a \to q_a^E \) and \( \varepsilon(q_a^E) \to q_a^E \) such as \( a \in \Sigma_0 \).
As \( \Delta^E \subset \Delta^F \) ans \( t'_i \in \mathcal{L}(\text{Th}_E) \), i.e. \( \sigma^*_E(t'_i) = q_f^E \).
So: \( \sigma^*_E(t'_i) = q_f^E \).

Moreover we have \( \varepsilon(q_f^E) \to q_f^E \in \Delta^E \), then: \( \sigma^*_E(t'_i) = q_f^E \), which means that \( t' \) is recognized by \( \mathcal{L}(\text{Th}_E) \).
Furthermore we have:

\[
\begin{align*}
\varepsilon\text{-closure}(t') &= \varepsilon\text{-closure}(\varepsilon(t'_i)_a\varepsilon(a)\ldots\varepsilon(c)\ldots) \\
                    &= \varepsilon\text{-closure}(t'_i)_a\ldots,_,c. \\
                    &= \varepsilon\text{-closure}(t'_i) = t.
\end{align*}
\]
Thus there exists \( t' \in \mathcal{L}(\text{Th}_E) \) such that \( \varepsilon\text{-closure}(t') = t. \)

\[
\square
\]

### Concatenation

\( E = F \cdot G \). Let \( t \in [E], t \in [F], t \in [G] \), it means \( t \in \{(t_F)_x\{t_1, \ldots, t_k\}\} \), such that \( t_i \in [G], i = 1 \ldots k \).

According to the induction hypothesis, there exists \( t'_1, t_1, \ldots, t_k \) with \( t'_i \in \mathcal{L}(\text{Th}_F) \) and \( t'_i \in \mathcal{L}(\text{Th}_G) \), \( i = 1 \ldots k \), such as \( \varepsilon\text{-closure}(t'_i) = t_F \) and \( \varepsilon\text{-closure}(t'_i) = t_k \).

Let us build two terms \( t''_i \) and \( t''_j \) by replacing each symbol \( x \) of arity 0 by \( \varepsilon(x) \), ie.

\[
\begin{align*}
t''_i &= (t'_i)_a\varepsilon(a)\ldots\varepsilon(c)\ldots \\
t''_j &= (t'_j)_a\varepsilon(a)\ldots\varepsilon(c)\ldots \\
t' &= \{\varepsilon(t''_i)_x\{t'_1, t'_2, \ldots, t'_k\}\}.
\end{align*}
\]

Let us prove that \( t' \in \mathcal{L}(\text{Th}_E) \).

According to the construction of Thompson’s automata of concatenation we replaced the transition \( a \to q_a^G \) dans \( \text{Th}_G \) by \( a \to q_a^E \) and \( \varepsilon(q_a^E) \to q_a^E \) in \( \text{Th}_E \) by \( a \in \Sigma_0 \).
And as we have \( \Delta^G \subset \Delta^E \) and \( \sigma^*_E(t'_i) = q_f^G \), then \( \sigma^*_E(t''_i) = q_f^G \).

According to the same construction, we replaced the transition \( a \to q_a^G \) by \( a \to q_a^E \) and \( \varepsilon(q_a^E) \to q_a^E \) for \( a \neq c \), where \( c \) is the symbol of concatenation.
If \( a = c \), this transition \( (c \to q_c^E) \) is replaced by \( \varepsilon(q_c^G) \to q_c^E \).

Since \( t'_i \) is recognized by \( \text{Th}_F \) by the induction hypothesis, then: \( \sigma^*_E(t'_i) = q_f^E \).
As we have \( \varepsilon(q_f^E) \to q_f^E \in \Delta^E \) for \( a \neq c \) and \( \varepsilon(q_f^G) \to q_f^G \in \Delta^E \) otherwise (if \( a = c \) and \( (\Delta^E \setminus \{c \to q_c^E\}) \subset \Delta^E \)), alors nous avons \( \sigma^*_E(t'') = q_f^F \).

Therefore \( \sigma^*_E(\varepsilon(t')) = q_f^G \) because \( \varepsilon(q_f^G) \to q_f^G \in \Delta^G \). So \( t' \) is recognized by \( \mathcal{L}(\text{Th}_E) \).
Thus there exists

For the case

\[
\epsilon\text{-closure}(\epsilon(t')) = \epsilon\text{-closure}(t')
\]

\[
\epsilon\text{-closure}(t') \in \{\epsilon\text{-closure}(\epsilon(t'_1), ..., \epsilon(t'_n))\}
\]

\[
\epsilon\text{-closure}(t'_1), ..., \epsilon\text{-closure}(t'_n)
\]

Let us replace terms \( t'_1 \) and \( t'_2 \) respectively by \( \epsilon\text{-closure}(t'_1, a, e, c) \) and \( \epsilon\text{-closure}(t'_2, a, e, c) \).

we will have: \( \epsilon\text{-closure}(t') \in \epsilon\text{-closure}(t'_1, ..., \epsilon\text{-closure}(t'_n)) \).

So \( t \in \{(t_F), (t_1, ..., t_k)\} \).

\[\Box\]

**Closure**

\( E = F^{ec} \). Let \( t \in [F^{ec}] \):

\[
t \in \begin{cases} [F^{0,c}] & \text{if } n = 0, \\ [F^{n,c}], n \in \mathbb{N}^* & \text{otherwise.} \end{cases}
\]

We will show that for all \( t \in F^{ec} \), there exists \( t' \in \mathcal{L}(\text{Th}_E) \), such as

\[\epsilon\text{-closure}(t') = t.\]

For the case \( n = 0 \), \( t \in [F^{0,c}] \), \( t^0 = c \), this case is obvious because we have:

\[\{c \rightarrow q^F_e, \epsilon(q^F_e) \rightarrow q^F_f\} \in \Delta^F.\]

Then \( t^0 = \epsilon(t^0) \).

So \( t^0 \) is recognized by the automaton \( \text{Th}_E \) and \( \epsilon\text{-closure}(t^0) = t^0 \).

For \( n = 1, t \in [F^{1,c}] \), \( t^1 = c, [F] \), we demonstrate that there exists

\[t^1 \in \mathcal{L}(\text{Th}_E) \text{ and } \epsilon\text{-closure}(t^1) = t^1.\]

\[t^1 \in \{c, t^1\} / t^1 \in [F] \text{ then } t^1 \in \{c, t^1, ..., t^1\}.\]

According to the induction hypothesis \( \{c, t^1\} \in \mathcal{L}(\text{Th}_F) \), it means \( \sigma^+_F(t^1) = q^F_f \). And depending on the construction of Thompson’s automata for the closure operation, we have \( \epsilon(q^F_f) \rightarrow q^F_f \in \Delta^F \), so \( \sigma^+_E(t^1) = q^F_f \). Therefore, \( t^1 \in \mathcal{L}(\text{Th}_E) \).

Furthermore, we have:

\[\epsilon\text{-closure}(t^1) \in \{\epsilon\text{-closure}(c, t^1, ..., t^1)\}\]

\[\epsilon\text{-closure}(c, \epsilon\text{-closure}(t^1, ..., t^1))\]

\[\epsilon\text{-closure}(t^1, ..., t^1)\]

For the case \( 1 < k \leq n \), the closure operation comes down to a concatenation. \( t^k = t^{k-1, c} [F] \).

Thus there exists \( t^k \in \mathcal{L}(\text{Th}_E) \) such that \( \epsilon\text{-closure}(t^k) = t^k \).
Appendix B: Proof of Lemma 2

We remind that proofs are accomplished by induction on the construction of the tree automaton.

Leaf tree

\[ E = a. \] This case is obvious because \( t = a \) and \( \varepsilon\text{-closure}(t) = [E] \).

Arity

\[ E = g(E_1, E_2, \ldots, E_n). \] We have \( t \in \mathcal{L}(\text{Th}_E) \) which means that \( t = g(t_1, t_2, \ldots, t_n) \).

According to the induction hypothesis, there exists \( t_1, t_2, \ldots, t_n \) such as:

\[ t_1 \in \mathcal{L}(\text{Th}_{E_1}) \] and \( \varepsilon\text{-closure}(t_1) \in [E_1] \).

\[ t_2 \in \mathcal{L}(\text{Th}_{E_2}) \] and \( \varepsilon\text{-closure}(t_2) \in [E_2] \).

\[ \vdots \]

\[ t_n \in \mathcal{L}(\text{Th}_{E_n}) \] and \( \varepsilon\text{-closure}(t_n) \in [E_n] \).

We have:

\[ g(\varepsilon\text{-closure}(t_1), \varepsilon\text{-closure}(t_2), \ldots, \varepsilon\text{-closure}(t_n)) \in [g(E_1, E_2, \ldots, E_n)]. \]

According to the definition 1 of the function \( \varepsilon\text{-closure}(t) \), we have:

\[ g(\varepsilon\text{-closure}(t_1), \varepsilon\text{-closure}(t_2), \ldots, \varepsilon\text{-closure}(t_n)) = \varepsilon\text{-closure}(g(t_1, t_2, \ldots, t_n)). \]

So, \( \varepsilon\text{-closure}(g(t_1, t_2, \ldots, t_n)) \in [g(E_1, E_2, \ldots, E_n)] \).

Therefore, \( \varepsilon\text{-closure}(g(t_1, t_2, \ldots, t_n)) \in [E] \).

Furthermore, we have: \( t = g(t_1, t_2, \ldots, t_n) \), then:

\[ \varepsilon\text{-closure}(t) \in [E]. \]

Union

\[ E = F + G. \] We have \( t \in \mathcal{L}(\text{Th}_E) \) which means that \( t = t_f \) or \( t = t_g \), where

\[ t_f \in \mathcal{L}(\text{Th}_F) \] and \( t_g \in \mathcal{L}(\text{Th}_G) \).

According to the induction hypothesis:

\[ t_f \in \mathcal{L}(\text{Th}_F) \] and \( \varepsilon\text{-closure}(t_f) \in [F] \).

\[ t_g \in \mathcal{L}(\text{Th}_G) \] and \( \varepsilon\text{-closure}(t_g) \in [G] \).

Without loss of generality, we suppose that \( t = t_f \).

We have \( \varepsilon\text{-closure}(t_f) \in [F] \), then \( \varepsilon\text{-closure}(t_f) \in [F + G] \).

So, \( \varepsilon\text{-closure}(t) \in [F + G] \).

As we have \( E = F + G \), then \( \varepsilon\text{-closure}(t) \in [E] \).

Concatenation

\[ E = F \cdot G. \] If \( t \in \mathcal{L}(\text{Th}_E) \) then \( t \) is of the form \( t_f \cdot t_g \).
According to the induction hypothesis:
$t_f \in \mathcal{L}(\text{Th}_F)$ et $\varepsilon$-closure$(t_f) \in [F]$.
$t_g \in \mathcal{L}(\text{Th}_G)$ et $\varepsilon$-closure$(t_g) \in [G]$.

We have $\varepsilon$-closure$(t_f) \varepsilon$-closure$(t_g) \in [F \varepsilon G]$.

According to the property 2, we have $\varepsilon$-closure$(t_f \cdot t_g) = \varepsilon$-closure$(t_f) \cdot \varepsilon$-closure$(t_g)$.

So, $\varepsilon$-closure$(t) \in [F \varepsilon G]$.

Then $\varepsilon$-closure$(t) \in [E]$. ☐

**Closure**

$E = F^\varepsilon$. If $t \in \mathcal{L}(\text{Th}_E)$, then $t$ is of the form: $t = t_f^n \cdot c$, $t_f \in [F]$ and

$$
t_f^n = \begin{cases} 
t_f^0 & = c, \\
t_f^1 & = (t_f^0 \cdot c)_t \\
t_f^n & = (t_f^{n-1} \cdot c)_t 
\end{cases} \text{ si } n \geq 2.
$$

According to the induction hypothesis, we have $t_f \in \mathcal{L}(\text{Th}_F)$ and $\varepsilon$-closure$(t_f) \in [F]$.

For the case $t = t_f^0 \cdot c$, we have $\varepsilon$-closure$(t_f) \varepsilon$-closure$(c) = c$.

We also have $c \in [F]$, then $c \in [F^0 \cdot c]$.

Therefore, $\varepsilon$-closure$(t) \in [E]$.

For $t = t_f^1 \cdot c = (t_f^0 \cdot c)_t = t_f$, we have $\varepsilon$-closure$(t) = \varepsilon$-closure$(t_f)$.

$\varepsilon$-closure$(t_f) \in [F]$ according to the induction hypothesis, then $\varepsilon$-closure$(t_f) \in [F^1 \cdot c]$, because $[F^1 \cdot c] = [F^0 \cdot c] \cup [F^0 \cdot F]$ according to the closure property of regular languages.

Therefore, $\varepsilon$-closure$(t) \in [E]$.

For $t_f^n$ with $n > 1$, closure is a special case of the concatenation, and this was proven before.

Then, for $t = t_f^n \cdot c$ and $E = F^\varepsilon$, $\varepsilon$-closure$(t) \in [E]$. ☐
On Computing Best Trees for Weighted Tree Automata

Johanna Björklund, Frank Drewes, Niklas Zechner
Department of Computing Science, Umeå University, 901 87 Umeå, Sweden
{johanna,drewes,zechner}@cs.umu.se

We generalise a search algorithm by Mohri and Riley from strings to trees. The original algorithm takes as input a weighted automaton \( M \) over the tropical semiring, together with an integer \( N \), and outputs \( N \) strings of minimal weight with respect to \( M \). In our setting, \( M \) defines a weighted tree language, and the output is a set of \( N \) trees with minimal weight. We prove that the algorithm is correct, and that its time complexity is a low polynomial in \( N \) and the relevant size parameters of \( M \).

1 Introduction

Tree automata are useful in natural language processing (NLP), not least to describe the derivation trees of context-free grammars in an automata-theoretic way. To allow analyses to be computed together with an associated confidence level or a probability, we can choose to equip transitions and final states with weights, i.e., to work with weighted tree automata (wta) [4]. This is convenient when there is a set of competing analyses to choose from, and we want to find an analysis that optimises some objective function \( f \). Huang and Chiang [5] observe that even when it is not tractable to compute \( f \) for every possible analysis, we may still obtain a satisfactory approximation by first ranking the candidate analyses according to a simpler function, such as can be computed by a wta, and then finding an \( N \)-best list \( a_1, \ldots, a_N \) according to this ranking, and finally optimising \( f \) over \( \{a_1, \ldots, a_N\} \). Examples include reranking the hypotheses produced by parsers or translation systems, where the reranking is based on auxiliary language models or evaluation scores orthogonal to the first round of analysis; see, e.g. [3, 8].

There are other situations in which an \( N \)-best analysis can be used for approximation. Suppose for instance that the analysis is computed by a cascade of computational modules, a common architecture for NLP systems [5]. Each module typically comes with its own objective function, and the goal is to optimise these jointly. Although it might not be possible to compute the full set of outputs from each module, we may again settle for the \( N \) best outputs from each module, and propagate them downstream. In their paper, Huang and Chiang provide several examples of this technique, including (i) joint parsing and semantic role labeling, and (ii) combined information extraction and coreference resolution.

In the majority of the above-mentioned applications, the weights represent probabilities and are as such taken from the interval of real values between zero and one. However, for the sake of numerical precision, negative log likelihoods are used in the actual computations, and the min operation is used to find the most likely analysis. This makes the min-plus semiring (or tropical semiring) \((\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)\) an appropriate structure for transition weights. Alternatively, the max-plus semiring \((\mathbb{R}_+ \cup \{-\infty\}, \max, +, -\infty, 0)\) may be used.

In this paper, we focus on the case where trees are associated with weights by means of a weighted tree automaton (wta) over the tropical semiring. Thus, the weight of a computation, called a run, is the sum of the weights of the rules applied, and the weight of a tree is the minimum of the set of all runs on that tree. Note that the latter is only relevant if the automaton is nondeterministic. In [5] Huang...
and Chiang give an $O(m + D \cdot N \log N)$ algorithm for (essentially) finding a set $S$ of $N$ best runs in an acyclic wta, where $m$ is the number of transitions and $D$ is the size of the largest run in $S$. However, as pointed out by Mohri and Riley [7], one would usually rather determine the $N$ best trees, because the trees correspond to the analyses and it is not very useful to obtain the same analysis twice in an $N$-best list just because it corresponds to several distinct runs of the nondeterministic automaton that implements the weight assignment. Unfortunately, determining the $N$ best trees is a harder problem. Part of the difficulty lies in the fact that weighted automata are not closed under determinisation. In fact, both in the string and in the tree case the set of weighted languages recognisable by deterministic weighted automata is a proper subset of those recognisable by nondeterministic weighted automata.

Mohri and Riley [7] solve the problem of finding the $N$ best strings, where the input is a weighted string automaton (wsa) over the tropical semiring (and the number $N$). To avoid computing redundant paths, they apply Dijkstra’s $N$-shortest paths algorithm to a determinised version of the input automaton. Their algorithm applies the determinisation algorithm under a lazy evaluation scheme to guarantee termination and keep the running time polynomial. We generalise this algorithm to weighted tree languages, while simplifying the technique by working directly with the input automaton rather than an on-the-fly determinisation. The frontier is no longer a set of paths, but rather a set of trees that are combined and recombined into new trees to drive the search. This increased dimensionality creates an efficiency problem which we solve by a pruning technique. Owing to space limitations, the proofs have been left out, but a detailed treatment is given in [1] and available as a technical report1.

2 Preliminaries

We write $\mathbb{N}$ for the set of non-negative integers and $\mathbb{R}_+$ for the set of non-negative reals; $\mathbb{R}_+^\infty$ denotes $\mathbb{R}_+ \cup \{\infty\}$. For $n \in \mathbb{N}$, $[n] = \{1, \ldots, n\}$. In particular, $[0] = \emptyset$. The number of elements of a (possibly infinite) set $S$ is written $|S|$. The empty string is denoted by $\lambda$.

The estimation of the running time of our algorithm contains the factor $\log r$, where $r$ is the maximum rank of symbols in the ranked alphabet considered (see below for the definitions). To avoid the technical problem that $\log 1 = 0$ we use the convention that, throughout this paper, $\log r$ abbreviates $\max(1, \log r)$.

For a set $A$, an $A$-labelled tree is a function $t : D \to A$ where $D \subseteq \mathbb{N}^*$ is such that, for every $v \in D$, there exists a $k \in \mathbb{N}$ with $\{i \in \mathbb{N} \mid v i \in D\} = [k]$. We call $D$ the domain of $t$ and denote it by $\text{dom}(t)$. An element $v$ of $\text{dom}(t)$ is called a node of $t$, and $k$ is the rank of $v$. The subtree of $t \in T_2$ rooted at $v$ is the tree $t/v$ defined by $\text{dom}(t/v) = \{u \in \mathbb{N}^* \mid vu \in \text{dom}(t)\}$ and $t/v(u) = t(vu)$ for every $u \in \mathbb{N}^*$. If $t(\lambda) = f$ and $t/i = t_i$ for all $i \in [k]$, where $k$ is the rank of $\lambda$ in $t$, then we denote $t$ by $f[t_1, \ldots, t_k]$. If $k = 0$, then $f[]$ is usually abbreviated as $f$. In other words, a tree $t$ with domain $\{\lambda\}$ is identified with $t(\lambda)$.

A ranked alphabet is a finite set of symbols $\Sigma = \bigcup_{k \in \mathbb{N}} \Sigma(k)$, partitioned into pairwise disjoint subsets $\Sigma(k)$. For every $k \in \mathbb{N}$ and $f \in \Sigma(k)$, the rank of $f$ is $\text{rank}(f) = k$. The set $T_2$ of all trees over $\Sigma$ contains all $\Sigma$-labelled trees $t$ such that the rank of every $v \in \text{dom}(t)$ coincides with the rank of $t(v)$. For a set $T$ of trees we denote by $T(\Sigma)$ the set of all trees $f[t_1, \ldots, t_k]$ such that $f \in \Sigma(k)$ and $t_1, \ldots, t_k \in T$.

Let $\Sigma$ be a ranked alphabet and let $\Box \not\in \Sigma$ be a special symbol of rank 0. The set of contexts over $\Sigma$ is the set $\Sigma_\Sigma$ consisting of all $c \in T_{\Sigma, \Sigma(\Box)}$ such that there is exactly one $v \in \text{dom}(c)$ with $c(v) = \Box$. The substitution of a tree $t$ for $\Box$ in $c$ is defined as usual, and is denoted by $c[t]$.

A weighted tree language over the tropical semiring is a mapping $L : T_2 \to \mathbb{R}_+^\infty$, where $\Sigma$ is a ranked alphabet. Such languages can be specified by the use of so-called weighted tree automata (wta), of which there exist variants with final weights and with final states. As shown by Borchardt [2] these two variants

---

1See http://www8.cs.umu.se/research/uminf/index.cgi?year=2014&number=22
are equivalent, and going from final weights to final states only requires a single additional state (which becomes the unique final state) and, in the worst case, twice as many transitions. This means that all results shown in this paper, including the running time estimations, hold for both types of wta.

Formally, a weighted tree automaton is a system $M = (Q, \Sigma, \delta, Q_f)$ where $Q$ is a finite set of states which are considered as symbols of rank 0; $\Sigma$ is a ranked alphabet of input symbols disjoint with $Q$; $\delta : \Sigma(Q) \times Q \rightarrow \mathbb{R}_+$ is the transition function; and $Q_f \subseteq Q$ is the set of final states. Note that the transition function $\delta$ can be specified as the set of all transition rules $f[q_1, \ldots, q_k] \rightarrow q$ such that $\delta(f[q_1, \ldots, q_k], q) = w \neq \infty$. In particular, transition rules whose weight is $\infty$ are not represented explicitly.

For convenience, we define the behaviour of $M$ on trees in $T_{\Sigma,Q}$ as opposed to just $T_{\Sigma}$, where states are considered to be symbols of rank 0. The set of runs of $M$ on $t \in T_{\Sigma,Q}$ is the set of all $Q$-labelled trees $\pi : dom(t) \rightarrow Q$ such that $\pi(v) = t(v)$ for all $v \in dom(t)$ with $t(v) \in Q$. A run $\pi$ is accepting if $\pi(\lambda) \in Q_f$. The weight of a run $\pi$ on a tree $t = f[t_1, \ldots, t_k]$ is defined as

$$w(\pi) = \sum_{v \in dom(t), t(v) \in \Sigma(q)} \delta(t(v)[\pi(v_1) \cdots \pi(v_k)], \pi(v)).$$

Now, let $M(t) = \min \{w(\pi) | \pi$ is an accepting run of $M$ on $t\}$ for every tree $t \in T_{\Sigma,Q}$. This defines the weighted tree language $\mathcal{W}_M : T_{\Sigma} \rightarrow \mathbb{R}_+$ recognised by $M$, namely $\mathcal{W}_M(t) = M(t)$ for all $t \in T_{\Sigma}$.

The problem we are concerned with in this paper is to compute $N$ minimal trees such that, according to $M$, there are no trees outside this set with smaller weight. For $N \in \mathbb{N}$, an acceptable solution is a set $T = \{t_1, \ldots, t_N\} \subseteq T_{\Sigma}$ such that $M(t_i) \leq M(t)$ for all $i \in [N]$ and $t \in T_{\Sigma} \setminus T$. Similarly, for $N = \infty$, we seek an infinite set $T = \{t_1, t_2, \ldots\} \subseteq T_{\Sigma}$ with $M(t_i) \leq M(t)$ for all $i \geq 1$ and $t \in T_{\Sigma} \setminus T$.

## 3 The Algorithm

We now present our algorithm for computing $N$ minimal trees with respect to a given wta. This is done in two steps: First a basic version is developed, which is later turned into a more efficient one by means of a pruning strategy. Correctness and efficiency are studied in Section 4. Throughout the paper, let $M = (Q, \Sigma, \delta, Q_f)$ be the wta given as input to the search algorithm. The letters $m, n$, and $r$ denote the number $|\delta|$ of transition rules, the number $|Q|$ of states, and the maximum rank of symbols in $\Sigma$.

Our algorithm explores its search space recursively. The frontier of the explored part is organised as a priority queue. The algorithm iteratively selects a promising tree $t$ from the queue, considers $t$ for output, puts it into a set $T$ of explored trees, and finally expands the frontier by all trees in $\Sigma(T)$ which have at least one occurrence of $t$ as a direct subtree. For $t \in T \subseteq T_{\Sigma}$ this expansion is defined as

$$\text{expand}(T, t) = \{f[t_1, \ldots, t_k] \in \Sigma(T) | t_i = t \text{ for at least one } i \in [k]\}.$$

To define our algorithm, it is convenient to consider two wtas $M^0$ and $M_q$, for every $q \in Q$. The wta $M^0$ is simply given by $M^0 = (Q, \Sigma, \delta, \{q\})$, i.e. $q$ becomes the unique final state. The wta $M_q$ is given by $M_q = (Q, \Sigma \cup \{\square\}, \delta \cup \{\square \rightarrow q\}, Q_f)$. Note that $M_q(c) = M(c[q])$ for all $c \in \Sigma$ and $q \in Q$.

The priority of a tree $t$ in our queue is primarily decided by the minimal value of $M(c[t])$, where $c$ ranges over all possible contexts. To determine this, we compute for every $q \in Q$ the minimal value of $M_q(c) + M^0(t)$. Since $M^0$ denotes the wta obtained from $M$ by taking $q$ as the unique final state, $M^0(t)$ is the minimal weight of all runs on $t$ whose root state is $q$. Since $M_q(c)$ is independent of $t$, a $c$ that minimises it can be calculated in advance using, e.g., Knuth’s extension of Dijkstra’s algorithm [6] (which, roughly speaking, computes the best derivation in a weighted context-free grammar). This yields the following lemma.
Lemma 1 A family of contexts \((c_q)_{q \in Q}\) such that \(M_q(c_q) = \min \{ M_q(c) \mid c \in C_{\Sigma} \}\) for each \(q \in Q\) can be computed in time \(O(\text{mr} \cdot (\log n + r))\).

In the rest of the paper, we frequently make use of the contexts \(c_q\), assuming that they have been computed for all \(q \in Q\). For a tree \(t\) in the frontier of our search space we are, intuitively, interested in the tree \(c[t]\) that has the least possible weight. Clearly, \(c\) can be assumed to be one of the contexts \(c_q\).

Thus, our aim has to be to determine the state \(q\) that minimises the weight of \(c_q[t]\).

Definition 1 (Optimal state) The mapping \(\text{optset}: T_{\Sigma} \rightarrow \text{pow}(Q)\), where \(\text{pow}(Q)\) is the powerset of \(Q\), is defined by

\[
\text{optset}(t) = \{ q \in Q \mid M_q(c_q) + M^0(t) = \min_{c \in C_{\Sigma}} M(c[t]) \}.
\]

In addition, let \(\text{opt}(t)\) denote an arbitrary but fixed element of \(\text{optset}(t)\), for every \(t \in T_{\Sigma}\).

We can now give our basic algorithm, which we formulate only for \(\text{wta}\) computing \textit{monotone} weighted tree languages. Here, a weighted tree language \(L\) is called \textit{monotone} if, for all trees \(t \in T_{\Sigma}\) and all \(c \in C_{\Sigma} \setminus \{\square\}\), \(L(t) \neq \emptyset\) implies \(L(c[t]) \geq L(t)\). To see that this does not diminish the usefulness of the algorithm, notice that an arbitrary input \(\text{wta}\) \(M\) can be made monotone as follows: Introduce a new symbol \(\text{out}\) of rank 1 and turn \(M\) into \(M'\) such that \(M'(t) = \infty\) and \(M'(\text{out}[t]) = M(t)\) for all \(t \in T_{\Sigma}\). This can easily be achieved by adding a new state \(q_t\), which becomes the unique final state, and transitions \(\text{out}[^{\text{q}}]_t \rightarrow q_t\) for \(q \in Q\). Then \(M'\) is monotone and if \(\text{out}[t_1], \ldots, \text{out}[t_N]\) are \(N\) trees of minimal weight with respect to \(M'\), then \(t_1, \ldots, t_N\) are minimal with respect to \(M\).

Our basic algorithm is presented in Algorithm 1. It maintains three data structures: \(T\) is a set of trees that represents the explored search space, \(K\) is a priority queue of trees in \(\Sigma(T)\), and \(C\) is a table containing the value \(M^0(t)\), for all \(q \in Q\) and \(t \in T \cup K\). The table \(C\) can easily be updated whenever new trees are added to \(K\). The priority order \(\leq_K\) of \(K\) is given by

\[ t <_K t' \Rightarrow \Delta(t) < \Delta(t') \text{ or } \Delta(t) = \Delta(t') \text{ and } t <_{\text{lex}} t' \]

where \(\Delta(s) = M(c_{\text{opt}(t)}[s])\) for all \(s \in T_{\Sigma}\).

Here, \(<_{\text{lex}}\) is any lexiographical order that orders trees first by size and then lexiographically. Note that the output condition in Line 8 cannot be replaced by the more intuitive \(\Delta(t) < \infty\) because it has to cover the case where \(\Delta(t) = \infty\) (which happens if there are fewer than \(N\) trees of finite weight).

Unfortunately, Algorithm 1 builds a large number of trees and is thus not very efficient. Therefore, we now give a more efficient version that works by repeatedly pruning the priority queue.

The idea of the pruning step is that a tree \(s\) can be discarded from the queue if we already have, for every state \(q \in \text{optset}(s)\), at least \(N\) other trees \(t <_K s\) such that \(q \in \text{optset}(t)\). Intuitively, in this case we have sufficiently many good alternatives to \(s\) in the formation of a set of minimal trees, so that \(s\) will not be needed. A polynomial runtime is thus obtained through the addition of a new procedure \textit{Prune} (see Algorithm 2). In Algorithm 1, we replace Line 3 by \(\text{Prune}(T, \text{enqueue}(K, \Sigma_0))\), and Line 12 by \(\text{Prune}(T, \text{enqueue}(K, \text{expand}(T, t)))\), thereby obtaining Algorithm 3 \textit{BestTrees} (not listed explicitly, again due to space limitations).

4 Correctness and Efficiency

Let us now establish the correctness of Algorithms 1 and 3, and then study the efficiency of the latter. For this, we assume that \(\Sigma \neq \Sigma_{(q)}\), so that \(T_{\Sigma}\) is infinite and hence \(N\) trees of minimal weight can always
Throughout this section we will write $N$ to be the number of active trees among $t$ in $T$. Let $l$ be such that $t \in T$ and consider the execution of BestTreesBasic, BestTreesBasic, BestTreesBasic, and terminates within $O(m)$ steps in this case. Throughout this section we will write $BestTreesBasic(M,N) = t_1,t_2,...,t_l$ or $BestTreesBasic(M,N) = t_1,t_2,...$ (and similarly for BestTrees) if running Algorithm 1 with the inputs $M$ and $N$ results in the (finite or infinite) sequence $t_1,t_2,...,t_l$ or $t_1,t_2,...$ of output trees.

Using the following simple lemma, we can prove the correctness of Algorithm 1.

**Lemma 2** Algorithm 1 never dequeues the same tree twice. Furthermore, if Algorithm 1 dequeues a tree $t$ in $T$, then it has previously dequeued all trees $s$ in $T$ such that $s <_K t$. In particular, if a tree in $t$ in $T$ is dequeued, then all trees $s$ in $T$ with $\Delta(s) < \Delta(t)$ have been dequeued earlier.

**Theorem 1 (Correctness of Alg. 1)** For all $N \in \mathbb{N}$, BestTreesBasic($M,N$) terminates and returns $N$ trees of minimal weight according to the wta $M$. Moreover, BestTreesBasic($M,\infty$) = $t_1,t_2,...$ consists of pairwise distinct trees such that, for each $i \in \mathbb{N}$ and every tree $t \in T \setminus \{t_1,...,t_l\}$, $M(t) \geq M(t_i)$.

In the following, let us say that a tree $s \in T$ is **discarded** in a run of BestTrees($M,N$) if it, at some stage, is considered in Line 2 of Algorithm 2, fulfills the pruning condition in Line 3, and is consequently removed from the queue in Line 4. Further, a tree $t \in T$ is **active** (with respect to the considered run of BestTrees($M,N$)) if it contains no discarded subtrees.

**Lemma 3** Let BestTreesBasic($M,\infty$) = $t_1,t_2,...$ and consider the execution of BestTrees($M,N$) for some $N > 0$. Let $l \in \mathbb{N} \cup \{\infty\}$ be the number of active trees among $t_1,t_2,...$, and let $i_j$ be such that $t_{i_j}$ is the $j$th active tree in $t_1,t_2,...$, for all $j \leq l$. Then BestTrees($M,N$) = $t_1,t_2,...,t_{\min(l,N)}$.
Using this, the correctness of Algorithm 3 is established.

**Theorem 2 (Correctness of Alg. 3)** For all $N \in \mathbb{N}$, BestTrees$(M, N)$ terminates and returns $N$ trees of minimal weight according to the input wta $M$. Moreover, BestTrees$(M, \infty) = t_1, t_2, \ldots$ consists of pairwise distinct trees such that, for each $i \in \mathbb{N}$ and every tree $t \in T \setminus \{t_1, \ldots, t_i\}$, $M(t) \geq M(t_i)$.

Let us now discuss the worst-case efficiency of BestTrees. A consequence of the pruning is that $T$ can only grow to contain $N \cdot n$ trees, since at this point, the pruning will discard everything that is left in the queue. Since each execution of the ‘while’ loop increases the size of $T$, the body of the ‘while’ loop in BestTrees is executed at most $N \cdot n$ times. Using Lemma 1 together with Lemma 4 below, as well as the fact that the main loop of Algorithm 3 is executed at most $Nn$ times, we obtain Theorem 3.

**Lemma 4** Prune$(K, \text{Expand}(T, t))$ is computable in time $O(\max(m \cdot (Nr + r \log r + N \log N), Nn^2))$.

**Theorem 3** BestTrees$(M, N)$ runs in time $O(\max(Nmn \cdot (Nr + r \log r + N \log N), N^2 n^3, mr^2))$.

It may be worthwhile to notice that the set $T$ of Algorithm 3 is subtree closed, meaning that $t_1, \ldots, t_k \in T$ for every tree $f[t_1, \ldots, t_k] \in T$. Since all output trees of Algorithm 3 are in $T$, this means that the output of Algorithm 3 can be represented as a packed forest with $|T|$ nodes, i.e., of size $\leq N \cdot n$.

## 5 Conclusion and future work

Future work includes the implementation and integration of the algorithm into an open-source library for formal tree languages. On the theoretical side, we are interested in seeing further generalisations of the search algorithm, for example, from trees to directed acyclic graphs, or from the tropical semiring to some encompassing family of extremal semirings.

### References


Construction of Tree Automata from a Regular Tree Expression

Ludovic Mignot Nadia Ouali Sebti Djelloul Ziadi
Laboratoire LITIS - EA 4108, Université de Rouen, Avenue de l’Université 76801 Saint-Étienne-du-Rouvray Cedex, France {ludovic.mignot,nadia.ouali-sebti,djelloul.ziadi}@univ-rouen.fr

There exist several methods of computing an automaton recognizing the language denoted by a given regular expression: non exhaustively. The position automaton $P$ due to Glushkov, the c-continuation automaton $C$ due to Champarnaud and Ziadi, the follow automaton $F$ due to Ilie and Yu and the equation automaton $E$ due to Antimirov. It has been shown that $P$ and $C$ are isomorphic and that $E$ (resp. $F$) is a quotient of $C$ (resp. of $P$). In this paper, we define from a given regular tree expression the $k$-position tree automaton $P$ and the follow tree automaton $F$. Using the definition of the equation tree automaton $E$ of Kuske and Meinecke and our previously defined $k$-C-continuation tree automaton $C$, we show that the previous morphic relations are still valid on regular tree expressions.

1 Introduction

Regular expressions are used in numerous domains of applications in computer science. They are an easy and compact way to represent potentially infinite regular languages, that are well-studied objects leading to efficient decision problems. The first approach of the computation of an automaton from regular expression is to determine particular properties over the syntactic structure of the regular expression $E$. Glushkov [7] proposed the computation of an automaton with $(n+1)$-states. Ilie and Yu showed in [8] how to reduce it by merging similar states. Another method is to compute the transition function of the automaton as follows. Basically, it is a computation that tries to determine what words $w$ can be accepted after reading a prefix $w$. The first author that introduced such a process is Brzozowski [2]. He showed how to compute a regular expression denoting $w^{-1}(L(E))$ from the expression $E$: this expression, denoted by $d_w(E)$, is called the derivative of $E$ with respect to $w$. Furthermore, the set of dissimilar derivatives, combined with reduction according to associativity, commutativity and idempotence of the sum, is finite and can lead to the computation of a deterministic finite automaton. Antimirov [1] computed the partial derivatives. These so-called derived terms produce the equation automaton. Finally, by deriving expressions after having them indexed, Champarnaud and Ziadi [4] computed the c-continuation automaton. The different morphic links between these four automata have been studied too: Ilie and Yu showed that the follow automaton is a quotient of the position automaton; Champarnaud and Ziadi proved that the position automaton and the c-continuation automaton are isomorphic and that the equation automaton is a quotient of the position automaton. Finally, using a join of the two previously defined quotients, Garcia et al. presented in [6] an automaton that is smaller than both the follow and the equation automata.

In this paper, we recall the study of these already known morphic links between different tree automata. We recall two tree automata constructions, the $k$-position automaton [14] and the follow automaton [14], and we recall their morphic links with two other already known automata constructions,

---

*D. Ziadi was supported by the MESRS - Algeria under Project 8/U03/7015.

Submitted to:
TTAFT 2015
the equation automaton of Kuske and Meinecke [10] and our \( k \)-C-continuation automaton [12, 13, 14]. Notice that a position automaton and a reduced automaton have already been defined in [11]. However, they are not isomorphic with the automata we define in this paper. This study is motivated by the development of a library of functions for handling rational kernels [5] in the case of trees. The first problem consists in converting a regular tree expression into a tree automaton. In Section 2, we recall the definitions of \( k \)-position automaton, follow automaton, equation automaton and of the \( k \)-C-continuation automaton; we also present the morphic links between these four automata in this section. It is proved that there is no morphic link between the follow automaton and the equation automaton. Moreover, we recall the computation of the Garcia et al. equivalence leading to a smaller automaton in Section 3. Then, in Section 4 we give the algorithms and the complexity of the computation of different tree automata. Finally, the different results described in this paper are given in the conclusion.

2 Tree Automata from Regular Expressions

In the following we use the definitions of a ranked alphabet, a tree, a finite tree automaton, a tree substitution, a c-product, an iterated c-product, a c-closure, a regular tree expression and its denoted language, a set of positions, a quotient of a tree automaton and a mapping \( h \) defined in [12, 13, 14]. A regular expression \( E \) is linear if every symbol of rank greater than 1 appears at most once in \( E \).

In this section, we show how to compute from a regular expression \( E \) several tree automata accepting \([E]_f\): we introduce two new constructions, the \( k \)-position automaton and the follow automaton of \( E \), and then we recall two already-known constructions, the equation [10] and the \( k \)-C-continuation [12] automata. Regular languages defined over the ranked alphabet \( \Sigma \) are exactly the languages denoted by a regular expression on \( \Sigma \). In what follows we only consider expressions without 0 or reduced to 0. The set of symbols in \( \Sigma \) that appear in an expression \( F \) is denoted by \( \text{First}(F) \).

2.1 The \( k \)-Position Tree Automaton

In this section, we show how to compute the \( k \)-position automaton of an expression \( E \), recognizing \([E]_f\). This is an extension of the well-known position automaton [7] for regular word expressions. In what follows, for any two trees \( s \) and \( t \), we denote by \( s \preceq t \) the relation “\( s \) is a subtree of \( t \)”. Let \( t = f(t_1, \ldots, t_n) \) be a tree. We denote by root(\( t \)) the root of \( t \), by \( k \)-child(\( t \)) the label of the \( k \)th child of \( f \) in \( t \), if it is the root of \( t_k \) if it exists, and by \( \text{Leaves}(t) \) the set of the leaves of \( t \), i.e. \( \{ s \in \Sigma_0 \mid s \preceq t \} \). Let \( E \) be a regular expression, \( 1 \leq k \leq m \) be two integers and \( f \) be a symbol in \( \Sigma_m \). The set \( \text{First}(E) \) is the subset of \( \Sigma \) defined by \( \{ \text{root}(t) \in \Sigma \mid t \in [E] \} \). The set \( \text{Follow}(E, f, k) \) is the subset of \( \Sigma \) defined by \( \{ g \in \Sigma \mid \exists t \in [E], \exists s \preceq t, \text{root}(s) = f, k \text{-child}(s) = g \} \). The set \( \text{Last}(E) \) is the subset of \( \Sigma \) defined by \( \text{Last}(E) = \bigcup_{c \in [E]} \text{Leaves}(t) \). Let us first show that the position functions First and Follow are inductively computable.

Let \( E \) be linear. The set \( \text{First}(E) \) can be computed as follows:

\[
\text{First}(0) = \emptyset, \text{First}(a) = \{ a \}, \text{First}(f(E_1, \ldots, E_m)) = \{ f \}
\]

\[
\text{First}(E_1 + E_2) = \text{First}(E_1) \cup \text{First}(E_2), \text{First}(E_1 \cdot c) = \text{First}(E_1) \cup \{ c \}
\]

\[
\text{First}(E_1 \cdot E_2) = \begin{cases} 
\text{First}(E_1) \setminus \{ c \} \cup \text{First}(E_2) & \text{if } c \in [E_1], \\
\text{First}(E_1) & \text{otherwise}.
\end{cases}
\]

Let \( 1 \leq k \leq m \) be two integers and \( f \) be a symbol in \( \Sigma_m \). The set of symbols \( \text{Follow}(E, f, k) \) can be computed inductively as follows:

\[
\text{Follow}(0, f, k) = \emptyset, \text{Follow}(a, f, k) = \emptyset
\]
derivatives are no longer sets of expressions, but sets of tuples of expressions.

In order to compute from E a tree automaton recognizing L(E), we have to compute the first and follow sets of E.

2.3 The Equation Tree Automaton

Theorem 2.

Proposition 3.

Proposition 1.

Let E be linear. The relation \( \sim \) is the largest similarity relation over \( \mathcal{L} \).

Proposition 2. Let E be linear. Then \( \mathcal{L}(\mathcal{F}_E) = [E] \).

2.2 The Follow Tree Automaton

In this section, we define the follow tree automaton which is a generalisation of the Follow automaton introduced by L. Ilie and S. Yu in [8] and that it is a quotient of the k-position automaton. Notice that in this automaton, states are no longer positions, but sets of positions and that we extend the definition of the functions \( \text{First} \) and \( \text{Follow} \) to the position \( q \).

The two functions \( \text{First} \) and \( \text{Follow} \) are sufficient to compute the \( k \)-position tree automaton of E. The \( k \)-position automaton \( \mathcal{F}_E \) is the automaton \( (Q, \Sigma, Q_T, \Delta) \) defined by

\[
Q = \{ f^k \mid f \in \Sigma_m \land 1 \leq k \leq m \} \cup \{ e^1 \} \quad \text{with} \quad e^1 \text{ a new symbol not in } \Sigma, \quad Q_T = \{ e^1 \}
\]

\[
\Delta = \{ (f^k, g^l, \ldots, g^n) \mid f \in \Sigma_m \land k \leq m \land g \in \Sigma_n \land g \in \text{Follow}(E, f, k) \}
\]

The \( \sim \) relation is defined as follows:

\[
\mathfrak{Q} = \{ \text{First}(E) \} \cup \bigcup_{f \in \Sigma_m} \{ \text{Follow}(E, f, k) \mid 1 \leq k \leq m \}, \quad Q_T = \{ \text{First}(E) \}
\]

\[
\Delta = \{ (\text{Follow}(E, g, l), f, \text{Follow}(E, f, 1), \ldots, \text{Follow}(E, f, m)) \mid f \in \Sigma_m \land f \in \text{Follow}(E, g, l) \land g \in \Sigma_m \land l \leq m \} \cup \{ \langle I, c \rangle \mid c \in I \land c \in \Sigma_0 \}
\]

Let us show that \( \mathcal{F}_E \) is a quotient of \( \mathcal{F}_E \) w.r.t. a similarity relation ; since this kind of quotient preserves the language, this method is consequently a proof of the fact that the language denoted by E is recognized by \( \mathcal{F}_E \).

A similarity relation over an automaton \( A = (Q, \Sigma, Q_T, \Delta) \) is an equivalence relation \( \sim \) over \( Q \) such that for any two states \( q \) and \( q' \) in \( Q \), \( q \sim q' \) if \( \forall f \in \Sigma_n, \forall (q_1, \ldots, q_n) \in \Delta \Rightarrow (q', f, q_1, \ldots, q_n) \in \Delta \). In other words, two similar states admit the same predecessors w.r.t. any symbol.

Proposition 1. Let \( \mathfrak{A} \) be an automaton and \( \sim \) be a similarity relation over \( \mathfrak{A} \). Then \( \mathcal{L}(\mathfrak{A}/\sim) = \mathcal{L}(\mathfrak{A}) \).

Proposition 2. Let E be linear. The relation \( \sim \) is the largest similarity relation over \( \mathcal{F}_E \).

Proposition 3. Let E be linear. The finite tree automaton \( \mathcal{F}_E/\sim \) is isomorphic to \( \mathcal{F}_E \).

2.3 The Equation Tree Automaton

In [10], Kuske and Meinecke extend the notion of word partial derivatives [1] to tree partial derivatives in order to compute from E a tree automaton recognizing [E]. Due to the notion of ranked alphabet, partial derivatives are no longer sets of expressions, but sets of tuples of expressions.
Let $\mathcal{N} = (E_1, \ldots, E_m)$ be a tuple of regular expressions, $F$ and $G$ be some regular expressions and $c \in \Sigma_0$. Then $\mathcal{N} \cdot F$ is the tuple $(E_1 \cdot F, \ldots, E_m \cdot F)$. For a set $\mathcal{N}$ of tuples of regular expressions, $\mathcal{N} \cdot F$ is the set $\mathcal{N} \cdot F = \{ \mathcal{N} \cdot F | \mathcal{N} \in \mathcal{N} \}$. Finally, $\text{SET}(\mathcal{N}) = \{ E_1, \ldots, E_m \}$ and $\text{SET}(\mathcal{N}) = \bigcup_{\mathcal{N} \in \mathcal{N}} \text{SET}(\mathcal{N})$.

Let $f$ be a symbol in $\Sigma_{=0}$. The set $f^{-1}(E)$ of tuples of regular expressions is defined as follows:

$$f^{-1}(0) = \emptyset, f^{-1}(F + G) = f^{-1}(F) \cup f^{-1}(G), f^{-1}(F^*) = f^{-1}(F) \cdot F^*.$$ 

$$f^{-1}(g(E_1, \ldots, E_m)) = \begin{cases} \{(E_1, \ldots, E_m)\} & \text{if } f = g, \\ \emptyset & \text{otherwise,} \end{cases}$$

$$f^{-1}(F \cdot G) = \begin{cases} f^{-1}(F) \cdot G & \text{if } c \notin \mathcal{F} \\ f^{-1}(F) \cdot G \cup f^{-1}(G) & \text{otherwise.} \end{cases}$$

The function $f^{-1}$ is extended to any set $S$ of regular expressions by $f^{-1}(S) = \bigcup_{E \in S} f^{-1}(E)$.

The partial derivative of $E$ w.r.t. a word $w \in \Sigma_{\geq 1}$, denoted by $\partial_w(E)$, is the set of regular expressions $\partial_w(E) = \begin{cases} \{E\} & \text{if } w = \varepsilon, \\ \text{SET}(f^{-1}(\partial_w(E))) & \text{if } w = uf, f \in \Sigma_{\geq 1}, u \in \Sigma^*, f^{-1}(\partial_w(E)) \neq \emptyset, \\ \{0\} & \text{if } w = uf, f \in \Sigma_{\geq 1}, u \in \Sigma^*, f^{-1}(\partial_w(E)) = \emptyset. \end{cases}$

The Equation Automaton of $E$ is the tree automaton $\mathcal{A}_E = (Q, \Sigma, Q_T, A)$ defined by $Q = \{ \partial_w(E) | w \in \Sigma_{\geq 1} \}$, $Q_T = \{ \varepsilon \}$, and

$$\Delta = \{(F, f, G_1, \ldots, G_m) | F \in Q, f \in \Sigma_{\geq 1}, m \geq 1, (G_1, \ldots, G_m) \in f^{-1}(F)\}$$

$$\cup \{(F, c) | F \in Q \land c \in ([F] \cap \Sigma_0)\}.$$ 

### 2.4. The $k$-C-Continuation Tree Automaton

In [10], Kuske and Meinecke show how to efficiently compute the equation tree automaton of a regular expression via an extension of Champarnaud and Ziadi’s C-Continuation [3, 4, 9]. In [12, 13, 14], we show how to inductively compute it. In this section, we prove that this automaton is isomorphic to the $k$-position tree automaton and we consider the following quotient: $0 \cdot \varepsilon = 0$. As we consider only regular expressions without 0 or reduced to 0 then if after the computation of $k$-C-Continuation we obtain expression of the form $0 \cdot \varepsilon$ we reduce it to 0.

**Definition 2** ([12, 13, 14]). Let $E \neq 0$ be linear. Let $k$ and $m$ be two integers such that $1 \leq k \leq m$. Let $f$ be in $(\Sigma_E \cap \Sigma_m)$. The $k$-C-continuation $C_{\kappa}(E)$ of $f$ in $E$ is the regular expression defined by:

$$C_{\kappa}(g(E_1, \ldots, E_m)) = \begin{cases} E_k & \text{if } f = g, \\ C_{\kappa}(E_j) & \text{if } f \in \Sigma_{E_1} \\ C_{\kappa}(E_1 + E_2) & \text{otherwise} \end{cases}$$

$$C_{\kappa}(E_1 \cdot E_2) = \begin{cases} C_{\kappa}(E_1) \cdot C_{\kappa}(E_2) & \text{if } f \in \Sigma_{E_1} \\ C_{\kappa}(E_2) & \text{if } f \in \Sigma_{E_2} \text{ and } c \in \text{Last}(E_1) \\ 0 & \text{otherwise} \end{cases}$$

By convention, we set $C_{\kappa}(E) = E$.

**Lemma 1.** Let $E$ be a regular expression without occurrences of 0 or reduced to 0. Then, $C_{\kappa}(E)$ is a regular expression without occurrences of 0 or reduced to 0.

Let us now show how to compute the $k$-C-Continuation tree automaton.

**Definition 3** ([12, 13, 14]). Let $E \neq 0$ be linear. The automaton $\mathcal{C}_E = (Q_E, \Sigma_E, \{ C_{\kappa}(E) \}, \Delta_E)$ is defined:

$$Q_E = \{ (f^k, C_{\kappa}(E)) | f \in \Sigma_{E_m}, 1 \leq k \leq m \} \cup \{ (\varepsilon, C_{\kappa}(E)) \}.$$ 

$$\Delta_E = \{ ((x, C_{\kappa}(E)), g, ((g^1, C_{\kappa}(E)), \ldots, (g^m, C_{\kappa}(E)))) | g \in \Sigma_{E_m}, m \geq 1, (C_{\kappa}(E), \ldots, C_{\kappa}(E)) \in g^{-1}(C_{\kappa}(E)) \cup \{ ((x, C_{\kappa}(E)), c) | c \in [C_{\kappa}(E)] \cap \Sigma_0 \} \}$$

**Theorem 3** ([12, 13, 14]). The automaton $\mathcal{C}_E$ accepts $[E]$.

Let $\sim$ be the equivalence relation over the set of states of $\mathcal{C}_E$ defined for any two states $(f^k, C_{\kappa}(E))$ and $(g^m, C_{\kappa}(E))$ by $(f^k, C_{\kappa}(E)) \sim (g^m, C_{\kappa}(E)) \iff h(C_{\kappa}(E)) = h(C_{\kappa}(E))$. 

Proposition 4 ([12, 13, 14]). The automaton $C_E/\sim_e$ is isomorphic to $s_A E$.

Proposition 5. The Follow tree automaton and the Equation Tree Automaton are incomparable though they are derived from two isomorphic automata, i.e. neither is a quotient of the other.

3 A smaller automaton

In [6] P. García et al. proposed an algorithm to obtain an automaton from a word regular expression. Their method is based on the computation of both the partial derivatives automaton and the follow automaton. They join two relations, the first relation is over the states of the word follow automaton and the second relation is over the word c-continuations automaton, in one relation denoted by $\equiv_V$. What we propose is to extend the relation $\equiv_V$ to the case of trees as follows:

$$C_f k j (E) \equiv_V C g p i (E) \iff \exists C_{h_k} (\bar{E}) \sim \varphi C_f j (\bar{E}) \lor \exists C_{h_p} (\bar{E}) \sim \varphi C_g p i (\bar{E})$$

The idea is to define the follow relation $\sim_{\varphi}$ over the states of the $k$-c-continuation automaton $C_E$ as follows: $C_f j (E) \sim_{\varphi} C_g p i (E) \iff \text{Follow}(C_f j (E), j, k) = \text{Follow}(C_g p i (E), g, p)$ such that we keep all the equivalent $k$-c-continuations in the merged states. The obtained automaton is denoted by $C_E/\sim_{\varphi}$. Then apply the relation $\sim_e$ (apply the mapping $h$) over the states of the automaton $C_E/\sim_{\varphi}$ and merge the states which have at least one expression in common.

4 Complexity of the computation of the tree automata

In [10], Kuske and Meinecke extend the algorithm based on the notion of word partial derivatives [1] to tree partial derivatives in order to compute from a regular expression $E$ a tree automaton recognizing $[E]$ with a complexity $O(|E|^2)$. Laugerotte et al. proposed an algorithm for the computation of the position tree automaton and the reduced tree automaton with an $O(||E|| \cdot |E|)$ space and time complexity in [11] with $||E||$ is the alphabetic width of $E$ and $|E|$ is its size. In [12, 13] Mignot et al. gave an efficient algorithm for the computation of the equation automaton using the $k$-c-continuations with an $O(||E|| \cdot |Q|)$ space and time complexity where $|Q|$ is the set of $k$-c-continuations of $E$. The algorithm proposed in [11] for the computation of the function Follow can be used in different constructions such us the equation automaton [10], $k$-c-continuation automaton [12, 13, 14] and Follow Automaton [14].

5 Conclusion

In this paper we define and recall different constructions of tree automata from a regular expression. The different automata and their relations (quotient, isomorphism) defined in this paper are represented in Figure 1, where our extension of García et al. construction is denoted by $\equiv_V$-NFA. We have shown that the $k$-position automaton and the $k$-c-continuations automaton are isomorphic, and that both the equation automaton and the follow automaton are different quotients of the $k$-position automaton.

Looking for reductions of the set of states, we applied the algorithm by García et al. [6] which allowed us to compute an automaton the size of which is bounded above by the size of the smaller of the follow and the equation automata.
Figure 1: Relations between the automata.

References


We propose a reduction of the minimization problem for a bottom-up deterministic tree automaton (DFTA) to the minimization problem for a string deterministic finite automaton (DFA). We proceed by a transformation of the tree automaton into a particular string automaton and then minimize the string automaton. We show that for our transformation, the minimization of the resulting string automaton coincides with minimization original tree automaton. We also discuss the complexity of this approach in different types of tree automata namely standard, acyclic, incremental, and incrementally constructed tree automata.

1 Introduction

Tree automata constitute a powerful theoretical tool used in many fields such as XML Schema [1, 2], natural language processing [3], verification techniques, and program analysis, but relatively few general tools and toolkits exist for algorithm experimentation. In this context, we develop a framework that allows manipulating tree automata and representing large amounts of data as trees.

Minimization is considered as a useful technique to compact the size of automata. In the literature, almost all the minimization techniques [4, 5, 6] are inspired by string automata minimization which was studied for the first time by Huffman and Moore [7]. Their algorithm is based on the definition of distinguishable pairs of states. At the end of the algorithm, all states judged indistinguishable are merged. Later, Hopcroft [8] defined a new algorithm which proceeds by refining the coarsest partition until no more refinements are possible.

Following the same steps, minimization techniques for tree automata have emerged. Early in 1967, Brainerd [4] proposed the first DFTA minimization method which we call standard method. Since then, several algorithms and implementations have been created, all following the same approach as Brainerd’s algorithm, e.g. Arbib [5], Gécseg and Steinby [6], and Comon et al. [9].

After that, Watson [10, 11] designed the first incremental minimization algorithm. It is based on a recursive function that decides if two states are equivalent. Unlike classical techniques, the

---

1Project 8/U03/7015 supported by the MESRS - Algeria.
process can be halted at any time and produces a valid tree automaton that recognizes the same language as the departure one. That algorithm was subsequently refined by Watson and Daciuk [12]. This incremental algorithm constitutes the basics of many other techniques [13, 14].

However, incrementality is an ambiguous term and has been used by Carrasco et al. [15] to interpret another manner of minimal automata building. In fact, this work is a generalization of a previous work [16] on strings. The “incremental” notion is employed here to describe the construction instead of the construction instead of states verification. Hence the minimal tree automaton is constructed by adding trees to a minimal tree automaton while maintaining minimality.

As mentioned above, in the wish to develop a toolkit to manipulate tree automata, our observation from minimization techniques studies is that there exists a degree of analogy with the string automata minimization\(^2\). The question is therefore: does there exist some transformation from a tree automaton (DFTA — a deterministic finite tree automaton) to a string one (DFA — a deterministic finite string automaton) while its minimization coincides with the minimization of the original one. The DFA can then be minimized using one of the well-known techniques, then the result will be transformed back to a DFTA which will be the wanted minimal tree automaton.

In fact, this idea is not completely novel. First, Carrasco et al. [17] define a “signature” to each state which represents the “behaviour” of a state in the minimization process. Next, Abdulla et al. [18] extend this notion to compute an equivalence relation between states called “upward bisimulation” in order to minimize nondeterministic finite tree automata (NFTA). They transform the computation of the equivalence relation to the resolution of a transition system which is similar with string automata. Therefore the complexity of this minimization is \(\Theta(|\mathcal{A}| |Q| \log(|Q|))\) where \(|\mathcal{A}|\) is the size of the automaton and \(|Q|\) the number of its states. This complexity can be mapped to deterministic finite tree automation (DFTA) by using Högberg et al. [19].

In this paper, we continue in this direction and we construct a string automaton which can fully replace the tree one for minimization purposes. Abdulla et al. defined an equivalent transition system that can be used to compute the Myhill-Nerode relation and discussed complexity for standard minimization. The focus of this present paper is on proving that tree automata minimization can be done through string automata minimization techniques which are well studied and the different implementation are available. After the definition of an associated string automaton to a given tree automaton and the proof that Myhill-Nerode congruence coincides in both automata. We show that for the deterministic minimization, the complexity is improved in the way that it is given in function of the string automaton instead of the tree automaton. Next, we show that the associated string automaton minimization coincides also with the acyclic, incremental minimizations. Finally we discuss the complexity in all of theses minimization classes and we show that some results are new and improved. Thus, it will be shown that the associated string automaton can fully replace the initial tree automaton in any minimization technique and reaches in almost all cases a better complexity.

The paper is organized as follows. Section 2 recalls some preliminaries on trees and their automata. Next, the standard minimization algorithm is given with a complexity discussion. Af-\(^2\)Similar approaches are being taken by several other tree automata researchers.
ter that, in Section 4, we detail the basics of our approach and the algorithm then we discuss its complexity in the deterministic case. Section 5 discusses the method impact in acyclic, incremental and incremental construction minimization techniques and reports their complexities. Finally, Section 6 presents some concluding remarks and suggestions for future work.

2 Preliminaries

A ranked alphabet is a pair \((\Sigma, \text{Arity})\) where \(\Sigma\) is a finite set of symbols and \(\text{Arity}\) is a mapping \(\text{Arity}: \Sigma \to \mathbb{N}\) where \(\mathbb{N}\) is the set of nonnegative integers. The arity of a symbol \(f \in \Sigma\) is noted \(\text{Arity}(f)\), the subset of \(p\)-ary symbols of \(\Sigma\) is \(\Sigma_p = \{ f \in \Sigma \mid \text{Arity}(f) = p\}\). We use the notation \(f, f(), f(, ), \ldots, f(, , , )\) respectively for constant, unary, binary, \ldots, \(p\)-ary symbols. For the sake of simplicity, we use just \(\Sigma\) to represent a ranked alphabet \((\Sigma, \text{Arity})\). The set of trees or terms \(T(\Sigma)\) over a ranked alphabet \(\Sigma\) is the smallest set satisfying \(\Sigma_0 \subseteq T(\Sigma)\) and if \(p \geq 1, f \in \Sigma_p\) and \(t_1, t_2, \ldots, t_p \in T(\Sigma)\) then \(f(t_1, t_2, \ldots, t_p) \in T(\Sigma)\). A tree language \(L\) is a subset of \(T(\Sigma)\). The set \(St(t)\) of subtrees of a tree \(t = f(s_1, \ldots, s_n)\) is defined by \(St(t) = \{t\} \cup \bigcup_{k=1}^n St(s_k)\). The set \(t(r \leftarrow s)\) is the set all trees in which we substitute every occurrence of the subtree \(r \in St(t)\) by the tree \(s\) once.

A bottom up finite tree automaton (FTA) over a ranked alphabet \(\Sigma\) is a tuple \(A = (Q, \Sigma, Q_f, \Delta)\) where \(Q\) is a finite set of states, \(Q_f \subseteq Q\) is the set of final states and \(\Delta \subseteq \bigcup_{n \geq 0} \Sigma_n \times S_{p+1}^n\), \(n \in \mathbb{N}\) is a finite set of transitions. The size of a transition \(\rho = (f, q_1, \ldots, q_n, q), f \in \Sigma, q, q_1, \ldots, q_n \in Q\) is \(|\rho| = n + 1\). Then the size of the automaton \(A\) is defined as

\[
|A| = \sum_{\rho \in \Delta} |\rho|
\]  

Now, we consider the deterministic FTA (DFTA)

The transition function \(d\) for a DFTA is:

\[
d: \bigcup_{n \geq 0} \Sigma_n \times Q^n \to Q
\]

\[
d(f, q_1, \ldots, q_n) = q, (f, q_1, \ldots, q_n, q) \in \Delta
\]

\(\Gamma(q) = \{(f, q_1, \ldots, q_n) \mid (f, q_1, \ldots, q_n, q') \in \Delta\}\) denotes the set of arguments extracted from transitions in which the state \(q\) appears but not as an output.

\(\text{occ}_d((f, q_1, \ldots, q_n)) = \{i \mid q_i = q\}\) denotes the set of positions of the state \(q\) in the argument \((f, q_1, \ldots, q_n)\).

Let \(\rho = (f, q_1, \ldots, q_n)\) be an argument, then \(\rho(q_i \leftarrow p) = (f, q_1, \ldots, q_{i-1}, p, q_{i+1}, \ldots, q_n)\) such that \(q_i = q\) denotes the argument computed by substituting \(q\) by \(p\) in a precise place \(i\) in \(\rho\).

In fact, some authors add a special state noted \(\bot\) to complete a tree automaton, this completion is usually used to define equivalence between states. Here, we use \(\Gamma\) to avoid the completion of DFTA and then to define states equivalence using only the existing transitions.

For \(t \in T(\Sigma)\), the output \(m_{\rho}(t)\) when \(A\) operates in \(Q\) is the state in \(Q\) recursively computed as:
Lemma 1  
Let \( m_A(t) = \begin{cases} d(t) & \text{if } t \in \Sigma_0 \\
(d, m_A(t_1), m_A(t_2), \ldots, m_A(t_n)) & \text{if } t = f(t_1, t_2, \ldots, t_n) \in T(\Sigma) - \Sigma_0 \end{cases} \) (4) 

A tree \( t \) is accepted by \( A \) if and only if \( m_A(t) \in Q_f \). The language accepted by \( A \) is: \( L(A) = \{ t \in T(\Sigma) \mid m_A(t) \in Q_f \} \). In the same way the accepted language (down language) by a state \( q \) is defined as follows: \( L^+(q) = \{ t(s \leftarrow #) \mid t \in T(\Sigma), s \in L^+(q) \} \) and \( m_A(t) \in Q_f \). Then, a state \( q \) is accessible if \( L^+(q) \neq \emptyset \). Furthermore, a state \( q \) is co-accessible if there exists \( r \in T(\Sigma \cup \{q\}) \) such that \( q \in St(r) \) and \( m_A(t) \in Q_f \). A state is useless if it is neither accessible nor co-accessible. Useless states and the transitions using them can be safely removed from \( Q \) and \( \Delta \) respectively without affecting \( L(A) \). We can remove all useless states in \( \Theta(|A|) \). Thus, we suppose throughout this paper that every tree automaton is free from useless states.

3 Tree automata minimization

As this work focusses on deterministic minimization, this section presents the standard deterministic approach and gives the most adopted algorithm [9]. In fact, this standard algorithm is a “reincarnation” of the first DFTA minimization due to Brainerd [20] from which every standard DFTA minimization algorithm is derived. We note that every deterministic tree automaton can be minimized by computing states equivalence classes and then merging equivalent states.

Let \( A = (Q, \Sigma, Q_f, \Delta) \) be a DFTA. We define over \( Q \) the following equivalence relation \( \equiv \).

\( p \equiv q \) if:

1. \( p \in Q_f \Leftrightarrow q \in Q_f \) and,
2. for all \( p \in \Gamma^+(p), i \in \text{occ}_p(p) : \rho(p ; i, q) \in \Gamma^+(q) \) and \( d(p) \equiv d(p ; i, q) \)

Minimization for DFTA was first discussed in the late 1960s by Brainerd [4], and standardised in [9, 17]. It computes the equivalence relation \( \equiv \) by successive approximations \((\equiv_j)_{j \geq 0}\):

1. \( p \equiv_0 q \) if and only if \( (p \in Q_f \Leftrightarrow q \in Q_f) \)
2. \( p \equiv_{j+1} q \) if and only if \( p \equiv_j q \) and for all \( p \in \Gamma^+(p), i \in \text{occ}_p(p) : \rho(p ; i, q) \in \Gamma^+(q) \) and \( d(p) \equiv_j d(p ; i, q) \)

The computation of the family \((\equiv_j)_{j \geq 0}\) can then be done by successive approximations until reaching the stable point, that is, some natural number \( k \) such that \( \equiv_k = \equiv_{k+1} \).

**Lemma 1**  For \( k \geq |Q| - 2 \), we have \( \equiv_{k+1} = \equiv_k \)**
Algorithm 1 describes in a general way the standard tree automata minimization. It iterates over a sequence of steps. First, the initial partition is set to \( \{ Q_f, Q - Q_f \} \). Next, at each iteration \( i \), the current partition \( P_i \) is split by computing \( \equiv_i \).

Let us recall that this standard algorithm is quadratic and needs \( O(|\mathcal{A}|^2) \). There exists several implementations of this standard algorithms like Carrasco et al. [17] which are quadratic too.

Furthermore, there exists other tree minimization techniques like acyclic, incremental and incrementally constructed ones. But before discussing them, let us introduce our transformation.

### 4 From DFTA to DFA

The main idea of our reduction is to create an associated string automaton to a FTA to be minimized. Instead of minimizing the wanted FTA, we proceed by minimizing its associated FA. In this section, we show how to construct this FA and we prove some efficient properties, the minimization of this FA is left to the next section.

This idea is not completely novel. First, Carrasco et al. [17] designed a “signature” for each state. We can say the signature plays the role of states “behaviour” in the minimization process.

After that, Abdulla et al. [18] use another way to identify states behaviour in order to provide a NFTA minimization. They compute a composed bisimulation relation which composes downward and upward bisimulation relations. The authors reduce the computation of the upward bisimulation to the resolution of word finite transition systems. The authors prove that for this transformation, the computation of the upward bisimulation can be done by computing an analog equivalence function on the word TS. This can be done using Tarjan-Paige algorithm [21].

For DFTA, this transformation holds using results of Högberg et al. [19]. They report that the upward bisimulation can compute the minimal DFTA.

Based on the previous works, we continue in this direction and we propose a transformation using a similar reduction to that proposed by Abdulla et al. that creates a valid string DFA to prove then that it can replace the DFTA to be minimized in any of minimization techniques and then proving that there is no need to re-implement those algorithms anew. We show also that in term of complexity, this transformation gives the same complexity as the direct techniques in some cases, and better complexity in other ones.

Indeed, our approach proceeds on two steps. First, we construct the equivalence relation \( \sim \) defined on the states of a DFTA \( \mathcal{A} \) and then we regroup states which are possibly equivalent according to the equivalence relation \( \equiv \). We show that \( (p \equiv q \Rightarrow p \sim q) \).

Next, using the relation \( \sim \) we construct a string automaton \( M_{\mathcal{A}} \) using the same states as \( \mathcal{A} \). Then we prove that two states in \( M_{\mathcal{A}} \) are equivalent by the Nerode equivalence relation \( \equiv \) if and only if they are equivalent by the equivalence relation \( \equiv \).

In the following definition, we associate to the transitions set \( \Delta \) a string language \( L_\Delta \) called “horizontal language”. For each transition \( \rho \in \Delta \), we deduce a set of strings \( L_\rho \). The union of all these languages \( L_\rho \) constitutes \( L_\Delta \). An equivalence relation \( \sim \) is defined using \( L_\Delta \) in which we keep for each state a list of strings from \( L_\Delta \) instead of keeping its signature.
From tree automata to string automata minimization

Definition 1 Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be DFTA. The horizontal language of $\Delta$ noted $L_\Delta$ is defined as follows:

\[
L_\Delta = \bigcup_{\mathcal{P} \in \Delta} L_\mathcal{P}
\]

(5)

\[
L_\mathcal{P} = \bigcup_{i=1}^{n} q_i f q_1 \cdots q_{i-1} \bullet q_{i+1} \cdots q_n
\]

(6)

where $\mathcal{P} = (f, q_1, \ldots, q_n, q)$ and $\bullet \not\in \Sigma_0 \cup Q$ is a special symbol.

Definition 2 Let $p, q \in Q$. We say that $p$ and $q$ are possibly equivalent (we note $p \sim q$), if and only if, $(p \in Q_f) \Leftrightarrow (q \in Q_f)$ and for all $f \in \Sigma, u, v \in Q^*$ : $pfu \bullet v \in L_\Delta \Leftrightarrow qfu \bullet v \in L_\Delta$.

Proposition 1 For $p, q \in Q$, we have $p \equiv q \Rightarrow p \sim q$

Lemma 2 The equivalence relation $\sim$ can be computed in $O(|A|)$.

Now, after the identification of the states that are possibly equivalent by $\equiv$, we associate for each state $q$ in a transition $(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n)$ the “letter” $f q_1 \cdots q_{i-1} \bullet q_{i+1} \cdots q_n$. Indeed, we transform the transition of the tree automaton to a transition of a string automaton.

We let $\overline{Q} = \{q \mid \exists p \neq q$ such that $q \sim p\}$. To each state, we associate an alphabet $\sigma_q$ defined as follows:

\[
\sigma_q = \{fu, v \mid \exists qfu \bullet v \in L_\Delta\}
\]

(7)

Proposition 2 We have $|\bigcup_{q \in \overline{Q}} \sigma_q| \leq |\mathcal{A}|$

The automaton $M_{\mathcal{A}}$ will be defined on the alphabet $\sigma = (\bigcup_{q \in \overline{Q}} \sigma_q)$. It is clear that the size of the alphabet $\sigma$ depends on the number of equivalence classes in $\sim$.

Indeed, the new alphabet coincides with the environment defined by Abdullah et al. [18] and we can construct an FA which represents the same transitions system. However, in deterministic case, there is no need to add transitions between states and environment because in this case the left and right sides are equal. Just transitions from environment to the output states are considered. It is possible to show that this construction minimizes the wanted DFTA using the fact that DFTA can be minimized using upward bisimulation (see Högb erg et al. works [19]) and then using Paige-Tarjan algorithm to achieve it. But, we prove below the equivalence between Nerode equivalence in the string and tree automata and we show that it is beneficial in other minimization algorithm—especially in the incremental construction of tree automata.

Definition 3 Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be a DFTA. The string automaton associated to $\mathcal{A}$ noted $M_{\mathcal{A}}$ is the tuple $M_{\mathcal{A}} = (Q', \sigma, \{i_1\}, F, \delta)$ where $Q' = Q \cup \{i_1\}$ is the state set, $\sigma = \bigcup_{q \in Q} \sigma_q \cup Q$ is the alphabet, $\{i_1\}$ is the initial state, $F = Q_f$ is the final states set and $\delta : Q' \times \sigma \rightarrow Q'$ is the transition function defined as follows.


• for all $q$ in $Q$: $d(q, a) = q' \text{ where } a = f_{q_1, \ldots, q_n}d(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n) = q'$.

• for all $q$ in $Q$: $\delta(i_s, q) = q$.

Let us notice that the state $i_s$, the alphabet symbols $Q$, and transitions leaving $i_s$ have no importance in the minimization process. We use them just to construct a habitual string automaton because it is usual to define string automata with an initial state. Moreover, we can see that states from $Q - \overline{Q}$ have no equivalent states because $\forall p, q \in Q - \overline{Q}: p \neq q$ then $p \neq q$. Here, no transition is outgoing from those states.

In the next section, we will prove that the associated string FA can fully replace DFTA minimization the minimization processes namely standard, acyclic, incremental and incrementally constructed minimization and then avoid the re-implementation of those algorithms. We discuss also the complexity of this transformation and we show that it have no negative influence on the time and space process.

5 Minimization techniques using the associated FA

In what follows, we will show and discuss how that the associated FA can fully replace the FTA to be minimized in the specified deterministic minimization techniques namely standard, acyclic, incremental and incrementally constructed minimization. We prove also that in some cases (Acyclic and incremental techniques) the complexities are improved.

5.1 DFTA standard minimization

We show in this part that the minimization of a given DFTA is no more than minimizing its associated string DFA. But first, let us show that the FA is deterministic.

**Proposition 3** The associated string automaton $M_{\mathcal{A}}$ of a DFTA $\mathcal{A}$ is deterministic.

We note that $i_s$ and all transitions outgoing from it are not considered in the minimization process. Then we use $\sigma$ in what follows to denote $\sigma - Q$.

After the string automaton $M_{\mathcal{A}}$ is defined, we show that the computation of the equivalence relation $\equiv$ defined on $\mathcal{A}$ can be done by computing the Nerode equivalence $\simeq$ relation defined on $M_{\mathcal{A}}$. $\simeq$ is defined as follows.

\[
p \equiv q \iff \left\{ \begin{array}{l}
p \in F \iff q \in F \\
\text{for all } a \in \sigma, \delta(p, a) \simeq \delta(q, a)
\end{array} \right.
\]

**Proposition 4** For $p, q \in Q$, we have $(p \equiv q) \iff (p \equiv q)$

We can also check this corollary:

**Corollary 1** Let $p \in Q - \overline{Q}, q \in Q (q \neq p)$ we have $p \neq q$. 
As a consequence of this Corollary and as mentioned above, only states in $\mathcal{Q}$ are considered in the minimization algorithm. We can easily check that $i_s$ is not equivalent to any state. Indeed, for each state $q \in \mathcal{Q}' - \mathcal{Q}$, its equivalence class is $[q]$. Using Hopcroft Algorithm [8], the automaton $M_{\mathcal{A}}$ can be minimized in $O(|\sigma||Q|\log|Q|)$.

**Lemma 3** The automaton $M_{\mathcal{A}}$ can be minimized in $O(|\sigma||Q|\log|Q|)$.

**Theorem 1** Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be a DFTA, $\mathcal{A}$ can be minimized in $\Theta(|\mathcal{A}| + |\sigma||Q|\log(|Q|))$.

Note that this complexity is the same as if Abdulla et al. [18] reduction is combined with results of Höfger et al. [19]. The only difference here is that the complexity is given in function of the string automaton which is the output of the transformation algorithm. Indeed, DFTA minimization has a major specificity compared with its homologous NFTA one: the left and right sides of two states in a transition arguments must be the same. Then in worst case, every transition can connect with $\min(\hat{r}, |\Delta|)$ transitions where $\hat{r}$ is the maximum rank of $\Sigma$. And as every connection between two transitions is counted once, then $|\sigma| = \frac{\min(\hat{r}, |\Delta|) \times |\Delta|}{2} < |\mathcal{A}|$.

### 5.2 Acyclic minimization

Acyclic automata are a beneficial data structure that represent and store finite set of objects. When objects are trees like in Xml, it is useful to store a finite XML files in a compact and small structure. Acyclic DFA (ADFA) minimization was largely discussed like in [22, 23, 24] and almost of these techniques have linear asymptotic complexity. Here we show how the associated FA can positively minimize an acyclic DFTA.

Although Proposition 4 is sufficient for proving the use of the associated DFA to minimize ADFTA, but let us recall some useful definitions.

**Definition 4** Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be a DFTA. Then $\mathcal{A}$ is acyclic (ADFTA) if and only if for all $q \in Q$, if $t \in L^1(q)$ then $S_{\mathcal{A}}(t) \cap L^1(q) = \{t\}$.

We can consider the following lemma (the proof is trivial and then omitted).

**Lemma 4** The associated DFA of a ADFTA is acyclic.

Using Proposition 4 we know that states from an ADFTA which are not in $\sim$ are distinguishable and cannot be merged during minimization since ADFTA are trivial case of DFTA.

Thus, after computing the associated string ADFA $\mathcal{M}_{\mathcal{A}}$ of a DFTA $\mathcal{A}$ using one of the string acyclic minimizations like [23, 24] in linear time:

**Theorem 2** A ADFTA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ can be minimized using its associated ADFA $\mathcal{M}_{\mathcal{A}} = (Q', \sigma, \{i_s\}, F, \delta)$ in $O(|\sigma||Q|)$. 
5.3 Incremental minimization

Incremental minimization is a useful technique in practice. It is used when minimization process may be halted in any time producing a reduced automaton in terms of states number and producing a valid one which recognizes the same language as the departure automaton.

In string case, Watson et al. [25] introduce for the first time the incremental version for cyclic DFA, but the complexity as reported by authors is exponential. Next, Watson et al. [12] improve this algorithm and give an almost quadratic implementation. After that, Almeida et al. [14] present the best known incremental implementation using the UNION-FIND algorithm.

However, in tree case, Cleophas et al. [13] generalize the incremental approach to trees and give a cubic implementation to this need.

Here also, we show that the incremental minimization can be done using the associated DFA and the complexity of this minimization is better than previous works on trees.

Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be a DFTA and $\mathcal{M}_\mathcal{A} = (Q', \sigma, \{i_s\}, F, \delta)$ be its associated DFA. We extend the transition function $\delta$ by $\delta'$ as follows. $\delta'(q,a) = \delta(q,a)$ where $a \in \sigma$ and $\delta'(q,ax) = \delta'(\delta(q,a), x)$ where $ax \in \sigma^+$. We define the right language of a state $q \in Q$ by: $L_r(q) = \{x \in \sigma^+ \mid \delta'(q,x) \in F\}$.

**Lemma 5** Let $p, q \in Q$ then $L_r(p) = L_r(q) \iff \overline{L}(p) = \overline{L}(q)$.

Using this lemma, we can compute $\text{equiv}(p,q)$ in $\mathcal{M}_\mathcal{A}$ instead of computing it in $\mathcal{A}$.

Thus, we can use the best-known complexity algorithm for DFA due to Almeida et al. [14] to minimize a DFTA by incrementally minimizing its associated DFA:

**Theorem 3** A DFTA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ can be incrementally minimized using its associated DFA $\mathcal{M}_\mathcal{A} = (Q', \sigma, \{i_s\}, F, \delta)$ in $O(|\sigma||Q|^2 \alpha(|Q|))$ where $\alpha$ is the inverse Ackermann time function.

5.4 Incremental construction of minimal tree automata

Incremental construction of automata is an important approach which is discussed in string and tree cases. It allows to add or delete words (resp. trees) to an existing minimal automaton. In other words if $\mathcal{A}$ is a DFA (resp. DFTA) and $w$ is a word (resp. $t$ is a tree) then the incremental construction consists on creating a new automaton which recognizes $L(\mathcal{A}) \cup \{w\}$ (resp. $L(\mathcal{A}) \cup \{t\}$) while maintaining minimality. First, incremental construction was presented by Daciuk et al. [26] for ADFA. Next, Carrasco et al. [16] generalize this notion to cyclic DFA. Later, they redefine the incremental construction for trees in [15].

**Theorem 4** Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be a minimal DFTA and $\mathcal{M}_\mathcal{A} = (Q', \sigma, \{i_s\}, F, \delta)$ be its associated minimal DFA then the minimal automaton that recognizes $L(\mathcal{A}) \cup \{t\}$ where $t$ is constructed in $O(|\Delta|^2 + |\mathcal{A}|)$.

Here, we conclude the results shown in this paper in the following theorem.

**Theorem 5 (main result)** Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be a DFTA and $\mathcal{M}_\mathcal{A} = (Q', \sigma, \{i_s\}, F, \delta)$ be its associated DFA then $\mathcal{A}$ can be minimized using $\mathcal{M}_\mathcal{A}$ in:
1. $O(|\sigma| |Q| \log(|Q|))$ (standard minimization),
2. $O(|\sigma| |Q|)$ if $\mathcal{A}$ is acyclic,
3. $O(|\sigma| |Q|^2 \alpha(|Q|))$ (incremental minimization),
4. If $\mathcal{A}$ is minimal, then the minimal DFTA recognizing $L(\mathcal{A}) \pm \{t \in T_S\}$ can be computed using $\mathcal{M}_\mathcal{A}$ in $O(|\Delta|^2 + |\mathcal{A}|)$ where $\hat{r}$ is the maximum rank of $\Sigma$.

6 Conclusion

In this paper, we have shown how the minimization problem of deterministic tree automata can be reduced to the minimization problem of deterministic string automata which is considered as well-studied since the 60s. Indeed, we use the environment (and the TS transformation) notion proposed by Abdulla et al. [18] to create a string alphabet which is read by an associated string automaton and then minimize it. Hence, we prove that there is actually no need to implement existing algorithms proposed for trees and exploit the large range of minimization algorithms for strings to add minimization procedures in tree toolkits. Moreover, We prove that DFTA minimization can be done in $O(|\mathcal{A}| + |\sigma| |Q| \log(|Q|))$, where $\sigma$ is the alphabet of the DFA $M_\mathcal{A}$ associated to $\mathcal{A}$ for standard minimization (which is considered in term of asymptotic complexity as the same as the best known one) and we show that the minimization using associated DFA gives better complexities in other existing minimization approaches namely acyclic and incremental minimization (which are clearly improved in this present paper).

In fact, it is interesting to study the average size of $\sigma$. This leads us to consider the problem of random generation of deterministic tree automata. But instead of the existing random generators in literature, no real generator is developed. We hope that we consolidate this work with an experimental tests and comparisons with other techniques after developing a such generator.

References


A Tree automata minimization algorithm

Algorithm 1 Tree automata minimization

1: function MINIMIZATION(DFTA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$) 
2: $P_0 \leftarrow Q$ 
3: $P_1 \leftarrow \{Q_f, Q - Q_f\}$ 
4: $i \leftarrow 1$ 
5: repeat 
6: create $P_{i+1}$ by refining $P_i$ so that $p \equiv_{i+1} q$ iff for all $p \in \Gamma(p), i \in \text{occ}_p(p) : p(p ; q) \in \Gamma(q)$ 
7: $d(p) \equiv d(p(p ; q))$ 
8: $i \leftarrow i + 1$ 
9: until ($P_i = P_{i-1}$) 
10: $Q_{\text{min}} \leftarrow \bigcup q \in Q$ 
11: $\Delta_{\text{min}} \leftarrow \{ (f, [q_1], \ldots, [q_n]) \mid (f, q_1, \ldots, q_n) \in \Delta \}$ 
12: $Q_{\text{min}} f \leftarrow \{q \mid q \in Q_f\}$ 
13: return $\mathcal{A}_{\text{min}} = (Q_{\text{min}}, \Sigma, Q_{\text{min}} f, \Delta_{\text{min}})$ 
14: end function

B Proofs of section 4

B.1 Proof of Proposition 1

Proof 1 Let $p,q \in Q$ such that $p \not\sim q$. Then, there exists $p f q_1 \ldots q_{i-1} \cdot q_{i+1} \ldots q_n \in L_{\Delta}$ but $p f q_1 \ldots q_{i-1} \cdot q_{i+1} \ldots q_n \notin L_{\Delta}$. Using Definition 1 then we have $(f, q_1, \ldots, q_{i-1}, p, q_{i+1}, \ldots, q_n, p') \in \Delta$ but no transition with the form $(f, q_1, \ldots, q_{i-1}, q, p_{i+1}, \ldots, q_n) (p ; q) \notin \Gamma(q)$ although $i \in \text{occ}_p((f, q_1, \ldots, q_{i-1}, p, q_{i+1}, \ldots, q_n))$. By states equivalence we have $p \not\equiv q$.

B.2 Proof of Lemma 2

Proof 2 As the language $L_{\Delta}$ is finite, equivalence relations $\sim$ can be computed in linear time on the size of $|\mathcal{A}|$. This can be done by minimizing an acyclic automaton which reads $L_{\Delta}$.

C Proofs of section 5

C.1 Proof of Proposition 3

Proof 3 By definition we have for all $q \in Q' - \overline{Q}, a \in \sigma : |\delta(q, a)| \leq 1$. Let $q \in \overline{Q}$ and assume that there exists a symbol $f_{u,v} \in \sigma$ such that $\delta(q, a) = \{q', q''\}$ with $q' \neq q''$. Let $u = q_1 \ldots q_{i-1}$
and \( v = q_{i+1} \ldots q_n \). Using Definition 1 we have then \((f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n, q') \in \Delta \) and \((f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n, q') \in \Delta \). This leads to a contradiction because the tree automaton \( \mathcal{A} \) is deterministic.

### C.2 Proof of Proposition 4

**Proof 4** It is well known that the Nerode equivalence \( \equiv \) can be computed by successive approximations \( \equiv_j \) defined as:

\[
p \equiv_0 q \quad \text{iff} \quad (p \in F \iff q \in F)
\]

\[
p \equiv_{j+1} q \quad \text{iff} \quad p \equiv_j q \text{ and for all } a \in \Sigma, p, q \in Q : \delta(p, a) \equiv_j \delta(q, a).
\]

Here, and as mentioned below, useless states which appear in the DFA are equivalent and distort the correctness of the approach. The solution is to initialize \( \equiv_0 \) with what follows. \( \equiv_0 = \overline{Q} \cup \{(q, q) \mid q \in Q \} \).

To prove this proposition we show that for all \( p, q \in Q, j \in \mathbb{N} : p \equiv_j q \Rightarrow p \equiv_j q \). The proof is done by induction on the definitions of \( \equiv_j \) and \( \equiv_j \). For the basic case \((j = 0)\), as \( Q_f = F \), for \( p, q \in Q \), we have \( p \equiv_0 q \Rightarrow p \equiv_0 q \). Assume now that for all \( p, q \in Q, (p \equiv_k q) \Rightarrow (p \equiv_k q) \) for some \( k \geq 0 \).

First, we prove that \((p \equiv_{k+1} q) \Rightarrow (p \equiv_{k+1} q)\):

Suppose that \((p \equiv_{k+1} q)\). By the successive approximations of \( \equiv \) we have

\[
(p \equiv_{k+1} q) \iff p \equiv_k q \text{ and for all } f_{a,v} \in \Sigma, \delta(p, f_{a,v}) \equiv_k \delta(q, f_{a,v})
\]

By applying induction hypothesis, we get that

\[
(p \equiv_{k+1} q) \iff p \equiv_k q \text{ and for all } f_{a,v} \in \Sigma, \delta(p, f_{a,v}) \equiv_k \delta(q, f_{a,v})
\]

Let \( u = q_1 \ldots q_{i-1} \) and \( v = q_{i+1} \ldots q_n \). Using the horizontal language definition we get that \( p f u v \equiv q f u v \in L_\Delta \). So, there exists two states \( p' \) and \( q' \) in \( Q \) such that \((f, q_1, \ldots, q_{i-1}, p, q_{i+1}, \ldots, q_n, p')\) and \((f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n, q')\) are in \( \Delta \) (see equation (6)). From Definition 3 we have \( p' = \delta(p, f_{u,v}) = d(f, q_1, \ldots, q_{i-1}, p, q_{i+1}, \ldots, q_n) \) and \( q' = \delta(q, f_{u,v}) = d(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n) \). Finally, we get then \((p \equiv_{k+1} q)\) by applying (13).

Now, the next step of the proof is to show that \((p \equiv_{k+1} q) \Rightarrow (p \equiv_{k+1} q)\). This proof can be done following the same steps as the first implication (this part is omitted).
1. Introduction

Multi bottom-up tree transducers were originally introduced and studied in [3, 14], albeit under different names. We consider the linear and extended variant, which we call MBOT for short. MBOT have good algorithmic properties [9, 15] and thus they were further developed into a formal model for tree-to-tree translation, which is a sub-discipline in syntax-based statistical machine translation (SMT). An open-source implementation of an SMT system based on shallow MBOT is available [5].

The semantics of our MBOT is defined by means of a derivation relation over sentential forms. We apply synchronous rewriting [6], which means that several parts of the sentential form develop (via the rules) at the same time. The left-hand side of a rule contributes to the input tree and the right-hand side to the output tree of the sentential form. For MBOT, the right-hand side consists of a vector of trees, so it can act simultaneously at several positions in the output tree. The input and output positions that are supposed to be developed in parallel are recorded by active links \((v, w)\), which relate a position \(v\) in the input tree to a position \(w\) in the output tree. After applying a rule using active links, those used links are disabled. Thus disabled links simply record all links that were active at some point during the derivation. In this way, we preserve all links and can later argue about their structure, which will allow us to prove properties about MBOT. Links are similar to origins of [1, 4, 10, 13]. A dependency computed by an MBOT is a triple which consists of an input tree, an output tree derived from it, and the set of all disabled links of the derivation.

Our first result is that the links in each dependency are organized hierarchically and that the distance between (input and output) link targets is bounded (Theorems 1 and 2). Then we provide generic linking theorems for \(\varepsilon\)-free MBOT which, given an MBOT that computes a tree relation with particular properties, predict certain natural links that must be present in the set of computed dependencies (Theorems 3 and 4). Theorem 3 concerns arbitrary compositions of \(\varepsilon\)-free XTOPR (which are \(\varepsilon\)-free MBOT whose right-hand sides contain at most one tree), whereas Theorem 4 concerns a single \(\varepsilon\)-free MBOT. In both cases, we assume that the computed tree relation contains a sub-relation that is obtained by plugging trees from a simple, yet infinite tree language into an input-output context pair. Finally, we demonstrate in Section 5 how to apply these linking theorems to show that certain tree relations cannot be computed by any \(\varepsilon\)-free MBOT or by any composition of \(\varepsilon\)-free XTOPR.

2. Preliminaries

We use the set \(\mathbb{N}\) of all nonnegative integers and the set \(\mathbb{N}_+\) of all positive integers. The composition of relations \(\rho\) and \(\rho'\) is denoted by \(\rho : \rho'\), and the inverse of the relation \(\rho\) is denoted by \(\rho^{-1}\).
The set of all finite words over $S$ is $S^*$, where $\varepsilon \in S^*$ is the empty word. The concatenation of the words $v, w \in S^*$ is $v\cdot w$ or simply $vw$. The length of a word $w \in S^*$ is denoted by $|w|$. Given a word (or vector) $w \in \Sigma^*$ and $1 \leq i \leq |w|$, we write $w_i$ for the $i^{th}$ letter in $w$.

In the following, let $\Sigma$ be an alphabet and $S$ be a set with $\Sigma \cap S = \emptyset$. The set $T_\Sigma(S)$ of $S$-trees indexed by $S$ is defined as usual, and we let $T_\Sigma^*(S)$. The set $\text{pos}(t) \subseteq \mathbb{N}_1^*$ of positions of $t$, the height $\text{ht}(t)$ of $t$, and the size $|t|$ of $t$ are defined in the standard way. Further, for $w \in \text{pos}(t)$, we denote by $t(w)$ the label of $t$ at $w$, and by $t_w$ the $w$-rooted subtree of $t$.

Positions are totally ordered by the lexicographic order $\preceq$ on $\mathbb{N}_1^*$ and partially ordered by the prefix order $\preceq$. Given a finite set $P \subseteq \mathbb{N}_1^*$ of positions, we let $\mathcal{F} = (w_1, \ldots, w_k)$ be the vector of the positions of $P$ in lexicographic order, where $P = \{w_1, \ldots, w_k\}$ with $w_1 \preceq \cdots \preceq w_k$. For a sequence $\mathcal{F} = (u_1, \ldots, u_n)$ of trees and positions $\mathcal{T} = (w_1, \ldots, w_n)$ of $t$ that are pairwise incomparable with respect to $\preceq$, we let $t[w]\mathcal{T}$ denote the tree obtained from $t$ by replacing (in parallel) all subtrees $t_w$ at $w_i$ for all $1 \leq i \leq n$. In the special case $n = 1$, we also use the notation $t[w]_1$.

For every $s \in S$, let $\text{pos}_s(t) = \{w \in \text{pos}(t) \mid t(w) = s\}$. If $|\text{pos}_s(t)| \leq 1$ for every $s \in S$, then the tree $t \in T_\Sigma(S)$ is linear, and we denote the set of all linear trees of $T_\Sigma(S)$ by $T_\Sigma^\text{lin}(S)$. We reserve the sets $X = \{x_i \mid i \in \mathbb{N}_\ast\}$ and $X_n = \{x_i \mid 1 \leq i \leq n\}$ of variables. A tree $t \in T_\Sigma(X_n)$ is an $n$-context over $\Sigma$ if $t$ is linear and all variables of $X_n$ occur in $t$. The set of all $n$-contexts over $\Sigma$ is denoted by $C_\Sigma(X_n)$. Given $c \in C_\Sigma(X_n)$ and $t_1, \ldots, t_n \in T_\Sigma$, we write $c[t_1, \ldots, t_n]$ for $c[w]\mathcal{T}$, where $\mathcal{T} = (t_1, \ldots, t_n)$ and $w = (w_1, \ldots, w_n)$ with $w_i \in \text{pos}_{x_i}(c)$ being the unique position of $x_i$ in $c$ for every $1 \leq i \leq n$.

3. Linear extended multi bottom-up tree transducers

A multi bottom-up tree transducer (for short: MBOT) is a tuple $M = (Q, \Sigma, I, R)$ where $Q$ is the alphabet of states, $I \subseteq Q$ contains the initial states, $\Sigma$ is the alphabet of input and output symbols such that $\Sigma \cap Q = \emptyset$, and $R \subseteq T_\Sigma^\text{lin}(Q) \times Q \times T_\Sigma(Q)^*$ is the nonempty, finite set of rules. We write $t \xrightarrow{\delta} \mathcal{T}$ for a rule $(\ell, q, \mathcal{T}) \in R$. We require that all states in $\mathcal{T}$ appear in $\ell$ for every $(\ell, q, \mathcal{T}) \in R$. If $|\mathcal{T}| \leq 1$ for all $(\ell, q, \mathcal{T}) \in R$, then $M$ is a (linear) extended top-down tree transducer with regular look-ahead [2, 8, 18] (for short: xtopR), and if $|\mathcal{T}| = 1$ for all $(\ell, q, \mathcal{T}) \in R$, then it is a (linear) nondeleting xtopR (for short: n-xtop). Finally, it is $\varepsilon$-free if $\ell \notin Q$ for all $(\ell, q, \mathcal{T}) \in R$. Each rule $(\ell, q, \mathcal{T}) \in R$ is a look-ahead rule because it can be used to check whether an input subtree belongs to a certain regular tree language [12]. For the remaining discussion, let $M = (Q, \Sigma, I, R)$ be an MBOT.

An example is the $\varepsilon$-free MBOT $M_{\alpha} = (\{q\}, \Sigma, \{q\}, R)$ with $\Sigma = \{\sigma, \gamma_1, \gamma_2, \alpha\}$ and the set $R$ of rules containing $\sigma(\alpha, q, \alpha) \xrightarrow{q} \sigma(q, \alpha, q), \gamma_1(q) \xrightarrow{q} \gamma_1(q), \gamma_2(q) \xrightarrow{q} \gamma_2(q) \cdot \gamma_2(q)$, and $\alpha \xrightarrow{\cdot} \alpha \cdot \alpha$ (see Figure 1).

A link is just an element $(v, w) \in \mathbb{N}_* \times \mathbb{N}_*$. A sentential form over $Q$ and $\Sigma$ is a tuple $(\xi, A, D, \zeta)$, where $\xi, \zeta \in T_\Sigma(Q)$ and $A, D \subseteq \text{pos}(\xi) \times \text{pos}(\zeta)$. Elements in $A$ and $D$ are called active and disabled links, respectively. We denote by $\mathcal{SF}(Q, \Sigma)$ the set of all sentential forms over $Q$ and $\Sigma$. The link structure $\text{links}_{v, w}(\ell, q, \mathcal{T})$ of the rule $(\ell, q, \mathcal{T}) \in R$ for positions $v$ and $w = (w_1, \ldots, w_j)$ with $v, w_1, \ldots, w_j \in \mathbb{N}_*$ is

$$\text{links}_{v, w}(\ell, q, \mathcal{T}) = \bigcup_{p \in Q} \left\{(v', w, w') \mid v' \in \text{pos}_p(\ell), w' \in \text{pos}_p(r_t)\right\}. $$
For the left-most rule $\rho$ presented in Figure 1 and the positions $v = 1.2$ and $\vec{w} = (2)$ we obtain the link structure $\text{links}_{v,\vec{w}}(\rho) = \{(1.2.2, 2.1), (1.2.2, 2.3)\}$. Figure 1 shows the links of input hierarchical because (2) requires the corresponding input-side property for the inverted set $\vec{A}$ if $\implies$ both $u$ and the set of dependencies $\text{D}(M)$ of sentential forms computed by $M$ is

$$\text{D}(M) = \{(\xi, A, D, \zeta) \in \text{SF}(Q, \Sigma) \mid \exists q \in I: \langle q, \{v, e\}, \emptyset, q \rangle \implies M \langle \xi, A, D, \zeta \rangle\},$$

and the set $\text{D}(M)$ of dependencies computed by $M$ is $\text{D}(M) = \{(t, D, u) \mid t, u \in T_\Sigma, (t, \emptyset, D, u) \in \text{SF}(M)\}$. Finally, the tree relation computed by $M$ is $M = \{(t, u) \mid (t, D, u) \in \text{D}(M)\}$.

A short derivation using the mbot $M_{ex}$ is shown in Figure 2. It results in the dependency $\langle t, \{(e, u), (2, 1), (2, 3), (2, 1.1.1), (2.1.3.1), (2.1.1.1.1), (2.1.1.3.1.1)\}, u \rangle$, where $t = \sigma(\alpha, \gamma_1(\gamma_2(\alpha)), \alpha)$ and $u = \sigma(\gamma_1(\gamma_2(\alpha)))$.

Next, we introduce some important properties for sets of links, sentential forms, and the set of dependencies computed by an mbot (see [16]). A set $L \subseteq \mathbb{N}_+ \times \mathbb{N}_+$ of links is (i) input hierarchical if $v_1 < v_2$ implies both $\vec{w}_1 < \vec{w}_2$ and that there exists $(v_1, w_1) \in L$ with $w_1 \leq w_2$, and (ii) strictly input hierarchical if $v_1 < v_2$ implies $w_1 < w_2$ and $v_1 = v_2$ implies that $w_1$ and $w_2$ are comparable with respect to $\leq$, for all $(v_1, w_1), (v_2, w_2) \in L$. A sentential form $(\xi, A, D, \zeta) \in \text{SF}(Q, \Sigma)$ is (strictly) input hierarchical whenever $A \cup D$ is. Finally, $\text{D}(M)$ has those properties if for each $(t, D, u) \in \text{D}(M)$ the corresponding sentential form $(t, \emptyset, D, u)$ has them [i.e., $M$ has them]. The property (strictly) output hierarchical can be defined by requiring the corresponding input-side property for the inverted set $L^{-1}$ of links, the inverted sentential form $(\xi, A^{-1}, D^{-1}, \xi)$, and the set $\text{D}(M)^{-1} = \{(u, D^{-1}, t) \mid (t, D, u) \in \text{D}(M)\}$.

The links $L$ illustrated in the last derivation step of Figure 2 are input hierarchical. They are not strictly input hierarchical because $(2, 1), (2.1.3.1.1) \in L$ violates the stricter condition. However, $L$ is strictly output hierarchical.

**Theorem 1 (see [16, Lm. 22])** Let $M$ be an mbot. (i) The set $\text{D}(M)$ is input hierarchical and strictly output hierarchical. (ii) If $M$ is an xtop, then $\text{D}(M)$ is also strictly input hierarchical. □

Let $b \in \mathbb{N}$. A sentential form $(\xi, A, D, \zeta) \in \text{SF}(Q, \Sigma)$ has (i) link distance $b$ in the input if for all links $(v_1, w_1), (v_1', w_2) \in A \cup D$ with $|v'| > b$ there exists a link $(v_1, v_2) \in A \cup D$ such that $v < v'$ and $1 \leq |v| \leq b$, and (ii) strict link distance $b$ in the input if for all positions $v_1, v_1' \in \text{pos}(\xi)$ with $|v'| > b$ there exists a link $(v_1, v_2) \in A \cup D$ such that $v < v'$ and $1 \leq |v| \leq b$. The set $\text{D}(M)$ of dependencies has those properties

![Figure 2: A derivation of the mbot $M_{ex}$. The active links are clearly marked, whereas disabled links are shown in light gray.](image-url)
if for each \((t, D, u) \in D(M)\) the corresponding sentential form \(\langle t, \emptyset, D, u \rangle\) has them. Moreover, \(D(M)\) is (strictly) link-distance bounded in the input if there exists an integer \(b \in \mathbb{N}\) such that it has (strict) link distance \(b\) in the input. A sentential form \(\langle \xi, A, D, \zeta \rangle\) and \(D(M)\) have (strict) link distance \(b\) in the output if \(\langle \zeta, A^{-1}, D^{-1}, \xi \rangle\) and \(D(M)^{-1}\) have (strict) link distance \(b\) in the input, respectively.

**Theorem 5** ([3, Sect. 3.4]) computed by a classical result of [3], which states that the class of tree relations computed by \(n\)-a classical result of [3] have (strict) link distance \(b\) in the input.

### 4. Linking theorems

Our linking theorems establish the existence of certain interrelated links, which are forced simply by a subset of the computed tree relation. We need the following utility definitions. A tree \(t \in T_{\Sigma}\) is a chain (or unary tree) if \(\text{pos}(t) \subseteq \{1\}^*\), and \(t\) is a binary tree if \(\text{pos}(t) \subseteq \{1, 2\}^*\). A tree language \(T \subseteq T_{\Sigma}\) is (i) unary shape-complete if for every chain \(t \in T_{\Sigma}\) there exists a tree \(t' \in T\) with \(\text{pos}(t') = \text{pos}(t)\), and (ii) binary shape-complete if for every binary tree \(t \in T_{\Sigma}\) there exists a tree \(t' \in T\) with \(\text{pos}(t') = \text{pos}(t)\).

We now start with a linking theorem for the composition of arbitrarily many \(\varepsilon\)-free \(\text{xtop}^R\). This theorem is only applicable to tree relations, which contain a sub-relation that is obtained with the help of an input and an output context into which we can plug trees from a unary shape-complete tree language. If such a tree relation \(\tau\) is computed by a composition \(\tau = M_1 \ldots M_k\) of \(\varepsilon\)-free \(\text{xtop}^R\) \(M_1, \ldots, M_k\), then we can deduce a dependency and the natural links relating the corresponding subtrees of the contexts.

**Theorem 3** Let \(k, n \in \mathbb{N}_+\) and \(M_1, \ldots, M_k\) be \(\varepsilon\)-free \(\text{xtop}^R\) over \(\Sigma\) such that

\[
\{\langle c[t_1, \ldots, t_n], c'[t_1, \ldots, t_n] \rangle \mid t_1 \in T_1, \ldots, t_n \in T_n\} \subseteq M_1 \ldots M_k
\]

for some \(c, c' \in C_{\Sigma}(X_\Sigma)\) and unary shape-complete tree languages \(T_1, \ldots, T_n \subseteq T_{\Sigma}\). Then there exist trees \(t_1 \in T_1, \ldots, t_n \in T_n\), dependencies \(\langle u_0, D_1, u_1 \rangle \in D(M_1), \ldots, (u_{k-1}, D_k, u_k) \in D(M_k)\) with \(u_0 = c[t_1, \ldots, t_n]\) and \(u_k = c'[t_1, \ldots, t_n]\), and a link \((v_{ji}, w_{ji}) \in D_i\) for each \(1 \leq i \leq k\) and \(1 \leq j \leq n\) such that (i) \(\text{pos}_{x_{ji}}(c') \leq w_{ji}\) for all \(1 \leq j \leq n\), (ii) \(v_{ji} \leq w_{ji-i-1}\) for all \(2 \leq i \leq k\) and \(1 \leq j \leq n\), and (iii) \(\text{pos}_{x_{ji}}(c) \leq v_{ji}\) for all \(1 \leq j \leq n\).

We know that \(\varepsilon\)-free \(\text{mbot}\) and several relevant subclasses (different from \(\text{xtop}^R\) and its subclasses) are closed under composition [9]. Therefore, our second linking theorem concerns a single \(\varepsilon\)-free \(\text{mbot}\).

**Theorem 4** Let \(n \in \mathbb{N}_+\) and \(M = (Q, \Sigma, I, R)\) be an \(\varepsilon\)-free \(\text{mbot}\) such that

\[
\{\langle c[t_1, \ldots, t_n], c'[t_1, \ldots, t_n] \rangle \mid t_1 \in T_1, \ldots, t_n \in T_n\} \subseteq M
\]

for some \(c, c' \in C_{\Sigma}(X_\Sigma)\) and binary shape-complete tree languages \(T_1, \ldots, T_n \subseteq T_{\Sigma}\). Then there exist trees \(t_1 \in T_1, \ldots, t_n \in T_n\), a dependency \(\langle c[t_1, \ldots, t_n], D, c'[t_1, \ldots, t_n] \rangle \in D(M)\) and a link \((v_{ji}, w_{ji}) \in D\) for every \(1 \leq j \leq n\) such that (i) \(\text{pos}_{x_{ji}}(c) \leq v_{ji}\) for all \(1 \leq j \leq n\) and (ii) \(\text{pos}_{x_{ji}}(c') \leq w_{ji}\) for all \(1 \leq j \leq n\).

### 5. Applications of the linking theorems

We present some applications of our linking theorems to existing results of the literature. We start with a classical result of [3], which states that the class of tree relations computed by \(\text{xtop}^R\) (as well as those computed by \(n\)-\(\text{xtop}\)) is not closed under composition.

**Theorem 5** ([3, Sect. 3.4]) The class of tree relations computable by \(\varepsilon\)-free \(\text{xtop}^R\) (or \(\varepsilon\)-free \(n\)-\(\text{xtop}\)) is not closed under composition.
The class of regularity preserving tree relations computable by $\varepsilon$-free MBOT is not closed under inverses.

**Proof.** Let $M_{\text{top}} = (Q, \Sigma, \{\varepsilon\}, R)$ be the $\varepsilon$-free MBOT with $Q = \{\varepsilon, p, q, r\}$ and $\Sigma = \{\varepsilon, \delta, \gamma, \alpha\}$, where $R$ contains exactly the rules $\delta(p, q) \rightarrow \delta(\varepsilon \rho(p, q))$, $\delta(p, q) \rightarrow \delta(p, \varepsilon \rho(q))$, $\delta(p, \varepsilon \rho(q)) \rightarrow \varepsilon \rho(p, q)$, $\delta(\varepsilon \rho(p, q)) \rightarrow \varepsilon \rho(p, q)$, and $\varepsilon = \varepsilon \rho(p, q)$ for every $x \in \{p, q, r\}$. We can check that $M_{\text{top}}$ is regularity preserving. The inverse $M_{\text{top}}^{-1}$, which is also regularity preserving, is illustrated in Figure 4. We suppose for the sake of a contradiction that there exists an $\varepsilon$-free MBOT $M = (Q, \Sigma, I, R)$ that computes $M_{\text{top}}^{-1}$. By Theorem 2(i) there...
exists a $b \in \mathbb{N}$ such that $D(M)$ has strict link distance $b$ in the output. Moreover, let $n > b + 2$, and we select the contexts $\alpha \in T$, $\gamma(t) \in T$ for all trees $t \in T$, and $\sigma(t_1, t_2) \in T$ for all trees $t_1, t_2 \in T$. By Theorem 4 there are trees $t_1, \ldots, t_n \in T$, a dependency $\langle [c_1], \ldots, [c_n], D, c' \rangle \in D(M)$, and a link $(v_j, w_j) \in D$ for every $1 \leq j \leq n$ such that (i) $\text{pos}_{x_j}(c) \leq v_j$ and (ii) $\text{pos}_{x_j}(c') \leq w_j$ for all $1 \leq j \leq n$ (see Figure 4). Based on those links, we can derive a contradiction because there exists a link $(v, w) \in D$ such that $\varepsilon < w < 2^{b+1}$.

Finally, we show that the relation $M_{\text{tpc}}$ cannot be computed by any composition of $\varepsilon$-free $\text{xtop}^R$ (see [17]). The authors believe that a proof approach based on the common fooling technique would be rather difficult (or even hopeless) as we would need to argue over several (at least 2) unknown intermediate trees.

**Theorem 7** ([17, Thm. 6]) The relation $M_{\text{tpc}}$ cannot be computed by any chain of $\varepsilon$-free $\text{xtop}^R$.  

**Proof (Sketch.)** As before we prove the statement by contradiction. Therefore, we assume that $M_{\text{tpc}}$ is computed by a composition of several $\varepsilon$-free $\text{xtop}^R$. By [11, Thm. 11] we know that three $\varepsilon$-free $\text{xtop}^R$ suffice, so there are $\varepsilon$-free $\text{xtop}^R M_1$, $M_2$, and $M_3$ over $\Sigma$ such that $M_{\text{tpc}} = M_1 \cdot M_2 \cdot M_3$. By Theorem 3 there are links, which can be used to derive a contradiction.

**References**