M1 Summer internship report
Computer Science Department, Boston College
June – August 2010

Advisor: Howard Straubing

An automaton model for forest algebras

by

Antoine Delignat-Lavaud
antoine@delignat-lavaud.fr

Contents

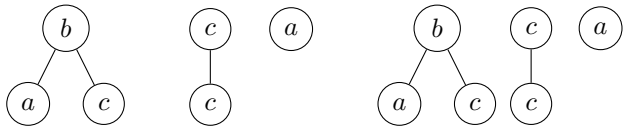
Internship description and acknowledgements

This report presents the results of a 10-week internship of the author at the Computer Science department of Boston College. Despite the small size of the department with less than a dozen permanent members, most of them working on subjects outside the theory of computation, the quality of reception was more worthy of an invited professor than of a mere first-year graduate student. In particular, working conditions were definitely a step up from what is usually offered in French institutions, especially to gratuate and doctoral students. However, working conditions would have meant nothing if not for the great availability of my advisor and his willingness to involve me in his research. For this I wish to thank Howard Straubing for making this internship a fruitful experience.

1 Introduction to forest algebras

It is well known from two celebrated results by Schützenberger and McNaughton that star-freeness, aperiodicity and first-order definability (with an order predicate on positions and letter-testing predicates) are equivalent notions for regular word languages. This theorem laid the foundation for the algebraic theory of regular languages. Many characterizations of regular language classes by algebraic (e.g. equations defining a pseudo-variety) or logical descriptions have been obtained using techniques and results from semigroup theory.

The goal of forest algebras is to provide a similarly universal algebraic framework to study regular tree languages. This introduction is a summary of a paper by Bojańczyk and Walukiewicz [1]; most proofs are omitted and can be found in their article. By forest, we mean an ordered collection of unranked ordered trees. We denote by $\mathcal{T}(A)$ and $\mathcal{F}(A)$ respectively the set of unranked ordered trees and forests over the alphabet A. There are two natural operations on forests: concatenation, which we denote $+$, although it is not necessarily commutative, consists of putting one forest after the other; while rooting, which we denote $\cdot$, is a left action of the alphabet A on $\mathcal{F}(A)$. If $t \in \mathcal{F}(A)$ and $a \in A$, $a \cdot t$ denotes the tree with root a and t as the subforest of descendants of a.



(a) A forest t_1 (b) A forest t_2 (c) The forest $t_1 + t_2$

Figure 1: Concatenation of forests

The empty forest is denoted 0, with the convention that $a \cdot 0 = a$ for all $a \in A$; it is clear that $(\mathcal{F}(A), 0, +)$ is a monoid. We can define the syntax of forests as $t ::= 0 \mid t + t \mid a \cdot t,\ a \in A$. This formalism is well-suited to build automata on forests (it corresponds to a bottom-up pass). However, rooting is too limited an operation for our algebraic approach. To understand this, consider the syntactic congruence on words: $w \equiv_L w'$ means that in any word x we can replace the subword w with w' (or w' with w) without changing the membership of x in L. As for forests, we cannot easily

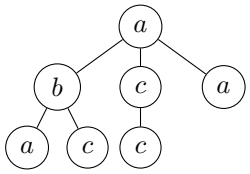
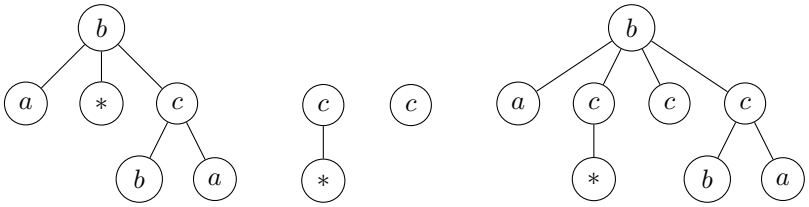


Figure 2: The rooting $a \cdot (t_1 + t_2)$

describe what it means to replace a subforest with another one in terms of rooting. For this purpose, we need to introduce *contexts*.

A context over an alphabet A is a forest over $A \uplus \{*\}$ where $*$ appears in exactly one leaf which we call the *hole*. We denote by $\mathcal{C}(A)$ the set of contexts over A and 1 the context with a single leaf labelled $*$. If $p, q \in \mathcal{C}(A)$, we define $p \circ q$ as the context obtained by replacing the hole in p by q; $(\mathcal{C}(A), 1, \circ)$ is then a monoid.

If $p \in \mathcal{C}(A)$ and $t \in \mathcal{F}(A)$, we can define $p \cdot t$ as the forest obtained by replacing the hole in p by t. This is a monoidal action of $\mathcal{C}(A)$ on $\mathcal{F}(A)$: if $q \in \mathcal{C}(A)$, $(p \circ q) \cdot t = p \cdot (q \cdot t)$.



(a) A context p (b) A context q (c) The context $p \circ q$

Figure 3: Composition of contexts

Definition 1. *A* forest algebra *is a tuple* $(H, V, \cdot, \mathrm{in_L}, \mathrm{in_R})$ *where:*

1. H *is the* horizontal monoid *denoted* $(H, 0, +)$.
2. V *is the* vertical monoid *denoted* $(V, 1, \circ)$.
3. $\cdot$ *is a left monoidal action of* V *on* H.
4. $\mathrm{in_L}, \mathrm{in_R} : H \to V$ *are such that* $\mathrm{in_L}(g) \cdot h = g + h$ *and* $\mathrm{in_R}(g) \cdot h = h + g$ *for all* $g, h \in H$.
5. *The action* $\cdot$ *is* faithful, *i.e. for every two distinct* $v, w \in V$, *there exists some* $h \in H$ *such that* $v \cdot h \neq w \cdot h$.

Remark 1. Despite the additive notation, we do not require $(H, +)$ to be commutative. Having H commutative corresponds to the case of unordered unranked forests. This case leads to major simplifications, some of which can be explained by the next remark.

Remark 2. Because of the insertion functions, the horizontal monoid is contained in the vertical monoid: for all $h \in H$, we have $\mathrm{in_L}(h) \cdot 0 = h + 0 = h$, i.e. any element from H is of the form $v \cdot 0$ with $v \in V$. Furthermore, insertion functions are injective monoid morphisms.

Remark 3. The faithfulness condition cannot be described equationally because of the existential quantifier, and it is not preserved by homomorphic image. However, if (H, V) satisfies all the axioms of a forest algebra but faithfulness, it can be transformed into a forest algebra $(H, V/{\sim_V})$ where

$$v_1 \sim_V v_2 \text{ if and only if } \forall h \in H, v_1 \cdot h = v_2 \cdot h$$

Remark 4. Because $\cdot$ is a monoidal action, we will often omit $\circ$ and $\cdot$ and write vwh instead of $(v \circ w) \cdot h = v \cdot (w \cdot h)$.

A *morphism* between two forest algebras (H, V) and (G, W) is a pair of monoid morphisms $(\alpha : H \to G, \beta : V \to W)$ that preserves the action and insertion functions. In fact, if α is surjective, the condition on the action is sufficient:

$$\forall h \in H, v \in V,\ \alpha(vh) = \beta(v)\alpha(h)$$

As it implies that for all $h, h' \in H$ and $g \in \mathrm{Im}(\alpha) = G$:

$$\begin{aligned} \beta(\mathrm{in_L}(h))\alpha(h') &= \alpha(\mathrm{in_L}(h)h') \\ &= \alpha(h + h') \\ \beta(\mathrm{in_L}(h))g &= \alpha(h) + g \\ &= \mathrm{in_L}(\alpha(h))g \\ \beta(\mathrm{in_L}(h)) &= \mathrm{in_L}(\alpha(h)) \end{aligned}$$

The last equality is derived from the previous one using the faithfulness condition. Notice that we used the same notation for the insertion functions and operations of each forest algebra. A similar argument yields $\beta(\mathrm{in_R}(h)) = \mathrm{in_R}(\alpha(h))$.

Remark 5. The morphism α is determined by β via

$$\alpha(h) = \alpha(h + 0) = \alpha(\mathrm{in_L}(h)0) = \beta(\mathrm{in_L}(h))0_G$$

Definition 2. *The* free forest algebra *over A, denoted A^Δ, is the forest algebra defined by:*

- *The horizontal monoid is $(\mathcal{F}(A), 0, +)$.*
- *The vertical monoid is $(\mathcal{C}(A), 1, \circ)$.*
- *The action is substitution of forests in contexts.*
- *The* $\mathrm{in_L}$ *(respectively* $\mathrm{in_R}$*) function transforms a forest into a context by adding a hole to the right (resp. left) of all the roots of the forest.*

It is clear A^Δ is a forest algebra, while the following lemma shows that it is indeed a free algebraic structure:

Lemma 1. For all forest algebras (H, V), any function $f : A \to V$ can be uniquely extended to a morphism $(\alpha, \beta) : A^\Delta \to (H, V)$ such that $\beta(a(*)) = f(a)$ for every $a \in A$.

The proof of this lemma relies on the fact that the context monoid is generated by rootings $a(*)$ and insertions $\mathrm{in_L}(t), \mathrm{in_R}(t)$, which are fixed by the axioms. Given a forest algebra (H, V), we will define a morphism $(\alpha, \beta) : A^\Delta \to (H, V)$ by the action of β on $a(*)$ or, by an abuse of notation, by giving $\beta(a)$.

Definition 3. *A set L of A-forests is said to be* recognized *by a forest algebra morphism $(\alpha, \beta) : A^\Delta \to (H, V)$ if there exists some $F \subseteq H$ such that $L = \alpha^{-1}(F)$. The morphism (α, β) is said to* recognize *L and F is called the* accepting set*. We also say that L is recognized by (H, V). In the case when (H, V) is finite, we say that L is* recognizable*.*

Definition 4. *Let L be a forest language over A. The* horizontal syntactic equivalence *relative to L is defined as follows: we say that two forests $t_1, t_2 \in \mathcal{F}(A)$ are L-equivalent, which we denote $t_1 \equiv_L t_2$, if $\forall p \in \mathcal{C}(A), pt_1 \in L \Leftrightarrow pt_2 \in L$.*

It can be easily verified that the horizontal syntactic equivalence is a congruence with respect to concatenation and substitution of forests. Hence, the action of contexts on $\equiv_L$-classes is well-defined. We can then use the method in remark 3 to make this action faithful:

Definition 5. *The* vertical syntactic equivalence *relative to L is defined as follows: we say that two contexts $p_1, p_2 \in \mathcal{C}(A)$ are L-equivalent, which we denote $p_1 \equiv_L p_2$, if $\forall t \in \mathcal{F}(A), p_1 t \in L \Leftrightarrow p_2 t \in L$.*

Definition 6. *The* syntactic forest algebra *of $L \subseteq \mathcal{F}(A)$ is the quotient of A^Δ by the syntactic congruences. Namely, it is the forest algebra (H^L, V^L) with $H^L = \mathcal{F}(A)/\equiv_L$ and $V^L = \mathcal{C}(A)/\equiv_L$. The canonical projection of this quotient (α^L, β^L) is called the* syntactic morphism*.*

The next proposition justifies that this construction has the expected minimality property of syntactic objects:

Proposition 1. The syntactic morphism of a forest language L recognizes L. Moreover, any morphism $(\alpha, \beta) : A^\Delta \to (H, V)$ that recognizes L can be extended by a morphism $(\alpha', \beta') : (H, V) \to (H^L, V^L)$ such that $\beta' \circ \beta = \beta^L$.

Remark 6. One may wonder what is the benefit of considering forests instead of trees. Indeed, it is possible to define tree algebras. In this case, the horizontal monoid is still the forest monoid, while contexts are trees with one hole at a leaf (the insertion functions are modified accordingly). The action of a context on a forest is then a tree, and a tree language is recognizable if is is recognized by a tree algebra morphism.

In this context, proposition 1 does not hold: there exist languages with two minimal recognizing tree algebras. One such example is the language of trees where only the root can have more than one child. There are syntactic classes for the empty forest, forests with one path, forests which are a concatenation of paths, and a class for everything else. If we denote $H = \{0, 1, 2, \bot\}$ the honrizontal monoid, then both $H \cup \{i\}$ and $H \cup \{f\}$ with $i + i = i$ and $f + f = \bot$ recognize L but have no common quotient recognizing L. The same issue arises if we say that a tree language is recognizable if it is the intersection of a recognizable forest language with the set of trees.

2 Automata over forests

In this section, we present two equivalent models for automata over forests. The first one is very algebraic and closely related to forest algebras while the second one is more suitable for practical purposes.

2.1 Forest automata

Definition 7. *A* forest automaton *is a tuple* $\mathcal{A} = \langle (Q, 0, +), A, \delta : A \times Q \to A, F \subseteq Q \rangle$, *where* $(Q, 0, +)$ *is the finite* state monoid*,* δ *is the* transition function *of the automaton.*

A forest automaton defines a map from $\mathcal{F}(A)$ to Q. This map, written $\cdot^{\mathcal{A}}$, is defined by induction on the structure of forests:

- $0^{\mathcal{A}} = 0$.
- $(t_1 + t_2)^{\mathcal{A}} = t_1^{\mathcal{A}} + t_2^{\mathcal{A}}$.
- $(a \cdot t)^{\mathcal{A}} = \delta(a, t^{\mathcal{A}})$.

The language accepted by $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \{t \in \mathcal{F}(A) \mid t^{\mathcal{A}} \in F\}$.

Proposition 2. A forest language is recognizable if and only if it is accepted by some forest automaton.

Proof: If a forest automaton $\mathcal{A} = \langle (Q, 0, +), A, \delta : A \times Q \to Q, F \subseteq Q \rangle$ accepts L, we define a forest algebra (H, V) as follows:

- The horizontal monoid H is $(Q, 0, +)$.
- The vertical monoid V is $(H^H, \mathrm{id}_H, \circ)$ the monoid of functions from H to H.
- The action is function application: $v \cdot h = v(h)$.
- The insertion functions are uniquely defined by the action.

It is clear that this defines a forest algebra. We now define a morphism $(\alpha, \beta) : A^\Delta \to (H, L)$ by $\beta(a) = \delta(a) \in H^H$. One can verify by induction that $t^{\mathcal{A}} = \alpha(t)$.

Conversely, if L is recognized by some morphism $(\alpha, \beta) : A^\Delta \to (H, V)$, i.e. $L = \alpha^{-1}(F)$ for some $F \subseteq H$, let $\mathcal{A} = \langle H, A, \delta : A \times H \to H, F \subseteq H \rangle$ where δ is defined by:

$$\delta(a, h) = \beta(a) \cdot h$$

such that $t^{\mathcal{A}} = \alpha(t)$. It follows that $\mathcal{L}(\mathcal{A}) = L$ is regular. ♢

2.2 Bottom-up deterministic forest automata

Definition 8. *A* bottom-up deterministic forest automaton *(BUDFA) is a tuple:*

$$\mathcal{A} = \langle S, Q, s_0 \in S, \gamma : S \times Q \to S, \lambda : S \times A \to Q, F \subseteq S \rangle$$

S *is the finite set of* horizontal states*,* s_0 *is the* initial state*,* F *is the subset of* accepting states*,* Q *is the finite set of* vertical states*.* γ *is a semiautomaton, its set of states is* S *and its alphabet is* Q*.* γ *defines a left monoidal action of* Q^* *on* S*; we will often write* $s \cdot q_1 \dots q_n$ *instead of* $\gamma(\gamma(\cdots\gamma(s, q_1)\cdots), q_{n-1}), q_n)$*.*

A BUDFA $\mathcal{A}$ defines a map from $\mathcal{F}(A)$ to S. This map, written $\cdot^{\mathcal{A}}$, is defined by induction on the structure of forests:

- $0^{\mathcal{A}} = s_0$.
- $(t_1 + a \cdot t_2)^{\mathcal{A}} = t_1^{\mathcal{A}} \cdot \lambda(t_2^{\mathcal{A}}, a)$.

It is possible to draw a BUDFA as follows: we start by drawing the horizontal automaton (S, Q, s_0, γ, F), each state s is then tagged with a sequence $a_1 : q_1, \dots, a_n : q_n$ such that for all $1 \leqslant i \leqslant n$, $\lambda(s, a_i) = q_i$.

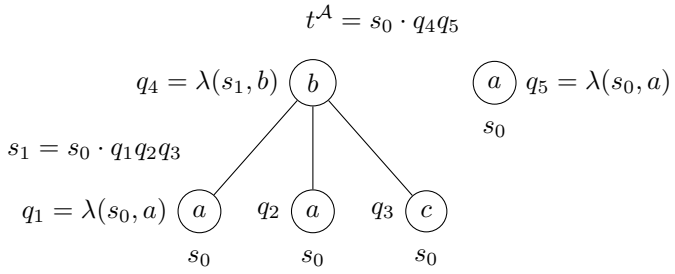


Figure 4: Evaluation of $t^{\mathcal{A}}$

Example 1. Let L_1 be the language of forests over $\{a, b\}$ in which any path contains at most one node labelled a. The BUDFA in figure 5 accepts L_1; figure 6 shows and accepting run of the automaton.

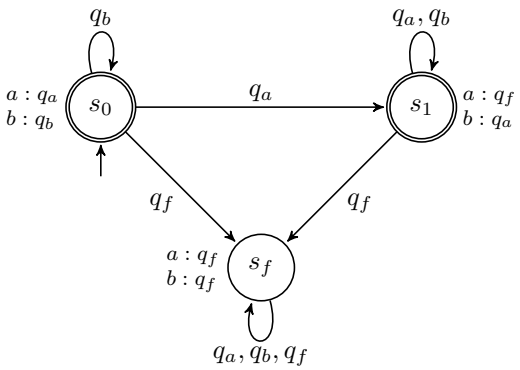


Figure 5: A BUDFA accepting L_1

Example 2. Let L_2 be the language of binary trees in which any maximal path has even length. The BUDFA in figure 7 accepts L_2.

A forest automaton

$$\mathcal{A} = \langle (Q, 0, +), A, \delta : A \times Q \to Q, F \subseteq Q \rangle$$

can be easily transformed into a BUDFA

$$\mathcal{A}' = \langle Q, Q, 0, \gamma : Q \times Q \to Q, \lambda : Q \times A \to Q, F \subseteq Q \rangle$$

by taking $\gamma(q_1, q_2) = q_1 + q_2$ and $\lambda(q, a) = \delta(a, q)$, so that $t^{\mathcal{A}} = t^{\mathcal{A}'}$ for any forest t. Conversely, it is possible to transform a BUDFA

$$\mathcal{A} = \langle S, Q, s_0, \gamma, \lambda, F \rangle$$

into a forest automaton. For $q \in Q$, let $\gamma_q : s \mapsto \gamma(s, q)$; $\gamma_q \in S^S$. Let

$$T = \{\gamma_q \mid q \in Q\} \subseteq S^S$$

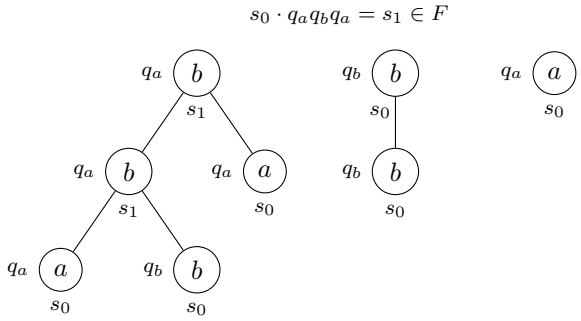


Figure 6: An accepting run of the automaton

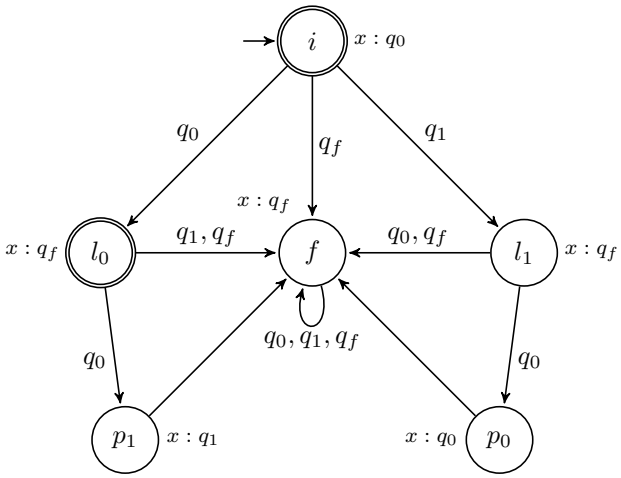


Figure 7: A BUDFA accepting L_2

We denote by $*$ the reverse composition of functions, i.e. if $f, g \in S^S$, $f * g = g \circ f$. $H = (\langle T \rangle, \mathrm{id}_S, *)$ denotes the submonoid of $(S^S, \mathrm{id}_S, *)$ generated by T. We now define the forest automaton

$$\mathcal{A}' = \langle H, A, \delta : A \times H \to H, F' \rangle$$

where $F' = \{f \in H | f(s_0) \in F\}$ and $\delta(a, f) = \gamma_{\lambda(f(s_0),a)}$.

We show by induction on t that $t^{\mathcal{A}} \in F$ if and only if $t^{\mathcal{A}'} \in F'$:

- If $t = 0$: $0^{\mathcal{A}} \in F \Leftrightarrow s_0 \in F \Leftrightarrow \mathrm{id}_S(s_0) \in F \Leftrightarrow \mathrm{id}_S \in F' \Leftrightarrow 0^{\mathcal{A}'} \in F'$.
- If $t = t_1 + a \cdot t_2$:

$$\begin{aligned}
(t_1 + a \cdot t_2)^{\mathcal{A}} &= \gamma(t_1^{\mathcal{A}}, \lambda(t_2^{\mathcal{A}}, a)) \\
(t_1 + a \cdot t_2)^{\mathcal{A}'} &= t_1^{\mathcal{A}'} * \delta(a, t_2^{\mathcal{A}'}) \\
&= t_1^{\mathcal{A}'} * \gamma_{\lambda(t_2^{\mathcal{A}'}, a)} \\
&= s \mapsto \gamma(t_1^{\mathcal{A}'}(s), \lambda(t_2^{\mathcal{A}'}, a))
\end{aligned}$$

The same construction can be used to transform a BUDFA into a forest algebra recognizing the same language: let H be the transition monoid of the horizontal DFA of

$\mathcal{A}$ (as defined above). If $w = q_1 \dots q_k \in Q^*$, we define $\gamma_w = \gamma_{q_1} * \cdots * \gamma_{q_k}$, in particular H is $(\langle \gamma_w \mid w \in Q^* \rangle, \mathrm{id}_S, *)$.

Let V be the submonoid of $(H^H, \mathrm{id}_H, \circ)$ generated by:

$$\begin{array}{rcl} & & H \longrightarrow H \\ v_a & : & h \longmapsto \gamma_{\lambda(h(s_0),a)} \\ v_{\mathrm{in_L}(t)} & : & h \longmapsto \gamma_t * h \\ v_{\mathrm{in_R}(t)} & : & h \longmapsto h * \gamma_t \end{array}$$

for all $a \in A, t \in \mathcal{F}(A)$. The action is $v \cdot h = v(h)$.

(H, V) is called the *transition forest algebra* of $\mathcal{A}$. It is the forest algebra associated with the forest automaton built from $\mathcal{A}$ as above, hence it recognizes the language accepted by $\mathcal{A}$.

2.3 Minimization of BUDFA

Let L be a recognizable forest language. Recall that the syntactic congruence $\equiv_L$ is then of finite index, since the syntactic forest algebra recognizes L.

A *left-context* is a context in which the hole has no sibling on its left. The set of left-contexts is denoted $\mathcal{C}_\ell(A)$. We define the equivalence relation on $\mathcal{F}(A)$: $t_1 \sim_L t_2$ if and only if for every left-context $p \in \mathcal{C}_\ell(A)$, $p \cdot t_1 \in L \Leftrightarrow p \cdot t_2 \in L$.

It is clear that $\equiv_L \subseteq \sim_L$, i.e. that $\sim_L$ is *coarser* than $\equiv_L$, and consequently it is of finite index.

We can restrict the syntactic congruence of forests to the set of trees. This equivalence relation on $\mathcal{T}(A)$ is still denoted $\equiv_L$.

The *minimal automaton* of L is then defined as follows:

$$\mathcal{A}_L = \langle \mathcal{F}(A)/\!\sim_L, \mathcal{T}(A)/\!\equiv_L, [0]_{\sim_L}, \gamma, \lambda, [L]_{\sim_L} \rangle$$

where $\lambda([t]_{\sim_L}, a) = [at]_{\equiv_L}$ and $\gamma([t_1]_{\sim_L}, [t_2]_{\equiv_L}) = [t_1 + t_2]_{\sim_L}$.

This automaton is well-defined because:

- $t_1 \sim_L t_2$ implies $at_1 \equiv_L at_2$: if $p \in \mathcal{C}(A)$, then $p \circ (a(*)) \in \mathcal{C}_\ell(A)$.
- $t_1 \sim_L t'_1$ and $t_2 \equiv_L t'_2$ implies $t_1 + t_2 \sim_L t'_1 + t'_2$: for all $p \in \mathcal{C}_\ell(A)$, since $p \circ \mathrm{in_R}(t_2) \in \mathcal{C}_\ell(A)$:

$$\begin{array}{rcl} (p \circ \mathrm{in_R}(t_2)) \cdot t_1 \in L & \Leftrightarrow & (p \circ \mathrm{in_R}(t_2)) \cdot t'_1 \in L \\ p \cdot (t_1 + t_2) \in L & \Leftrightarrow & p \cdot (t'_1 + t_2) \in L \\ (p \circ \mathrm{in_L}(t_1)) \cdot t_2 \in L & \Leftrightarrow & (p \circ \mathrm{in_L}(t'_1)) \cdot t'_2 \in L \\ p \cdot (t_1 + t_2) \in L & \Leftrightarrow & p \cdot (t'_1 + t'_2) \in L \end{array}$$

The first step of minimization is to trim the automaton of its non-reachable states. The following algorithm, inspired by [2], can be used to compute the reachable states of a BUDFA $\mathcal{A}$: we maintain two subsets $S' \subseteq S$ and $Q' \subseteq Q$ of reachable states. They are initialized to $S' = \{s_0\}$ and $Q' = \{\lambda(s_0, a) \mid a \in A\}$. A breadth-first search is performed in the horizontal DFA of $\mathcal{A}$ starting from s_0 using transitions in Q'. Whenever a new state $s \in S$ is reached, it is added to S' and all the $q_1, \dots, q_p$ that

appear in the tag $a_1 : q_1, \ldots, a_p : q_p$ of s are added to Q'. Then, any previously ignored transition labelled with some q_i during the BFS is explored.

To measure the complexity of algorithms on BUDFA, we define the size of a BUDFA to be the size of the horizontal transition table, i.e. $|S| \cdot |Q|$. The size of the output function is not relevant because there are at least as many vertical states as letters in the alphabet in a useful automaton. The trimming algorithm is then linear because each transition is explored at most once.

Let $\mathcal{A} = \langle S, Q, s_0, \gamma, \lambda, F\rangle$ be a trimmed BUDFA that recognizes L. We can build a morphism of BUDFA from $\mathcal{A}$ to $\mathcal{A}_L$ as follows: for $s \in S, q \in Q$ let

$$\begin{aligned} T_s &= \{t \in \mathcal{F}(A) \mid t^{\mathcal{A}} = s\} \subseteq \mathcal{F}(A) \\ T_q &= \{at \in \mathcal{F}(A) \mid \lambda(t^{\mathcal{A}}, a) = q\} \subseteq \mathcal{T}(A) \end{aligned}$$

The core of the proof of minimality of $\mathcal{A}_L$ is contained in the following lemma:

Lemma 2. Let $s \in S$, $q \in Q$.

1. If $t_s, t'_s \in T_s$ then $t_s \sim_L t'_s$.
2. If $t_q, t'_q \in T_q$ then $t_q \equiv_L t'_q$.

Proof: Let $t_s, t'_s \in T_s, t_q, t'_q \in T_q, p \in \mathcal{C}_\ell(A), r \in \mathcal{C}(A)$.

1. When $\mathcal{A}$ reads $p \cdot t_s$ or $p \cdot t'_s$, since the hole has no left sibling, the horizontal DFA reaches the state s after reading t_s or t'_s; hence $(p \cdot t_s)^{\mathcal{A}} = (p \cdot t'_s)^{\mathcal{A}}$.
2. When $\mathcal{A}$ reads $r \cdot t_q$ or $r \cdot t'_q$, the horizontal DFA ends up in the same state if the hole was replaced by t_q or t'_q since both are labelled by the same q. ♢

Because of this lemma, the map $s \mapsto [T_s]_{\sim_L}$, $q \mapsto [T_q]_{\equiv_L}$ defines a surjective morphism from $\mathcal{A}$ to $\mathcal{A}_L$: s_0 is mapped to $[0]_{\sim_L}$ and F is mapped to $[L]_{\equiv_L}$.

Moreover, the following equivalence relations on S and Q:

$$\begin{aligned} s \sim_{\mathcal{A}} s' \quad &\text{if and only if} \quad [T_s]_{\sim_L} = [T_{s'}]_{\sim_L} \\ q \sim_{\mathcal{A}} q' \quad &\text{if and only if} \quad [T_q]_{\equiv_L} = [T_{q'}]_{\equiv_L} \end{aligned}$$

can be used to minimize any trimmed BUDFA $\mathcal{A}$ recognizing L; $\mathcal{A}/{\sim_{\mathcal{A}}}$ is then isomorphic to $\mathcal{A}_L$.

Proposition 3. Algorithm 1 computes the relation $\sim_{\mathcal{A}}$ given a trimmed BUDFA $\mathcal{A}$.

Proof: Let us define the relation $\sim^i$ by:

$$\begin{aligned} \sim^i \;=\; &\{(s_1, s_2) \in S^2 \mid \{s_1, s_2\} \text{ is not marked after i steps of the loop on line 3}\} \\ &\uplus \{(q_1, q_2) \in Q^2 \mid \{q_1, q_2\} \text{ is not marked after i steps of the loop on line 3}\} \end{aligned}$$

Clearly, any pair marked by the algorithm is not equivalent for $\sim_{\mathcal{A}}$, i.e. $\sim_{\mathcal{A}} \subseteq \sim^i$. We define two families of contexts by induction:

$$\begin{aligned} \mathcal{C}_s^0 &= \mathcal{C}_q^0 = \{*\} \\ \mathcal{C}_s^{i+1} &= \mathcal{C}_s^i \cup \{p \circ \mathrm{in_R}(t) \mid p \in \mathcal{C}_s^i, t \in \mathcal{T}(A)\} \cup \{p \circ a(*) \mid p \in \mathcal{C}_q^i, a \in A\} \\ \mathcal{C}_q^{i+1} &= \mathcal{C}_q^i \cup \{p \circ \mathrm{in_L}(t) \mid p \in \mathcal{C}_s^i, t \in \mathcal{F}(A)\} \end{aligned}$$

Algorithm 1 Computation of $\sim_{\mathcal{A}}$

1: Mark all pairs $\{s_1, s_2\} \in S^2$ such that $s_1 \in F$ and $s_2 \notin F$.
2: Mark all pairs $\{q_1, q_2\} \in Q^2$ such that $\gamma(s_0, q_1) \in F$ and $\gamma(s_0, q_2) \notin F$.
3: **repeat**
4: **for all** $a \in A$ **do**
5: **for all** unmarked $\{s_1, s_2\} \subseteq S$ **do**
6: Mark $\{s_1, s_2\}$ if $\{\lambda(s_1, a), \lambda(s_2, a)\}$ is marked
7: **end for**
8: **end for**
9: **for all** $q \in Q$ **do**
10: **for all** unmarked $\{s_1, s_2\} \subseteq S$ **do**
11: Mark $\{s_1, s_2\}$ if $\{\gamma(s_1, q), \gamma(s_2, q)\}$ is marked
12: **end for**
13: **end for**
14: **for all** $s \in S$ **do**
15: **for all** unmarked $\{q_1, q_2\} \subseteq Q$ **do**
16: Mark $\{q_1, q_2\}$ if $\{\gamma(s, q_1), \gamma(s, q_2)\}$ is marked
17: **end for**
18: **end for**
19: **until** No new pair is marked
20: **return** $\{(s_1, s_2) \in S^2 \mid \{s_1, s_2\} \text{ is not marked}\} \uplus$
$\{(q_1, q_2) \in Q^2 \mid \{q_1, q_2\} \text{ is not marked}\}$

Notice that all contexts in $\mathcal{C}_s^i$ are left contexts. We also define:

$$\begin{array}{ll} s \sim_{\mathcal{A}}^i s' & \text{if and only if} \quad \forall p \in \mathcal{C}_s^i, p \cdot t_s \in L \Leftrightarrow p \cdot t_{s'} \in L \\ q \sim_{\mathcal{A}}^i q' & \text{if and only if} \quad \forall p \in \mathcal{C}_q^i, p \cdot t_q \in L \Leftrightarrow p \cdot t_{q'} \in L \end{array}$$

In the above definition, t_s and t_q denote any element of respectively T_s and T_q. We show by induction on $i \in \mathbb{N}$ that $\sim^i \subseteq \sim_{\mathcal{A}}^i$.

- Assume $s \sim^0 s'$, i.e. $s \in F \Leftrightarrow s' \in F$. Then $t_s \in L \Leftrightarrow t_{s'} \in L$, which is equivalent to $s \sim_{\mathcal{A}}^0 s'$ since $\mathcal{C}_s^0 = \{*\}$. Similarly, if $q \sim^0 q'$, i.e. $\gamma(s_0, q) \in F \Leftrightarrow \gamma(s_0, q') \in F$, then $t_q \in L \Leftrightarrow t_{q'} \in L$, which is equivalent to $q \sim_{\mathcal{A}}^0 q'$ since $\mathcal{C}_q^0 = \{*\}$
- Let $i \in \mathbb{N}$, assume that $\sim^i \subseteq \sim_{\mathcal{A}}^i$ and $s \not\sim_{\mathcal{A}}^i s'$. There exists some $p \in \mathcal{C}_s^{i+1}$ such that for instance $p \cdot t_s \in L$ and $p \cdot t_{s'} \notin L$.
  - If $p \in \mathcal{C}_s^i$ then $s \not\sim^i s'$: $\{s, s'\}$ was marked at step i hence $s \not\sim^{i+1} s'$.
  - If $p = p' \circ a(*)$ with $p' \in \mathcal{C}_q^i$ then $p' \circ t_{\lambda(s,a)} \in L$ and $p' \circ t_{\lambda(s',a)} \notin L$. But then, the pair $\{s, s'\}$ is marked at step $i+1$ of the algorithm because of line 6, hence $s \not\sim^{i+1} s'$.
  - If $p = p' \circ \mathrm{in}_{\mathrm{R}}(t_q)$ with $p' \in \mathcal{C}_s^i$ and $t_q \in T_q$ then $p' \circ t_{\gamma(s,q)} \in L$ and $p' \circ t_{\gamma(s',q)} \notin L$. But then, the pair $\{s, s'\}$ is marked at step $i+1$ of the algorithm because of line 11, hence $s \not\sim^{i+1} s'$.

  The other implication $q \not\sim_{\mathcal{A}}^{i+1} q' \Rightarrow q \not\sim^{i+1} q'$ is similar. It follows that $\sim^{i+1} \subseteq \sim_{\mathcal{A}}^{i+1}$.

It remains to show that there exists some i such that $\sim_{\mathcal{A}}^i = \sim_{\mathcal{A}}$. If p is a context, $|p|_*$ denotes the depth of the hole in p (starting from 0 if the hole is a root) and ∇p denotes the maximum number of siblings in p. If $p \cdot t_s \in L \Leftrightarrow p \cdot t_{s'} \in L$ holds for all contexts p

such that $\nabla p \leqslant |S|$ and $|p|_* \leqslant |Q|$, it also holds for all contexts. If i is sufficiently large, $\mathcal{C}_s^i$ and $\mathcal{C}_q^i$ contain all contexts such that $\nabla p \leqslant |S|$ and $|p|_* \leqslant |Q|$, hence $\sim_{\mathcal{A}}^i = \sim_{\mathcal{A}}$. This also shows that the loop on line 3 is run at most $\max(|S|, |Q|)$ times. ♢

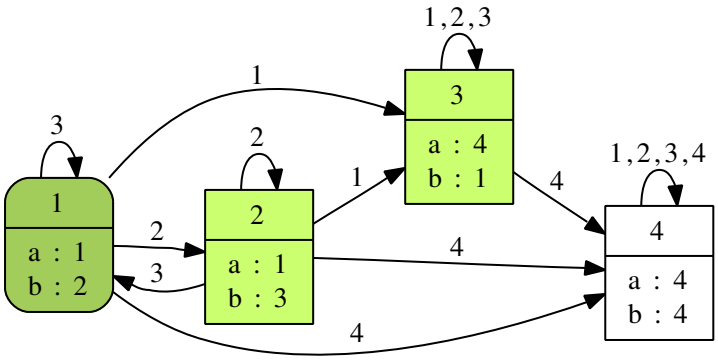


Figure 8: A non minimal BUDFA

Example 3. The automaton in figure 8 recognizes forests over $\{a, b\}$ in which no pair of nodes labelled by a are in a same path. The following table shows the relation $\sim^i$ at each step of the algorithm:

	S	Q
$\sim^0$	1 2 3 | 4	1 2 3 | 4
$\sim^1$	1 2 | 3 | 4	1 | 2 3 | 4
$\sim_{\mathcal{A}}$	1 2 | 3 | 4	1 | 2 3 | 4

At step 1, the S-pairs $\{1,3\}$ and $\{2,3\}$ are marked because $\lambda(1,a) = \lambda(2,a) = 1$, $\lambda(3,a) = 4$ and the Q-pair $\{1,4\}$ is marked. Then, the Q-pairs $\{1,2\}$ and $\{1,3\}$ are marked because $\gamma(1,1) = 3$, $\gamma(1,3) = 1$ and the S-pair $\{1,3\}$ is marked; $\gamma(2,1) = 3$, $\gamma(2,2) = 2$ and the S-pair $\{2,3\}$ is marked. No other pair is marked at the next step, hence only S-states 1 and 2 and Q-states 2 and 3 are equivalent.

Algorithm 2 Computing $\sim_A$ by refinement

1: $\mathcal{R}_s := \{F, F^c\}$, $\mathcal{R}_q := \{Q\}$
2: **repeat**
3: $\quad \mathcal{R}_s := \mathcal{R}'_s$, $\mathcal{R}_q := \mathcal{R}'_q$
4: $\quad$ Define $\mathcal{R}'_s$ by $s\mathcal{R}'_s s'$ if $s\mathcal{R}_s s'$ and
5: $\quad\quad$ **For all** $q \in Q$, $a \in A$, $\lambda(s,a)\mathcal{R}_q\lambda(s',a)$ and $\gamma(s,q)\mathcal{R}_s\gamma(s',q)$
6: $\quad$ Define $\mathcal{R}'_q$ by $q\mathcal{R}'_q q'$ if $q\mathcal{R}_q q'$ and
7: $\quad\quad$ **For all** $s \in S$, $\gamma(s,q)\mathcal{R}_s\gamma(s,q')$
8: **until** $\mathcal{R}'_s = \mathcal{R}_s$ and $\mathcal{R}'_q = \mathcal{R}_q$
9: **return** $\mathcal{R}_s$, $\mathcal{R}_q$

Although algorithm 1 is easy to prove and well-suited for minimization by hand, it is much too slow. By splitting classes instead of marking pairs, it is easy to build

an equivalent algorithm that runs in quadratic time. This variant is presented in algorithm 2; it is clear that $s\mathcal{R}'_s s'$ at step i of algorithm 2 if and only if the pair $\{s, s'\}$ was not marked at step i of algorithm 1. Quadratic complexity can be achieved by using a data structure such that splitting a $\mathcal{R}_s$-class B can be done in $O((|A|+|Q|)|B|)$ and splitting a $\mathcal{R}_q$-class C can be done in $O(|S||C|)$. Implementation details can be found in papers focusing on the word case, e.g. [3].

Algorithm 3 Generalized Hopcroft minimization

$LS := \emptyset$, $LQ := \emptyset$
$PS := \{F, F^c\}$, $PQ := \{Q\}$
$ADD(LS, \min(F, F^c))$
while $LS \neq \emptyset$ or $LQ \neq \emptyset$ **do**
  if $LQ = \emptyset$ **then**
    $C := EXTRACT(LS)$
    for all $B \in PS$ **do**
      for all $q \in Q$ **do**
        $D := \gamma_q^{-1}(C) = \bigcup_{s' \in C} \{s \in S \mid \gamma(s, q) = s'\}$
        $C0 := B \cap D$; $C1 := B \cap D^c$
        if $\min(C0, C1) \neq \emptyset$ **then**
          $REPLACE(PS, B, \max(C0, C1))$
          $ADD(PS, \min(C0, C1))$; $ADD(LS, \min(C0, C1))$
        end if
      end for
    end for
    for all $B \in PQ$ **do**
      for all $s \in S$ **do**
        $D := \gamma_s^{-1}(C) = \bigcup_{s' \in C} \{q \in Q \mid \gamma(s, q) = s'\}$
        $C0 := B \cap D$; $C1 := B \cap D^c$
        if $\min(C0, C1) \neq \emptyset$ **then**
          $REPLACE(PQ, B, \max(C0, C1))$
          $ADD(PQ, \min(C0, C1))$; $ADD(LQ, \min(C0, C1))$
        end if
      end for
    end for
  else
    $C := EXTRACT(LQ)$
    for all $B \in PS$ **do**
      for all $a \in A$ **do**
        $D := \lambda_a^{-1}(C) = \bigcup_{q \in C} \{s \in S \mid \lambda(s, a) = q\}$
        $C0 := B \cap D$; $C1 := B \cap D^c$
        if $\min(C0, C1) \neq \emptyset$ **then**
          $REPLACE(PS, B, \max(C0, C1))$
          $ADD(PS, \min(C0, C1))$; $ADD(LS, \min(C0, C1))$
        end if
      end for
    end for
  end if
end while
return PS, QS

Proposition 4. Given a trimmed BUDFA $\mathcal{A}$, algorithm 3 computes the relation $\sim_{\mathcal{A}}$ and can be implemented to run in $O(|\mathcal{A}| \log |\mathcal{A}|)$ time.

Proof: We first show the algorithm is correct. Notice that in algorithm 1, $\sim^i \neq \sim^{i+1}$ if and only if at least one of the following statement is true:

1. $\exists B, C \in S/\sim^i, \exists q \in Q, \gamma(B,q) \cap C \neq \emptyset$ and $\gamma(B,q) \not\subseteq C$;
2. $\exists B \in Q/\sim^i \exists C \in S/\sim^i, \exists s \in S, \gamma(s,B) \cap C \neq \emptyset$ and $\gamma(s,B) \not\subseteq C$;
3. $\exists B \in S/\sim^i \exists C \in Q/\sim^i, \exists a \in A, \lambda(B,a) \cap C \neq \emptyset$ and $\lambda(B,a) \not\subseteq C$.

We say that (C,q), (C,s) and (C,a) are *splitters* for the conditions 1,2 and 3. We also define the refinements w.r.t. a splitter set for each condition:

- If $B, (C,q)$ satisfy 1, let $B_{(C,q)} = B \cap \gamma_q^{-1}(C)$ and $B^{(C,q)} = B \setminus B_{(C,q)}$
- If $B, (C,s)$ satisfy 2, let $B_{(C,s)} = B \cap \gamma_s^{-1}(C)$ and $B^{(C,s)} = B \setminus B_{(C,s)}$
- If $B, (C,a)$ satisfy 3, let $B_{(C,a)} = B \cap \lambda_a^{-1}(C)$ and $B^{(C,a)} = B \setminus B_{(C,a)}$

We also use the notation B' and B'' for the refinements of B if there is no ambiguity on the splitter. If B is a class of horizontal states of $\sim$ a partition of S and Q, we denote $spl(B,\sim)$ the set of splitters of B in $\sim$, i.e. the set of all splitters (C,s) and (C,a) such that 1 or 3 holds. Similarly, if B is a class of vertical states $spl(B,\sim)$ is the set of splitters (C,q) such that 2 holds. The repeat loop of algorithm 2 can be restated as follows: for each class B of $\mathcal{R}_s$ and $\mathcal{R}_q$ and for all splitter $(C,x) \in spl(B, \mathcal{R}_s \cup \mathcal{R}_q)$, split B into $B_{(C,x)}$ and $B^{(C,x)}$.

In algorithm 3, we use a dual approach focusing on the splitters. Like in [3], we define *objects of refinement in* $\sim$ by $obj(C,x,\sim) = \{B \in \sim |\ (C,x) \in spl(B,\sim)\}$. Because of the above remark, the following algorithm computes $\sim_{\mathcal{A}}$:

1: $\sim := \{F, F^c\} \uplus \{Q\}$
2: **while** $\exists (C,x)$ with $C \in \sim$ and $x \in S \uplus Q \uplus A$ such that $obj(C,x,\sim) \neq \emptyset$ **do**
3: **for all** $B \in obj(C,x,\sim)$ **do**
4: Replace B with $B_{(C,x)}$ and $B^{(C,x)}$ in $\sim$
5: **end for**
6: **end while**

This is in substance algorithm 3, except for how we look for splitters (C,x) such that $obj(C,x,\sim) \neq \emptyset$. Notice that if we refine a class B into $B_{(C,x)}$ and $B^{(C,x)}$, then for all $D \subseteq B_{(C,x)}$ or $D \subseteq B^{(C,x)}$, $D \notin obj(C,x,\sim)$. In other words, once a splitter is used to refine a class, it is not a splitter of any subset of the refined class. Furthermore, the sets of splitters $\{(B,x),(B',x)\}$, $\{(B,x),(B'',x)\}$ and $\{(B',x),(B'',x)\}$ all lead to the same refinement of $\sim$. We can then use a set L of candidate splitters. A candidate splitter can only be added once to L, and if it splits a class B, then it suffices to add one of (B',x) or (B'',x) to L. To minimize the number of candidate splitters to test we always chose to add the smallest possible candidate.

In algorithm 3, instead of using splitters (C,x), we only add the set C to the waiting set and we test all possible (C,x) when processing C. This does not change the complexity in the worst case, and the algorithm becomes more concise. Otherwise, this is exactly what we described above, but with proper distinction of the 3 different kinds of splitters. This concludes the corectness proof.

Regarding the complexity, we rely on the fact that for each respective kind of splitter (C,q), (C,s) and (C,a), it is known (from the word case) that it suffices to process

respectively $O(|Q|\log|S|)$, $O(|S|\log|Q|)$ and $O(|A|\log|S|)$ candidates to fully refine $\sim$. Using appropriate data structures, one can implement the algorithm such that each kind of splitter can be processed in time respectively $O(|\gamma_q^{-1}(C)|)$, $O(|\gamma_s^{-1}(C)|)$ and $O(|\lambda_a^{-1}(C)|)$. The total complexity of the algorithm is then $O(|\mathcal{A}|\log|\mathcal{A}|)$. $\diamondsuit$

2.4 BUNFA and determinization

Definition 9. *A* bottom-up non-deterministic forest automaton *(BUNFA) is a tuple:*

$$\mathcal{A} = \langle S, Q, I \subseteq S, \gamma : S \times Q \to 2^S, \lambda : S \times A \to 2^Q, F \subseteq S \rangle$$

γ and λ are extended into maps respectively from $2^S \times 2^Q$ to 2^S and from $2^S \times A$ to 2^Q by:

$$\begin{aligned} \gamma(T,R) &= \bigcup_{s\in T}\bigcup_{q\in R}\gamma(s,q) \\ \lambda(T,a) &= \bigcup_{s\in T}\lambda(s,a) \end{aligned}$$

A BUNFA defines a map from $\mathcal{F}(A)$ to 2^S, by induction on the structure of forests:

- $0^{\mathcal{A}} = I$
- $(t_1 + a \cdot t_2)^{\mathcal{A}} = \gamma(t_1^{\mathcal{A}}, \lambda(t_2^{\mathcal{A}}, a))$

A forest t is accepted by $\mathcal{A}$ if $t^{\mathcal{A}} \cap F \neq \emptyset$.

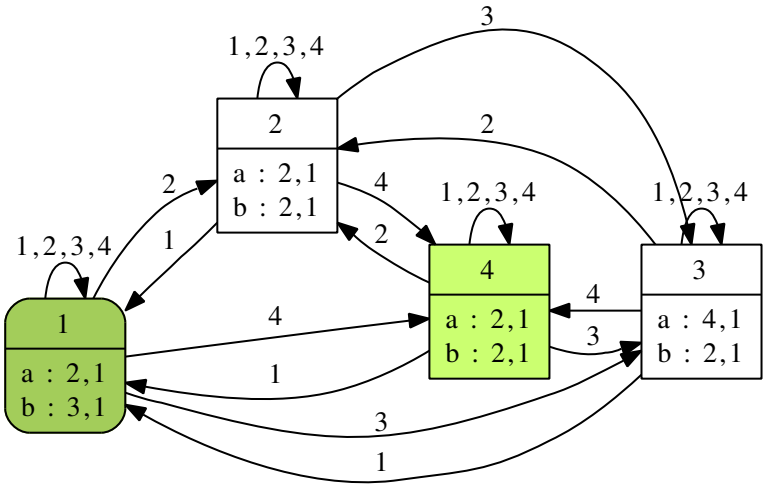


Figure 9: A BUNFA

Example 4. The automaton in figure 9 recognizes forests over $\{a, b\}$ containing a path from a leaf to the root that ends in *ab*. It was generated from an automaton accepting the word *ab*.

Proposition 5. Given a BUNFA $\mathcal{A}$, algorithm 4 computes a BUDFA that accepts the same language as $\mathcal{A}$.

Algorithm 4 Determinization algorithm

1: $Q' := \emptyset$
2: **for all** $s \in S$, $a \in A$ **do**
3: $\quad q' := \lambda(s, a)$
4: $\quad$ **if** $q' \notin Q'$ **then**
5: $\quad\quad ADD(Q', q')$
6: $\quad\quad$ **for all** $q'' \in Q'$ **do**
7: $\quad\quad\quad$ **if** $q' \cup q'' \notin Q'$ **then**
8: $\quad\quad\quad\quad ADD(Q', q' \cup q'')$
9: $\quad\quad\quad$ **end if**
10: $\quad\quad$ **end for**
11: $\quad$ **end if**
12: **end for**
13: $L := \{I\}$, $S' := \{I\}$
14: **repeat**
15: $\quad s' := EXTRACT(L)$
16: $\quad$ **for all** $q' \in Q'$ **do**
17: $\quad\quad s'' := \bigcup_{s \in s'} \bigcup_{q \in q'} \gamma'(s, q)$
18: $\quad\quad$ **if** $s'' \notin S'$ **then**
19: $\quad\quad\quad ADD(S', s'')$, $ADD(L, s'')$
20: $\quad\quad$ **end if**
21: $\quad$ **end for**
22: **until** $L = \emptyset$
23: Define $\gamma' : \begin{array}{ccc} S' \times Q' & \longrightarrow & S' \\ (s', q') & \longmapsto & \bigcup_{s \in s'} \bigcup_{q \in q'} \gamma(s, q) \end{array}$
24: Define $\lambda' : \begin{array}{ccc} S' \times A & \longrightarrow & Q' \\ (s', a) & \longmapsto & \bigcup_{s \in s'} \lambda(s, a) \end{array}$
25: **return** $(S', Q', I, \gamma', \lambda', \{s' \in S' \mid s' \cap F \neq \emptyset\})$

Proof: First, we remark that the BUDFA $\mathcal{A}_p = (2^S, 2^Q, I, \gamma, \lambda, F_p)$ where $F_p = \{P \subseteq S \mid P \cap F \neq \emptyset\}$ recognizes the same language as $\mathcal{A}$, but it requires the computation of a huge number of non-reachable states.

There are different approaches to reduce the number of useless states created during the determinization process. In this algorithm, we first determinize λ: we create a new vertical state for each subset of Q that appears in λ. Additionally, we create new vertical states for any union of such subsets. It is not guaranteed that those states are reachable, but their usefulness will be explained below.

The next step is to determinize the extended horizontal NFA (S, Q', γ, I, F), using the standard determinization procedure. We obtain a DFA (S', Q', γ', I, F'). But it remains to extend the output function λ to $S' \times A$. Because we had created vertical states for all the unions of $\lambda(s, a)$ for all $s \in S, a \in A$, we know that for all $s' \in S', a \in A$, $\bigcup_{s \in s'} \lambda(s, a) \in Q'$, hence λ can be extended into $\lambda' : S' \times A \to Q'$ and $(S', Q', I, \gamma', \lambda', F')$ is a BUDFA that accepts the same language as $\mathcal{A}$. $\diamondsuit$

Instead of creating vertical states for the unions of all $\lambda(s, a)$ for all $s \in S, a \in A$, we could have restarted the determinization process on the automaton $(S', Q', I, \gamma', \lambda : S' \times Q' \to 2^{Q'}, F')$. This approach does yield a determinized automaton, but it can have more than 2^S horizontal states, most of which are either equivalent of non-

reachable. Because $|S| \gg |Q|$ in all practical cases, the cost of creating more vertical states than necessary is offset by the reduction of the number of horizontal states.

2.5 Computing the syntactic forest algebra of a language

In the case of word languages, it is well known that the syntactic monoid is isomorphic to the transition monoid of the minimal automaton accepting the language. In this section, we show how this result can be extended to forest algebras.

$\mathcal{A}_L = \langle \mathcal{F}(A)/\sim_L, \mathcal{T}(A)/\equiv_L, [0]_{\sim_L}, \gamma, \lambda, [L]_{\sim_L} \rangle$ denotes the minimal BUDFA of a recognizable forest language L, and its transition forest algebra is defined as follows:

- The horizontal monoid is $H = (\langle \gamma_{[t]_{\equiv_L}} \mid t \in \mathcal{T}(A) \rangle, \mathrm{id}, *)$, where

$$\gamma_{[t_2]_{\equiv_L}}([t_1]_{\sim_L}) = [t_1 + t_2]_{\sim_L}$$

  for all $[t_1]_{\sim_L} \in \mathcal{F}(A)/\sim_L$, $[t_2]_{\equiv_L} \in \mathcal{T}(A)/\equiv_L$
- The vertical monoid V is the submonoid of $(H^H, \mathrm{id}, \circ)$ generated by rootings and insertions, as explained below.
- The action is function application.

To simplify notation, we will write γ_t instead of $\gamma_{[t]_{\equiv_L}}$ if $t \in \mathcal{T}(A)$. Furthermore, we extend this notation to forests as follows:

- If t is the empty forest, then γ_t is id.
- If $t = as$, let $\gamma_t = \gamma_{\lambda(\gamma_s([0]_{\sim_L}),a)}$.
- Otherwise, there exists some $n \in \mathbb{N}^*$ such that $t = t_1 + \cdots + t_n$ where $t_1, \ldots, t_n \in \mathcal{T}(A)$. We then define $\gamma_t = \gamma_{[t_1]_{\equiv_L}} * \cdots * \gamma_{[t_n]_{\equiv_L}}$.

Notice that if $t_1, t_2 \in \mathcal{T}(A)$ and $[t]_{\sim_L} \in \mathcal{F}(A)/\sim_L$,

$$\begin{aligned} \gamma_{t_1} * \gamma_{t_2}([t]_{\sim_L}) &= \gamma_{t_2}(\gamma_{t_1}([t]_{\sim_L})) \\ &= \gamma_{t_2}([t + t_1]_{\sim_L}) \\ &= [t + t_1 + t_2]_{\sim_L} \end{aligned}$$

Hence, for any forest t_2, $\gamma_{t_2}([t_1]_{\sim_L}) = [t_1 + t_2]_{\sim_L}$.

Lemma 3. Let $t_1, t_2 \in \mathcal{F}(A)$. Then $[t_1]_{\equiv_L} = [t_2]_{\equiv_L}$ if and only if $\gamma_{t_1} = \gamma_{t_2}$.

Proof: It suffices to show that $t_1 \equiv_L t_2$ if and only if for all $t \in \mathcal{F}(A)$, $[t + t_1]_{\sim_L} = [t + t_2]_{\sim_L}$. The "if" part is obvious since $\equiv_L \subseteq \sim_L$, while the "only if" part was already proven to show that the minimal automaton is well defined. ♢

We will now build a similar correspondence between contexts and elements of V. For $a \in A$, let

$$\begin{aligned} & & H &\longrightarrow H \\ v_a &: & \gamma_s &\longmapsto \gamma_{as} \\ v_{\mathrm{in_L}(t)} &: & \gamma_s &\longmapsto \gamma_{t+s} \\ v_{\mathrm{in_R}(t)} &: & \gamma_s &\longmapsto \gamma_{s+t} \end{aligned}$$

It is easy to see that any context can be uniquely decomposed (up to commutation of left and right insertions) in a product of factors of one of the following forms: $a(*)$,

$\mathrm{in_L}(t)$ and $\mathrm{in_R}(t)$ for some $t \in \mathcal{F}(A)$. For $p \in \mathcal{C}(A)$, we define v_p by induction on the previous decomposition. The basis cases are already handled; if $p = p_1 \circ p_2$, let $v_p = v_{p_1} \circ v_{p_2}$. Notice that $v_{\mathrm{in_L}(t)}$ and $v_{\mathrm{in_R}(t')}$ commute for all $t, t' \in \mathcal{F}(A)$. Using these notations, V is exactly $\langle v_a, v_{\mathrm{in_L}(t)}, v_{\mathrm{in_R}(t)} \mid a \in A, t \in \mathcal{F}(A)\rangle$.

Lemma 4. Let $p_1, p_2 \in \mathcal{C}(A)$. Then $[p_1]_{\equiv_L} = [p_2]_{\equiv_L}$ if and only if $v_{p_1} = v_{p_2}$.

Proof: We first show by induction on $p \in \mathcal{C}(A)$ that $\forall t \in \mathcal{F}(A), \gamma_{p\cdot t} = v_p(\gamma_t)$:

- If $p = *$, this is obvious.
- If $p = \mathrm{in_L}(t')$, $\gamma_{p\cdot t} = \gamma_{t'+t} = v_{\mathrm{in_L}(t')}(\gamma_t)$. The case $p = \mathrm{in_R}(t')$ is similar.
- If $p = a(*)$, $v_p(\gamma_t) = \gamma_{\lambda(\gamma_t([0]_{\sim_L}),a)} = \gamma_{at}$.
- If $p = p_1 \circ p_2$, $\gamma_{(p_1\circ p_2)\cdot t} = \gamma_{p_1\cdot(p_2\cdot t)} = v_{p_1}(v_{p_2}(\gamma_t))$ by induction hypothesis, while $v_{p_1\circ p_2} = v_{p_1} \circ v_{p_2}$.

$v_{p_1} = v_{p_2}$ if and only if for all $t \in \mathcal{F}(A), v_{p_1}(\gamma_t) = v_{p_2}(\gamma_t)$, i.e. $\gamma_{p_1\cdot t} = \gamma_{p_2\cdot t}$.
From the previous lemma, $\gamma_{p_1\cdot t} = \gamma_{p_2\cdot t}$ if and only if, $p_1 \cdot t \equiv_L p_2 \cdot t$. ♢

The two previous lemmas can be combined into

Theorem 1. *The syntactic forest algebra and the transition forest algebra of the minimal automaton of a forest language L are isomorphic. The isomorphism of forest algebras is given by:*

$$(\alpha, \beta) : \begin{array}{ccc} (H^L, V^L) & \longrightarrow & (H, V) \\ ([t]_{\equiv_L}, [p]_{\equiv_L}) & \longmapsto & (\gamma_t, v_p) \end{array}$$

Proof: We have already seen that α and β are two well-defined monoid isomorphisms. It remains to verify that $\alpha([p]_{\equiv_L} \cdot [t]_{\equiv_L}) = \beta([p]_{\equiv_L})\alpha([t]_{\equiv_L})$, in other words that $\gamma_{p\cdot t} = v_p(\gamma_t)$. This was already proven in the previous lemma. ♢

Remark 7. In the above construction, assuming $|S| = n$, since H is the transition monoid of the horizontal DFA of $\mathcal{A}$, $|H| \leqslant n^n$ and this bound is tight. However, when considering the size of a forest algebra, only the size of the vertical monoid is important as it contains the horizontal monoid. But since V is a submonoid of H^H, it can potentially have up to $n^{n^{n+1}}$ elements.

In fact, there is a much lower bound. Assume $S = \{0, \ldots, n-1\}$ and $s_0 = 0$. Let $I = \langle v_{\mathrm{in_L}(t)}, v_{\mathrm{in_R}(t)} \mid t \in \mathcal{F}(A)\rangle$. Since left and right insertions commute, there is an injective map $I \to \langle v_{\mathrm{in_L}(t)}\rangle \times \langle v_{\mathrm{in_R}(t)}\rangle$, hence $|I| \leqslant |H|^2 \leqslant n^{2n}$.

Let $R = \{F \in H^H \mid F(\gamma)$ only depends on the value of $\gamma(0)\}$. If $F \in R$, F is essentially a map from S to H, hence $|R| \leqslant (n^n)^n = n^{n^2}$. Furthermore, R is a left ideal of V that contains all rootings and products of rootings.

Let $F \in V$. Either F contains no rooting, or it can be written $F' \circ v_a \circ J$ with $J \in I$ (J can be the insertion of an empty forest). Since $F' \circ v_a \in R$, it follows that:

$$|V| \leqslant n^{2n}(1 + n^{n^2})$$

Although we believe $|V| = \Omega(n^{n^2})$ in the worst case, we do not think the above bound is tight and a more detailed analysis of $|V|$ remains to be conducted.

2.6 Logical characterization

It is well-known that a word language is accepted by a finite state automaton if and only if it can be defined by some monadic second order (MSO) sentence with a binary successor predicate on the positions of letters and a family of letter-testing unary predicates.

In this section, we show that a forest language is recognizable if and only if it can be defined by a MSO sentence with two binary predicates (next-sibling and child) and a collection of unary label-testing predicates.

First, we define the syntax of first-order formulas over this signature. We will denote this logic $FO[N, C]$:

$$\varphi ::= x = y \mid N(x, y) \mid C(x, y) \mid L_a(x), a \in A \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x\varphi$$

We will also use the usual abbreviations $\varphi \vee \varphi, \varphi \to \varphi$ and $\forall x\varphi$. We interpret $FO[N, P]$ formulas as follows: we consider a forest t over the alphabet A to be a map from a prefix-closed domain $\mathcal{P}\text{os}(t) \subseteq \mathbb{N}^*$ to A. The domain of the interpretation $\mathcal{I}$ is $\mathbb{N}^*$. The interpretation of N and C are given by:

$$\begin{array}{lll} \forall p, p' \in \mathbb{N}^*, N^{\mathcal{I}}(p, p') & \text{if and only if} & \exists p'' \in \mathbb{N}^*, \exists i \in \mathbb{N}, p = p'' \cdot i \wedge p' = p'' \cdot (i+1) \\ \forall p, p' \in \mathbb{N}^*, C^{\mathcal{I}}(p, p') & \text{if and only if} & \exists i \in \mathbb{N}, p = p' \cdot i \end{array}$$

To deal with free variables occuring in the formula, we define $\mathcal{V}$-forests over A as forests t over the extended alphabet $A \times 2^{\mathcal{V}}$ given a finite set $\mathcal{V}$ of first-order variables such that if $\{(a_i, U_i), i \leqslant n\}$ denotes the set of labels in t, $U_i \cap U_j = \emptyset$ for all $i \neq j$ and $\bigcup_{i \leqslant n} U_i = \mathcal{V}$.

Let $\varphi \in FO[N, C]$ be a first-order formula, $\mathcal{V}$ denotes the set of free variables in φ. We assume bound variables are pairwise distinct and do not appear in $\mathcal{V}$. If t is a $\mathcal{V}$-forest, we define $t \models_{\mathcal{I}} \varphi$ (read t is a model of φ or t satisfies φ with respect to the interpretation $\mathcal{I}$) by induction on the syntax of φ:

- $t \models_{\mathcal{I}} L_a(x)$ if and only if $\exists p \in \mathcal{P}\text{os}(t), t(p) = (a, U)$ with $x \in U$.
- If P is a binary predicate, $t \models_{\mathcal{I}} P(x, y)$ if and only if $P^{\mathcal{I}}(p_x, p_y)$, where p_x is the position such that $t(p_x) = (a_i, U_i)$ with $x \in U_i$.
- $t \models_{\mathcal{I}} \varphi_1 \wedge \varphi_2$ if and only if $t \models_{\mathcal{I}} \varphi_1$ and $t \models_{\mathcal{I}} \varphi_2$.
- $t \models_{\mathcal{I}} \neg\varphi$ if and only if $t \not\models_{\mathcal{I}} \varphi$.
- $t \models_{\mathcal{I}} \exists x\varphi$ if and only if $\exists i \leqslant n, t_i \models_{\mathcal{I}} \varphi$ where t_i is the $\mathcal{V} \cup \{x\}$-forest obtained by replacing the label (a_i, U_i) in t by $(a_i, U_i \cup \{x\})$.

We define the language accepted by φ as:

$$L_\varphi = \{t \in \mathcal{F}(A \cup \mathcal{V}(\varphi)) \mid t \models_{\mathcal{I}} \varphi\}$$

where $\mathcal{V}(\varphi)$ denotes the set of free variables of φ. If φ has no free variable, we say that φ is a sentence and the language accepted by φ is a subset of $\mathcal{F}(A)$.

We now define the syntax of the monadic second-order logic fragment $MSO[N, C]$:

$$\Phi ::= \varphi \in FO[N, C] \mid X(x) \mid \exists X\Phi$$

Uppercase letters $X, Y \ldots$ are used to denote second-order variables. We define the semantic of $MSO[N, C]$ formulas on $(\mathcal{V}, \mathcal{W})$-forests, i.e. forests t over the extended

alphabet $A \times 2^{\mathcal{V}} \times 2^{\mathcal{W}}$ such that the restriction of t to labels in $A \times 2^{\mathcal{V}}$ is a $\mathcal{V}$-forest and $\mathcal{W}$ is a finite set of second-order variables. Our definition of satisfiability is extended as follows:

- $t \models_{\mathcal{I}} \varphi \in FO[N, C]$ if and only if the restriction of t to labels in $A \times 2^{\mathcal{V}}$ is a model of φ.
- $t \models_{\mathcal{I}} X(x)$ if and only if there exists a position $p \in \mathcal{P}\text{os}(t)$ such that $t(p) = (a_i, V_i, W_i)$ with $x \in V_i$ and $X \in W_i$.
- $t \models_{\mathcal{I}} \exists X \Phi$ if and only if there exists a set of positions $P \subseteq \mathcal{P}\text{os}(t)$ such that the $(\mathcal{V}, \mathcal{W} \cup \{X\})$-forest obtained by replacing all labels $t(p) = (a_p, V_p, W_p)$ for $p \in P$ by $(a_p, V_p, W_p \cup \{X\})$ satisfies Φ.

We also define the language accepted by $\Phi \in MSO[S, N]$ to be the set of $\mathcal{V}(\Phi), \mathcal{W}(\Phi)$-forests that satisfy Φ, where $\mathcal{W}(\Phi)$ denotes the set of free second-order variables in Φ. It is a forest language if Φ is a sentence. Just like the class of regular language is equal to the class of $MSO[S]$-definable languages, a forest language is regular if and only if it is $MSO[N, C]$-definable. The proof of this fact, adapted from [4], is almost the same for words and forests.

Proposition 6. If the BUDFA $\mathcal{A} = (S, Q, s_1, \gamma, \lambda, F)$ accepts the forest language L, there exists a $MSO[N, C]$ sentence Φ such that $L = L_\Phi$.

Proof: We will build a formula that states there exists an accepting run of the automaton. Assume $|S| = n$, $|Q| = m$. We will use two families of second-order variables $(S_i)_{i \leqslant n}$ and $(Q_i)_{i \leqslant m}$. $Q_i(x)$ will be true if the subtree at position x is assigned the vertical state q_i by the automaton. $S_i(x)$ will be true if after reading the sequence of the vertical states of the left-siblings of x then the vertical state of x the horizontal DFA is in state s_i. The first step is to ensure each node gets at most one horizontal and vertical state:

$$\Phi_1 := \forall x \left(\bigwedge_{1 \leqslant i \neq j \leqslant n} S_i(x) \to \neg S_j(x) \right) \wedge \left(\bigwedge_{1 \leqslant i \neq j \leqslant m} Q_i(x) \to \neg Q_j(x) \right)$$

We will need macros to select nodes that have no left or right sibling: $last(x) := \neg \exists y, N(x, y)$, $first(x) := \neg \exists y, N(y, x)$. The next formula means the run ends in an accepting state:

$$\Phi_2 := Qx, (last(x) \wedge \neg \exists y, C(x, y)) \to \bigvee_{s_i \in F} S_i(x)$$

The quantifier Q is $\forall$ if $s_1 \in F$ and $\exists$ otherwise. The next step is to ensure that the state assignement is consistant with λ and γ:

$$\begin{aligned}
\Phi_2 &:= \forall x, \bigwedge_{\lambda(s_1, a) = q_j} (\neg \exists y, C(y, x) \wedge L_a(x)) \to Q_j(x) \\
\Phi_3 &:= \forall x, \bigwedge_{\gamma(s_1, q_i) = s_j} (first(x) \wedge Q_i(x)) \to S_j(x) \\
\Phi_4 &:= \forall x, (\exists y, N(x, y)) \to \bigwedge_{\gamma(s_i, q_j) = s_k} (S_i(x) \wedge Q_j(y)) \to S_k(y) \\
\Phi_5 &:= \forall x, (\exists y, C(x, y) \wedge last(x)) \to \bigwedge_{\lambda(s_i, a) = q_j} (S_i(x) \wedge L_a(y)) \to S_j(y)
\end{aligned}$$

The $MSO[N, C]$ sentence:

$$\Phi := \exists S_1 \cdots \exists S_n \exists Q_1 \cdots \exists Q_m \Phi_1 \wedge \Phi_2 \wedge \Phi_3 \wedge \Phi_4 \wedge \Phi_5$$

is satisfied by a forest t if and only if there exists an accepting run of $\mathcal{A}$ on t. $\diamondsuit$

Proposition 7. Given an $MSO[N, C]$ sentence Φ, there exists a BUNFA that accepts L_Φ.

Proof: We proceed by induction on Φ. Because free variables will appear in the induction, we show instead that for any formula Φ we can build an automaton that accepts the $(\mathcal{V}(\Phi), \mathcal{W}(\Phi))$-forests defined by Φ. It is easy to build a BUNFA $\mathcal{A}_v$ that accepts the language $\mathcal{L}$ of all $(\mathcal{V}(\Phi), \mathcal{W}(\Phi))$-forests.

- If $\Phi = L_a(x)$, a BUDFA with 2 horizontal and 2 vertical states can easily test if there is a label (a, V, W) with $x \in V$. The intersection with $\mathcal{A}_v$ accepts L_Φ.
- Similarly, if $\Phi = N(x, y)$, $\Phi = C(x, y)$, $\Phi = (x = y)$ or $\Phi = X(x)$ a BUDFA that accepts L_φ can be easily constructed.
- If $\Phi = \Phi_1 \wedge \Phi_2$, it is easy to build a BUDFA that accepts $L_{\varphi_1} \cap L_{\varphi_2} \cap \mathcal{L}$. Similarly, if $\Phi = \neg\Psi$, it is easy to build a BUDFA that accepts $\mathcal{L} \setminus L_\Psi$.
- If $\Phi = \exists x \Psi$, Let $(S, Q, s_0, \gamma, \lambda, F)$ be a BUDFA that accepts Ψ. We consider the BUNFA $(S \times \{0, 1\} \uplus \{s_\perp\}, Q \times \{0, 1\} \uplus \{q_\perp\}, (s_0, 0), \gamma', \lambda', F \times \{1\})$ where $\gamma'(s_\perp, *) = \gamma'(*, q_\perp) = s_\perp$, $\lambda'(s_\perp, *) = q_\perp$ and the value of γ' and λ' on non-sink states is given in the tables below:

γ'	$(s, 0)$	$(s, 1)$
$(q, 0)$	$(\gamma(s, q), 0)$	$(\gamma(s, q), 1)$
$(q, 1)$	$(\gamma(s, q), 1)$	$s_\perp$

λ'	$(s, 0)$	$(s, 1)$
(a, V, W)	$\left\{\begin{array}{c}(\lambda(s, (a, V \cup \{x\}, W)), 1) \\ (\lambda(s, (a, V, W)), 0)\end{array}\right\}$	$(\lambda(s, (a, V, W)), 1)$

  This automaton accepts $(\mathcal{V}(\Psi) \setminus \{x\}, \mathcal{W}(\Psi))$-forests t such that if x is added to one of the middle components of a label, the resulting $(\mathcal{V}(\Psi), \mathcal{W}(\Psi))$-forest is accepted by $\mathcal{A}$, i.e. it recognizes $L_{\exists x, \Psi}$.
- If $\Phi = \exists X, \Psi$, we use a similar construction on the third component of labels.

Like in the word case, each quantifier can potentially cause an exponential blowup of the number of states. $\diamondsuit$

2.7 Comparison with existing automaton models

BUDFA are extremely similar to different well-studied models of tree automata. However, slight variations in the model can cause dramatic changes to the efficiency (and in some cases, the realizability) of critical operations, particularly minimization.

In this section, we will compare BUDFA with the most widespread and well-understood unranked tree automaton models, following the survey in chapter 8 of [5]. The first such model is finite hedge automata (interestingly, the term hedge denotes what we call forests), also called unranked tree automata (UTA). A non-deterministic FHA

over A is a tuple $(Q, \delta : Q \times A \to \mathrm{Rec}(Q), F)$, where $\mathrm{Rec}(Q)$ denotes the family of regular languages over Q. A run of the NFHA is a labelling $\lambda : \mathcal{P}\mathrm{os}(t) \to Q$ such that for every $p \in \mathcal{P}\mathrm{os}(t)$, if n denotes the number of children of p, then $\lambda(p \cdot 1) \cdots \lambda(p \cdot n) \in \delta(\lambda(p), t(p))$. We will write $a(R) \to q$ instead of $\delta(q, a) = R$. In particular, a leaf labelled by a can be assigned the state q if $\varepsilon \in R$. The automaton accepts a tree if there is a run such that the root is labelled with a state in F.

Compared to BUDFA, there is a major difference: there is no direct link between the horizontal and vertical states, since the NFHA only handle the vertical aspect of the automaton, while the horizontal level is solely specified by the regular languages R. As a result, there are different kinds of FHA depending on how the regular languages of the transitions are represented. However, even if the transition languages are represented by deterministic finite state automata, and we assume determinism on the vertical level (i.e. if $a(L_1) \to q_1$ and $a(L_2) \to q_2$ then $q_1 \neq q_2$ or $L_1 \cap L_2 = \emptyset$), the resulting model of "deterministic" FHA does not have a unique minimal automaton, as shown in [6]. It is however possible to minimize the number of vertical states using the analog of the vertical syntactic congruence for trees, but the minimization of the whole representation is NP-complete.

BUDFA are almost identical to *deterministic stepwise automata* (DSA). A DSA is a tuple $(Q, A, (\mathcal{D}_a)_{a \in A}, F)$ where each $\mathcal{D}_a$ is the horizontal automaton with output $\mathcal{D}_a = (S_a, Q, s_a^{in}, \delta_a : S_a \times Q \to S_a, \lambda_a : S_a \to Q)$. It is quite easy to transform a DSA into a BUDTA (the analog of a BUDFA for trees: the automaton accepts based on vertical states instead of horizontal states). The resulting BUDTA has at most $\sum_{a \in A} |S_a|$ horizontal states and $|A| \cdot |Q|$ vertical states: Let $S = \{s_0\} \biguplus_{a \in A} S_a$, $Q' = Q \times A$, $\lambda(s \in S_a, a) = (\lambda_a(s), a)$, $\gamma(s \in S_a, (q, a)) = \delta_a(s, q)$, $\lambda(s_0, a) = (\lambda_a(s_a^{in}), a)$, $\gamma(s_0, (q, a)) = \delta_a(s_a^{in}, q)$.

It might seem that DSA are more concise than BUDTA. However, assume that $\mathcal{A}$ is a BUDTA such that $\lambda(s, a) = \lambda(s, b)$ for all $s \in S$ and $a, b \in A$. A DSA would need a total of $|S| \cdot |A|$ horizontal states to recognize the same language. More importantly, assuming the BUDTA is minimal, all those horizontal states would be non-equivalent, while when converting a DSA to a BUDFA, there is a chance that some of the added vertical states can be reduced, and it is almost certain that many horizontal states are equivalent (at least the sink states can be merged). From an algebraic point of view, adding a horizontal state means increasing the dimension of the transformation space in which the horizontal monoid of the transition forest algebra lives, while adding a vertical state adds a generator but does not increase the dimension. It is thus desirable to have as few horizontal states as possible.

The problem of DSA minimization is studied in [2], in which an analog of algorithm 1 is presented, although its proof is quite sketchy. The authors of this paper claim the algorithm can be implemented in $O(|\mathcal{A}|^2)$ while asking whether if it is possible to improve this complexity. We claim algorithm 3 can be implemented in $O(|\mathcal{A}| \log |\mathcal{A}|)$.

It is shown in [5] that a DSA can be transformed into a deterministic tree automaton on binary trees, using the external encoding: the tree $a(a_1 + \cdots + a_n)$ is encoded into $@(@(\cdots @(a, a_1) \cdots, a_{n-1}), a_n)$, where the letters are nullary symbols and $@$ is a new binary symbol. There is a similar encoding of (non empty) forests into binary trees that preserves recognizability: the forest $t = a(t_1 + \cdots + t_n)$ is encoded into $E(t) = @(+(\cdots + (E(t_1), E(t_2)) \cdots, E(t_n)), a)$, with $E(a) = a' \in A'$ a disjoint copy of A. Let $\mathcal{A} = (S, Q, s_0, \gamma, \lambda, F)$ be a BUDFA over the alphabet A. We build a deterministic tree automaton $(S \uplus Q \uplus A \uplus \bot, \Delta, F)$ over the signature $\{a : 0, a \in A, a' :$

$0, a' \in A', + : 2, @ : 2\}$. There are three kinds of rules: $a' \to \lambda(s_0, a)$ for all $a' \in A'$, $a \to a$ for all $a \in A$, $+(s, q) \to \gamma(s, q)$ for all $s \in S, q \in Q$ and $@(s, a) \to \lambda(s, a)$ for all $s \in S$, $a \in A$. The missing transitions lead to the sink state $\perp$. It is easy to see that this encoding is bijective and preserves regularity.

3 The `forestalg` GAP package

GAP [7] is a software suite for computational discrete algebra. We created a GAP package dedicated to the study of forest algebras and forest automata. The package can be downloaded from the author's webpage[1]. The reason we chose GAP is because the ultimate goal of this research is to characterize tree logics and families of regular tree languages using properties of their syntactic forest algebras, and GAP provides a large number of tools to study the properties of algebraic objects.

At the time this report was written, the package supported the following features:

1. Forest algebras: it is possible to define a forest algebra by the generators of its horizontal and vertical monoid and an explicit description of the action. Given a forest algebra, one can extract its horizontal or vertical monoid and action, compute the action of a given vertical element on an horizontal element, and test whether the horizontal or vertical monoid is commutative, aperiodic, idempotent, $\mathcal{R}$-, $\mathcal{L}$- or $\mathcal{J}$-trivial.

   It is also possible to draw the Green's relations and the Cayley graph of the horizontal and vertical monoid.

2. BUDFA: it is possible to input BUDFA and BUNFA. If the input automaton is non-deterministic, it will be transformed into a BUDFA using algorithm 4 and trimmed of its non-reachable states.

   There are functions to create two specific kinds of BUDFA: one for tree languages (given the size of the alphabet, returns a BUDFA that recognizes trees), and one for path languages (given a DFA or a rational expression, returns a BUDFA that accepts forests that contain a path labelled by a word accepted by the DFA or rational expression).

   A function is provided to test whether a given forest is accepted by a BUDFA. Given two automata, it is also possible to build a BUDFA that recognizes the union and intersection of their accepted languages.

   The package also implements the minimization algorithm 3, however because of a lack of efficient data structure for sets and partitions in GAP, the complexity of this implementation is not optimal. It is still possible to minimize BUDFA such that $|S| \cdot |Q| \leqslant 2^{16}$ in less than a minute.

   Finally, there are functions to draw BUDFA, and what we call the BUDFA action (a graph in which each node is an element of the horizontal monoid of the transition forest algebra and edges are labelled by the generators of the vertical monoid, i.e. the left and right insertions and the rootings).

Figure 10 shows a BUDFA drawn by the package and its action. For an extended description of the package and its functions, the manual is given in the appendix of this report.

[1] `http://antoine.delignat-lavaud.fr`

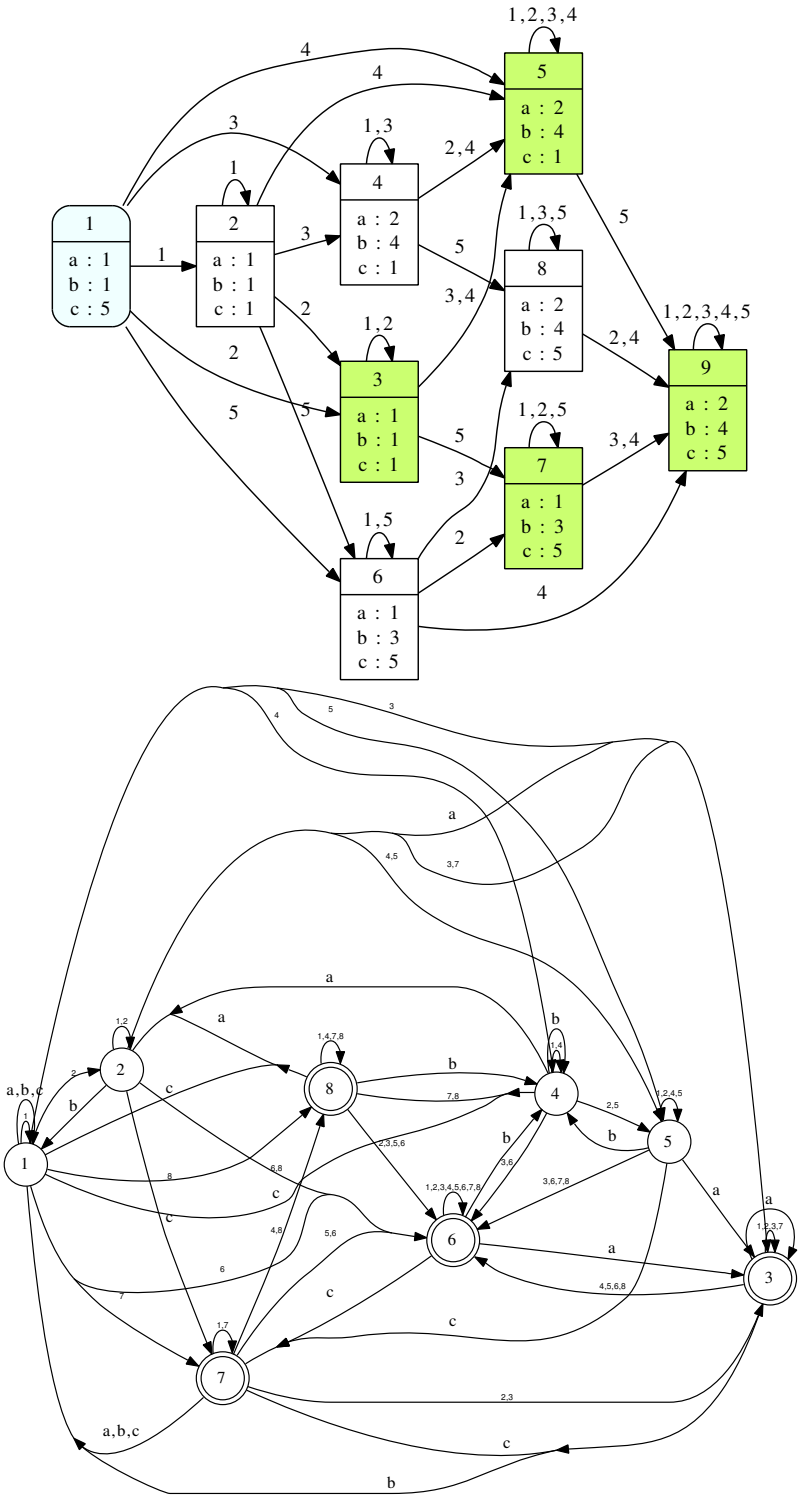


Figure 10: A BUDFA for the maximal path language $(a + c)a^+b^+$ and its action

4 Conclusion and further research

In this report, we presented what we belive to be a robust automaton model for an algebraic study of regular forest languages. Although the main feature of this model is its algebraic soundness, its behavior with respect to determinization and minimization is remarkably good. For instance, our implementation of algorithm 3 showed a loglinear behavior on all the instances we tested it with (including big automata obtained from the determinization of complex BUNFA containing many equivalent states). However, the task of analyzing the precise complexity of the algorithm using proper data structures for handling the partition and set operations remains to be performed.

Our work on automata also produced some interesting byproducts. For instance, we have seen that if a forest language is recognized by a BUDFA with n states, then its vertical syntactic monoid has $O(n^{n^2+o(n^2)})$ elements. It would be interesting to see if the given bound on $|V|$ can be improved or if we can find a family of BUDFA such that the bound is tight.

Another byproduct of this research is that for every recognizable forest language L, there exists an integer n such that H^L is isomorphic to a submonoid T_h of $\mathcal{T}_n = \{1 \dots n\}^{\{1 \dots n\}}$ and V^L is isomorphic to a submonoid T_v of $\mathcal{T}_m$ where $m = |H^L|$. There also exists a bijection $\pi : T_h \to \{1 \dots m\}$ such that the action of (T_h, T_v) is function application, i.e. $v \cdot h = \pi^{-1}(v(\pi(h)))$, in other words the action of the forest algebra is implicitly contained in the vertical monoid.

Our GAP package for forest algebras could be improved in different ways. First, it is missing a function to create an BUDFA from a given formula in $MSO[N, C]$ (or another fragment of logic, like EF or CTL). Moreover, the package is lacking specific tools to analyze forest algebras as a whole. All the provided tools work either on the horizontal or vertical monoid. The problem to solve here is what exactly should we be looking for in the forest algebra that can give us information about the recognized language? A possible lead to investigate is the action preorder and equivalence: $h_1 \preceq_V h_2$ if there exists some $v \in V$ such that $h_1 = v \cdot h_2$, and $h_1 \approx_V h_2$ if and only if $h_1 \preceq_V h_2 \preceq_V h_1$. Papers focusing on characterizing forest languages definable in a given logic, such as [8] or [9], can give indications on what to look for.

References

[1] Mikołaj Bojańczyk and Igor Walukiewicz. Forest algebras. In Jörg Flum, Erich Grädel, and Thomas Wilke, editors, *Logic and automata, history and perspectives*, volume 2 of *Texts in Logic and Games*, pages 107–131. Amsterdam University Press, 2008.

[2] Julien Cristau, Christof Löding, and Wolfgang Thomas. Deterministic automata on unranked trees. In *In Proceedings of the 15th International Symposium on Fundamentals of Computation Theory (FCT), LNCS*, pages 68–79. Springer, 2005.

[3] Timo Knuutila. Re-describing an algorithm by hopcroft. *Theor. Comput. Sci.*, 250(1-2):333–363, 2001.

[4] Howard Straubing. *Finite automata, formal logic, and circuit complexity.* Birkhauser Verlag, Basel, Switzerland, Switzerland, 1994.

[5] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[6] Wim Martens and Joachim Niehren. On the Minimization of XML-Schemas and Tree Automata for Unranked Trees. *Journal of Computer and System Science*, 73:550–583, 2007. Special issue of DBPL 05.

[7] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.12*, 2008.

[8] Michael Benedikt and Luc Segoufin. Regular tree languages definable in fo and in fomod. *ACM Trans. Comput. Log.*, 11(1), 2009.

[9] Mikolaj Bojanczyk, Howard Straubing, and Igor Walukiewicz. Wreath products of forest algebras, with applications to tree logics. In *LICS*, pages 255–263, 2009.

Appendix

Manual of the `forestalg` package

ForestAlg

Version 1.0

July, 2010

Antoine Delignat-Lavaud

Antoine Delignat-Lavaud — Email: antoine.delignat-lavaud@ens-cachan.fr
— Homepage: http://antoine.delignat-lavaud.fr/

Copyright

Acknowledgements

This package was created using the ideas of Howard Straubing, Igor Walukiwicz, Mikolaj Bojanczyk and Jean-Eric Pin, among others. It also heavily relies on packages automata and sgpviz by M. Delgado, S. Linton and J. Morais.

Colophon

This package was created during a 6-weeks internship of the author at Boston College under the supervision of Pr. Howard Straubing.

Contents

Chapter 1

Reference manual

1.1 Operations on forests

Forests are ordered collections of unranked trees over some alphabet. Concatenation of forests (putting a forest after another) is denoted additively (although it is not a commutative operation) while the a-rooting of a forest f (the tree of root a and having f as children) is denoted multiplicatively.

1.1.1 Forest

◊ `Forest(`*`Expression`*`)` (function)

Creates a forest object. `Expression` can either be a string containing the decomposition of the forest into sums and rootings, following this syntax:

```
Forest ::= a | Forest+Forest | a(Forest)
```

or it can use the internal list representation; labels are lower case characters, e.g. `'a'`, rootings are stored in list in which the first value is the label of the rooting node and the second is the list describing the rooted forest. Finally, the sum of two forests is the concatenation of the list that describe them

Example

```
gap> f := Forest("a+b(a(b)+b(a+a))+c(c(c+a)+b)");
a+b(a(b)+b(a+a))+c(c(c+a)+b)
gap> g := Forest(['b', ['a',['b','b']]]);
b+a(b+b)
```

1.1.2 \+

◊ `\+(`*`Forest1, Forest2`*`)` (function)

Returns the concatenation of `Forest1` and `Forest2` into a new forest;

Example

```
gap> f+g;
a+b(a(b)+b(a+a))+c(c(c+a)+b)+b+a(b+b)
```

1.1.3 ForestRooting

◊ ForestRooting(*Char, Forest*) (function)

Returns the rooting of Forest under the lowercase character Char.

Example

```
gap> ForestRooting('b', f);
b(a+b(a(b)+b(a+a))+c(c(c+a)+b))
```

1.1.4 WriteDotForest

◊ WriteDotForest(*Forest, File*) (function)

This will convert the forest Forest into a DOT graph file for the GraphViz library. File is the name of the output file, this function returns the temporary directory where the DOT file was written.

Example

```
gap> WriteDotForest(f, "forest.dot");
dir("/tmp/tmp.sGATtd/")
```

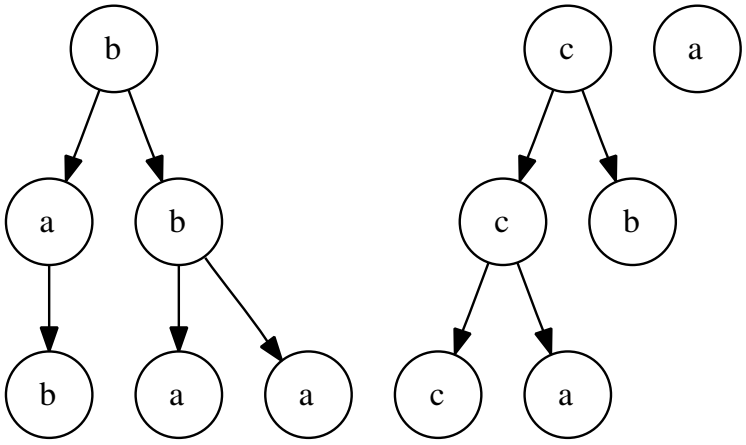


Figure 1.1: Output of the DrawForest function

1.1.5 DrawForest

◊ DrawForest(*Forest*) (function)

This will convert the forest Forest into a DOT graph file, run GraphViz on the resulting file and display the produced image in one of the PS viewer found on your system.

Example

```
gap> DrawForest(f);
```

```
Displaying file: /tmp/tmp.2eSCn4/forest.dot.ps
```

1.2 Operations on forest algebras

Forest algebras are pairs (H,V) of monoids (horizontal and vertical) with a faithful monoidal action of V on H and a pair of morphism from H to V called the left and right insertions in_L and in_R such that $\mathrm{in}_L(g)\cdot h = g+h$ and $\mathrm{in}_R(g)\cdot h = h+g$ for all $g,h\in H$. Forests algebras are created by giving the horizontal and vertical monoid, along with the action (the insertion functions are uniquely defined by the action and its axioms). The action table is only given for the generators of the vertical semigroup (recall that all semigroups in GAP are generated by some transformations).

1.2.1 ForestAlgebra

◊ `ForestAlgebra(`*`Horizontal, Vertical, Action`*`)` (function)

Creates a forest algebra structure. `Horizontal` and `Vertical` are both in the `Semigroup` category, while `Action` is a matrix with m rows and n columns where m is the number of generators of `Vertical` and n is the number of elements in `Horizontal`. If `E` denotes the list of elements of `Horizontal` and `G` the list of generators of `Vertical`, `Action[i][j]=k` if and only if the action of `G[i]` on `E[k]` is `E[k]`.

Example

```
gap> H := Monoid([Transformation([2,2,3]), Transformation([3,3,3])]);
<monoid with 2 generators>
gap> V := Monoid([Transformation([2,2,3]), Transformation([3,3,3]),
        Transformation([2,3,3])]);
<monoid with 3 generators>
gap> Elements(H);
[Transformation([1..3]), Transformation([2,2,3]), Transformation([3,3,3])]
gap> A := [[1, 2, 3 ],[2, 2, 3],[3, 3, 3],[2, 3, 3]];
gap> HV := ForestAlgebra(H, V, A);
<H has 3 generators and 3 elements, V has 4 generators>
```

1.2.2 Display

◊ `Display(`*`ForAlg`*`)` (function)

Prints the multiplication table of the horizontal and vertical monoid, then the action table of `ForestAlg`.

Example

```
gap> Display(HV);
[ [  1,  2,  3 ],
  [  2,  2,  3 ],
  [  3,  3,  3 ] ]
[ [  1,  2,  3,  4 ],
  [  2,  2,  4,  4 ],
```

```
    [  3,  3,  4,  4 ],
    [  4,  4,  4,  4 ] ]
  [ [  1,  2,  3 ],
    [  2,  2,  3 ],
    [  3,  3,  3 ],
    [  2,  3,  3 ] ]
```

1.2.3 ForestHorizontalMonoid

◊ ForestHorizontalMonoid(*ForAlg*) (function)

Returns the horizontal monoid of ForestAlg.

Example

```
gap> Elements(ForestHorizontalMonoid(HV));
[ Transformation( [ 1, 2, 3 ] ), Transformation( [ 2, 2, 3 ] ),
  Transformation( [ 3, 3, 3 ] ) ]
gap> IsCommutative(ForestHorizontalMonoid(HV));
true
```

1.2.4 ForestVerticalMonoid

◊ ForestVerticalMonoid(*ForAlg*) (function)

Returns the vertical monoid of ForestAlg.

Example

```
gap> GreensJClasses(ForestVerticalMonoid(HV));
[{Transformation([3,3,3])},{Transformation([2,3,3])},
 {Transformation([2,2,3])},{Transformation([1,2,3])}]
```

1.2.5 ForestAction

◊ ForestAction(*ForAlg*) (function)

Returns the action table of ForestAlg.

1.2.6 ForestApplyAction

◊ ForestApplyAction(*ForAlg, H, V*) (function)

Computes the action of V on H in ForestAlg. H can either be an element of the horizontal monoid of ForAlg (i.e. a transformation) or its index in the Elements of the horizontal monoid. V must be a transformation in the vertical monoid of ForAlg, this function finds a decomposition of V using generators of the vertical monoid and applies the action table on this decomposition.

Example

```
gap> Elements(ForestVerticalMonoid(HV));
[Transformation([1,2,3]), Transformation([2,2,3]), Transformation([2,3,3]),
  Transformation([3,3,3])]
gap> ForestApplyAction(HV, Transformation([1,2,3]), Transformation([2,3,3]));
Transformation([2,2,3])
```

1.2.7 IsHCommutative

◊ IsHCommutative(*ForAlg*) (function)
◊ IsVCommutative(*ForAlg*) (function)

Tests if the horizontal or vertical monoid of ForAlg is commutative.

1.2.8 IsHAperiodic

◊ IsHAperiodic(*ForAlg*) (function)
◊ IsVAperiodic(*ForAlg*) (function)

Tests if the horizontal or vertical monoid of ForAlg is aperiodic.

1.2.9 IsHIdempotent

◊ IsHIdempotent(*ForAlg*) (function)
◊ IsVIdempotent(*ForAlg*) (function)

Tests if the horizontal or vertical monoid of ForAlg is idempotent.

1.2.10 IsHRTrivial

◊ IsHRTrivial(*ForAlg*) (function)
◊ IsVRTrivial(*ForAlg*) (function)

Tests if the horizontal or vertical monoid of ForAlg is $\mathcal{R}$-trivial.

1.2.11 IsHLTrivial

◊ IsHLTrivial(*ForAlg*) (function)
◊ IsVLTrivial(*ForAlg*) (function)

Tests if the horizontal or vertical monoid of ForAlg is $\mathcal{L}$-trivial.

1.2.12 IsHJTrivial

◊ IsHJTrivial(*ForAlg*) (function)
◊ IsVJTrivial(*ForAlg*) (function)

Tests if the horizontal or vertical monoid of ForAlg is $\mathcal{J}$-trivial.

1.3 Operations on forest automata

A bottom-up deterministic forest automaton is a tuple $(S,Q,s_0,A,\gamma,\lambda,F)$ where S is a finite set of horizontal states, $s_0 \in S$ is the initial state, $F \subseteq S$ is the set of accepting states, Q is a finite set of vertical states, $\gamma : S \times Q \to S$ is a semiautomaton on set S with alphabet Q, A is the alphabet of input forests and $\lambda : S \times A \to Q$ is the vertical output function.

1.3.1 ForestAutomaton

◊ ForestAutomaton(*Det, S, Q, A, T, L, i, F*) (function)

Creates a forest automaton. Det is a boolean and must be set to true if the automaton is deterministic and false otherwise. If you input a non deterministic forest automatonm, it will be immediately transformed into an equivalent minimal deterministic automaton. S, Q and A are integers that denote respectively the number of horizontal and vertical states of the automaton and the size of the alphabet (notice that the set of horizontal states is then $\{1 \dots S\}$ and the set of vertical states is $\{1 \dots Q\}$, the distinction between each kind of states is given by the context). T is a matrix with Q rows and S columns such that T[i][j]=k if and only if $\gamma(j,i) = k$ if the automaton is deterministic, of if $\gamma(j,i) \in k$ otherwise. L is also a matrix with A rows and S columns such that L[i][j]=k if and only if $\lambda(j,a_i) = k$ if the automaton is deterministic, or if $\lambda(j,a_i) \in k$ otherwise. i is the initial state (or set of initial states if the automaton is non deterministic). F is a list of accepting states.

Example

```
gap> A := ForestAutomaton(true, 6, 3, 1,
 [[2,5,6,6,6,6],[3,6,4,6,6,6],[6,6,6,6,6,6]],
 [[1,3,3,1,2,3]],1,[2]);
<Forest automaton on {a}>
< deterministic automaton on 3 letters with 6 states >
gap> B := ForestAutomaton(true, 4, 4, 2,
 [[3,3,3,4],[2,2,3,4],[1,1,3,4],[4,4,4,4]],
 [[1,1,4,4],[2,3,1,4]],1,[1,2,3]);
<Forest automaton on {ab}>
< deterministic automaton on 4 letters with 4 states >
gap> C := ForestAutomaton(true, 3, 3, 2,
 [[1,2,3],[2,2,3],[3,3,3]], [[2,3,3],[1,2,3]],1,[1,2]);
<Forest automaton on {ab}>
< deterministic automaton on 3 letters with 3 states >
gap> ND := ForestAutomaton(false, 4, 2, 2,
  [[[1,2],,4,],[1,3,,]], [[1,,1,],[2,2,,]],1,[4]);
<Forest automaton on {ab}>
< deterministic automaton on 3 letters with 5 states >
```

1.3.2 Display

◊ Display(*ForAut*) (function)

Displays the transition table of ForAut, its initial and accepting states and output function.

Example

```
gap> Display(A);
   |  1  2  3  4  5  6
-----------------------
 1 |  2  5  6  6  6  6
 2 |  3  6  4  6  6  6
 3 |  6  6  6  6  6  6
Initial state:   [ 1 ]
Accepting state: [ 2 ]
Output function :
[ [  1,  3,  3,  1,  2,  3 ] ]
```

1.3.3 TreesForestAutomaton

◊ TreesForestAutomaton(*n*) (function)

Returns the minimal automaton that recognizes the language of all trees on an alphabet with n letters.

Example

```
gap> TreesForestAutomaton(4);
<Forest automaton on {abcd}>
< deterministic automaton on 1 letters with 3 states >
```

1.3.4 ReachableStatesForestAutomaton

◊ ReachableStatesForestAutomaton(*A*) (function)

Trims A of its non reachable states. This function is automatically called after determinization or before minimization.

1.3.5 ProductForestAutomaton

◊ ProductForestAutomaton(*A1, A2, F*) (function)

Given two forest automata A1 and A2 on the same alphabet, computes the product automaton of A1 and A2 using F as the set of accepting states. F must be a list of products of states from A1 and A2, e.g. [[1,3], [2,5]...]. This function mainly exists to provide the union and intersection operators. The resulting automaton is trimmed but not minimized; it is not minimal in general.

1.3.6 *

◊ *(*A1, A2*) (function)

Returns the minimal automaton that recognizes the intersection of the languages accepted by A1 and A2. The resulting automaton is trimmed but not minimized; it is not minimal in general.

Example

```
gap> A*TreesForestAutomaton(1);
<Forest automaton on {a}>
< deterministic automaton on 3 letters with 6 states >
```

1.3.7 \+

◊ \+(*A1, A2*) (function)

Returns the minimal automaton that recognizes the union of the languages accepted by A1 and A2. The resulting automaton is trimmed but not minimized; it is not minimal in general.

Example

```
gap> B+C;
<Forest automaton on {ab}>
< deterministic automaton on 3 letters with 3 states >
```

1.3.8 ForestAutomatonAccepts

◊ ForestAutomatonAccepts(*ForAut, For*) (function)

Returns true if ForAut accepts the forest For and false otherwise.

Example

```
gap> f := Forest("a(a(a+a)+a(a(a(a+a)+a(a+a))+a))");
a(a(a+a)+a(a(a(a+a)+a(a+a))+a))
gap> ForestAutomatonAccepts(A, f);
true
```

1.3.9 MinimalForestAutomaton

◊ MinimalForestAutomaton(*ForAut*) (function)

Computes the minimal forest automaton of ForAut. It is safe to call this function multiple times on the same automaton as it stores the result of its computation. This function does not change the name of states for automaton that are already reduced.

Example

```
gap> MinimalForestAutomaton(B);
<Forest automaton on {ab}>< deterministic automaton on 3 letters with 3 states >
gap> B;
<Forest automaton on {ab}>< deterministic automaton on 4 letters with 4 states >
```

1.3.10 TransitionForestAlgebra

◊ TransitionForestAlgebra(*ForAut*) (function)

Returns the transition forest algebra of ForAut.

Example

```
gap> HV := TransitionForestAlgebra(A);
<H has 3 generators and 5 elements, V has 4 generators>
gap> IsHAperiodic(HV);
true
```

1.3.11 SyntacticForestAlgebra

◊ SyntacticForestAlgebra(*ForAut*) (function)

Minimizes ForAut and returns its transition forest algebra, i.e. the syntactic forest algebra of the language recognized by ForAut.

Example

```
gap> HV := SyntacticForestAlgebra(B);
<H has 4 generators and 4 elements, V has 7 generators>
gap> IsHCommutative(HV);
false
```

1.3.12 HorizontalForestAutomaton

◊ HorizontalForestAutomaton(*ForAut*) (function)

Returns an Automaton object (as used by the automata package) describing the horizontal automaton of ForAut, i.e. the automaton (S, Q, γ, s_0, F).

Example

```
gap> ForestHorizontalAutomaton(C);
< deterministic automaton on 3 letters with 3 states >
```

1.3.13 ExistsPathInLanguageAutomaton

◊ ExistsPathInLanguageAutomaton(*Regular, Maximal*) (function)

Regular can either be an automaton or a rational expression describing a regular language L. Computes a forest automaton that accepts forests containing a path labelled with a word in L. Maximal is a boolean, if true such a path must be maximal, it can be any path otherwise. The labelling of a path is read from the root to the leaves. Because the resulting automaton can be huge, it is trimmed but not minimized.

Example

```
gap> AB := RationalExpression("ab");
ab
gap> Aab := ExistsPathInLanguageAutomaton(AB, false);
<Forest automaton on {ab}>
< deterministic automaton on 3 letters with 4 states >
gap> ForestAutomatonAccepts(Aab, Forest("ba"));
true
gap> ForestAutomatonAccepts(Aab, Forest("bab"));
true
gap> Aabmax := ExistsPathInLanguageAutomaton(AB, true);
<Forest automaton on {ab}>
< deterministic automaton on 3 letters with 5 states >
gap> ForestAutomatonAccepts(Aabmax, Forest("bab"));
false
```

1.3.14 WriteDotForestAutomaton

◊ WriteDotForestAutomaton(*ForAut, File*) (function)

Creates a GraphViz Dot file describing the automaton ForAut. The ouput in written in File in a temporary directory. This function returns the directory where File was written.

Example

```
gap> WriteDotForestAutomaton(A, "automaton_1");
dir("/tmp/tmp.0YxjY3/")
```

1.3.15 DrawForestAutomaton

◊ DrawForestAutomaton(*ForAut*) (function)

Uses GraphViz to draw the automaton ForAut and displays the resulting postscript file.

Example

```
gap> DrawForestAutomaton(A);
Displaying file: /tmp/tmp.sX024c/forest_automaton.dot.ps
```

1.3.16 WriteDotForestAction

◊ WriteDotForestAction(*ForAut, File[, IgnElts]*) (function)

This function computes the horizontal transformation monoid of ForAut and how the left and right insertions and rootings act on its elements. The ouput is a GraphViz Dot file written in File. This function accepts an optional argument IgnElts; it can either be a integer or a list or integers that correspond to elements of the horizontal transformation monoid (using the order given by the

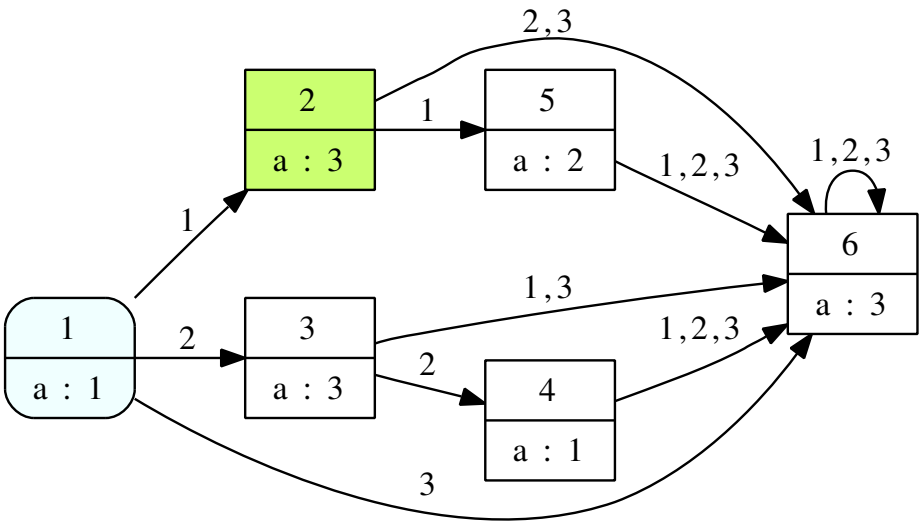


Figure 1.2: Output of the DrawForestAutomaton function

`Elements` function) to be ignored, for instance those corresponding to sink states. This function returns the temporary directory where `File` was written. *Important remark: this function does not minimize the automaton*

Example

```
gap> WriteDotForestAction(A, "action_1");
dir("/tmp/tmp.0YxjY3/")
```

1.3.17 DrawForestAutomatonAction

◊ `DrawForestAutomatonAction(`*`ForAut[, IgnElts]`*`)` (function)

Similar to `WriteDotForestAction`, but draws a postscript file of the graph and displays it.

Example

```
gap> DrawForestAutomatonAction(A);
gap> DrawForestAutomatonAction(A, 5);
gap> DrawForestAutomatonAction(B, 4);
```

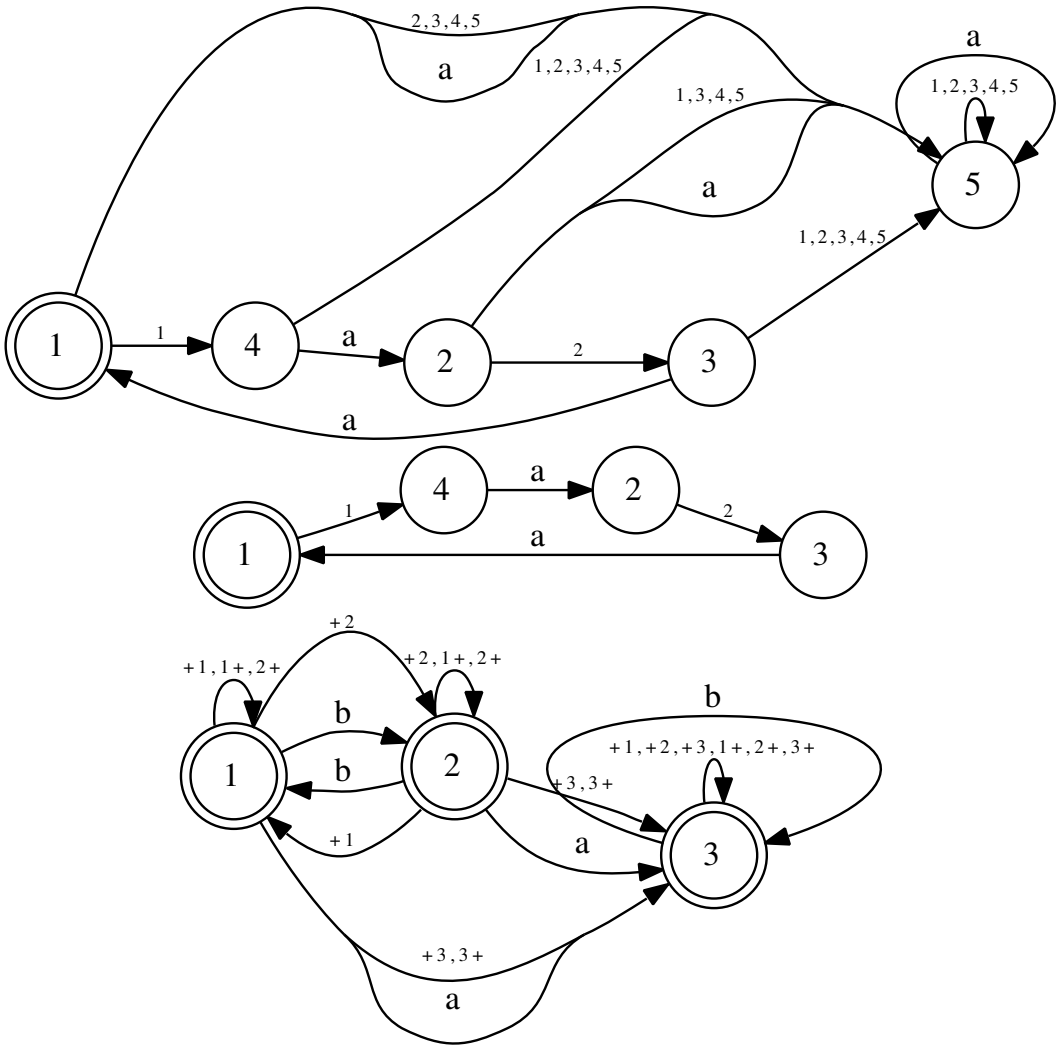


Figure 1.3: Output of the DrawForestAutomatonAction function

Index