CIT. Journal of Computing and Information Technology, Vol. 24, No. 4, December 2016, 311–322
doi: 10.20532/cit.2016.1002867

Efficient Implementation for Deterministic Finite Tree Automata Minimization

Younes Guellouma and Hadda Cherroun

Laboratoire LIM, Université Amar Telidji de Laghouat, Laghouat, Algérie

We address the problem of deterministic finite tree automata (DFTA) minimization. We describe a new alternative to implement both standard and incremental tree automata minimization using a well-defined graph representing the automaton to be minimized. We show that the asymptotic complexity of the standard implementation is linearithmic and the incremental one is $O(n^3 \log(n))$ where n is the DFTA size.

ACM CCS (2012) Classification: Theory of Computation → Formal languages and automata theory → Automata extensions

Theory of Computation → Design and analysis of algorithms → Mathematical optimization → Discrete optimization

Theory of Computation → Design and analysis of algorithms → Graph algorithms analysis

Keywords: automata theory, minimization, trees, asymptotic complexity

1. Introduction

Tree automata constitute a powerful theoretical tool for several real applications such as XML schemata [1], natural languages processing [2], verification techniques, program analysis etc. In this paper, we focus on the minimization of deterministic finite tree automata (DFTA). In the literature, all the minimization techniques [3], [4], [5] are largely inspired by finite string automata (FSA) minimization which was studied for the first time by Huffman and Moore [6]. Their algorithm was based on the definition of distinguishable pairs of states. At the end of the algorithm, all states judged undistinguishable were merged. Later, Hopcroft [7] defined a new algorithm which proceeds by refining the coarsest partition until no more refinements are possible.

Following the same steps, new minimization techniques for tree automata have emerged. Early in 1967, Brainerd [3] proposed the first DFTA minimization method which we call *standard method*. Since these several algorithms and implementations have been created, but they all followed the same approach as Brainerd's algorithm like Arbib [4], Gésceg and Steinby [5] and Comon *et al.* [8].

After that, Watson [9] designed a new minimization method called *incremental technique*, in which they use a recursive function that decides if two states are equivalent. This method constitutes the basics of many other techniques [10], [11].

Another particular case is the deterministic acyclic tree automata. It was first defined on strings [12]. Next, many other works proposed standard and incremental procedures for this case like in [13].

Concerning complexity, almost all the authors have dedicated large part to complexity study and in many cases. It was maintained by empiric tests and experimental evaluations. In cyclic case, Hopcroft minimization [14] remains the most efficient algorithm known for solving this problem for cyclic automata, the worst-case complexity of this minimization is $\mathcal{O}(a{\cdot}n{\cdot}\log(n))$ where a is the size of the alphabet and n the number of the states of the automaton to be minimized. For acyclic automata, the minimization can be done in linear time (see [15], [16]).

However, in tree case, the complexity of standard minimization needs a quadratic time. Although the numbers of works and improvements inspired by the standard method like the technique of Carrasco *et al.* [17], the whole

asymptotical complexity remains quadratic. But, Abdullah *et al.* [18] introduced a method in which they adapted Paige-Tarjan algorithm [19] to refine an equivalence relation in order to minimize a non-deterministic automata (NFTA). Since the deterministic automata is a particular case of non-deterministic automata, the last approach works well for it and gives the best known complexity in tree case which is $\mathcal{O}(r{\cdot}m{\cdot}\log(n))$ where r is the maximum rank of the alphabet, m is the size of the automaton and n is the number of its states (see [20]).

In this paper, we present an efficient implementation of DFTA using usual graph properties. We give a general algorithm for the standard minimization and we show that the asymptotic complexity of this construction reaches the best known one which can be deduced by combining Abdullah *et al.* [18] and Högberg *et al.* [20] works on NFTA. For the second, we extend materials presented in [21] and we detail a new incremental DFTA minimization approach. Then we discuss its application in cyclic and acyclic automata. We show that this new algorithm improves the complexity given by [10], [21] and gives a new complexity when dealing with the acyclic case.

The paper is organized as follows. In Section 2, we give some preliminaries and basic properties on DFTA. Afterward, we recall the standard algorithm of DFTA minimization in Section 3. Then, in Section 4 we present a graph view of a DFTA and introduce some usual properties. Moreover, we give a general algorithm for the standard minimization. Finally, Section 5 is allocated to incremental minimization of DFTA. We conclude in Section 6.

2. Preliminaries

Given a set of symbols Σ called *alphabet*, the set of unranked ordered trees T_Σ over Σ is defined inductively as follows: $\Sigma \subset T_\Sigma$ and $\forall\, \sigma \in \Sigma$, $t_1, \ldots, t_n \in T_\Sigma$, $n \in \mathbb{N} : \sigma(t_1, \ldots, t_n) \in T_\Sigma$.

An alphabet is ranked if there exists a mapping *Arity*: $\Sigma \to \mathbb{N}$ which associates to each symbol an arity representing the "fixed" number of its children in a constructed tree. $\hat{r}(\Sigma) = \max_{\sigma\in\Sigma} Arity(\sigma)$ is the maximum rank of the alphabet Σ (we use $\hat{r}$ if Σ is known). A tree set T_Σ is called "ranked" if and only if: $\forall\, \sigma(t_1, \ldots, t_n) \in T_\Sigma$, $n = arity(\sigma)$.

The subset of p-ary symbols of Σ is $\Sigma_p = \{\sigma \in \Sigma \mid Arity(\sigma) = p\}$. We use the notation $\sigma, \sigma(\,), \sigma(\,,\,), \ldots, \sigma(\,, \ldots,\,)$ respectively for constant, unary, binary, ..., p-ary symbols. The set of ranked ordered trees T_Σ over a ranked alphabet Σ is the smallest set satisfying: $p \geq 0$, $\sigma \in \Sigma_p$ and $t_1, t_2, \ldots, t_p \in T_\Sigma$ then $\sigma(t_1, t_2, \ldots, t_p) \in T_\Sigma$. A *tree language* L is a subset of T_Σ.

The set $Sub(t)$ of all *subtrees* of the tree $t = \sigma(s_1, \ldots, s_n)$ is: $Sub(t) = \{t\} \cup \bigcup_{k=1}^{n} Sub(s_k)$. The tree $t(r \leftarrow s)$ is the tree in which every occurrence of r is substituted by the tree s. We define a new substitution as follows: every tree $u \in t(r \Leftarrow s)$ is obtained by substituting just one occurrence of the tree r by s. We note that $\Leftarrow$ defines a set of trees.

Example 1. Let $t = f(a, g(b, a))$ be a tree. $t(a \leftarrow g(b)) = f(g(b), g(b, g(b)))$ but $t(a \Leftarrow g(b)) = \{f(g(b), g(b, a)), f(a, g(b, g(b)))\}$.

A *bottom up finite tree automaton* (FTA) over an alphabet Σ is a tuple $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ where Q is a finite set of *states* ($Q \cap \Sigma = \varnothing$), $Q_f \subseteq Q$ is the set of final states and $\Delta \subset \bigcup_{n\geq 0} \Sigma \times Q^n$, $n \in \mathbb{N}$ is a finite set of transitions. If Σ is ranked, then the transitions set is written as $\Delta \subset \bigcup_{n\geq 0} \Sigma \times Q^{n+1}$, $n \in \mathbb{N}$.

In what follows, we consider only the ranked alphabet. The same results can be generalized to unranked tree automata by adding some restrictions.

The size of a transition $\rho = (\sigma, q_1, \ldots, q_n, q)$, $\sigma \in \Sigma_n$, $q, q_1, \ldots, q_n \in Q$ is $|\rho| = n + 1$. Then, the size of the automaton $\mathcal{A}$ is defined as

$$|\mathcal{A}| = \sum_{\rho\in\Delta} |\rho| \tag{1}$$

We can bound the size of an automaton:

$$|\mathcal{A}| \leq \hat{r}\,|\Delta| \tag{2}$$

The *transition function* δ for a FTA is:

$$\delta : \bigcup_{n\geq 0} \Sigma_n \times Q^n \to 2^Q$$

$$\delta(\sigma, q_1, \ldots, q_n) = \{q \mid (\sigma, q_1, \ldots, q_n, q) \in \Delta\}$$

$(\sigma, q_1, \ldots, q_n)$ is called *argument*.

A FTA is a deterministic bottom-up finite tree automaton (DFTA) if $\forall\, \rho \in \bigcup_{n\geq 0} \Sigma \times Q^n : |\delta(\rho)| \leq 1$. In other words, for every transition in Δ there is at most one possible output, that is,

if $(\sigma, q_1, ..., q_n, p)$, $(\sigma, q_1, ..., q_n, q) \in \Delta$, then $p = q$. A non-deterministic FTA is denoted NFTA. We note that the function δ of a DFTA has at most one output, then it is defined on: $\delta : \bigcup_{n\geq 0} \Sigma_n \times Q^n \rightarrow Q$.

Let $\rho = (\sigma, q_1, ..., q_n)$ be an argument, then $\rho_{p:i} = (\sigma, q_1, ..., q_{i-1}, p, q_{i+1}, ..., q_n)$ denotes the argument obtained by substituting q_i by p at a precise place i in ρ.

For $t \in T_\Sigma$, the *output* $m_{\mathcal{A}}(t)$ is computed recursively as follows: Let $t = \sigma(t_1, t_2, \ldots, t_n) \in T_\Sigma$. Then:

$$m_{\mathcal{A}}(t) = \delta(\sigma, m_{\mathcal{A}}(t_1), m_{\mathcal{A}}(t_2), \ldots, m_{\mathcal{A}}(t_n)) \quad (3)$$

A tree t is accepted by $\mathcal{A}$ if and only if $m_{\mathcal{A}}(t) \in Q_f$.

The language accepted by $\mathcal{A}$ is: $L(\mathcal{A}) = \{t \in T_\Sigma \mid m_{\mathcal{A}}(t) \in Q_f\}$.

In the same way, the accepted language (down language) of a state q is defined as follows: $L^{\downarrow}(q) = \{t \in T_\Sigma \mid m_{\mathcal{A}}(t) = q\}$.

The residual (up) language of a state q is defined as follows:

$$L^{\uparrow}(q) = \bigcup_{\substack{t \in T(\Sigma) \\ s \in L^{\downarrow}(q) \\ m_{\mathcal{A}}(t) \in Q_f}} t(s \Leftarrow \#) \quad (4)$$

Mind that $\# \notin \Sigma$ is a special symbol. Every tree from $L^{\uparrow}(q)$ has a single leaf labeled #. Substituting this symbol by any tree from $L^{\downarrow}(q)$ produces a tree that is accepted by the automaton in question. A DFTA $\mathcal{A}$ is *acyclic* (ADFTA) if and only if:

$$\forall q \in Q,\ t \in L^{\uparrow}(q) \Rightarrow Sub(t) \cap L^{\downarrow}(q) = \{t\} \quad (5)$$

Recall that ADFTA satisfy some properties like the finiteness of the accepted languages, the absence of cycles (the output function m is non-recursive), etc.

A state q is *accessible* if $L^{\downarrow}(q) \neq \emptyset$ and is *co-accessible* if $L^{\uparrow}(q) \neq \emptyset$. A state is *useless* if it is neither accessible nor co-accessible. Useless states and the transitions including them can be safely removed from Q and Δ without affecting $L(\mathcal{A})$. We can remove all useless states in $O(|\mathcal{A}|)$ [17]. Thus, we suppose throughout this paper that every tree automaton is free from useless states.

3. Tree Automata Minimization

A deterministic tree automaton can be minimized by removing all its useless states and computing its Nerode equivalence relation [22]. Indeed, the Nerode equivalence reflects states having the same behaviour in terms of tree acceptance. In other words, two states p and q are equivalent w.r.t Nerode relation if and only if $L^{\uparrow}(p) = L^{\uparrow}(q)$. Therefore, equivalent states can be safely merged without affecting the accepted language. The computation of this relation can be achieved by means of two different approaches. The first one that we call "standard approach" computes a series of equivalence relations (or partitions) $\equiv_0, \equiv_1, \ldots$. The relation $\equiv_0$ is the coarsest one. At each step, the relation $\equiv_{i+1}$ is obtained by splitting the relation $\equiv_i$ while respecting that $\equiv_i$ is coarser than $\equiv_{i+1}$. This approach seeks to reach a relation gathering all equivalent states that respect minimization in one partition.

The second one called "incremental approach" starts with a singleton partition for each state and iteratively merges partitions that are shown to be equivalent. The process stops when the greatest fixed-point is reached.

Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be a DFTA. $\equiv$ over Q is an equivalence relation such that $p \equiv q$ implies:

1. $p \in Q_f \Leftrightarrow q \in Q_f$ and,
2. for all $\rho \in \bigcup_{n\geq 1} \Sigma_n \times Q^n : \delta(\rho_{p:i}) \equiv \delta(\rho_{q:i})$ for all $1 \leq i \leq n$.

Minimization for DFTA was first discussed in the late 1960s by Brainerd [3], and standardised in [8], [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 $\rho \in \bigcup_{n\geq 1} \Sigma_n \times Q^n : \delta(\rho_{p:i}) \equiv_j \delta(\rho_{q:i})$ for all $1 \leq i \leq n$.

The computation of the family $(\equiv_j)_{j\geq 0}$ can then be done by successive approximations until reaching the stable point.

Lemma 1. For $k \geq |Q| - 2$, we have $\equiv_{k+1} = \equiv_k$.

To prove this lemma, one has to use the property that each cycle's length cannot exceed $|Q| - 2$ (the number of nodes in one graph).

Algorithm 1. Tree autoamata minimization.

function MinimizationDFTA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$
$P_0 \leftarrow Q$
$P_1 \leftarrow \{Q_f, Q - Q_f\}$
$i \leftarrow 1$
repeat
create P_{i+1} by refining P_i so that $p \equiv_{i+1} q$ iff for all $\rho \in \bigcup_{n\geq 1} \Sigma_n \times Q^n$ $\delta(\rho_{p:i})$ and $\delta(\rho_{q:i})$ are in the same partition in P_i
$i \leftarrow i + 1$
until $(P_i = P_{i-1})$
$Q_{\min} \leftarrow \bigcup_{q\in Q} [q]$
$\Delta_{\min} \leftarrow \{(\sigma, [q_1], ..., [q_n]) \mid (\sigma, q_1, ..., q_n) \in \Delta\}$
$Q_{\min f} \leftarrow \{[q] \mid q \in Q_f\}$
return $\mathcal{A}_{\min} = (Q_{\min}, \Sigma, Q_{\min f}, \Delta_{\min})$
end function

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 $\{Q\}$, and the second one to $\{Q_f, Q - Q_f\}$. Next, at each iteration i, the current partition P_i is split by computing $\equiv_i$. $[q]$ denotes the Nerode equivalence class of the state q.

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

Concerning incremental technique for strings, Watson [9] introduced for the first time the incremental version for cyclic DFA. After that, Almeida *et al.* [11] presented a new split-based incremental implementation using the union-find algorithm. Recently, Garcia *et al.* [23] proposed a new algorithm that outperforms that of Almeida *et al.* in some contexts.

However, in the tree case, Cleophas *et al.* [10] generalized the incremental approach to trees.

Here, the trick is to compute the equivalence between pairs of states separately instead of refining states sets. This is based on the following idea.

Lemma 2. Let $p, q \in Q$. Then, $p \equiv q \Leftrightarrow L^{\uparrow}(p) = L^{\uparrow}(q)$.

The equivalence between states can be computed recursively using the following property [10].

$$equiv(p,q) = \bigwedge_{\rho\in\Sigma_n\times Q^n,\ 0\leq i\leq n} equiv(\delta(\rho_{p:i}), \delta(\rho_{q:i})) \wedge ((p \in Q_f) \Leftrightarrow (q \in Q_f))$$

Recall that in the presence of cycles, the recursive call may be infinite. Thus, Cleophas *et al.* [10] used an extended recursive function $equiv(p, q, k)$ when k is initialized with $|Q| - 2$. At each step, k is decreased until it reaches 0.

4. Efficient Implementation of the Standard Minimization

In this section, we give an efficient data structure in favour of minimization and we introduce some useful properties. Next, we describe an $n \log n$ implementation of tree automata minimization after discussing the standard case.

Any DFTA can be seen as a *bipartite graph* according to the following definition.

Definition 1. The graph $G_{\mathcal{A}} = (X, lab, A, \gamma)$ associated to a DFTA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ is defined as follows:

- The Σ-nodes set X_Σ is any set, such that there exists a bijection $X_\Sigma \rightarrow \Delta$.
- $X = X_\Sigma \bigcup Q$ is the nodes set. Q is the state-nodes set (every state is a node in the graph).

- lab: $X \rightarrow \Sigma \bigcup Q$ is the labels function that returns the label of each node from X.
- $A \subseteq (X_\Sigma \times Q) \bigcup (Q \times X_\Sigma)$ is the arcs set.
- $\gamma : A \rightarrow \mathbb{N}$ is a function that affects a weight for each arc.

$G_{\mathcal{A}}$ is built as follows: with each transition $(\sigma, q_1, ..., q_n, q)$, we associate a set of arcs $\{(q, x_\sigma)\} \bigcup \{(x_\sigma, q_i) \mid 0 \leq i \leq n\}$, such that:

- $lab\,(x_\sigma) = \sigma$.
- $\forall\, q_i, lab\,(q_i) = q_i, lab\,(q) = q$.
- $\gamma\,(q, x_\sigma) = 0$.
- $\gamma\,(x_\sigma, q_i) = i, 1 \leq i \leq n$.

Mind that we use a different Σ-node for each transition. Here we have $|X_\Sigma| = |\Delta|$. Indeed, Σ-nodes are labelled with the first symbol of a transition. For this reason, two Σ-nodes may have the same label.

4.1. Efficient Initialization

Obviously, we can meet only one equivalence class (if all states are final and share the same up language). Otherwise, this number can be equal at most to the number of states (if the processed automaton is already minimal). However, the number of equivalence classes can previously be bounded in order to accelerate the refinement process. Here we introduce some properties necessary but not sufficient to decide if two states are equivalent or not.

But first, let us recall an interesting property defined by Carrasco *et al.* [17].

$$sig\,(q) = \{(\sigma, k) \mid \exists (\sigma, q_1, ..., q_n) \in \Delta,\ q_k = q, 1 \leq k < n\} \bigcup \begin{cases} \{(\#, 1)\} & \text{if } q \in Q_f \\ \varnothing & \text{otherwise} \end{cases}$$

Here the symbol # is used to distinguish final states from non-final ones.

In fact, the signature $sig\,(q)$ computes the similarity of left and right sides between states' occurrences in the transitions set. We propose a "vertical" verification based on what we call *states level*.

Definition 2. The paths set of a state q, denoted $\hat{q}$, is the set of all finite sequences $q_1, ..., q_n, q$ such that $q_1 \in Q_f, q_2, ..., q_n \in Q, n \leq |Q|$ and for each q_{i-1}, q_i there exists a Σ-node x such that $(q_{i-1}, x), (x, q_i) \in A$.

In other words, only arcs of non-null weight are considered in the path from a final state to the wanted one.

Definition 3. Let $q \in Q$ be a state. Then the level of the state q is defined as $\mathcal{L}(q) = min_{s \in \hat{q}} \|s\|$. $\|s\|$ is the length of the sequence s.

Lemma 3. The levels of a DFTA states can be computed in linear time.

Obviously, the states level can be computed "on the fly" by executing a breadth-first search starting from final states. A special node may be added to give an entry to all final states nodes.

Consequently, the following property remains true.

Lemma 4. Let $p, q \in Q$. Then, $p \equiv q \Rightarrow (sig\,(p) = sig\,(q) \wedge \mathcal{L}\,(p) = \mathcal{L}\,(q))$.

Proof. To prove this lemma, we need only to show that $\mathcal{L}$ (p) $\neq \mathcal{L}\,(q) \Rightarrow p \not\equiv q$. See [17] for the second part. Let $p, q \in Q$.

Assume that $\mathcal{L}\,(p) \neq \mathcal{L}\,(q)$. Then, the minimal path between p and one final state is different from the minimal path between q and another one. Resolving the two paths leads to this fact: there exists $t_1 \in \mathcal{L}_p^{\uparrow}$, $t_2 \in \mathcal{L}_q^{\uparrow}$ such that we find two branches in t_1 and t_2 of different sizes. We can directly conclude that $\mathcal{L}_p^{\uparrow} \neq \mathcal{L}_q^{\uparrow}$ and $p \not\equiv q$.

4.2 Standard Implementation

By means of the associated graph proposed above, we see that the properties of equivalence can easily be extracted and checked. Thus, the representation promotes the minimization process.

We define some properties simplifying the verification of possible equivalencies between the states.

Definition 4. Let q be a state-node such that $(x, q) \in A$. Then the neighbours set of q with respect to $x \in X_\Sigma$ is $N_x\,(q) = \{(p, \gamma\,(x, p)) \mid (x, p) \in A, p \neq q\}$.

Proposition 1. Let $p, q \in Q$. Then $p \equiv q \Rightarrow (\forall\, x \in X_\Sigma, \exists\, y \in X_\Sigma : lab\,(x) = lab\,(y) \wedge N_x\,(p) = N_y\,(q))$.

Proof. This case is trivial. Let $p, q \in Q$ such that $\exists x, y \in X_\Sigma : lab(x) = lab(y) \wedge N_x(p) \neq N_y(q)$ then using Definition 4 there exits $(p', \gamma(x, p'))$ in $N_x(p)$ which does not figure in $N_y(q)$. This leads to the fact that the transition $(lab(x), q_1, \ldots, q_n) \in \Delta$ such that $p, q_{(\gamma(x,p'))} \in q_1, \ldots, q_n$ does not satisfy the defintion of $\equiv$.

We put $p \sim q \Leftrightarrow \forall x \in X_\Sigma, \exists y \in X_\Sigma : lab(x) = lab(y) \wedge N_x(p) = N_y(q)$. Thus, this equivalence can be used in an alternative implementation by refining the initial partition of the minimization process until reaching the fixed point.

4.3 An $n \log n$ Algorithm

Now, we adapt the smallest half processing strategy involved in DFA Hopcroft minimization [7] to trees. Indeed, this strategy resolves effectively the problem of partition refinement. It consists of using a splitter (an argument in our case) to split a partition, while ensuring that pairs of states present in the new partition are necessarily inequivalent. Then, the smallest partition is chosen to continue the process.

Definition 5. Let $x \in X_\Sigma$, then the predecessor $pred(x)$ of x is $pred(x) = q$ such that $(q, x) \in A$. The successors set $succ(x)$ of x is $succ(x) = \{(q, \gamma(x, q)) \mid (x, q) \in A\}$.

Recall that the predecessor of a state-node represents the output of one transition that is unique (determinism).

To each symbol $\sigma \in \Sigma$ a set of Σ-nodes denoted $F_\sigma = \{x \mid lab(x) = \sigma\}$ is associated.

We give a general algorithm (Algorithm 2). The function initialize partitions the states set according to Lemma 4. Concerning the function *smallest*, it returns the smallest partition in the set given as a parameter. The algorithm deals with a set of couples from P: $2^Q \times X_\Sigma$ which is used by the function *reset* so that the equivalence classes of a given iteration can be computed.

Theorem 1. Algorithm 2 effectively minimizes an automaton $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ in $\mathcal{O}(|\mathcal{A}|\log(|Q|))$.

This result can be checked using the same steps as in [7] and then reaches $\mathcal{O}(|Q|\log(|Q|))$

Algorithm 2. Half processing standard minimization.

```
P ← initialize(Q)
S ← smallest(P)
Treat ← ∅
for all σ ∈ Σ do
   Treat ← Treat ∪ {(S, σ)}
end for
while Treat ≠ ∅ do
   let T = (S, σ) ∈ Treat
   Treat = Treat − T
   for all p ∈ P and σ ∈ Σ do
      let h ∈ p
      for i = 1..Arity(σ) do
         p1 ← {q | ∃x ∈ F_σ : N_x(q) = succ(x) − (q, i) ∧ pred(x) ∈ S}
         p2 ← P − p1
      end for
      if ∃y ∈ X_Σ : (p, y) ∈ T
         replace (p, y) by (p1, y), (p2, y) in T
      else
         T ← T ∪ smallest(p1, p2)
      end if
      reset(P)
   end for
end while
```

time complexity. However, instead of using a symbol from the alphabet to split states sets – like in string case, one can use the property $\exists x \in F_\sigma : N_x(q) = succ(x) - (q,i) \wedge pred(x) \in$ S. Then, due to the maximum rank of the alphabet, and the size of $X_\Sigma = |\Delta|$, the complexity will be $\mathcal{O}((\hat{r}\,|\Delta| + |Q|)\log(|Q|)) = \mathcal{O}(|\mathcal{A}|\log(|Q|))$. Mind that if $|\Sigma|$ is not considered as constant, the whole complexity will be $\mathcal{O}(|\Sigma||\mathcal{A}|\log(|Q|))$.

5. New Incremental Implementation

In this section, we present an efficient implementation of the incremental DFTA minimization. It intends to reduce the asymptotic complexity of that proposed by Cleophas *et al.* [10] and that mentioned in [21]. It improves the incremental algorithm by means of the proposed associated graph introduced above. In what follows, we consider the DFTA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ and its associated graph $G_\mathcal{A} = (X, lab, \mathcal{A}, \gamma)$.

In fact, since the algorithm of Cleophas does not propose any storage technique, the worst-case complexity is $\mathcal{O}(|\mathcal{A}|^{|Q|-2}|Q|^2)$.

However, despite its high complexity, this algorithm can be halted at any time resulting in a partially minimized automaton.

Reducing extra computations has been massively studied for FSA [24] and for DFTA [23]. One of the proposed improvements is the use of a particular set $\mathcal{D}$ representing *distinguishability*. However, the set $\mathcal{D}$ may be greedy in memory, it is clear that the relation $\not\equiv$ is not an equivalence or an order relation ($\not\equiv$ is not reflexive and it is not transitive).

First, we define two particular sets: $\mathcal{J}$ for equivalent states and $\mathcal{D}$ for distinguishable ones.

$$\mathcal{J}(p) = \{q \in Q \mid p \equiv q\} \cup \{p\}$$
$$\mathcal{D}(p) = \{q \in Q \mid p \not\equiv q\}$$

Next, we can intuitively deduct the following properties:

Let $p, q \in Q$:

$$(\mathcal{J}(p) \cap \mathcal{J}(q) \neq \varnothing) \Rightarrow p \equiv q$$
$$(\mathcal{D}(p) \cap \mathcal{J}(q) \neq \varnothing) \Rightarrow p \not\equiv q$$

Indeed, before verifying if two states are equivalent, one can check these properties in order to decide if the recursive call has to be executed. By implementing $\mathcal{J}$ and $\mathcal{D}$ as bit-vectors, we can get at most $\log(|Q|)$ verifications at each step.

We note that Lemma 4 can be used to initialize $\mathcal{D}$ with the states having different signatures.

By analogy to FSA [9], incremental technique may be very useful when dealing with both cyclic and acyclic cases. Thus, detecting cycles is useful in order to identify the nature of the DFTA to be minimized.

For this reason, we propose the following lemma which can be considered as a direct consequence of the pumping lemma on trees (see [8]).

Lemma 5. Let $q \in Q$. Let $t \in L^{\downarrow}(q)$, then if $\exists\, q' \in Q, r \in Sub(t) : t(r \Leftarrow t) \in L^{\downarrow}(q')$ then q appears in a cycle and $L^{\downarrow}(q)$ is infinite.

The proof of this lemma is simple. Knowing that the automaton is deterministic, if a tree belonging to a state's down language is a subtree of another tree accepted by the same state, this tree must be met more than once in one traversal (path).

Consequently, if the DFTA is cyclic, then there exist some states in which computation of δ runs infinitely. In terms of graphs, this is expressed obviously through graph cycles. As a direct result of this lemma:

Corollary 1. The DFTA $\mathcal{A}$ is cyclic if and only if there exists $q \in Q$ such that we can find a sequence $q, q_1, \ldots, q_n, q \in \hat{q}, q_1, \ldots, q_n \in$ Q.

However, cycles number in a graph can grow exponentially w.r.t arcs number [25]. Fortunately, some search algorithms can naturally detect sets of nodes participating in a cycle. One excellent solution is to compute the *strongly connected graph*. There exist several linear algorithms in literature that find strongly-connected components of a graph in linear time like [26]. In our case, this step takes $\mathcal{O}(|Q|+|\mathcal{A}|)$ (arcs number depends on transitions size).

Now, taking into account that the computation of strongly connected components gives a partition of the graph nodes, we put $\Pi_Q(G_\mathcal{A})$ the cycle-states set partitioning of the DFTA which contains portions of Q. $\pi(q)$ is the strongly connected component containing the state q. Therefore, we have:

Lemma 6. Let $\rho = (\sigma, q_1, \ldots, q_n)$ be an argument. Let $p, q \in Q$ such that $\delta(\rho_{p:i}) = p'$ and $\delta(\rho_{q:i}) = q'$. Then if $p' \in \pi(p) \wedge q' \notin \pi(q)$ then $p \not\equiv q$.

According to $(\equiv_j)_{j\geq 0}$, p' and q' should be equivalent. However, there exists a cycle from p to p', but not from q to q'. By the same argument, p can be reached from p', but it is not the case for q and q'. Therefore, we can say that p and q cannot be equivalent.

Corollary 2. The DFTA is acyclic if and only if $\forall \pi \in \Pi, |\pi| = 1$.

It is clear that no strongly connected component exists. Every partition contains only one state.

We extend the predecessor notion to state-nodes.

Definition 6. Let $q \in Q$ be a state-node, then the predecessors set $pred(q)$ of q is:

$$pred(q) = \{x \in X_\Sigma \mid (p, x) \in A\}$$

However, despite the use of $\mathcal{J}$ and $\mathcal{D}$, the global complexity of the computation remains high (two states may be verified more than once).

To solve this problem, we define a special tree *Trace* storing for every inspected pair of states, the path of computation and the recursion depth. *addNode* (P, Q) is a function that adds the states pair Q to the pair P as a subtree in *Trace*. We use also a function *subTree* (*node*) to regroup all pairs figuring in the subtree of the *node* in *Trace*.

Trace is used afterwards to decide the equivalence of states pairs invoked in the recursive call *equiv*.

We are now ready to give the improved equivalence function presented by Algorithm 3. *update* $(p, q,$ **B**$)$ is a function that updates the sets $\mathcal{J}$ and $\mathcal{D}$ for a pair (p, q) of states according to the given boolean **B**. **B** is set to *true* when p and q are equivalent. This one represents the knowledge about equivalence relation between p and q deducted previously in the process.

Algorithm 3. New equivalence function.

```
function equiv(p,q,k)
if  k = 0 then
    eq ← ((p ∈ Qf) ∧ (q ∈ Qf))
else if  (J(p) ∩ J(q) ≠ ∅) or {p, q} ∈ S then
    eq ← true
else if  (D(p) ∩ J(q) ≠ ∅) then
    eq ← false
else
    S ← S ∪ {{p, q}}
    eq ← ((p ∈ Qf) ∧ (q ∈ Qf))
    for x ∈ pred(p) do
      if ∃y ∈ pred(q), lab(x) = lab(y) ∧ Nx(p) = Ny(q) ∧ |pred(p)| = |pred(q)| then
        eq ← eq ∧ equiv(pred(x), pred(y), k − 1)
      end if
      if ¬eq then
        update(p, q, false)
        break;    # The loop is stopped here, {pred(x), pred(y)} must not be added to {p, q} in Trace.
      end if
      addNode({p, q}, {pred(x), pred(y)})
   end for
   S ← S − {{p, q}}
end if
return (eq)
end function
```

Algorithm 4. Trace treatment.

```
function Treat (Trace)
 if ∄ node ∈ Trace marked |Q| − 2 then
  for all p, q ∈ subTree (Trace) do
  update (p, q, true)
  end for
   else
    for all p, q ∈ Subtree (Trace) do
     if ∃ subTree (p, q) marked |Q| − 2 then
          Stack (p, q) ← {(N, depth (N)) | N ∈ subTree (p, q)}
     else
        update (p, q, true)
     end if
    end for
 end if
end function
```

When the function *equiv* is called, *Trace* is created, containing all tested pairs during recursive calls. To avoid the reuse of these states, we introduce a new algorithm (Algorithm 4) that treats *Trace*. Here, we meet three cases:

1. *equiv* returns *true*, then all pairs located in *Trace* will be equivalent,
2. *equiv* returns *false* (*equiv* meets a break) and the height of *Trace* is less than $|Q| - 2$, then all subtrees concerned by the break receive *false*. The remaining pairs receive *true*,
3. *equiv* returns *false* and the height of *Trace* is $|Q| - 2$, then all subtrees participating in cycles (ensured before this step by CCSC computation) and those concerned by the break are stored in a special matrice *Stack*. The remaining pairs receive *true*.

For implementation reasons, we can add a special markup of *Trace*: the recursion depth (parameter k) is associated to each stored node.

We can restore every verified path from *Stack* through the algorithm *CallStack* (Algorithm 5).

Now, we propose a specific preorder to optimize *equiv* calls.

Let $p, q \in Q$. Then we put $p \preceq q \Leftrightarrow \mathcal{L}(p) \leq \mathcal{L}(q)$. It is clear that this preorder is total.

Corollary 3. There exists $p \in Q$ such that $\forall q \in Q : q \preceq p \wedge \mathcal{L}(p) \leq |Q|$.

Here, we can see that the state's level is bounded by $|Q|$. This value comes from the longest path without cycles in a graph. Its value cannot exceed the number of its nodes.

Following this preorder, we give the general rules of the incremental minimization process:

1. Start with testing final states equivalences ($\forall p, q \in Q$: $\mathcal{L}(p) = \mathcal{L}(q) = 0$).
2. In each iteration, treat states leading to states already inspected during the previous steps. In other words, when an iteration i is processed, treat states in the set $\{p, \mathcal{L}(p) = i\}$.

Algorithm 6 represents the main function. The function *CSCC* ($G_{\mathcal{A}}$) computes the strongly connected components of the graph associated

Algorithm 5. Verify stack.

```
function CallStack (p, q)
 for all ((q1, q2), k)) ∈ Stack (p, q) do
    eq ← eq ∧ equiv (q1, q2, k)
 end for
 return eq
end function
```

to the DFTA in question. Regarding the function $InOut(p, q)$, it returns a boolean indicating if p and q are equivalent or not, according to lemma 4 and lemma 6.

The defined main function uses two sets: V for verified states and R for states concerned by the next step. First, R is initialized by final states couples Q_f ($\mathcal{L}(q \in Q_f) = 0$). Next, in each iteration, couples from R^2 are verified and stored in V. R is then updated to the next level. Finally, the algorithm ends when all possibilities are verified (V is stable). By verifying states in the indicated way, we can note that the main function can localize co-accessible states and then ignore useless states.

To accelerate the algorithm and avoid pairs double-checking, *Stack* can be verified before any call of *equiv* in the main function and in the equivalence function.

Theorem 2. A DFTA can be minimized using Algorithm 6 in $\mathcal{O}(\hat{r}|\mathcal{A}| + |Q|^3 \log(|Q|))$.

The main function executes exactly $|Q| \times |Q - 1|/2$ calls. In each one, a $|Q - 2|$ depth (ensured by k) new calls are made. Indeed, the predecessors and the neighbours' sets (~ computation) can be computed before the incremental process. Mind that this can be done in $\mathcal{O}(\hat{r}\,|\mathcal{A}|)$ because for every Σ-node, $\hat{r}$ states can be connected and the size of the whole graph is $|\mathcal{A}|$. Whereof, testing of these sets (Algorithm 3) can be done in a constant time. Furthermore, all duplicated computations can be avoided using *Trace*, *Stack* and the two guards $\mathcal{D}$ and $\mathcal{J}$.

The following result is very important when dealing with the acyclic case.

Corollary 4. An ADFTA can be minimized in $\mathcal{O}(|\mathcal{A}| \log(|Q|))$.

Algorithm 6. New main function.

```
function Main
CSCC(G_A)
S ← ∅
  for all q ∈ Q do
    J(q) ← q
    D(q) ← ∅
  end for
    R ← Q_f     # The initial level is 0
    V ← ∅
    V' ← ∅
    repeat
     T = {(p, p) | p ∈ Q}
     for all (p, q) ∈ R^2 − (V ∪ T) do
       Trace ← ∅
       if InOut(p, q) then
          update(p, q, false)
       else
          update(p, q, equiv(p, q, |Q| − 2))
          Treat(Trace)
       end if
     end for
     H ← R
     V' ← V
     V ← V ∪ R^2
     R ← {q | ∃ q' ∈ H, n ∈ N, x ∈ X_Σ: (x, q'), (q, x) ∈ A}     # We have L(q) = L(q') + 1
    until V = V'
end function
```

Indeed, it is not necessary to use *Trace*, *Stack* and the depth guard k in the acyclic case because no cycles are met. A *native* equivalence function *equiv* (p, q) can be used instead of the improved one. Detecting DFTA nature can be done instantly by analysing the result of $CSCC(G_{\mathcal{A}})$. Here, just $\mathcal{J}$ and $\mathcal{D}$ computations are needed at each step.

6. Conclusion

In this work, we present an efficient implementation of DFTA minimization based on a simple graphic view of the automaton to be minimized. We show a general algorithm of the standard technique and we outline an adapted method based on the smallest half-strategy processing. The time complexity is $\mathcal{O}(|\mathcal{A}| \log(|Q|))$.

The second part is dedicated to the incremental technique. We give an improved implementation based on graph properties and avoidance of some computations. The complexity of this algorithm is $\mathcal{O}(|Q|^3 \log(|Q|))$. Moreover, we show that this technique minimizes an ADFTA in $\mathcal{O}(|\mathcal{A}| \log(|Q|))$.

References

[1] D. Duarte, "A Method for Evolution of XML Schemas Preserving the Validity of Documents", PhD thesis, Université François-Rabelais, 2005 (in French).

[2] M. Tommasi, "Tree Structures and Automatic Learning" PhD thesis, Université Charles de Gaulle – Lille 3, 2006 (in French).

[3] W. S. Brainerd, "Tree Generating Systems and Tree Automata", PhD thesis, Purdue University, 1967.

[4] M. A. Arbib and Y. Give'on, "Algebra Automata I: Parallel Programming as a Prolegomena to the Categorical Approach", *Information and Control*, vol. 12, no. 4, pp. 331–345, 1968. http://dx.doi.org/10.1016/S0019-9958(68)90374-4

[5] F. Gécseg and M. Steinby, "Minimal Ascending Tree Automata", *Acta Cybernetica*, vol. 4, pp. 37–44, 1980.

[6] E. F. Moore, "Gedanken-Experiments on Sequential Machines", in *Automata Studies*, Princeton University, pp. 129–153, 1956. http://dx.doi.org/10.2307/2964500

[7] J. E. Hopcroft, "An *n*log*n* Algorithm for Minimizing States in a Finite Automaton", Tech. Rep., Stanford, CA, USA, 1971.

[8] H. Comon *et al.*, "Tree Automata Techniques and Applications", release October, 2007.

[9] B. W. Watson, "An Incremental DFA Minimization Algorithm", in *Finite State Methods in Natural Language Processing*, 2001.

[10] L. G. Cleophas *et al.*, "On Minimizing Deterministic Tree Automata", in J. Zdarek and J. Holub (Eds.), *Proceedings of the Prague Stringology Conference*, Prague, 2009, pp. 173–182.

[11] Marco Almeida *et al.*, "Incremental DFA Minimization", *RAIRO − Theoretical Informatics and Applications*, pp. 173–186, 2014. http://dx.doi.org/10.1051/ita/2013045

[12] B. W. Watson, "A Fast New Semi-Incremental Algorithm for the Construction of Minimal Acyclic DFAs", *Automata Implementation, Volume 1660 of the series Lecture Notes in Computer Science*, pp. 121–132, 1999. http://dx.doi.org/10.1007/3-540-48057-9_11

[13] B. W. Watson, "A Taxonomy of Algorithms for Constructing Minimal Acyclic Deterministic Finite Automata", *South African Computer Journal*, vol. 27, pp. 12–17, 2001.

[14] J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages and Computation", Addison-Wesley, 1979.

[15] J. Bubenzer, "Minimization of Acyclic DFAs", *Proceedings of the Prague Stringology Conference*, Prague, 2011, pp. 29–31.

[16] D. Revuz, "Minimisation of Acyclic Deterministic Automata in Linear Time", *Theor. Comput. Sci.*, vol. 92, no. 1, pp. 181–189, 1992. http://dx.doi.org/10.1016/0304-3975(92)90142-3

[17] R. C. Carrasco *et al.*, "An Implementation of Deterministic Tree Automata Minimization", in J. Holub and J. Zdarek (Eds.), *CIAA, Volume 4783 of Lecture Notes in Computer Science*, pp. 122–129, Springer, 2007. http://dx.doi.org/10.1007/978-3-540-76336-9_13

[18] P. A. Abdulla *et al.*, "Bisimulation Minimization of Tree Automata", *International Journal of Foundations of Computer Science*, vol. 18, no. 4, pp. 699–713, 2007. http://dx.doi.org/10.1142/S0129054107004929

[19] R. Paige and R. E. Tarjan, "Three Partition Refinement Algorithms", *SIAM Journal on Computing*, vol. 16, no. 6, pp. 973–989, 1987. http://dx.doi.org/10.1137/0216062

[20] J. Högberg *et al.*, "Backward and Forward Bisimulation Minimization of Tree Automata", *Theoretical Computer Science*, vol. 410, no. 37, pp. 3539–3552, 2009. http://dx.doi.org/10.1016/j.tcs.2009.03.022

[21] Y. Guellouma *et al.*, "A Fast Implementation of Incremental Minimization of Tree Automata", I*nternational Conference on Innovations in Information Technology (IIT)*, pp. 322–327, 2012. http://dx.doi.org/10.1109/INNOVATIONS.2012.6207759

[22] D. Kozen, "On the Myhill-Nerode Theorem for Trees", *Bulletin of the EATCS*, vol. 47, pp. 170–173,1992.

[23] P. Garca *et al.*, "A Split-Based Incremental Deterministic Automata Minimization Algorithm", *Theory of Computing Systems*, vol. 57, no. 2, pp. 319–336, 2015. http://dx.doi.org/10.1007/s00224-014-9588-y

[24] B. W. Watson., "A Taxonomy of Finite Automata Minimization Algorithms", *Computing Science Note 93/44*, Eindhoven University of Technology, The Netherlands, 1993.

[25] D. B. Johnson, "Finding All the Elementary Circuits of a Directed Graph" *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975. http://dx.doi.org/10.1137/0204007

[26] R. Tarjan, "Depth-First Search and Linear Graph Algorithms", *SIAM Journal on Computing*, 1972. http://dx.doi.org/10.1137/0201010

Received: November 2015
Revised: September 2016
Accepted: September 2016

Contact addresses:
Younes Guelloma
Laboratoire LIM
Université Amar Telidji de Laghouat
BP 37G, Route de Ghardaïa
03000 Laghouat, Algérie
e-mail: y.guellouma@mail.lagh-univ.dz

Hadda Cherroun
Laboratoire LIM
Université Amar Telidji de Laghouat
BP 37G, Route de Ghardaïa
03000 Laghouat, Algérie
e-mail: hadda_cherroun@mail.lagh-univ.dz

Younes Guellouma was born in Laghouat, Algeria, in 1984. He is an assistant in the Department of Mathematics and Computer sciences at the University of Laghouat. He also works in the laboratory of mathematics and informatics at the University of Laghouat. He got his Engineer degree in 2006 and his Master's degree in 2009 from the same university. He is currently a PhD student. His research topics are theory of automata, tree automata algorithmic and optimization methods.

Hadda Cherroun is a full professor in the Department of Computer Science at Amar Telidji University in Algeria. She received her Engineer degree (1991), MS (1997), and PhD (2007) in computer science from the University of Sciences and Technology Houari Boumediene in Algeria. Between 1991 and 1997 she worked as an engineer at Amar Telidji University. She is teaching various courses on algorithms and their analysis, parallel systems, and architectures. Dr. Cherroun leads many national projects. She was a member of the Trans-European Mobility Programme for University Studies (TEMPUS), Middle East and North Africa (MEDA) project: Ide@. Her main research interests lie with the design of algorithms for many domains, particularly scheduling, compiler technology, networks, distributed systems, and machine learning.