Algorithmica
DOI 10.1007/s00453-008-9172-4

Incremental Construction of Minimal Tree Automata

Rafael C. Carrasco · Jan Daciuk · Mikel L. Forcada

Received: 16 October 2006 / Accepted: 15 January 2008

Abstract We describe an algorithm that allows the incremental addition or removal of unranked ordered trees to a minimal frontier-to-root deterministic finite-state tree automaton (DTA). The algorithm takes a tree t and a minimal DTA A as input; it outputs a minimal DTA A' which accepts the language $L(A)$ accepted by A incremented (or decremented) with the tree t. The algorithm can be used to efficiently maintain dictionaries which store large collections of trees or tree fragments.

Keywords Deterministic tree automata · Incremental construction of minimal tree automata

1 Introduction

Some natural language processing applications, for example data-oriented parsing [3], require the storage of large collections of parse tree fragments. A data structure that is able to hold unranked ordered tree data efficiently is a minimal frontier-to-root deterministic tree automaton (DTA), much in the same way as a minimal deterministic finite-state automaton (DFA) [1], stores strings efficiently [11]. To start, each

R.C. Carrasco (✉)
Dep. de Lenguajes y Sistemas Informáticos, Universidad de Alicante,
Carretera de San Vicente s/n, 03071 Alicante, Spain
e-mail: carrasco@dlsi.ua.es

J. Daciuk
Knowledge Engineering Department, Gdańsk University of Technology, Ul. G. Narutowicza 11/12,
80-952 Gdańsk, Poland
e-mail: jandac@eti.pg.gda.pl

M.L. Forcada
Dep. de Llenguatges i Sistemes Informàtics, Universitat d'Alacant, 03071 Alacant, Spain
e-mail: mlf@dlsi.ua.es

subtree which is common to several trees in the collection is assigned a single state, as happens in prefix-tree DFAs, retrieval trees or tries [12, 13] with prefixes which are common to several strings in a dictionary. Furthermore, as happens in DFAs with string prefixes, the number of such states is minimized by assigning a single state to sets of subtrees that may appear interchangeably in the collection. There exists a standard procedure [4, 15] to obtain the minimal DTA that can be efficiently implemented [5]. In contrast, root-to-frontier deterministic tree automata are strictly less powerful than (frontier-to-root) DTAs; in particular, there exist finite tree languages that are not accepted by any root-to-frontier deterministic tree automaton [8].

The construction of minimal automata from collections in the traditional way requires considerable amounts of time and memory. To alleviate the problem in DFAs, *incremental algorithms* have been designed that directly construct minimal automata, going through a construction and local minimization cycle each time a new data item—a word or a string—is added to the language of the automaton. Two families of algorithms for construction of acyclic DFAs exist: those that require data to be lexicographically sorted [7, 10], and those that accept data without that restriction [2, 10, 18, 19]. Both kinds of algorithms have been extended to the case where strings are added to a cyclic DFA [6, 9, 14]. Of course, only the algorithms accepting unsorted input are amenable to removal. Here, we study the addition and removal of unsorted trees to a cyclic minimal DTA.

This paper is organized as follows: Sect. 2 describes the notation used; Sect. 3 presents the new algorithm for incremental construction of DTA and Sect. 4 includes a theoretical justification; finally, conclusions are presented in Sect. 5.

2 Notation

Given an *alphabet*, that is, a finite set of symbols (also called *labels*) $\Sigma = \{\sigma_1, \dots, \sigma_{|\Sigma|}\}$, we define T_Σ as the set of unranked ordered trees with labels in Σ: every symbol $\sigma \in \Sigma$ belongs to T_Σ and every $\sigma(t_1 \cdots t_m)$ such that $\sigma \in \Sigma$, $m > 0$ and $t_1, \dots, t_m \in T_\Sigma$ is also a tree in T_Σ. The trees so defined are ordered (the order of children is relevant) and unranked (there is no *a priori* bound on the number of children of a node). The *size* $|t|$ of $t \in T_\Sigma$ is $|t| = 1$ if $t = \sigma$ and $|t| = 1 + \sum_{k=1}^{m} |t_k|$ if $t = \sigma(t_1, \dots, t_m)$.

In the following, we will assume that Σ is given and will call *trees* all elements in $t \in T_\Sigma$. Any subset of T_Σ will be called a *tree language*. In particular, the language $\mathrm{sub}(t)$ of the *subtrees* of a tree t is recursively defined as follows:

$$\mathrm{sub}(t) = \begin{cases} \{\sigma\} & \text{if } t = \sigma \in \Sigma, \\ \{t\} \cup \bigcup_{k=1}^{m} \mathrm{sub}(t_k) & \text{if } t = \sigma(t_1 \cdots t_m) \in T_\Sigma - \Sigma. \end{cases} \tag{1}$$

A *finite-state frontier-to-root tree automaton* is defined as $A = (Q, \Sigma, \Delta, F)$, where $Q = \{q_1, \dots, q_{|Q|}\}$ is a finite set of *states*, $\Sigma = \{\sigma_1, \dots, \sigma_{|\Sigma|}\}$ is the *alphabet*, $F \subseteq Q$ is the subset of *accepting states*, and $\Delta = \{\tau_1, \dots, \tau_{|\Delta|}\} \subset \Sigma \times Q^* \times Q$ is a finite set of *transitions* (where Q^* denotes, as usually, $\bigcup_{m=0}^{\infty} Q^m$).

Each transition $\tau_n = (\sigma, i_1, \ldots, i_m, j)$ may be seen as consisting of an *argument* $\arg(\tau_n) = (\sigma, i_1, \ldots, i_m)$ and an *output* $\text{out}(\tau_n) = j$. The *size* of the automaton is defined as

$$|A| = \sum_{n=1}^{|\Delta|} |\tau_n|, \tag{2}$$

where the size $|\tau_n|$ of a transition $\tau_n = (\sigma, i_1, \ldots, i_m, j)$ is $m + 2$.

The automaton A is a *deterministic finite-state frontier-to-root tree automaton*, or deterministic tree automaton (DTA) for short, if for every argument there is at most one possible output, that is, for all $\sigma \in \Sigma$, for all $m \geq 0$ and for all $(i_1, \ldots, i_m) \in Q^m$, there is at most one $j \in Q$ such that $(\sigma, i_1, \ldots, i_m, j) \in \Delta$. In a DTA, the transition output for argument $(\sigma, i_1, \ldots, i_m)$ will be denoted with $\delta_m(\sigma, i_1, \ldots, i_m)$ and

$$\delta_m(\sigma, i_1, \ldots, i_m) = \begin{cases} j & \text{if } j \text{ is the only state in } Q \\ & \text{such that } (\sigma, i_1, \ldots, i_m, j) \in \Delta, \\ \perp & \text{if no such } j \text{ exists.} \end{cases} \tag{3}$$

In the following, the symbol $\perp$ will be interpreted as a special *absorption state* such that $\perp \in Q - F$. With this assumption, Δ remains finite but the output for all possible transition arguments is a state in Q.

The output $A(t)$ when a DTA A operates on $t \in T_\Sigma$ is the state in Q recursively computed as

$$A(t) = \begin{cases} \delta_0(\sigma) & \text{if } t = \sigma \in \Sigma, \\ \delta_m(\sigma, A(t_1), \ldots, A(t_m)) & \text{if } t = \sigma(t_1 \cdots t_m) \in T_\Sigma - \Sigma. \end{cases} \tag{4}$$

Note that, in contrast to DFAs, the DTA does not specify the initial state as every tree is processed starting at its leaves and each leaf label σ determines a state $\delta_0(\sigma)$ that is used as input for the ensuing computation.

For every DTA A, the tree *language* $L_A(q)$ *accepted* at state $q \in Q$ is the subset of T_Σ with output q

$$L_A(q) = \{t \in T_\Sigma : A(t) = q\} \tag{5}$$

and the tree language $L(A)$ accepted by A is the subset of trees in T_Σ accepted at the states in F

$$L(A) = \bigcup_{q \in F} L_A(q) = \{t \in T_\Sigma : A(t) \in F\}. \tag{6}$$

Two DTA accepting identical languages are said to be *equivalent*, that is, $A \equiv B$ if and only if $L(A) = L(B)$.

For every $t \in T_\Sigma$, A^t will denote the DTA $(Q^t, \Sigma, \Delta^t, F^t)$ accepting $L(A^t) = \{t\}$:

$$\begin{aligned} Q^t &= \text{sub}(t) \cup \{\perp^t\}, \\ \Delta^t &= \{(\sigma, t_1, \ldots, t_m, \sigma(t_1 \cdots t_m)) : \sigma(t_1 \cdots t_m) \in \text{sub}(t)\}, \\ F^t &= \{t\}. \end{aligned} \tag{7}$$

For every pair of DTA $A = (Q^A, \Sigma, \Delta^A, F^A)$ and $B = (Q^B, \Sigma, \Delta^B, F^B)$, a *product* of A and B is any DTA $P = (Q, \Sigma, \Delta, F)$ such that $P(t) = (A(t), B(t))$. For instance, the *union DTA* $P^+(A, B)$ is the product DTA with

$$\begin{aligned}
&Q = Q^A \times Q^B, \\
&\Delta = \big\{(\sigma, (i_1, j_1), \ldots, (i_m, j_m), (p, q)) \in \Sigma \times Q^* \times Q : \\
&\qquad p = \delta_m^A(\sigma, i_1, \ldots, i_m), q = \delta_m^B(\sigma, j_1, \ldots, j_m) \wedge (p, q) \neq (\perp^A, \perp^B)\big\}, \\
&F = \big(F^A \times Q^B\big) \cup \big(Q^A \times F^B\big).
\end{aligned} \tag{8}$$

As in the case of DFA [16], it is not difficult to show that $L(P^+(A, B)) = L(A) \cup L(B)$. Analogously, the *difference DTA* $P^-(A, B)$ is defined with accepting states $F = (F^A \times Q^B) - (Q^A \times F^B)$ and accepts $L(A) - L(B)$.

In a DTA A, a state q is *inaccessible* if $L_A(q) = \emptyset$, that is, if there is no tree t in T_Σ such that $A(t) = q$. Therefore, inaccessible states and the transitions using them can be safely removed from Q and Δ respectively without affecting $L(A)$. If q is accessible and C_q denotes the number of transitions in Δ with output q then $C_q > 0$ if and only if $q \neq \perp$ because Δ does not contain transitions with output $\perp$ and the accessibility condition $L_A(q) \neq \emptyset$ requires, for $q \neq \perp$, the existence of at least one transition with output q. It is worth noting that, in contrast to DFAs, in DTAs operating on unranked trees, the absorption state $\perp$ is always accessible because Δ is a finite subset of $\Sigma \times Q^* \times Q$ and there is an infinite number of arguments leading to $\perp$.

An accessible state $q \in Q$ is said to be *coaccessible* if there is at least one tree $t \in L(A)$ containing a subtree s such that $q = A(s)$. States which are not coaccessible (and accessible) are *useless*. In particular, as no transition in Δ contains $\perp$, the absorption state is useless. The identification of inaccessible and useless states can be done—see, for instance, reference [5]—in time $\mathcal{O}(|A|)$. Note that, after useless states have been removed, $|L_A(q)| = 1$ implies $C_q = 1$ but the converse is not necessarily true.

The standard algorithm to minimize DTA [4, 8, 15] removes inaccessible states and then partitions Q into equivalence classes by iterative refinement until it becomes a congruence. A *congruence* $\equiv$ on $A = (Q, \Sigma, \Delta, F)$ is an equivalence relation such that $p \equiv q$ implies

$$p \in F \;\leftrightarrow\; q \in F \tag{9}$$

and for all $m > 0$, all $k \leq m$ and all $(\sigma, r_1, \ldots, r_m) \in \Sigma \times Q^m$

$$\delta_m(\sigma, r_1, \ldots, r_{k-1}, p, r_{k+1} \ldots, r_m) \equiv \delta_m(\sigma, r_1, \ldots, r_{k-1}, q, r_{k+1} \ldots, r_m). \tag{10}$$

3 Adding and Deleting a Tree

This section describes the basic procedure to add or delete a tree t from the language accepted by a minimal DTA A. In particular, the algorithm that takes a minimal DTA A and a tree t as input and outputs a new minimal DTA A' such that $L(A') = L(A) \cup \{t\}$ is shown in Fig. 1.

```
Algorithm AddTree
Input: a minimal DTA $A$ and a tree $t \notin L(A)$.
Output: a new minimal DTA $A'$ such that $L(A') = L(A) \cup \{t\}$.
Method:

1. (* Create $A^\dagger$ such that $L(A^\dagger) = L(A) \cup \{t\}$ *)
   (a) $(A^\dagger, \Theta) \leftarrow \texttt{split}(A, t)$.
   (b) Add $A^\dagger(t)$ to $F^\dagger$ and $\Theta$.
2. (* Local minimization *)
   (a) Create the register $R \leftarrow Q^\dagger - \{A^\dagger(s) : s \in \mathrm{sub}(t)\}$.
   (b) $A' \leftarrow \texttt{minim}(A^\dagger, R, \Theta)$.
   (c) Return $A'$.
```

Fig. 1 Algorithm AddTree to add a tree to the language accepted by a minimal DTA. The algorithm can be also used for tree removal provided that step 1(b) adds $A^\dagger(t)$ to Θ but removes it from $F^\dagger$

The construction of A' is performed in two phases. The first one (recursive splitting) builds the union automaton $A^\dagger = P^+(A, A^t)$ satisfying $L(A^\dagger) = L(A) \cup \{t\}$ or, in the case of tree removal, $A^\dagger = P^-(A, A^t)$ such that $L(A^\dagger) = L(A) - \{t\}$. In both cases, t has a dedicated state in $A^\dagger$ so that $L_{A^\dagger}(A^\dagger(t)) = \{t\}$. As will be seen below, and was the case also for string automata [6], there is no need to explicitly build the whole union DTA, which will be instead obtained as a modified version of the original automaton A.

The second phase (local minimization) transforms $A^\dagger$ into a minimal DTA A'. For this purpose, any minimization algorithm, including an incremental one [20] could be used. However, as A was already minimal, a more effective procedure inspired in the local minimization described for the incremental construction of DFAs [6, 10] can be implemented. The key to local minimization is to keep a register R containing mutually inequivalent states so that only the equivalence of states which are not in R must be checked. After the recursive splitting, only states used when the automaton operates on the tree can be in $Q - R$ and, as they usually represent a minor fraction of the states in the automaton, checking the equivalence of most pairs of states is avoided.

This local minimization is also incremental but as will be seen later and in contrast with the incremental algorithm presented in [20], the pairwise comparison of states does not require further recursive calls. Thus, to avoid confusion, we use the name *local minimization* for our procedure.

Looking for an equivalent state in the register R can be a slow process which often returns no state. For instance, if all transitions in Δ containing $q \in Q$ in the argument have identical output in Δ and in $\Delta^\dagger$ and the outputs are in R then q cannot be equivalent to any other state in R. Therefore, in order to accelerate this search, an auxiliary set Θ is built during the recursive splitting which will be used later during the local minimization phase. After the split, this set contains those states in $Q^\dagger$ which belong to the argument of a transition whose output is a new state, that is, a state in $Q^\dagger - Q$.

The procedures to create the union DTA and to minimize it locally are described in more detail in the following subsections.

3.1 Splitting the Automaton

Building the product DTA $A^\dagger$ generates only four different types of accessible states:

1. The new absorption state $\perp^\dagger = (\perp, \perp^t)$, which will never appear in $\Delta^\dagger$.
2. One state of the form $(q, \perp^t)$ for every $q \in Q - \{\perp\}$ such that $L_A(q) \not\subseteq \text{sub}(t)$. The transitions in $\Delta^\dagger$ containing these *intact states* in the argument are completely determined by the analogous transitions in Δ. Moreover, $(q, \perp^t) \in F^\dagger \leftrightarrow q \in F$. Therefore, q and $(q, \perp^t)$ are equivalent.[1] If A is large, these are the great majority of accessible states, so it is convenient to build $A^\dagger$ as a modified A, as it is done in our algorithm.
3. One state $(\perp, s)$ for every $s \in \text{sub}(t)$ such that $A(s) = \perp$. These are *new states*, only one of which, $(\perp, t)$, can be in $F^\dagger$. The transitions containing them in the argument are determined by the analogous transition in Δ^t so that they may be considered to be equivalent[2] to the state s in A^t.
4. One state $(A(s), s)$ for every $s \in \text{sub}(t)$ such that $A(s) \neq \perp$. If $A(s) = q$ and $C_q = 1$ then $L_{A^\dagger}((q, s)) = L_A(q) = \{s\}$; therefore (q, s) can be identified with the state $q \in Q$ and our algorithm does not need to create a new state. We will call these states *recycled states*. However, in case $|L_A(q)| > 1$, (q, s) will be called a *cloned state* as transitions in Δ with q in the argument will spawn multiple transitions in $\Delta^\dagger$.

The remaining states in the product DTA (a great majority, as for most $(q, s) \in Q \times Q^t$ one has $A(s) \neq q$), are inaccessible and can be safely discarded; therefore the procedures described below will carefully avoid creating them.

Then, the splitting procedure which creates the new set of states and transitions in $A^\dagger$ from those in A can be performed recursively, as in algorithm `split` shown in Fig. 2. Indeed, if $s = \sigma(s_1, \ldots, s_m)$ and, for all $1 \le k \le m$, $L_A(r_k) = \{s_k\}$ with $r_k = A(s_k)$, then one finds only three different cases:

- The output $\delta_m(\sigma, r_1, \ldots, r_m)$ is $\perp$, that is, $C_q = 0$. Then, $(\perp, s)$ is a new state in $A^\dagger$ which can be identified with s, as discussed above.
- There is a single transition in Δ with output $q = \delta_m(\sigma, r_1, \ldots, r_m)$, that is, $C_q = 1$. Then, $L_A(q) = \{s\}$ and the split procedure can return A without further modification. Therefore, q is a *recycled* state.
- There is more than one transition in Δ with output $q = \delta_m(\sigma, r_1, \ldots, r_m)$, that is, $C_q > 1$. In such case, the new DTA $A^\dagger$ contains a destination state $n = (q, s)$ and $\Delta^\dagger$ a transition $(\sigma, r_1, \ldots, r_m, n)$. Therefore, $A^\dagger$ satisfies $L_{A^\dagger}(n) = \{s\}$. However, q may belong to the argument of a number of transitions in Δ and then, $\Delta^\dagger$ must include analogous transitions with n in the argument. The procedure to create these new transitions will be called *cloning* and is described in detail in the next section.

[1] States in two different DTA A and B are said to be equivalent if they satisfy conditions (9–10) in the *sum DTA* with $Q = Q^A \cup Q^B$, $F = F^A \cup F^B$ and $\Delta = \{(\sigma, i_1, \ldots, i_m, j) \in \Delta^A \cup \Delta^B : m > 0\}$.

[2] In the case of tree removal, the equivalence holds only if F^t is redefined as empty.

```
Function split
Input: A DTA $A$ and a subtree $s = \sigma(s_1, \ldots, s_m)$.
Output:  A DTA $A^\dagger$ such that $L(A^\dagger) = L(A)$ and $L_{A^\dagger}(A^\dagger(t)) = \{t\}$;
         A set $\Theta = \{q \in Q^\dagger : \exists(\sigma, i_1, \ldots, i_m, j) \in \Delta^\dagger : \exists k \leq m : i_k = q \wedge j \in Q^\dagger - Q\}$
Method:

1. Create an empty set $\Theta$.
2. For $k = 1$ to $m$ do
   • $(A, \theta) \leftarrow$ split$(A, s_k)$;
   • $r_k \leftarrow A(s_k)$
   • Add all states in $\theta$ to $\Theta$.
3. $q \leftarrow \delta_m(\sigma, r_1, \ldots, r_m)$.
4. If $C_q \neq 1$ then
   (a) Add a new state $n$ to $Q$.
   (b) If $C_q = 0$ then
         • Add $(\sigma, r_1, \ldots, r_m, n)$ to $\Delta$.
       else (* $C_q > 1$ *)
         • $A \leftarrow$ clone$(A, q, n)$.
         • Replace $(\sigma, r_1, \ldots, r_m, q)$ with $(\sigma, r_1, \ldots, r_m, n)$ in $\Delta$.
   (c) Add $r_1, \ldots, r_m$ to $\Theta$.
5. Return $(A, \Theta)$.
```

Fig. 2 Function `split` computing the product DTA

Note that for the efficient computation of $C_q = |\{\tau_n \in \Delta : \text{out}(\tau_n) = q\}|$ it suffices to keep a set of counters $\{C_1, \ldots, C_{|Q|}\}$ that are updated whenever a new transition is added to or removed from Δ. In addition to the computation of $A^\dagger$, function `split` builds the subset $\Theta \subseteq Q^\dagger$ which will be used later during local minimization: every time `split` creates a transition whose output is a new state, the states in the argument of the transition are added to Θ.

3.2 Cloning States

In case $|L_A(q)| > 1$, more than one tree is accepted at state q. Thus, if $q = A(s)$ with $s \in \text{sub}(t)$, there exists at least one $s' \neq s$ such that $A(s') = A(s)$. Splitting q so that s is accepted at a new state n would inadvertently remove some trees from $L(A)$ (those trees where s replaces s'). Therefore, it is necessary to add new transitions with n in the argument.

From a complementary perspective, for every transition in Δ containing q in the argument, $\Delta^\dagger$ will include a transition with $n = (q, s)$ and another transition with (q, q') in the argument with $q' = A^t(s')$. States (q, s) and (q, q') are equivalent in the product DTA $A^\dagger$ before $A^\dagger(t)$ is added or removed from $F^\dagger$. For the sake of simplicity, let us assume that $C_q = 2$. If $(q, q') = (q, \perp^t)$ then (q, q') is an intact state equivalent to q; however, if $s' \in \text{sub}(t)$ then (q, s') can be identified with the

```
Function clone
Input: a DTA $A$, a state $q \in Q$, and a state $n \in Q$ without transitions in $\Delta$.
Output: an equivalent DTA $\widehat{A}$ where $n$ and $q$ are equivalent.
Method:

1. If $q \in F$ then add $n$ to $F$.
2. While there exist $m > 0$, $k \leq m$ and $(\sigma, i_1, \ldots, i_m, j) \in \Delta$ such that $i_k = q$ and
   $\delta_m(\sigma, i_1, \ldots, i_{k-1}, n, i_{k+1}, \ldots, i_m) = \perp$ add a new transition $(\sigma, i_1, \ldots, i_{k-1}, n,$
   $i_{k+1}, \ldots, i_m, j)$ to $\Delta$.
3. Return $A$.
```

Fig. 3 Function `clone`

recycled state q, provided that A is modified so that $L_A(q) = \{s'\}$. In both cases, n will be called a *clone* of q in $A^\dagger$. The generalization for $C_q > 2$ is straightforward.

Function `clone`, shown in Fig. 3, modifies A so that n becomes trivially equivalent to q just by adding n to F if q is in F and by adding the missing transitions to Δ, that is, creating new transitions from the existing ones by replacing one q in the argument with n. Note that transitions recently added to Δ are also checked at step 2 of `clone`. For instance, when making state q_3 a clone of q_1—as in the example in Sect. 3.4—if $\delta_2(a, q_1, q_1) = q_2$ then step 2 of `clone` adds three transitions to Δ with output q_2: (a, q_3, q_1, q_2), (a, q_1, q_3, q_2) and (a, q_3, q_3, q_2). As cloning cannot modify inequivalences $p \not\equiv q$ for states p and q other than n, then, Θ remains unchanged.

3.3 Local Minimization

The union DTA $A^\dagger$ obtained after step 1 of Algorithm `AddTree` accepts $L(A) \cup \{t\}$ but is not necessarily minimal. A register R containing mutually inequivalent states—that is, the part of the automaton that cannot be further minimized—can be used to speed up the minimization process as the minimization will only check if unregistered states are equivalent to a state in R. In such case, they are merged with the equivalent one; otherwise, they are added to the register.

After `split`, R contains only those states in $Q^\dagger$ which are not used in the computation of $A^\dagger(t)$, that is, $R = Q^\dagger - \{A^\dagger(s) : s \in \text{sub}(t)\}$. As will be proven later, the class of states in the register (that is, whether they are accepting or not) is not changed by `split` and any transitions in Δ containing them in the argument remain also unchanged in $\Delta^\dagger$ (and lead to states which are also in R).

Local minimization is performed in our algorithm by means of function `minim` (Fig. 4) which takes three parameters as input: the automaton $A^\dagger$, its register R and the auxiliary set Θ. It returns the minimal automaton A' after merging pairs of equivalent states.

A useful property of the input is that if the states in $Q^\dagger - R$ are processed in topological order, then the equivalence check in (10) is non-recursive, that is, the transition outputs are equivalent if and only if they are identical. The topological ordering is such that, for all $s \in \text{sub}(t)$ and all $s' \in \text{sub}(s)$, either $A^\dagger(s')$ is after $A^\dagger(s)$

```
Function minim
Input: a DTA $A^\dagger$, a register $R$ and an auxiliary set of states $\Theta$.
Output: a minimal equivalent DTA $A'$.
Method:

1. $\Omega \leftarrow$ TopologicalSort$(Q - R)$ (* Create agenda *).
2. While $\Omega \neq \emptyset$ do
   (a) Pop the first item $n$ in $\Omega$ and let $(\sigma, r_1, \dots, r_m, n)$ be the only transition with
       output $n$ in $\Delta^\dagger$.
   (b) $q \leftarrow$ findEquiv$(A^\dagger, R, \Theta, n)$.
   (c) If $q \neq n$ then (* Merge $q$ and $n$ *)
       • Replace $(\sigma, r_1, \dots, r_m, n)$ with $(\sigma, r_1, \dots, r_m, q)$ in $\Delta^\dagger$.
       • Remove $n$ from $Q^\dagger$, $F^\dagger$, $\Delta^\dagger$ and $\Theta^\dagger$.
       • Add $r_1, \dots, r_m$ to $\Theta^\dagger$
       else
       • Add $n$ to $R$
3. Return $A^\dagger$.
```

Fig. 4 Function `minim` for local minimization

```
Function findEquiv
Input: The DTA $A^\dagger$, its register $R$, the auxiliary set $\Theta$ and a state $p \in Q^\dagger$.
Output: A state $q$ equivalent to $p$ in $A^\dagger$.
Method:

1. If $p \in \Theta$ and there is $q \in R$ such that $q \in F$ if and only if $p \in F$ and for all
   $(\sigma, i_1, \dots, i_m, j) \in \Delta$ and for all $k \leq m$ such that $i_k = p$ or $i_k = q$

       $\delta_m(\sigma, i_1, \dots, i_{k-1}, p, i_{k+1} \dots, i_m) = \delta_m(\sigma, i_1, \dots, i_{k-1}, q, i_{k+1} \dots, i_m)$

   then return $q$.
2. Otherwise return $p$.
```

Fig. 5 Function `findEquiv`

or $s' = s$. Therefore, the minimization procedure creates a stack Ω (called *agenda*) loaded with all states in $Q^\dagger - R$ in topological order. In practice, the initial contents of Ω—as well as R—can be easily computed by function `split`.

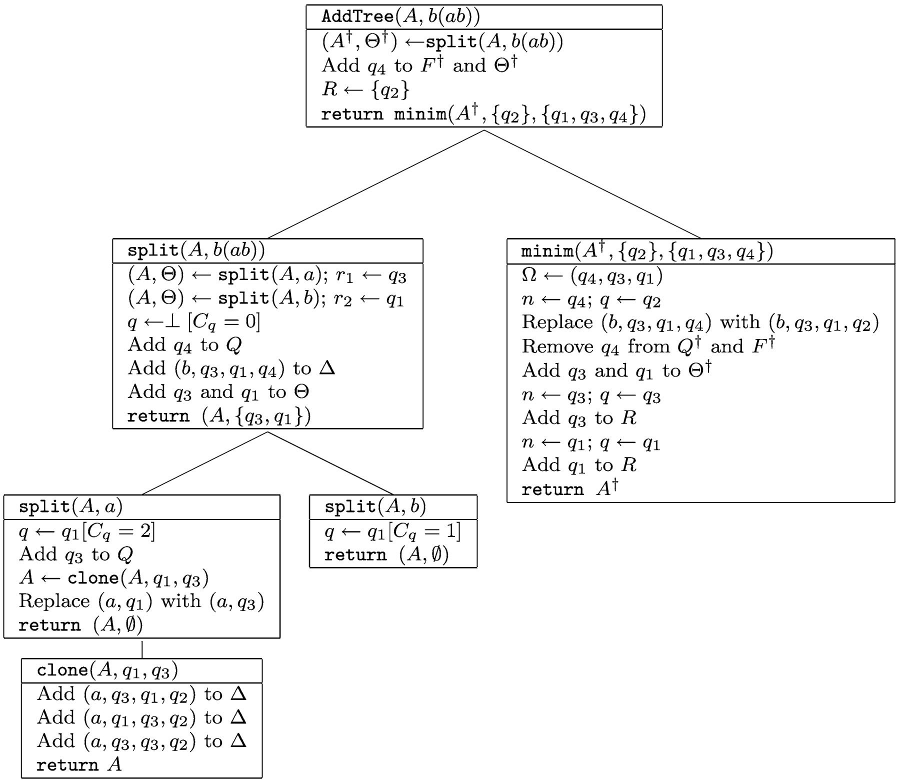
The set Θ contains states appearing in the argument of transitions with new output—therefore, including also new states—plus the state $A^\dagger(t)$, added to $F^\dagger$ and to $\Theta^\dagger$ at step 1(b) in `AddTree`. Every time an equivalent state $q \in R$ is found for a state $n \in Q^\dagger - R$, the state n is merged with its equivalent one q and states $r_1, \dots, r_m$ in the single transition $(\sigma, r_1, \dots, r_m, n)$ with output n may become equivalent to some state in R and are then added to Θ.

The search for equivalent states in the register R can be performed efficiently by implementing *signatures* (which may be used to obtain *hash codes* such as the ones used to speed up searches). For instance, the signature $\text{sig}(q)$ of a state q could be defined as

$$\text{sig}(q) = \{(\sigma, m, k) : \exists(\sigma, i_1, \ldots, i_m, j) \in \Delta^\dagger : \exists k \le m : i_k = q\}. \tag{11}$$

If the output of transitions containing q is in R, a state $p \in R$ cannot be equivalent to q unless $\text{sig}(p) = \text{sig}(q)$. Therefore, the search for an equivalent state in R can be limited to the subset of states in R having identical signature and class (accepting or non-accepting), a subset that will always be smaller. As the signature of a state does not depend on any transition output, the local minimization does not change

Input: $A = (Q, \Sigma, \Delta, F)$	Output: $A' = (Q', \Sigma, \Delta', F')$
$Q = \{q_1, q_2\}$	$Q' = \{q_1, q_2, q_3\}$
$\Sigma = \{a, b\}$	$\Delta' = \{(a, q_3), (b, q_1), (a, q_1, q_1, q_2),$ $(a, q_1, q_3, q_2), (a, q_3, q_1, q_2),$ $(a, q_3, q_3, q_2), (b, q_3, q_1, q_2)$
$\Delta = \{(a, q_1), (b, q_1), (a, q_1, q_1, q_2)\}$	
$F = \{q_2\}$	$F' = \{q_2\}$



```
AddTree($A, b(ab)$)
  ($A^\dagger, \Theta^\dagger$) ←split($A, b(ab)$)
  Add $q_4$ to $F^\dagger$ and $\Theta^\dagger$
  $R \leftarrow \{q_2\}$
  return minim($A^\dagger, \{q_2\}, \{q_1, q_3, q_4\}$)
```

```
split($A, b(ab)$)
  ($A, \Theta$) ← split($A, a$); $r_1 \leftarrow q_3$
  ($A, \Theta$) ← split($A, b$); $r_2 \leftarrow q_1$
  $q \leftarrow \perp$ [$C_q = 0$]
  Add $q_4$ to $Q$
  Add $(b, q_3, q_1, q_4)$ to $\Delta$
  Add $q_3$ and $q_1$ to $\Theta$
  return ($A, \{q_3, q_1\}$)
```

```
minim($A^\dagger, \{q_2\}, \{q_1, q_3, q_4\}$)
  $\Omega \leftarrow (q_4, q_3, q_1)$
  $n \leftarrow q_4$; $q \leftarrow q_2$
  Replace $(b, q_3, q_1, q_4)$ with $(b, q_3, q_1, q_2)$
  Remove $q_4$ from $Q^\dagger$ and $F^\dagger$
  Add $q_3$ and $q_1$ to $\Theta^\dagger$
  $n \leftarrow q_3$; $q \leftarrow q_3$
  Add $q_3$ to $R$
  $n \leftarrow q_1$; $q \leftarrow q_1$
  Add $q_1$ to $R$
  return $A^\dagger$
```

```
split($A, a$)
  $q \leftarrow q_1[C_q = 2]$
  Add $q_3$ to $Q$
  $A$ ← clone($A, q_1, q_3$)
  Replace $(a, q_1)$ with $(a, q_3)$
  return ($A, \emptyset$)
```

```
split($A, b$)
  $q \leftarrow q_1[C_q = 1]$
  return ($A, \emptyset$)
```

```
clone($A, q_1, q_3$)
  Add $(a, q_3, q_1, q_2)$ to $\Delta$
  Add $(a, q_1, q_3, q_2)$ to $\Delta$
  Add $(a, q_3, q_3, q_2)$ to $\Delta$
  return $A$
```

Fig. 6 Trace of AddTree$(A, b(ab))$

signatures, and they can be computed (and stored) in advance and updated when required during the split phase.[3]

3.4 A Simple Example

As an illustration of the whole addition process, Fig. 6 shows a trace of the algorithm `AddTree` when the tree $t = b(ab)$ is added to the minimal DTA accepting the language $\{a(aa), a(ab), a(ba), a(bb)\}$.

States in $Q^\dagger$ and their corresponding states in $Q \times Q^t$ are $q_1 = (q_1, b)$, $q_2 = (q_2, \perp^t)$, $q_3 = (q_1, a)$ and $q_4 = (\perp, b(ab))$. Cloning q_3 and q_1 implicitly generates the following transitions in $A^\dagger$: $(a, (1, b), (1, a), (2, a(ba)))$,$(a, (1, a), (1, b), (2, a(ab)))$ and $(a, (1, b), (1, b), (2, a(bb)))$. Later, local minimization merges q_4 and q_2.

Note that while the original DTA in the example has only three transitions, the addition of a single subtree a of size one duplicates the number of transitions and gives a new automaton, the one returned by $\texttt{split}(A, a)$, with $F^\dagger = \{q_1, q_2\}$ and

$$\Delta^\dagger = \{(a, q_3), (b, q_1), (a, q_1, q_1, q_2), (a, q_3, q_1, q_2), (a, q_1, q_3, q_2), (a, q_3, q_3, q_2)\}.$$

This illustrates the fact that, in some cases, the size of the output A' can be exponentially larger than the size of the input, with a direct effect on the computational cost of the algorithm.

4 Algorithm Analysis

The next proposition shows the main features of the output DTA $A^\dagger$ obtained after function `split` is applied.

Proposition 1 *A call to the procedure* `split` *with parameters A (a minimal DTA) and t (a tree) outputs a new automaton $A^\dagger = \texttt{split}(A, t)$ such that*

1. *$A^\dagger$ is deterministic, $Q^\dagger$ contains no inaccessible states and $\Delta^\dagger$ contains no transitions with output $\perp$.*
2. *$L_{A^\dagger}(q) = L_A(q)$ for all $q \in Q$ such that $|L_A(q)| = 1$.*
3. *$L_{A^\dagger}(A^\dagger(s)) = \{s\}$ for all $s \in \mathrm{sub}(t)$.*
4. *$L_A(q) \subseteq L_{A^\dagger}(q) \cup \mathrm{sub}(t)$ for all $q \in Q$.*

Proof 1. The automaton remains deterministic as the modifications to Δ do not add transitions with already existing arguments: step 2 of `clone` explicitly checks that no transition exists with the same argument and step 3(b) of `split` only adds a transition with argument $(\sigma, r_1, \ldots, r_m)$ if $\delta_m(\sigma, r_1, \ldots, r_m) = \perp$.

Inaccessible states can be generated whenever a new state is added to Q or a transition is removed from Δ. As the input DTA contains no inaccessible states, it is

[3] If n is a clone of q then n and q have identical signature; when $(\sigma, r_1, \ldots, r_m, n)$ is added to Δ, (σ, m, k) is added to $\mathrm{sig}(r_k)$.

trivially true that the automaton contains no inaccessible states just after the step 2 in function `split` when called with input $s = \sigma$ (then, $m = 0$) and can be inductively assumed when called with $m > 0$.

If q is accessible, there is a sequence $j_1, \dots, j_N$ of different states with $j_N = q$ and with $j_1 = \delta_0(\sigma)$ for some $\sigma \in \Sigma$ such that, for all n, j_{n+1} is the output of a transition containing j_n in the argument. Therefore, as `clone` does not add states or remove transitions, every new state n created by `split` becomes accessible when the transition $(\sigma, r_1, \dots, r_m, n)$ is added to Δ in step 4(b).

As a consequence, replacing a transition $(\sigma, r_1, \dots, r_m, q)$ when $C_q > 1$ can render no state $p \neq q$ inaccessible because cloning q creates, among others, transitions with all appearances of q in the argument replaced with n. Furthermore, $C_q > 1$ implies that, after cloning q, there is at least one transition in Δ with output q and no q in the argument and, therefore, also q remains accessible. Thus, the automaton returned contains no inaccessible states.

Finally, all transitions added by the algorithm have as output either a new state (`split`) or the output of a transition already in Δ (`clone`).

2. If $|L_A(q)| = 1$ then $L_A(q)$ contains a single tree z and, therefore, $C_p = 1$ for all states[4] p in $S_q = \{A(s) : s \in \text{sub}(z)\}$. On the one hand, no step in function `split` can modify a transition in Δ with output p if $C_p = 1$. On the other hand, cloning a transition with output p requires that there is a state r in the argument with $C_r > 1$ in contradiction with the fact that $|L_A(p)| = 1$. Thus, no transitions with output in S_q are added to or removed from Δ and $L_A(q) = \{z\}$ remains unchanged.

3. Recursively, `split`(A, s) is called for all $s \in \text{sub}(t)$. Let $s = \sigma \in \Sigma$ and $q = A(\sigma)$. If $C_q = 1$ then $L_A(q) = \{\sigma\}$, `split` returns $A^\dagger = A$ and, trivially, $A^\dagger(\sigma) = q$ and $L_{A^\dagger}(q) = \{\sigma\}$. Otherwise, `split` creates a new state n with a single transition (σ, n) in $\Delta^\dagger$ and, then, $A^\dagger(\sigma) = n$ and $L_{A^\dagger}(n) = \{\sigma\}$.

Let $s = \sigma(s_1, \dots, s_m)$, $q = A(s)$ and assume $L_A(r_k) = \{s_k\}$ for $r_k = A(s_k)$ after step 2 in function `split`. As discussed in the proof of Proposition 1.2, $L_A(r_k)$ cannot be modified after `split`(A, s_k) is complete since, then, $|L_A(s_k)| = 1$. If $C_q = 1$ then `split` returns $A^\dagger = A$, $q = A^\dagger(s)$ and the only transition with output q remains $(\sigma, r_1, \dots, r_m, q)$; therefore, $L_{A^\dagger}(q) = \{\sigma(s_1, \dots, s_m)\}$. Otherwise `split` creates a new state n with a single transition $(\sigma, r_1, \dots, r_m, n)$ in $\Delta^\dagger$ and $A^\dagger(s) = n$; therefore, $L_{A^\dagger}(n) = \{\sigma(s_1, \dots, s_m)\}$.

4. This statement can be proved analogously to the previous one. As `clone` does not add to Δ transitions with output n, then $L_{\widehat{A}}(n)$ remains empty with $\widehat{A} =$ `clone`(A, q, n) and $L_{\widehat{A}}(p) = L_A(p)$ for all $p \in Q$. Thus, languages $L_A(p)$ can be only modified at step 4(b) in function `split`, provided that there is a state $q \in Q$ such that $C_q > 1$. In that case, replacing $(\sigma, r_1, \dots, r_m, q)$ with $(\sigma, r_1, \dots, r_m, n)$ removes the subtree $s = \sigma(s_1, \dots, s_m)$ from $L_A(q)$. Further recursive effects do not take place because for every q in the argument of a transition there is a cloned transition where q is replaced with n. Therefore, `split` may only remove subtrees of t from the languages accepted by states in the input DTA A. □

[4]This can be easily proven by *reductio ad absurdum*: in a DTA A without inaccessible states, $C_p > 1$ implies $|L_A(p)| > 1$.

Corollary 1 *The DTA $A^\dagger = \texttt{split}(A, t)$ satisfies $A^\dagger(s) = A(s)$ for all $s \notin \mathrm{sub}(t)$.*

Proof Let $q = A(s)$ and recall that, according to Proposition 1.4, $L_A(q) \subseteq L_{A^\dagger}(q) \cup \mathrm{sub}(t)$. Therefore, if $s \notin \mathrm{sub}(t)$ then $s \in L_{A^\dagger}(q)$ and $A^\dagger(s) = A(s)$. □

Corollary 2 *The DTA $A' = (Q^\dagger, \Sigma, \Delta^\dagger, F^\dagger \cup \{A^\dagger(t)\})$ and the DTA $A'' = (Q^\dagger, \Sigma, \Delta^\dagger, F^\dagger - \{A^\dagger(t)\})$ accept respectively $L(A') = L(A) \cup \{t\}$ and $L(A'') = L(A) - \{t\}$.*

Proof On the one hand, the state $q = A^\dagger(t)$ is in F' but not in F'' and, according to Proposition 1.3, $L_{A^\dagger}(q) = \{t\}$ and, therefore, $t \in L(A')$ and $t \notin L(A'')$.

On the other hand, for any $q \in F$, the subtrees $s \in \mathrm{sub}(t) \cap L_A(q)$ are not necessarily in $L_{A^\dagger}(q)$. However, in such a case, $A^\dagger(s)$ is a new state $n \notin Q$ and, according to Proposition 1.3, $L_{A^\dagger}(n) = \{s\}$. As the first step in `clone` guarantees that n is in $F^\dagger$, then s is in $L(A^\dagger)$. □

As a consequence of the previous corollary, efficient removal of trees may be accomplished also by means of algorithm `AddTree` just by changing step 1(b) so that it removes $A^\dagger(t)$ from $F^\dagger$ (and adds it to Θ).

Corollary 3 *All subtrees $s \in \mathrm{sub}(t)$ generate a different output in $Q^\dagger$ when $A^\dagger$ operates on them and, then, a topological order can be defined in $B = \{A^\dagger(s) : s \in \mathrm{sub}(t)\}$.*

Proof As there is a one-to-one mapping between $\mathrm{sub}(t)$ and B, the hierarchical relations of subtrees in t can be directly mapped to the states in B. □

The following proposition justifies that the local minimization performed by function `minim` is correct.

Proposition 2 *The following statements remain true during the execution of* `minim`$(A^\dagger, R, \Theta)$

1. *All states in $Q^\dagger$ are either in the register R or in the agenda Ω.*
2. *If n is the top element in the stack Ω then $|L_{A^\dagger}(n)| = 1$, the states appearing in the argument of the transition with output n are not in R and all transitions in $\Delta^\dagger$ where n appears in the argument have their output in R.*
3. *All states in the register R are mutually inequivalent.*
4. *If n is the top element in the stack Ω and $n \in Q^\dagger - \Theta$ no state equivalent to n can be found in R. Otherwise, the search for an equivalent state in R can be done non-recursively.*
5. *The language $L(A^\dagger)$ accepted by the DTA remains unchanged.*

Proof 1. Initially, Ω contains all elements in $Q^\dagger - R$. Whenever a state is popped from Ω, it is either removed from $Q^\dagger$ or added to R.

2. From Proposition 1.3, $L_{A^\dagger}(n)$ contains initially a single subtree $s \in \mathrm{sub}(t)$. Transitions are only modified by `minim` when equivalent states are merged but, due

to the topological sort, no transition with output in $\{A^\dagger(s') : s' \in \mathrm{sub}(s)\}$ is changed before n is popped from Ω and, then, $L_{A^\dagger}(n)$ remains unchanged and Ω topologically sorted.

As a consequence, there is a single transition $(\sigma, r_1, \ldots, r_m, n)$ with output n, and all states r_k in its argument satisfy $r_k = A^\dagger(s_k)$ for some subtree $s_k \in \mathrm{sub}(t)$.

Assume that there is a transition $(\sigma, i_1, \ldots, i_m, j)$ and $k \leq m$ such that $i_k = n$ and $j \in Q^\dagger - R$. Then, according to Proposition 2.1, j must be in Ω and $L_{A^\dagger}(j)$ contains a single tree $s = \sigma(s_1, \ldots, s_m)$. As n is also in Ω there is s' such that $L_{A^\dagger}(n) = \{s'\}$ and $s' = s_k$. However, $s' \in \mathrm{sub}(s)$ is in contradiction with $j = A^\dagger(s)$ being below the top state $n = A^\dagger(s')$ in the topologically sorted stack Ω.

3. As A is a minimal DTA, initially, all pairs of states p, q in R satisfy at least one of the following:

(a) $p \in F$ and $q \notin F$ or vice versa. Then, p and q will remain inequivalent in $A^\dagger$ because no state removed from or added to $F^\dagger$ is in R.
(b) There exist $m > 0$, $k \leq m$ and $(\sigma, r_1, \ldots, r_m) \in \Sigma \times Q^m$ such that

$$\delta_m(\sigma, r_1, \ldots, r_{k-1}, p, r_{k+1} \ldots, r_m) \neq \delta_m(\sigma, r_1, \ldots, r_{k-1}, q, r_{k+1} \ldots, r_m).$$

In this case, the inequality will remain true in $A^\dagger$ because no state in R is used by transitions whose output in $\Delta^\dagger$ differs from that in Δ and the outputs are in R.

The automaton $A^\dagger$ is later modified by `minim` but mutual inequivalence is explicitly required for new additions to R and merging equivalent states does not modify the inequivalences.

4. On the one hand, as the output of transitions using n are in R, the arguments used in the previous paragraph to justify that two states p and q in R are mutually inequivalent are also valid for $p \in R$ and n.

On the other hand, whenever function `findEquiv` is called, all transitions using n have an output in R and, therefore, if the output states are inequivalent, they must be different states. Therefore, no further recursive computation of equivalence is necessary.

5. The only place where $A^\dagger$ is modified is when a state n is replaced with an equivalent q one found in the register. In such case, if n is in $F^\dagger$ so it is q and for all transitions using n in the argument, there is another transition with all n replaced with q so that, n can be safely deleted along with its outgoing transitions without changing the language. □

Corollary 4 *The output $A' =$ `minim`$(A^\dagger, R, \Theta)$ is a minimal DTA such that $L(A') = L(A) \cup \{t\}$.*

Proof According to Proposition 2, $L(A') = L(A^\dagger)$ and all states in R are mutually inequivalent. Upon termination (after at most $|t|$ iterations), Ω is necessarily empty and, therefore, Q' coincides with R. □

5 Conclusions and Future Work

Deterministic frontier-to-root tree automata may be seen as a very efficient way of storing (possibly infinite) tree collections, much in the same way as deterministic finite-state automata are a very efficient way of storing a (possibly infinite) language or dictionary.

Analogously to the way in which Carrasco and Forcada [6], and Daciuk and van Noord [11] described how to incrementally build a minimal DFA from strings, this paper describes an algorithm to build incrementally a minimal DTA from trees; that is, an algorithm that adds or removes a tree from the language accepted by a DTA which is already minimal, and delivers a DTA which is also minimal.

A number of possible improvements are possible, which will be reported elsewhere; we are currently working on:

- exploring the advantages of sorting trees before adding them to the automaton; in reference [10] this was shown to speed up the incremental construction of DFAs;
- extending the algorithm to efficiently add finite languages of trees by building the analogous to a prefix tree DFA and merging it with the minimal DTA.

Finally, we plan to use DTA to implement a perfect hashing function for a set of trees, much in the same way as a minimal deterministic string automaton may have its transitions or states labeled with positive integers [17] that, added, compute a unique number for each string.

Acknowledgements This work was supported by the Spanish Ministerio de Educación y Ciencia through grants TIC2003-08681-C02-01 and TIN2006-15071-C03-01.

References

1. Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation and Compiling. Vol. I: Parsing. Prentice-Hall, London (1972)
2. Aoe, J., Morimoto, K., Hase, M.: An algorithm for compressing common suffixes used in tree structures. Syst. Comput. Jpn. **24**(12), 31–42 (1993) (translated from Trans. IEICE **J75-D-II**(4), 770–779, 1992)
3. Bod, R.: A computational model of language performance: Data oriented parsing. In: Proceedings of the 14th Conference on Computational Linguistics, pp. 855–859. Association for Computational Linguistics, Morristown (1992)
4. Brainerd, W.S.: The minimalization of tree automata. Inf. Control **13**(5), 484–491 (1968)
5. Carrasco, R.C., Daciuk, J., Forcada, M.L.: An implementation of deterministic tree automata minimization. In: Holub., J., Zdárek, J. (eds.) CIAA2007, 12th International Conference on Implementation and Application of Automata Proceedings. Lecture Notes in Computer Science, vol. 4783, pp. 122–129. Springer, New York (2007)
6. Carrasco, R.C., Forcada, M.L.: Incremental construction and maintenance of minimal finite-state automata. Comput. Linguistics **28**(2), 207–216 (2002)
7. Ciura, M., Deorowicz, S.: How to squeeze a lexicon. Softw. Pract. Experience **31**(11), 1077–1090 (2001)
8. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata (1997), release 1 October 2002
9. Daciuk, J.: Comments on incremental construction and maintenance of minimal finite-state automata by R.C. Carrasco and M.L. Forcada. Comput. Linguistics **30**(2), 227–235 (2004)

10. Daciuk, J., Mihov, S., Watson, B.W., Watson, R.E.: Incremental construction of minimal acyclic finite-state automata. Comput. Linguistics **26**(1), 3–16 (2000)
11. Daciuk, J., van Noord, G.: Finite automata for compact representation of language models in NLP. In: Implementation and Application of Automata, 6th International Conference, CIAA 2001. Lecture Notes in Computer Science, vol. 2494, pp. 65–73. Springer, New York (2002)
12. de la Briandais, R.: File searching using variable length keys. In: Proceedings of the Western Joint Computer Conference, pp. 295–298 (1959)
13. Fredkin, E.: Tree memory. Commun. ACM **3**(9), 490–499 (1960)
14. Garrido-Alenda, A., Forcada, M.L., Carrasco, R.C.: Incremental construction and maintenance of morphological analysers based on augmented letter transducers. In: Proceedings of TMI 2002, Theoretical and Methodological Issues in Machine Translation, Keihanna/Kyoto, Japan, March 2002, pp. 53–62 (2002)
15. Gécseg, F., Steinby, M.: Tree Automata. Akadémiai Kiadó, Budapest (1984)
16. Hopcroft, J., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
17. Lucchesi, C.L., Kowaltowski, T.: Applications of finite automata representing large vocabularies. Softw. Pract. Experience **23**(1), 15–30 (1993)
18. Revuz, D.: Dynamic acyclic minimal automaton. In: Yu, S., Paun, A. (eds.) CIAA 2000, Fifth International Conference on Implementation and Application of Automata. Lecture Notes in Computer Science, vol. 2088, pp. 226–232. Springer, New York (2000)
19. Sgarbas, K., Fakotakis, N., Kokkinakis, G.: Two algorithms for incremental construction of directed acyclic word graphs. Int. J. Artif. Intell. Tools **4**(3), 369–381 (1995)
20. Watson, B.W., Daciuk, J.: An efficient incremental DFA minimization algorithm. Nat. Lang. Eng. **9**(1), 49–64 (2003)