Incremental Construction and Maintenance of Minimal Finite-State Automata

Rafael C. Carrasco*
Universitat d'Alacant

Mikel L. Forcada†
Universitat d'Alacant

Daciuk et al. [Computational Linguistics 26(1):3–16 (2000)] describe a method for constructing incrementally minimal, deterministic, acyclic finite-state automata (dictionaries) from sets of strings. But acyclic finite-state automata have limitations: For instance, if one wants a linguistic application to accept all possible integer numbers or Internet addresses, the corresponding finite-state automaton has to be cyclic. In this article, we describe a simple and equally efficient method for modifying any minimal finite-state automaton (be it acyclic or not) so that a string is added to or removed from the language it accepts; both operations are very important when dictionary maintenance is performed and solve the dictionary construction problem addressed by Daciuk et al. as a special case. The algorithms proposed here may be straightforwardly derived from the customary textbook constructions for the intersection and the complementation of finite-state automata; the algorithms exploit the special properties of the automata resulting from the intersection operation when one of the finite-state automata accepts a single string.

1. Introduction

In a recent paper in this journal, Daciuk et al. (2000) describe two methods for constructing incrementally minimal, deterministic, acyclic finite-state automata (dictionaries) from sets of strings: The first method adds strings in dictionary order, and the second one is for unsorted data. Adding an entry is an important dictionary maintenance operation, but so is removing an entry from the dictionary, for example, if it is found to be incorrect. Since ordering cannot obviously be expected in the removal case, we will focus on the second, more general problem (a solution for which has already been sketched by Revuz [2000]).

But dictionaries or acyclic finite automata have limitations: For instance, if one wants an application to accept all possible integer numbers or Internet addresses, the corresponding finite-state automaton has to be cyclic. In this article, we show a simple and equally efficient method for modifying *any* minimal finite-state automaton (be it acyclic or not) so that a string is added to or removed from the language it accepts. The algorithm may be straightforwardly derived from customary textbook constructions for the intersection and the complementation of finite-state automata; the resulting algorithm solves the dictionary construction problem addressed by Daciuk et al.'s (2000) second algorithm as a special case.

* Departament de Llenguatges i Sistemes Informàtics, Universitat d'Alacant, E-03071 Alacant, Spain. E-mail: carrasco@dlsi.ua.es
† Departament de Llenguatges i Sistemes Informàtics, Universitat d'Alacant, E-03071 Alacant, Spain. E-mail: mlf@dlsi.ua.es.

© 2002 Association for Computational Linguistics

This article has the following parts. In Section 2, we give some necessary mathematical preliminaries. The minimal automata resulting from adding or removing a string are described in detail in Section 3; the algorithms are described in Section 4. In Section 5, one addition and one removal example are explained in detail, and some closing remarks are given in Section 6.

2. Mathematical Preliminaries

2.1 Finite-State Automata and Languages

As in Daciuk et al. (2000), we will define a deterministic finite-state automaton as $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, $q_0 \in Q$ is the start state, $F \subseteq Q$ is a set of accepting states, Σ is a finite set of symbols called the alphabet, and δ: $Q \times \Sigma \rightarrow Q$ is the next-state mapping. In this article, we will define δ as a *total* mapping; the corresponding finite-state automaton will be called **complete** (Revuz 2000). This involves no loss of generality, as any finite-state automaton may be made complete by adding a new **absorption** state $\perp$ to Q, so that all undefined transitions point to it and $\delta(\perp, a) = \perp$ for all $a \in \Sigma$. Using complete finite-state automata is convenient for the theoretical discussion presented in this article; real implementations of automata and the corresponding algorithms need not contain an explicit representation of the absorption state and its incoming and outgoing transitions.

For complete finite-state automata, the extended mapping δ^*: $Q \times \Sigma^* \rightarrow Q$ (the extension of δ for strings) is defined simply as

$$\begin{aligned} \delta^*(q, \epsilon) &= q \\ \delta^*(q, ax) &= \delta^*(\delta(q, a), x) \end{aligned} \tag{1}$$

for all $a \in \Sigma$ and $x \in \Sigma^*$, with ϵ the empty or null string. The language accepted by automaton M

$$\mathcal{L}(M) = \{w \in \Sigma^*\colon\ \delta^*(q_0, w) \in F\} \tag{2}$$

and the right language of state q

$$\vec{\mathcal{L}}(q) = \{x \in \Sigma^*\colon\ \delta^*(q, x) \in F\} \tag{3}$$

are defined as in Daciuk et al. (2000).

2.2 Single-String Automaton

We also find it convenient to define the (complete) **single-string automaton** for string w, denoted $M_w = (Q_w, \Sigma, \delta_w, q_{0w}, F_w)$, such that $\mathcal{L}(M_w) = \{w\}$. This automaton has $Q_w = \mathrm{Pr}(w) \cup \{\perp_w\}$, where $\mathrm{Pr}(w)$ is the set of all prefixes of w and $\perp_w$ is the absorption state, $F_w = \{w\}$, and $q_{0w} = \epsilon$ (note that nonabsorption states in Q_w will be named after the corresponding prefix of w). The next-state function is defined as follows

$$\delta(x, a) = \begin{cases} xa & \text{if } x, xa \in \mathrm{Pr}(w) \\ \perp_w & \text{otherwise} \end{cases} \tag{4}$$

Note that the single-string automaton for a string w has $|Q_w| = |w| + 2$ states.

2.3 Operations with Finite-State Automata

2.3.1 Intersection Automaton. Given two finite-state automata M_1 and M_2, it is easy to build an automaton M so that $\mathcal{L}(M) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$. This construction is found

in formal language theory textbooks (Hopcroft and Ullman 1979, page 59) and is referred to as *standard* in papers (Karakostas, Viglas, and Lipton 2000). The (complete) **intersection automaton** has $Q = Q_1 \times Q_2$, $q_0 = (q_{01}, q_{02})$, $F = F_1 \times F_2$, and $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ for all $a \in \Sigma$, $q_1 \in Q_1$ and $q_2 \in Q_2$.

2.3.2 Complementary Automaton. Given a complete finite-state automaton M, it is easy to build its **complementary automaton** $\bar{M}$ so that $\mathcal{L}(\bar{M}) = \Sigma^* - \mathcal{L}(M)$; the only change is the set of final states: $\bar{F} = Q - F$ (Hopcroft and Ullman 1979, page 59). In particular, the **complementary single-string automaton** M_{-w} accepting $\Sigma^* - \{w\}$ is identical to M_w except that $F_{-w} = Q - \{w\}$.

2.3.3 Union Automaton. The above constructions may be combined easily to obtain a construction to build, from two complete automata M_1 and M_2, the (complete) **union automaton** M such that $\mathcal{L}(M) = \mathcal{L}(M_1) \cup \mathcal{L}(M_2)$. It suffices to consider that, for any two languages on Σ^*, $L_1 \cup L_2 = \Sigma^* - (\Sigma^* - L_1) \cap (\Sigma^* - L_2)$. The resulting automaton M is identical to the intersection automaton defined above except that $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$.

3. Adding and Removing a String

3.1 Adding a String

Given a (possibly cyclic) minimal complete finite-state automaton M, it is easy to build a new complete automaton M' accepting $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$ by applying the union construct defined above to M and the single-string automaton M_w. The resulting automaton $M' = (Q', \Sigma, \delta', q_0', F')$, which may be minimized very easily (see below), has four kinds of states in Q':

- States of the form $(q, \perp_w)$ with $q \in Q - \{\perp\}$, equivalent to those nonabsorption states of M that are not reached by any prefix of w; they will be called **intact** states, because they have the same transition structure as their counterparts in M (that is, if $\delta(q, a) = q'$, then $\delta'((q, \perp_w), a) = (q', \perp_w)$) and belong to F' if $q \in F$. As a result, they have exactly the same right languages, $\vec{\mathcal{L}}((q, \perp_w)) = \vec{\mathcal{L}}(q)$, because all of their outgoing transitions go to other intact states. Furthermore, each state $(q, \perp_w)$ has a different right language; therefore, no two intact states will ever be merged into one by minimization (intact states may, however, be eliminated, if they become unreachable, as we will describe below). For large automata (dictionaries) M, these are the great majority of states (the number of intact states ranges between $|Q| - |w| - 1$ and $|Q|$); therefore, it will be convenient in practice to consider M' as a modified version of M, and it will be treated as such in the algorithms found in this article.
- States of the form (q, x) with $q \in Q - \{\perp\}$ and $x \in \Pr(w)$, such that $\delta^*(q_0, x) = q$; they will be called **cloned** states, inspired by the terminology in Daciuk et al. (2000); the remaining states in $(Q - \{\perp\}) \times \Pr(w)$—the great majority of states in $Q \times Q_w$—may safely be discarded because they are unreachable from the new start state $q_0' = (q_0, \epsilon)$, which itself is a cloned state. Cloned states are modified versions of the original states $q \in Q - \{\perp\}$: All of their outgoing transitions point to the corresponding intact states in Q', that is, $(\delta(q, a), \perp_w)$, except for the transition with symbol $a : xa \in \Pr(w)$, which

now points to the corresponding cloned state $(\delta(q,a), xa)$, that is,

$$\delta'((q,x),a) = \begin{cases} (\delta(q,a), xa) & \text{if } xa \in \Pr(w) \\ (\delta(q,a), \perp_w) & \text{otherwise} \end{cases} \quad (5)$$

Cloned states are in F' if the corresponding original states are in F; in addition, if there is a cloned state of the form (q,w), then it is in F'. There are at most $|w|+1$ cloned states.

- States of the form $(\perp, x)$, with $x \in \Pr(w)$. These states will be called **queue** states; states of this form appear when the string w is not in $\mathcal{L}(M)$ (the pertinent case, because we are adding it) and only if in the original automaton $\delta^*(q_0, x) = \perp$ for some $x \in \Pr(w)$. Only the final queue state $(\perp, w)$—if it exists—is in F'. There are at most $|w|$ queue states.
- The new absorption state $\perp' = (\perp, \perp_w) \notin F$.

This automaton has to be minimized; because of the nature of the construction algorithm, however, minimization may be accomplished in a small number of operations. It is not difficult to show that minimization may be performed by initializing a list R called the **register** (Daciuk et al. 2000) with all of the intact states and then testing, one by one, queue and cloned states (starting with the last queue state $(\perp, w)$ or, if it does not exist, the last clone state (q,w), and descending in $\Pr(w)$) against states in the register and adding them to the register if they are not found to be equivalent to a state in R. (Performing this check backwards avoids having to test the equivalence of states by visiting their descendants recursively: see the end of Section 4.1.) Minimization (including the elimination of unreachable states in M') appears in Section 4 as part of the string addition and removal algorithms.

3.2 Removing a String

Again, given a (possibly cyclic) minimal complete finite-state automaton M, it is easy to build a new complete automaton M' accepting $\mathcal{L}(M') = \mathcal{L}(M) - \{w\} = \mathcal{L}(M) \cap (\Sigma^* - \{w\})$ by applying the intersection construct defined above to M and M_{-w}. The resulting automaton has the same sets of reachable states in Q' as in the case of adding string w and therefore the same close-to-minimality properties; since w is supposed to be in $\mathcal{L}(M)$, however, no queue states will be formed. (Note that, if $w \notin \mathcal{L}(M)$, a nonaccepting queue with all states eventually equivalent to $\perp' = (\perp, \perp_w)$ may be formed.) The accepting states in F' are intact states $(q, \perp_w)$ and cloned states (q,x) with $q \in F$, except for state (q,w). Minimization may be performed analogously to the string addition case.

4. Algorithms

4.1 Adding a String

Figure 1 shows the algorithm that may be used to add a string to an existing automaton, which follows the construction in Section 3.1. The resulting automaton is viewed as a modification of the original one: Therefore, intact states are not created; instead, unreachable intact states are eliminated later. The register R of states not needing minimization is initialized with Q. The algorithm has three parts:

- First, the cloned and queue states are built and added to Q by using function clone() for all prefixes of w. The function returns a cloned state

(with all transitions created), if the argument is a nonabsorption state in $Q - \{\perp\}$, or a queue state, if it operates on the absorption state $\perp \in Q$.

- Second, those intact states that have become unreachable as a result of designating the cloned state q'_0 as the new start state are removed from Q and R, and the start state is replaced by its clone. Unreachable states are simply those having no incoming transitions as constructed by the algorithm or as a consequence of the removal of other unreachable states; therefore, function unreachable() simply has to check for the absence of incoming transitions. Note that only intact states $(q, \perp_w)$ corresponding to q such that $\delta^*(q_0, x) = q$ for some $x \in \Pr(w)$ may become unreachable as a result of having been cloned.
- Third, the queue and cloned states are checked (starting with the last state) against the register using function replace_or_register(), which is essentially the same as the nonrecursive version in the second algorithm in Daciuk et al. (2000) and is shown in Figure 2. If argument state q is found to be equivalent to a state p in the register R, function merge(p, q) is called to redirect into p those transitions coming into q; if not, argument state q is simply added to the register. Equivalence is checked by function equiv(), shown in Figure 3, which checks for the equivalence of states by comparing *(1)* whether both states are accepting or not, and *(2)* whether the corresponding outgoing transitions lead to the same state in R. Note that outgoing transitions cannot lead to equivalent states, as there are no pairs of different equivalent states in the register $(\forall p, q \in R, \text{equiv}(p, q) \Rightarrow p = q)$ and backwards minimization guarantees that the state has no transitions to unregister states.

Finally, the new (minimal) automaton is returned. In real implementations, absorption states are not explicitly stored; this results in small differences in the implementations of the functions clone() and equiv().

4.2 Removing a String

The algorithm for removing a string from the language accepted by an automaton M' differs from the previous algorithm only in that the line

$$F \leftarrow F - \{q_{\text{last}}\}$$

has to be added after the first end_for. Since the string removal algorithm will usually be asked to remove a string that was in $\mathcal{L}(M)$, function clone() will usually generate only cloned states and no queue states (see Section 3.2 for the special case $w \notin L(M)$).

5. Examples

5.1 Adding a String

Assume that we want to add the string `bra` to the automaton in Figure 4, which accepts the set of strings $(\texttt{ba})^+ \cup \{\texttt{bar}\}$ (for clarity, in all automata, the absorption state and all transitions leading to it will not be drawn). The single-string automaton for string `bra` is shown in Figure 5. Application of the first stages of the string addition algorithm leads to the (unminimized) automaton in Figure 6. The automaton has, in addition to the set of intact states $\{(0, \perp_w), \ldots, (5, \perp_w)\}$, two cloned states ($(0, \epsilon)$ and $(1, \texttt{b})$) and two queue states ($(\perp, \texttt{br})$ and $(\perp, \texttt{bra})$). As a consequence of the designation of $(0, \epsilon)$ as the

algorithm **addstring**
Input: $M = (Q, \Sigma, \delta, q_0, F)$ (minimal, complete), $w \in \Sigma^*$
Output: $M' = (Q', \Sigma, \delta', q_0', F')$ minimal, complete, and such that $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$
$R \leftarrow Q$ [initialize register]
$q_0' \leftarrow \text{clone}(q_0)$ [clone start state]
$q_{\text{last}} \leftarrow q_0'$
for $i = 1$ to $|w|$
$q \leftarrow \text{clone}(\delta^*(q_0, w_1 \cdots w_i))$ [create cloned and queue states; add clones of accepting states to F]
$\delta(q_{\text{last}}, w_i) \leftarrow q$
$q_{\text{last}} \leftarrow q$
end_for
$i \leftarrow 1$
$q_{\text{current}} \leftarrow q_0$
while($i \leq |w|$ and unreachable(q_{current}))
$q_{\text{next}} \leftarrow \delta(q_{\text{current}}, w_i)$
$Q \leftarrow Q - \{q_{\text{current}}\}$ [remove unreachable state from Q and update transitions in δ]
$R \leftarrow R - \{q_{\text{current}}\}$ [remove also from register]
$q_{\text{current}} \leftarrow q_{\text{next}}$
$i \leftarrow i + 1$
end_while
if unreachable(q_{current})
$Q \leftarrow Q - \{q_{\text{current}}\}$
$R \leftarrow R - \{q_{\text{current}}\}$
end_if
$q_0 \leftarrow q_0'$ [replace start state]
for $i = |w|$ downto 1
replace_or_register($\delta^*(q_0, w_1 \cdots w_i)$) [check queue and cloned states one by one]
end_for
return $M = (Q, \Sigma, \delta, q_0, F)$
end_algorithm

Figure 1
Algorithm to add a string w to the language accepted by a finite-state automaton while keeping it minimal.

function **replace_or_register**(q)
if $\exists p \in R$: equiv(p, q) then
merge(p, q)
else
$R \leftarrow R \cup \{q\}$
end_if
end_function

Figure 2
The function replace_or_register().

```
function equiv(p,q)
        if (p ∈ F ∧ q ∉ F) ∨ (p ∉ F ∧ q ∈ F) return false
        for all symbols a ∈ Σ
            if δ(p,a) ≠ δ(q,a) return false
        end_for
        return true
end_function
```

Figure 3
The function equiv(p,q).

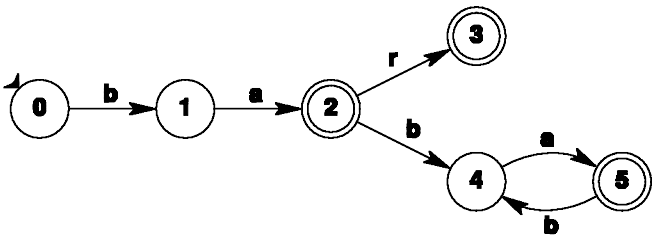

Figure 4
Minimal automaton accepting the set of strings $(\texttt{ba})^+ \cup \{\texttt{bar}\}$.

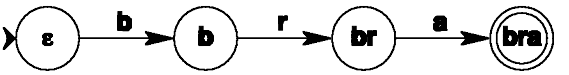

Figure 5
Single-string automaton accepting string `bra`.

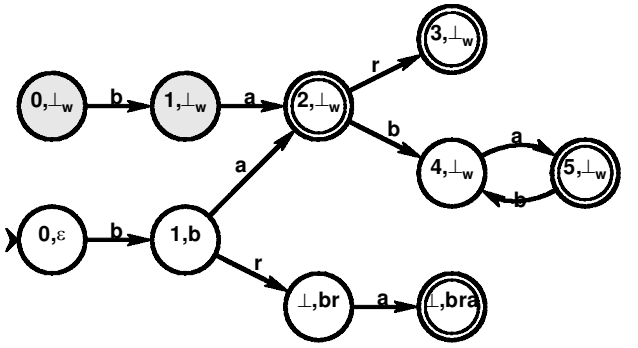

Figure 6
Unminimized automaton accepting the set $(\texttt{ba})^+ \cup \{\texttt{bar}\} \cup \{\texttt{bra}\}$. Shadowed states $(0, \perp_w)$ and $(1, \perp_w)$ have become unreachable (have no incoming transitions) and are eliminated in precisely that order.

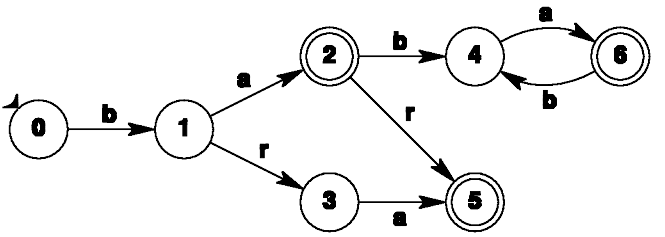

Figure 7
Minimal automaton accepting the set $(\texttt{ba})^+ \cup \{\texttt{bar}\} \cup \{\texttt{bra}\}$.

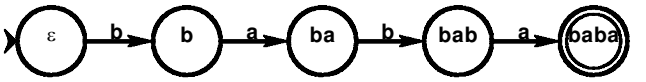

Figure 8
Single-string automaton accepting the string `baba`.

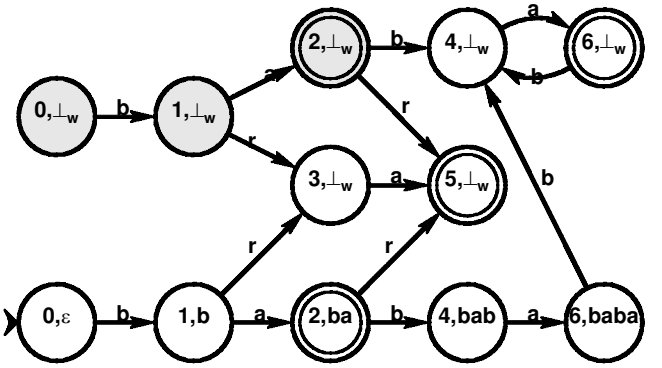

Figure 9
Unminimized automaton accepting the set $(\texttt{ba})^+ \cup \{\texttt{bar}\} \cup \{\texttt{bra}\} - \{\texttt{baba}\}$. Shadowed states $(0, \perp_w)$, $(1, \perp_w)$, and $(2, \perp_w)$ have become unreachable (have no incoming transitions) and are eliminated in precisely that order.

new start state, shadowed states $(0, \perp_w)$ and $(1, \perp_w)$ become unreachable (have no incoming transitions) and are eliminated in precisely that order in the second stage of the algorithm. The final stage of the algorithm puts intact states into the register and tests queue and cloned states for equivalence with states in the register. The first state tested is $(\perp, \texttt{bra})$, which is found to be equivalent to $(3, \perp_w)$; therefore, transitions coming into $(\perp, \texttt{bra})$ are made to point to $(3, \perp_w)$. Then, states $(\perp, \texttt{br})$, $(1, \texttt{b})$ and $(0, \epsilon)$ are tested in order, found to have no equivalent in the register, and added to it. The resulting minimal automaton, after a convenient renumbering of states, is shown in Figure 7.

5.2 Removing a String

Now let us consider the case in which we want to remove string `baba` from the language accepted by the automaton in Figure 7 (the single-string automaton for `baba` is shown in Figure 8). The automaton resulting from the application of the initial (construction) stages of the automaton is shown in Figure 9. Note that state $(6, \texttt{baba})$ is marked as nonaccepting, because we are removing a string. Again, as a consequence of the designation of $(0, \epsilon)$ as the new start state, shadowed states $(0, \perp_w)$, $(1, \perp_w)$,

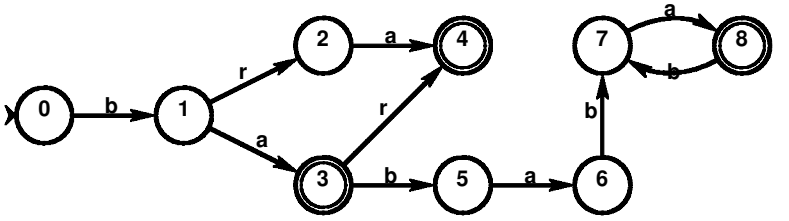

Figure 10
Minimal automaton accepting the set $(\texttt{ba})^+ \cup \{\texttt{bar}\} \cup \{\texttt{bra}\} - \{\texttt{baba}\}$.

and $(2, \perp_w)$ become unreachable (have no incoming transitions) and are eliminated in precisely that order in the second stage of the algorithm. The last stage of the algorithm puts all intact states into the register, checks cloned states $(6, \texttt{baba})$, $(4, \texttt{bab})$, $(2, \texttt{ba})$, $(1, \texttt{b})$ and $(0, \epsilon)$ (no queue states, since `baba` is accepted by the automaton in Figure 7), and finds none of them to be equivalent to those in the register, to which they are added. The resulting minimal automaton is shown in Figure 10.

6. Concluding Remarks

We have derived, from basic results of language and automata theory, a simple method for modifying a minimal (possibly cyclic) finite-state automaton so that it recognizes one string more or one string less while keeping the finite-state automaton minimal. These two operations may be applied to dictionary construction and maintenance and generalize the result in Daciuk et al.'s (2000) second algorithm (incremental construction of acyclic finite-state automata from unsorted strings) in two respects, with interesting practical implications:

- The method described here allows for the addition of strings to the languages of cyclic automata (in practice, it may be convenient to have cycles in dictionaries if we want them to accept, for example, all possible integer numbers or Internet addresses). In this respect, the algorithm presented also generalizes the string removal method sketched by Revuz (2000) for acyclic automata.
- Removal of strings is as easy as addition. This means that, for example, the detection of an erroneous entry in the dictionary does not imply having to rebuild the dictionary completely.

The asymptotic time complexity of the algorithms is in the same class ($O(|Q||w|)$) as that in Daciuk et al. (2000), because the slowest part of the algorithm (the last one) checks all queue and cloned states ($O(|w|)$) against all states of the register ($O(|Q|)$). As suggested by one of the reviewers of this article, an improvement in efficiency may be obtained by realizing that, in many cases, cloned states corresponding to the shortest prefixes of string w are not affected by minimization, because their intact equivalents have become unreachable and therefore have been removed from the register; the solution lies in identifying these states and not cloning them (for example, Daciuk et al.'s [2000] and Revuz's [2000] algorithms do not clone them).

As for the future, we are working on an adaptation of this algorithm for the maintenance of morphological analyzers and generators using finite-state nondeterministic letter transducers (Roche and Schabes 1997; Garrido et al. 1999).

Acknowledgments

The work reported in this article has been funded by the Spanish Comisión Interministerial de Ciencia y Tecnología through grant TIC2000-1599. We thank the two reviewers for their suggestions and Colin de la Higuera for his comments on the manuscript.

References

Daciuk, Jan, Stoyan Mihov, Bruce W. Watson, and Richard E. Watson. 2000. Incremental construction of minimal acyclic finite-state automata. *Computational Linguistics*, 26(1):3–16.

Garrido, Alicia, Amaia Iturraspe, Sandra Montserrat, Hermínia Pastor, and Mikel L. Forcada. 1999. A compiler for morphological analysers and generators based on finite-state transducers. *Procesamiento del Lenguaje Natural*, 25:93–98.

Hopcroft, John E. and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

Karakostas, George, Anastasios Viglas, and Richard J. Lipton. 2000. On the complexity of intersecting finite state automata. In *Proceedings of the 15th Annual IEEE Conference on Computational Complexity (CoCo'00)*, pages 229–234.

Revuz, Dominique. 2000. Dynamic acyclic minimal automaton. In *Preproceedings of CIAA 2000: Fifth International Conference on Implementation and Application of Automata*, pages 226–232, London, Ontario, July 24–25.

Roche, Emmanuel and Yves Schabes. 1997. Introduction. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*. MIT Press, pages 1–65.