Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT•

Science of Computer Programming 48 (2003) 81 – 97

Science of Computer Programming

www.elsevier.com/locate/scico

A new algorithm for the construction of minimal acyclic DFAs

Bruce W. Watson[a,b,*]

[a]*Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa*
[b]*FST Labs & Ribbit Software Systems Inc., Canada*

Received 20 November 2000; received in revised form 21 February 2001; accepted 16 November 2002

Abstract

We present a semi-incremental algorithm for constructing minimal acyclic deterministic finite automata. Such automata are useful for storing sets of words for spell-checking, among other applications. The algorithm is semi-incremental because it maintains the automaton in nearly minimal condition and requires a final minimization step after the last word has been added (during construction).

The algorithm derivation proceeds formally (with correctness arguments) from two separate algorithms, one for minimization and one for adding words to acyclic automata. The algorithms are derived in such a way as to be combinable, yielding a semi-incremental one. In practice, the algorithm is both easy to implement and displays good running time performance.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Minimal acyclic deterministic finite automata; Incremental algorithm; Recursive algorithm; Algorithmics

1. Introduction

In this paper, we present a semi-incremental algorithm for constructing minimal acyclic deterministic finite automata (ADFAs). By their acyclic nature, they represent finite languages, and are therefore useful in applications such as stringology [7], computational linguistics [18], information retrieval [2], data-structures [12], computational

* Corresponding author. Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa.
E-mail addresses: bwatson@cs.up.ac.za, watson@fst-labs.com (B.W. Watson).

0167-6423/03/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0167-6423(03)00012-1

biology [15,22] and compilers [1]. In such applications, the automata can grow extremely large (with more than 10^6 states), and are difficult to store without first applying a minimization procedure. In traditional minimization techniques, the unminimized ADFA is first constructed and then minimized. Unfortunately, the unminimized ADFA can be very large indeed—sometimes even too large to fit within the virtual address space of the host machine. As a result, incremental techniques for minimization (i.e. the ADFA is minimized during its construction) become interesting. Incremental algorithms usually have *some* overhead—if the unminimized ADFA fits easily within physical memory, it is still faster to use nonincremental techniques. On the other hand, with very large ADFAs, the incremental techniques may be the *only* option. Because of their commercial value, actual implementations of such algorithms are typically proprietary.

The algorithm presented in this paper is *semi*-incremental (as opposed to *fully* incremental, or just *incremental*) because it maintains the ADFA in a nearly minimal condition while words are added, but requires a simple 'final' step to achieve full minimality after all words have been added. It achieves this by requiring the words to be added in *any* order of decreasing length. A balance exists between the work done while adding each word (semi-incremental requires less work) and the work done in the final step (fully incremental requires no work here). Benchmarking indicates that the semi-incremental performs well compared with fully incremental ones.

In order to derive the algorithm, we proceed in three stages:

(1) Derive an efficient algorithm for minimizing ADFAs.
(2) Derive a simple algorithm for adding new words to an ADFA, assuming certain conditions.
(3) Combine the algorithms derived in the first two steps. By-design, the first two algorithms will manipulate the ADFA in a fashion that makes them combinable.

1.1. Related work and a short history

The next paragraphs present a chronological summary of the various algorithms for ADFA construction. Ref. [28] presents a taxonomy of the algorithms.

In the early-1990s, Revuz derived the first linear ADFA minimization algorithms [23,24]. The primary algorithm presented by Revuz uses an ordering of the words to quickly compress the endings of the words within the dictionary. By the mid-1990s, several groups were working independently on incremental algorithms—most of which are the same or very similar. In Greece, Sgarbas et al. derived a generalized (independent of word order) algorithm and presented it in [26]. In Japan, Park et al. were also deriving a related generalized algorithm [21]. At Marne-la-Valée in France, a group (including Revuz) was continuing work on related algorithms. In 1996, R.E. Watson and I also derived an incremental algorithm. Unlike many of the other derivations of related algorithms, ours also provides facilities for removing words from the language accepted by the automaton, while maintaining minimality. Also in 1996–1997, Daciuk independently derived the generalized incremental algorithm. In addition, Daciuk derived a new incremental algorithm which adds the words in lexicographic order. Simultaneously, Mihov had also derived the sorted algorithm, publishing it as [19]. Daciuk and Mihov went on to publish the algorithms in their dissertations as [8] and [20], respectively. Daciuk, Mihov, R.E. Watson and I published a joint paper on the generalized and the sorted algorithms in [9,10]. Independently in 1997, in the field of verification, Holzmann and Puri [16] discovered a restricted form of the algorithm, in which all words accepted by the automaton are the same length. In 1998, I sketched the semi-incremental algorithm (of the current paper) in [27]. Ciura and Deorowicz independently discovered the sorted algorithm, benchmarked it by building automata for several dictionaries and published the results as a technical report [6]. In 2000, Revuz presented essentially the generalized algorithm [25]—though he also sketched word deletion algorithms similar to those previously derived by Watson and Watson. In 2001, Graña et al. summarized some of the current results and made improvements to several of the algorithms [13]. Recently, the generalized algorithm was straightforwardly extended by Carrasco and Forcada to handle *cyclic* automata [5]. In [30], I gave an elegant recursive incremental algorithm, while [31] contains another algorithm based upon Brzozowski's minimization algorithm [3,4].

1.2. Structure of this paper

This paper is structured as follows:

- Section 2 presents the necessary definitions of ADFAs.
- Section 3 derives a procedure for minimizing an ADFA.
- Section 4 derives a procedure for adding words to an ADFA.
- Section 5 combines the algorithms from Section 3 and 4 to yield the semi-incremental algorithm.
- Section 6 gives a detailed example.
- Section 7 provides some details on running time and implementation issues for the algorithm.
- Section 8 presents the conclusions of this paper.

This paper is an extended presentation of [27].

2. Mathematical preliminaries

In this paper, we consider acyclic deterministic finite automata (ADFAs). The algorithm is readily extended to work with acyclic deterministic *transducers*, though such an extension is not considered here.

A deterministic finite automaton (DFA) is a 5-tuple $(Q, \Gamma, \delta, q_0, F)$ where:

- Q is the finite set of states.
- Γ is the input alphabet. We choose this instead of the more traditional Σ since we will use that letter for 'summation' in the section on running time analysis.
- $\delta \in Q \times \Gamma \longrightarrow Q \cup \{\perp\}$ is the transition function. We use $\perp$ to designate the invalid state.

- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of *final* states.

Throughout this paper, we will consider a specific ADFA $(Q, \Gamma, \delta, q_0, F)$. To make some definitions simpler, we will use the shorthand Γ_q to refer to the set of all alphabet symbols which appear as out-transition labels from state q. Formally,

$$\Gamma_q = \{ a \mid a \in \Gamma \wedge \delta(q,a) \neq \perp \}.$$

The right language of a state q, written $\overrightarrow{\mathscr{L}}(q)$, is the set of all words spelled out on paths from q to a final state. We define predicate *Equiv* to be 'equivalence' of states:

$$Equiv(p,q) \equiv \overrightarrow{\mathscr{L}}(p) = \overrightarrow{\mathscr{L}}(q)$$

We can also give an inductive definition of $\overrightarrow{\mathscr{L}}$:

$$\overrightarrow{\mathscr{L}}(q) = \left(\bigcup_{a \in \Gamma_q} \{a\} \cdot \overrightarrow{\mathscr{L}}(\delta(q,a)) \right) \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F, \\ \emptyset & \text{if } q \notin F. \end{cases}$$

(Note that $\{a\} \cdot \overrightarrow{\mathscr{L}}(\delta(q,a))$ indicates concatenation of an a to the left of the words in language $\overrightarrow{\mathscr{L}}(\delta(q,a))$.) Intuitively, a word z is in $\overrightarrow{\mathscr{L}}(q)$ if and only if

- z is of the form az' where $a \in \Gamma$ is a label of an out-transition from q to $\delta(q,a)$ (i.e. $a \in \Gamma_q$) and z' is in the right language of $\delta(q,a)$, or
- $z = \varepsilon$ and q is a final state.

The quantifications are frequently presented in the style given in [14]. All of the algorithms presented in this paper are in the form of the guarded command language—see [11,14].

3. Minimizing an ADFA

In this section, we derive a procedure for minimizing an ADFA. We begin with an abstract algorithm (whose correctness is easily determined) and refine the abstract details to yield an efficient algorithm.

The primary definition of minimality of an ADFA M (indeed, this definition applies to *any* DFA, not just acyclic ones) is:

$$(\forall M' : M' \text{ is equivalent to } M : |M| \leqslant |M'|),$$

where equivalence of DFAs means that they accept the same language. This definition is difficult to manipulate (in deriving an algorithm), and so we consider one written in terms of the right languages of states. Using right languages (and the Myhill-Nerode theorem—see [17, Section 3.4]), minimality can also be written as the following predicate (which we call postcondition R):

$$(\forall\, p, q \in Q : p \neq q : \neg Equiv(p,q)).$$

(Additionally, we require that there are no useless states, though this additional restriction is not usually written and we ignore it in the rest of this paper since, by design, our algorithms have no way of creating useless states.)

To achieve postcondition R, we introduce a procedure minimize which:

- assumes the ADFA $(Q, \Gamma, \delta, q_0, F)$ as a global data-structure;
- takes (as first parameter) a set of states U, called the *unique* states, which are pairwise inequivalent; that is

$$(\forall\ p, q \in U : p \neq q : \neg Equiv(p, q)).$$

  We refer to this as I_1, since we will use it as part of our invariant.
- does not shrink the set U of pairwise inequivalent states;
- takes (as second parameter) another set of states V (which is disjoint from U, i.e. $U \cap V = \emptyset$) which are to be made pairwise unique—those which are not unique will be removed since they are redundant.

In the following presentation of the algorithm, we do not give all of the shadow variables or the complete (and lengthy) invariants for a full correctness argument:

Algorithm 3.1.

```
proc minimize($U, V$) $\rightarrow$
    { invariant: $U \cap V = \emptyset \wedge I_1$
      variant: $|V|$ }
    do $V \neq \emptyset \rightarrow$
        let $q : q \in V$;
        $V := V - \{q\}$;
        if $(\exists\ p : p \in U : Equiv(p, q)) \rightarrow$
            { $q$ is redundant }
            let $p : p \in U \wedge Equiv(p, q)$;
            redirect all of $q$'s in-transitions to $p$;
            remove state $q$, since it is redundant
        [] $\neg(\exists\ p : p \in U : Equiv(p, q)) \rightarrow$
            { $q$ is unique against $U$ }
            $U := U + \{q\}$
        fi
    od
    { $V = \emptyset$ }
corp
```

Note that the invariant is also a precondition of the entire procedure. Invoking the procedure as minimize$(\emptyset, Q)$ would clearly minimize the entire ADFA.

In the first branch of the alternation (**if-fi** statement), redirecting transitions can be made fast by storing the reverse transition graph as well as the (normal) forward transition edges.

The main difficulty with this algorithm is testing *Equiv*. Fortunately, we can use the inductive definition of $\overrightarrow{\mathscr{L}}$. We can begin rewriting *Equiv* as follows:

$$\begin{array}{ll}
 & \mathit{Equiv}(p,q) \\
\equiv & \quad \langle \text{definition of } \mathit{Equiv} \rangle \\
 & \overrightarrow{\mathscr{L}}(p) = \overrightarrow{\mathscr{L}}(q) \\
\equiv & \quad \langle \text{the inductive definition of } \overrightarrow{\mathscr{L}} \rangle \\
 & (\varepsilon \in \overrightarrow{\mathscr{L}}(p) \equiv \varepsilon \in \overrightarrow{\mathscr{L}}(q)) \wedge \Gamma_p = \Gamma_q \wedge \\
 & (\forall a : a \in \Gamma_p \cap \Gamma_q : \{a\}\overrightarrow{\mathscr{L}}(\delta(p,a)) = \{a\}\overrightarrow{\mathscr{L}}(\delta(q,a))) \\
\equiv & \quad \langle \text{the inductive definition of } \varepsilon \in \overrightarrow{\mathscr{L}}(p) \rangle \\
 & (p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge \\
 & (\forall a : a \in \Gamma_p \cap \Gamma_q : \{a\}\overrightarrow{\mathscr{L}}(\delta(p,a)) = \{a\}\overrightarrow{\mathscr{L}}(\delta(q,a))) \\
\equiv & \quad \langle \text{for two languages } L_0, L_1\text{: } (\{a\}L_0 = \{a\}L_1) \equiv (L_0 = L_1) \rangle \\
 & (p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge \\
 & (\forall a : a \in \Gamma_p \cap \Gamma_q : \overrightarrow{\mathscr{L}}(\delta(p,a)) = \overrightarrow{\mathscr{L}}(\delta(q,a))) \\
\equiv & \quad \langle \text{definition of } \mathit{Equiv} \rangle \\
 & (p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge \\
 & (\forall a : a \in \Gamma_p \cap \Gamma_q : \mathit{Equiv}(\delta(p,a), \delta(q,a))).
\end{array}$$

Clearly, the last step has yielded a recursive definition which, while implementable (since we have *acyclic* DFAs and the recursion will end), is not very efficient.[1] Fortunately, the acyclicity also yields a way to implement this efficiently by restricting the invariant in the algorithm (as in the following paragraphs).

Given the invariant on pairwise inequivalent state set U, the evaluation of $\mathit{Equiv}(p,q)$ would be much simpler if we were assured that, for all $a \in \Gamma$, $\delta(p,a)$ and $\delta(q,a)$ were already in U (i.e. the children $\delta(p,a)$ and $\delta(q,a)$ are pairwise unique). Since we will use this requirement extensively, we write it as predicate $P_1(r)$ (for state r):

$$(\forall a : a \in \Gamma_r : \delta(r,a) \in U)$$

Assuming $P_1(p) \wedge P_1(q)$ we rewrite

$$\begin{array}{ll}
 & \mathit{Equiv}(\delta(p,a), \delta(q,a)) \\
\equiv & \quad \langle \text{definition of } \mathit{Equiv} \rangle \\
 & \overrightarrow{\mathscr{L}}(\delta(p,a)) = \overrightarrow{\mathscr{L}}(\delta(q,a)) \\
\equiv & \quad \langle \text{property of } U \text{ stated in invariant } I_1 \text{ and assumption } P_1(p) \wedge P_1(q) \rangle \\
 & \delta(p,a) = \delta(q,a).
\end{array}$$

[1] Interestingly, this would lead to the algorithm in [29].

Here, by introducing the assumption of $P_1(p) \wedge P_1(q)$, we have successfully removed the recursion in *Equiv*. Of course, it remains to ensure that this assumption holds. We consider the two assumed conjuncts separately.

To ensure that $P_1(q)$, we introduce an invariant I_2:

$$(\forall\, r,a : r \in V, a \in \Gamma_r : \delta(r,a) \in V \vee \delta(r,a) \in U)$$

I_2 is also a precondition. (Note that $U \cup V$ is not necessarily equal to Q, so this is not as simple as it looks.) I_2 is trivially true if r has no out-transitions or $V = \emptyset$ (there is no $r \in V$). Given I_2 and the acyclic property of ADFAs, we know that for any $r \in V$, there will be a chain of r's descendants (in terms of δ) eventually ending with one in U or one in V that has no descendants. From this, we conclude that we can always choose *some* $q \in V$ such that $P_1(q)$.

Turning to $P_1(p)$, we introduce another invariant I_3:

$$(\forall\, r : r \in U : P_1(r)).$$

We can now summarize our derivation from the last two derivations above, assuming $P_1(p) \wedge P_1(q)$:

$$\begin{aligned}
&(p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge \\
&(\forall\, a : a \in \Gamma_p \cap \Gamma_q : Equiv(\delta(p,a), \delta(q,a))) \\
\equiv \quad & \langle \text{invariants } I_2 \text{ and } I_3 \text{, assumption } P_1(p) \wedge P_1(q) \text{, previous derivation} \rangle \\
&(p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge \\
&(\forall\, a : a \in \Gamma_p \cap \Gamma_q : \delta(p,a) = \delta(q,a)).
\end{aligned}$$

We define the last predicate to be our new predicate $P_2(p,q)$:

$$(p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge (\forall\, a : a \in \Gamma_p \cap \Gamma_q : \delta(p,a) = \delta(q,a)).$$

Given this, we can rewrite our procedure

Algorithm 3.2.

```
proc minimize'$(U,V) \rightarrow$
    { invariant: $U \cap V = \emptyset \wedge I_1 \wedge I_2 \wedge I_3$
      variant: $|V|$ }
    do $V \neq \emptyset \rightarrow$
        let $q : q \in V \wedge P_1(q)$;
        $V := V - \{q\}$;
        if $(\exists\, p : p \in U : P_2(p,q)) \rightarrow$
            { $q$ is redundant }
            let $p : p \in U \wedge P_2(p,q)$;
            { $Equiv(p,q)$ }
            redirect all of $q$'s in-transitions to $p$;
            remove state q, since it is redundant
        [] $\neg(\exists\, p : p \in U : P_2(p,q)) \rightarrow$
            { $q$ is unique against $U$ }
            $U := U + \{q\}$
        fi
    od
    $\{\ V = \emptyset\ \}$
corp
```

This algorithm is still invoked as $\mathsf{minimize}'(\emptyset, Q)$.

Because of I_2 and the $P_1(q)$ assumption, this algorithm operates in a bottom-up fashion[2] on the set V, where (by precondition I_2) U is 'hanging' below V.

Instead of V being a set, we can reinforce this 'bottom-up' order by stipulating that V is a stack with those items on the top of the stack being topologically 'lower' (closer to U, and thus further from q_0) than the items lower in the stack—i.e. if r is on top of the stack, then $P_1(r)$.

Before restructuring the algorithm to use a stack, we rewrite V as a stack in invariant I_2 (I_3 remains the same). I_2', states that:

$$(\forall\, r, a : r \in V \wedge a \in \Gamma_r : (\delta(r,a) \text{ is higher than } r \text{ in } V \vee \delta(r,a) \in U)).$$

This gives the algorithm (where [] is the empty stack):

Algorithm 3.3.

```
proc $\mathsf{minimize}''(U, V) \rightarrow$
    { invariant: $U \cap V = \emptyset \wedge I_1 \wedge I_2' \wedge I_3$
      variant: $|V|$ }
    do $V \neq [] \rightarrow$
        $q := \mathsf{pop}(V)$;
        $\{\ P_1(q)\ \}$
        if $(\exists\, p : p \in U : P_2(p,q)) \rightarrow$
            { $q$ is redundant }
            let $p : p \in U \wedge P_2(p,q)$;
            $\{\ Equiv(p,q)\ \}$
            redirect all of $q$'s in-transitions to $p$;
            remove state q, since it is redundant
        [] $\neg(\exists\, p : p \in U : P_2(p,q)) \rightarrow$
            { $q$ is unique against $U$ }
            $U := U + \{q\}$
        fi
    od
    $\{\ V = []\ \}$
corp
```

[2] We say *bottom-up* to mean from final states towards the start state q_0, if we imagine q_0 at the top of the page, with the final states towards the bottom.

To minimize the entire ADFA, we invoke it with $\mathsf{minimize}''(\emptyset, V)$, where V is a preorder[3] stacking of the states Q.

4. Adding Words to an ADFA

At first glance, in adding a word w to an ADFA we would simply trace out the path (corresponding to w), from the start state q_0, and make the resulting state a final one (add it to F). Unfortunately, this may inadvertently add more than one word to the ADFA in the event that some state on the path has more than one in-transition (it is a confluence state).

Under certain conditions, we can, however, use such a simple procedure. A sufficient condition is that, along the w path there must be no state q with more than one in-transition (this is known as the *confluence* condition). We can state this condition formally as $P_3(w)$ (noting that we begin indexing with 0):

$$(\forall\, i : 0 \leqslant i < |w| : \mathsf{pred}(\delta^*(q_0, w_{[0...i)})) \leqslant 1),$$

where $\mathsf{pred}(q)$ is the number of *predecessors* (in-transitions) of state q and δ^* is the usual reflexive and transitive closure of the transition function δ.

Condition P_3 is trivially guaranteed if we are adding word to a *trie*—a tree-structured ADFA (with no confluence states at all). We state condition P_3 because we will be simultaneously minimizing the ADFA, and so we may have a situation where not all states will have at most one predecessor.

We can now give the procedure add_word which takes the word, adds it to the ADFA (which is taken to be a global data-structure) and returns the state at which the word ends (**cand** is *conditional conjunction*):

Algorithm 4.1.

```
func add_word($w$) : $Q$ $\rightarrow$
    { $P_3(w)$ }
    $q, i := q_0, 0$;
    { invariant: $q = \delta^*(q_0, w_{[0..i)})$
      variant: $|w| - i$ }
    do $i < |w|$ cand $\delta(q, w_i) \neq \perp$ $\rightarrow$
        $q, i := \delta(q, w_i), i + 1$
    od;
    if $i < |w|$ $\rightarrow$
        { $\delta(q, w_i) = \perp$ }
        do $i < |w|$ $\rightarrow$
            $q, i :=$ build_state($q, w_i$), $i + 1$
        od
    [] $i = |w|$ $\rightarrow$ skip
```

[3] *Any* preorder will do, since the out-transitions from any given state are unordered.

```
          fi;
          { $q = \delta^*(q_0, w)$ }
          $F := F \cup \{q\}$;
          return $q$
     cnuf
```

Function build_state is a function for extending the ADFA with a new transition. It returns the newly created state. The first loop proceeds as far as possible in the existing ADFA, while the second loop extends the ADFA, as necessary, to accommodate the rest of w.

5. Combining the algorithms

If we are simultaneously minimizing and adding words, we must co-ordinate the two algorithms. Recall, from the last version of the minimizing algorithm—Algorithm 3.3—that only states in U have been minimized (and may, therefore, have more than one predecessor) and that set U is grown bottom-up (towards q_0). To synchronize the algorithms, it makes sense to add the word-set W such that their final states are also added bottom-up.

One possible way to order W is to add the words in *any* order of decreasing length.[4] In this case, while adding w, we will not pass through any final states, and it will be safe to minimize the portion of the ADFA below the top-most (closest to q_0) final states. We call these top-most final states the *upper final states frontier* since they form the set of final states which appear first over all paths leading away from q_0. The states below this frontier will become our minimization set U.

After invoking add_word(w), the states at and below the returned state (inclusive) are safe for minimization and can be added to the stack (in preorder) using this procedure (which initially takes the returned value of add_word and the empty stack):

Algorithm 5.1.

```
     proc build_stack($q$,$X$) : Stack of states →
          push($X$,$q$);
          for $a : a \in \Gamma_q$ →
               if $\delta(q,a) \in F$ → skip
               [] $\delta(q,a) \notin F$ → $X$ := build_stack($\delta(q,a)$,$X$)
               fi
          rof;
          return $X$
     corp
```

Along each q-to-leaf path, this procedure stops when it encounters another final state (which, by our invariant, will already be in U).

[4] We say *any*, since there are usually several such possible orderings.

After adding word w, we minimize the following stack:

$$\mathsf{build\_stack}(\mathsf{add\_word}(w), []).$$

After adding all of the words, we still need to minimize the states above the upper final states frontier. This is done with an additional invocation of build_stack and minimize$''$.

For our final algorithm we maintain invariant I_4 with the following conjuncts:

- U is the set of states in the upper final states frontier and below; and
- $(Q, \Gamma, \delta, q_0, F)$ accepts all words in *Done*; and
- $To\_do \cap Done = \emptyset$; and
- $To\_do \cup Done = W$.

This yields our final, combined, semi-incremental algorithm:

Algorithm 5.2.

```
{ we have an empty ADFA with a single start state }
$U := \emptyset$;
$To\_do, Done := W, \emptyset$
{ invariant: $I_4$
  variant: $|To\_do|$ }
do $W \neq \emptyset \rightarrow$
    let $w : w \in To\_do \wedge w$ is any longest word in $To\_do$;
    $To\_do, Done := To\_do - \{w\}, Done \cup \{w\}$;
    $\mathsf{minimize}''(U, \mathsf{build\_stack}(\mathsf{add\_word}(w), []))$
od;
{ all have been minimized except those above the final states frontier }
$\mathsf{minimize}''(U, \mathsf{build\_stack}(q_0, []))$
{ $R$ }
```

6. An example

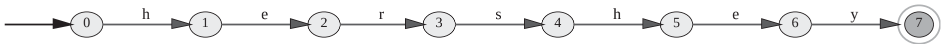
In this section, we present a complete example of automata constructed by the algorithm for the words

$$\{hershey, heresy, here, hers, they\}$$

While any order of decreasing length is permitted, we choose to use the order given above.

6.1. *Adding* hershey

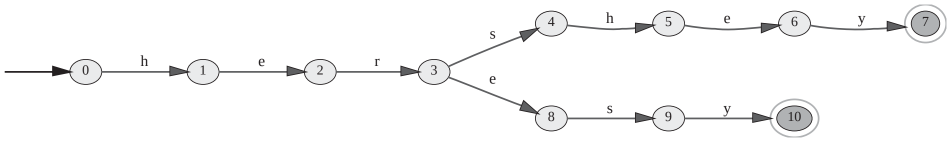
Initially, add_word constructs automaton



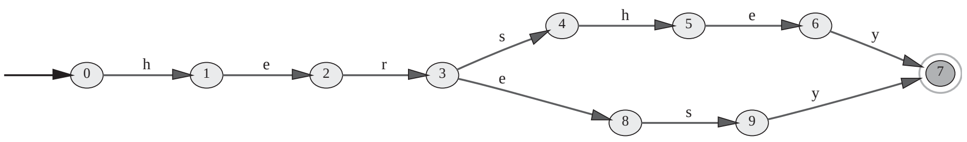
while build_stack yields the stack with one element [7], for state 7. The set of unique states U is initially empty and minimize$''$ cannot combine any states at this point, but simply adds state 7 to the unique set.

6.2. *Adding* heresy

Adding word *heresy* gives automaton



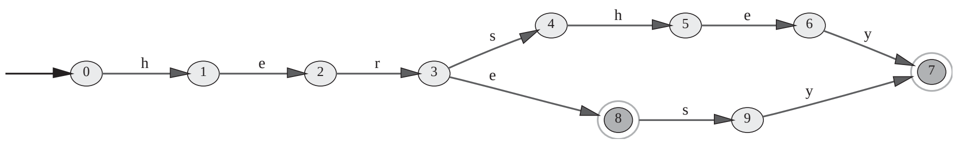
while build_stack gives the single element stack [10]. In minimize$''$, state 10 is found to be equivalent to state 7 (already in U) and 10 is eliminated in favour of 7, giving



The unique states remain $U = \{7\}$.

6.3. *Adding* here

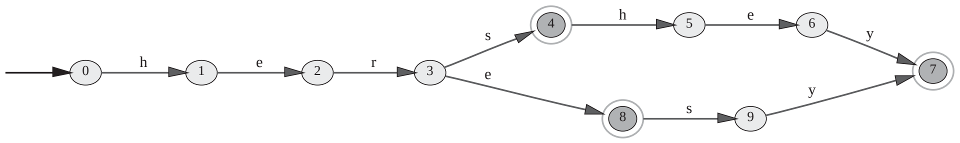
Adding *here* to the automata does not build new states, but makes 8 a final state:



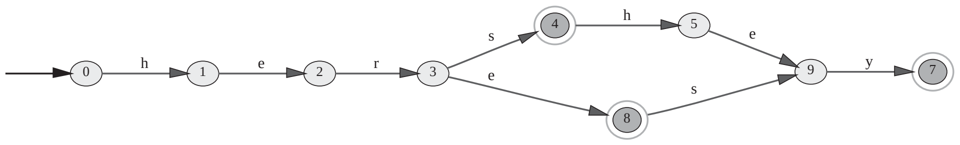
The resulting stack, from top to bottom, is [9, 8]. Procedure minimize$''$ is unable to combine states 9 or 8 and the automaton remains the same with $U = \{7, 8, 9\}$.

6.4. *Adding* hers

Word *hers* is added, making state 4 final:



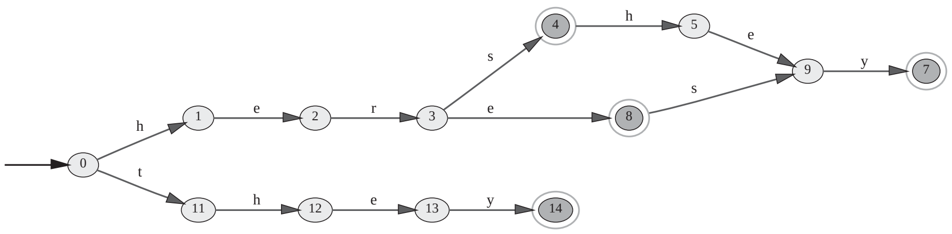
with the resulting stack (from top to bottom) $[6,5,4]$. Using unique set $U=\{7,8,9\}$, $\mathsf{minimize}''$ finds 6 to be equivalent to 9 and eliminates 6, giving



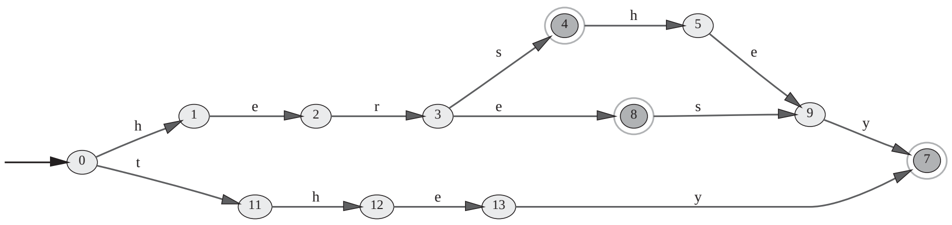
The unique state set is $U=\{4,5,7,8,9\}$.

6.5. *Adding* they

Word *they* is added, giving automaton



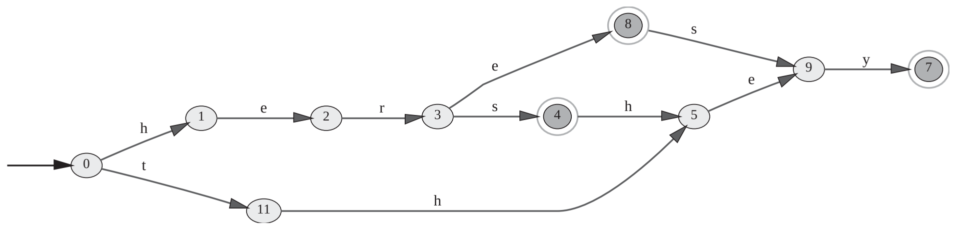
The resulting stack is $[14]$, and state 14 is eliminated in favour of state 7, giving



We still have $U=\{4,5,7,8,9\}$.

6.6. *The final minimization step*

Now that all of the words have been added, the final step is to push the remaining states on the stack, giving $[13,12,11,3,2,1,0]$ (note that $[3,2,1,13,12,11,0]$ is another possible stack), and to invoke $\mathsf{minimize}''$. States 13 and 9 are found to be equivalent and 13 is eliminated. Subsequently, states 12 and 5 are found to be equivalent and 12 is eliminated. No other states are equivalent—in particular, notice that 11 and 4 are inequivalent because 4 is final while 11 is not. The final automaton is



This automaton is minimal and set U consists of all of the states.

7. Implementation and performance

We begin with an analysis of the running time of the final algorithm (Algorithm 5.2). For this, we make the following assumptions:

- A state can be created or destroyed in constant time.
- For a given state and alphabet symbol, δ can be evaluated in constant time.
- The stack operations take constant time.
- We explicitly store the reverse transitions, in addition to the (normal) forward transitions δ. This allows us to perform some other operations in constant time. The additional space required can be deallocated once the automaton is constructed.

We have the following sub-analyses:

- The outer repetition executes exactly $|W|$ times.[5]
- For any word w, add_word(w) adds $\mathcal{O}(|w|)$ states and takes the same order time.
- Each state is pushed onto the stack exactly once (thereafter it is placed in U or removed due to redundancy, and by acyclicity never appears in the stack again).
- An invocation minimize$''(U,V)$ takes $\mathcal{O}(|\Gamma|\cdot|V|)$ time. The factor $|\Gamma|$ is due to the test $P_2(p,q)$ and the (possible) elimination of p and redirection of its in-transitions, while $|V|$ is due to the outer loop of that procedure.

Here, we have assumed that some elementary operations (such as set membership in U, stack operations and automata transitions) can be done in constant time.

The total time taken by add_word (and also the total number of states) is

$$\mathcal{O}\left(\sum_{w:w\in W}|w|\right).$$

This also happens to be the time taken (in total) by build_stack and the order of the total stack size.

[5] We will actually ignore this and separately calculate the total running time of the function invocations.

Given that each state is pushed onto the stack exactly once, the total time taken by minimize is

$$\mathcal{O}\left(|\Gamma| \cdot \sum_{w:w\in W} |w|\right).$$

It follows that the total running time of this algorithm is

$$\mathcal{O}\left(|\Gamma| \cdot \sum_{w:w\in W} |w| + \sum_{w:w\in W} |w| + \sum_{w:w\in W} |w|\right)$$

or, equivalently

$$\mathcal{O}\left(|\Gamma| \cdot \sum_{w:w\in W} |w|\right).$$

Interestingly, this running time is asymptotically the same as the running time of the best known non-incremental algorithms for ADFAs (they are also linear in the size of the ADFA, whose construction is in-turn linear in $\sum_{w:w\in W} |w|$). Naturally, the semi-incrementality takes its toll in the constant factor.

8. Conclusions

We have presented a semi-incremental algorithm for acyclic deterministic finite automata—indeed, it appears to be the first such algorithm published. The following aspects of the algorithm are noteworthy:

- The simplicity of the algorithm goes hand-in-hand with the formal derivation and presentation of correctness arguments.
- In order to formally derive a (semi-)incremental algorithm, novel techniques were used, such as: separately developing component algorithms that are nondeterministic and using their nondeterminism to synchronize them (co-operatively) into the incremental one.
- The running time of the algorithm is asymptotically as good as the best non-incremental ones.

Acknowledgements

Nanette Y. Saes and Richard E. Watson were kind enough to proofread this paper. The referees also provided a great number of useful comments (including some modifications to the example), and Jan Daciuk's recent work inspired me to write up this algorithm.

References

[1] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, Reading, MA, 1988.

[2] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, Addison-Wesley, Reading, MA, 1999.

[3] J.A. Brzozowski, Canonical regular expressions and minimal state graphs for definite events, MRI Symposia Series, Vol. 12, Polytechnic Press, Polytechnic Institute of Brooklyn, 1962, pp. 529–561.

[4] J.A. Brzozowski, Regular expression techniques for sequential circuits, Ph.D. Thesis, Princeton University, Princeton, New Jersey, June 1962.

[5] R.C. Carrasco, M.L. Forcada, Incremental construction and maintenance of minimal finite-state automata, Comput. Linguistics 28 (1) (2002) 207–216.

[6] M. Ciura, S. Deorowicz, Experimental study of finite automata storing static lexicons, Technical Report, Silesian Technical University, Poland, November 1999.

[7] M.A. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, Oxford, 1994.

[8] J. Daciuk, Incremental construction of finite-state automata and transducers, and their use in the natural language processing, Ph.D. Thesis, Technical University of Gdańsk, Poland, 1998.

[9] J. Daciuk, B.W. Watson, R.E. Watson, Incremental construction of minimal acyclic finite state automata and transducers, in: L. Karttunen, K. Oflazer (Eds.), Proc. Int. Workshop on Finite State Methods in Natural Language Processing, Ankara, Turkey, 1998, pp. 48–56.

[10] J. Daciuk, S. Mihov, B.W. Watson, R.E. Watson, Incremental construction of minimal acyclic finite state automata, Comput. Linguistics 26 (1) (2000) 3–16.

[11] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[12] G.H. Gonnet, R. Baeza-Yates, Handbook of Algorithms and Data Structures (In Pascal and C), 2nd Edition, Addison-Wesley, Reading, MA, 1991.

[13] J. Graña, F.M. Barcala, M.A. Alonso, Compilation methods of minimal acyclic finite-state automata for large dictionaries, in: B.W. Watson, D. Wood (Eds.), Proc. Sixth Conf. on Implementations and Applications of Automata, Pretoria, South Africa, 2001, pp. 116–129.

[14] D. Gries, The Science of Computer Programming, 2nd Edition, Springer, Berlin, 1980.

[15] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, Cambridge, 1997.

[16] G.J. Holzmann, A. Puri, A minimized automaton representation of reachable states, Software Tools Technol. Transfer 3 (1998) 1, URL citeseer.nj.nec.com/holzmann97minimized.html.

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

[18] D. Jurafsky, J.H. Martin, Speech and Language Processing, Prentice-Hall, Englewood Cliffs, NJ, 2000.

[19] S. Mihov, Direct building of minimal automaton for given list, Technical Report, Bulgarian Academy of Science, 1999.

[20] S. Mihov, Direct building of minimal automaton for given list, Ph.D. Thesis, Bulgarian Academy of Science, 1999.

[21] K.-H. Park, J.-I. Aoe, K. Morimoto, M. Shishibori, An algorithm for dynamic processing of DAWGs, Internat. J. Comput. Math. 54 (1994) 155–173.

[22] P. Pevzner, Computational Molecular Biology: An Algorithmic Approach, MIT Press, Cambridge, MA, 2000.

[23] D. Revuz, Dictionnaires et lexiques: méthodes et algorithmes, Ph.D. Thesis, Institut Blaise Pascal, LITP 91.44, Paris, France, 1991.

[24] D. Revuz, Minimisation of acyclic deterministic automata in linear time, Theoret. Comput. Sci. 92 (1992) 181–189.

[25] D. Revuz, Dynamic acyclic minimal automaton, in: D. Wood, S. Yu (Eds.), Proc. Fifth Conf. on Implementations and Applications of Automata, London, Canada, 2000, pp. 226–232.

[26] K. Sgarbas, N. Fakotakis, G. Kokkinakis, Two algorithms for incremental construction of directed acyclic word graphs, Internat. J. Artificial Intelligence Tools 4 (1995) 369–381.

[27] B.W. Watson, A fast new semi-incremental algorithm for the construction of minimal acyclic DFAs, in: D. Wood, D. Maurel (Eds.), Proc. Third Workshop on Implementing Automata, Lecture Notes in Computer Science, Vol. 1660, Springer, Rouen, France, 1998, pp. 91–98.

[28] B.W. Watson, A taxonomy of algorithms for constructing minimal acyclic deterministic finite automata, South African Comput. J. 27 (2001) 12–17.
[29] B.W. Watson, An incremental DFA minimization algorithm, in: L. Karttunen, K. Koskenniemi, G. van Noord (Eds.), Proc. Second Int. Workshop on Finite State Methods in Natural Language Processing, Helsinki, Finland, 2001.
[30] B.W. Watson, A new recursive algorithm for building minimal acyclic deterministic finite automata, in: C. Martin-Vide, V. Mitrana (Eds.), Grammars and Automata for String Processing: From Mathematics and Computer Science to Biology and Back, Gordon and Breach, London, 2002.
[31] B.W. Watson, A fast and simple algorithm for constructing minimal acyclic deterministic finite automata, J. Universal Comput. Sci. 8 (2000) 2, URL www.jucs.org.