# Incremental Construction of Minimal Acyclic Finite-State Automata

Jan Daciuk
Technical University of Gdańsk*

Stoyan Mihov
Bulgarian Academy of Sciences†

Bruce W. Watson
University of Pretoria§

Richard E. Watson‡

*In this paper, we describe a new method for constructing minimal, deterministic, acyclic finite-state automata from a set of strings. Traditional methods consist of two phases: the first to construct a trie, the second one to minimize it. Our approach is to construct a minimal automaton in a single phase by adding new strings one by one and minimizing the resulting automaton on-the-fly. We present a general algorithm as well as a specialization that relies upon the lexicographical ordering of the input strings. Our method is fast and significantly lowers memory requirements in comparison to other methods.*

## 1 Introduction

Finite state automata are used in a variety of applications, including aspects of natural language processing (NLP). They may store sets of words, with or without annotations such as the corresponding pronunciation, base form, or morphological categories. The main reasons for using finite state automata in the NLP domain are that their representation of the set of words is compact and that looking up a string in a dictionary represented by a finite-state automaton is very fast — proportional to the length of the string. Of particular interest to the NLP community are deterministic, acyclic, finite-state automata, which we call **dictionaries**.

Dictionaries can be constructed in various ways — see Watson (1993a; 1995) for a taxonomy of (general) finite state automata construction algorithms. A word is simply a finite sequence of symbols over some alphabet and we do not associate it with a meaning in this paper. A necessary and sufficient condition for any deterministic automaton to be acyclic is that it recognizes a finite set of words. The algorithms described here construct automata from such finite sets.

The Myhill-Nerode theorem (see Hopcroft and Ullman (1979)) states that among the many deterministic automata that accept a given language, there is a unique automaton (excluding isomorphisms) that has a minimal number of states. This is called the **minimal** deterministic automaton of the language.

The generalized algorithm presented in this paper has been independently developed by Jan Daciuk of the Technical University of Gdańsk, and by Richard Watson and

* Department of Applied Informatics, Technical University of Gdańsk, Ul. G. Narutowicza 11/12, PL80-952 Gdańsk, Poland; e-mail: jandac@pg.gda.pl
† Linguistic Modelling Laboratory, LPDP — Bulgarian Academy of Sciences, Bulgaria; e-mail: stoyan@lml.bas.bg
‡ e-mail: watson@OpenFIRE.org
§ Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa, e-mail: watson@cs.up.ac.za

© 2000 Association for Computational Linguistics

Bruce Watson (then of the IST Technologies Research Group) at Ribbit Software Systems Inc. The specialized (to sorted input data) algorithm was independently developed by Jan Daciuk and by Stoyan Mihov of the Bulgarian Academy of Sciences. Jan Daciuk has made his C++ implementations of the algorithms freely available for research purposes at www.pg.gda.pl/∼jandac/fsa.html.[1] Stoyan Mihov has implemented the (sorted input) algorithm in a Java package for minimal acyclic finite-state automata. This package forms the foundation of the Grammatical Web Server for Bulgarian (at origin2000.bas.bg) and implements operations on acyclic finite automata, such as union, intersection and difference, as well as constructions for perfect hashing. Commercial C++ and Java implementations are available via www.OpenFIRE.org. The commercial implementations include several additional features such as a method to remove words from the dictionary (while maintaining minimality). The algorithms have been used for constructing dictionaries and transducers for spell checking, morphological analysis, two-level morphology, restoration of diacritics, perfect hashing, and document indexing. The algorithms have also proven useful in numerous problems outside the field of NLP, such as DNA sequence matching and computer virus recognition.

An earlier version of this paper, authored by Daciuk, Watson, and Watson, appeared at the International Workshop on Finite-state Methods in Natural Language Processing in 1998 — see Daciuk, Watson, and Watson (1998).

## 2 Mathematical Preliminaries

We define a deterministic finite state automaton to be a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where $Q$ is a finite set of states, $q_0 \in Q$ is the start state, $F \subseteq Q$ is a set of final states, $\Sigma$ is a finite set of symbols called the *alphabet* and $\delta$ is a partial mapping $\delta : Q \times \Sigma \longrightarrow Q$ denoting transitions. When $\delta(q, a)$ is undefined, we write $\delta(q, a) = \perp$. We can extend the $\delta$ mapping to partial mapping $\delta^* : Q \times \Sigma^* \longrightarrow Q$ as follows (where $a \in \Sigma$, $x \in \Sigma^*$):

$$\begin{aligned} \delta^*(q, \varepsilon) &= q \\ \delta^*(q, ax) &= \begin{cases} \delta^*(\delta(q, a), x) & \text{if } \delta(q, a) \neq \perp \\ \perp & \text{otherwise} \end{cases} \end{aligned}$$

Let DAFSA be the set of all deterministic finite state automata in which the transition function $\delta$ is acyclic — there is no string $w$ and state $q$ such that $\delta^*(q, w) = q$.

We define $\mathcal{L}(M)$ to be the language accepted by automaton $M$:

$$\mathcal{L}(M) = \{ x \in \Sigma^* \mid \delta^*(q_0, x) \in F \}$$

The size of the automaton, $|M|$, is equal to the number of states, $|Q|$. $\mathcal{P}(\Sigma^*)$ is the set of all languages over $\Sigma$. Define the function $\overrightarrow{\mathcal{L}} : Q \longrightarrow \mathcal{P}(\Sigma^*)$ to map a state $q$ to the set of all strings on a path from $q$ to any final state in $M$. More precisely,

$$\overrightarrow{\mathcal{L}}(q) = \{ x \in \Sigma^* \mid \delta^*(q, x) \in F \}$$

$\overrightarrow{\mathcal{L}}(q)$ is called the **right language** of $q$. Note that $\mathcal{L}(M) = \overrightarrow{\mathcal{L}}(q_0)$. The right language of a state can also be defined recursively:

$$\overrightarrow{\mathcal{L}}(q) = \{ a \overrightarrow{\mathcal{L}}(\delta(q, a)) \mid a \in \Sigma \wedge \delta(q, a) \neq \perp \} \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}$$

1 The algorithms in Daciuk's implementation differ slightly from those presented here, as he uses automata with *final transitions*, not final states. Such automata have fewer states and fewer transitions than traditional ones.

One may ask whether such a recursive definition has a unique solution. Most texts on language theory, for example Arbib, Moll and Kfoury (1988), show that the solution is indeed unique — it is the least fixed-point of the equation.

We also define a property of an automaton specifying that all states can be reached from the start state:

$$Reachable(M) \equiv \forall_{q \in Q} \exists_{x \in \Sigma^*} (\delta^*(q_0, x) = q)$$

The property of being a minimal automaton is traditionally defined as follows (see Watson (1993b; 1995)):

$$Min(M) \equiv \forall_{M' \in \mathrm{DAFSA}} (\mathcal{L}(M) = \mathcal{L}(M') \Rightarrow |M| \leq |M'|)$$

We will, however, use an alternative definition of minimality, which is shown to be equivalent:

$$Minimal(M) \equiv (\forall_{q,q' \in Q} (q \neq q' \Rightarrow \overrightarrow{\mathcal{L}}(q) \neq \overrightarrow{\mathcal{L}}(q'))) \wedge Reachable(M)$$

A general treatment of automata minimization can be found in Watson (1995). A formal proof of the correctness of the following algorithm can be found in Mihov (1998).

## 3 Construction from Sorted Data

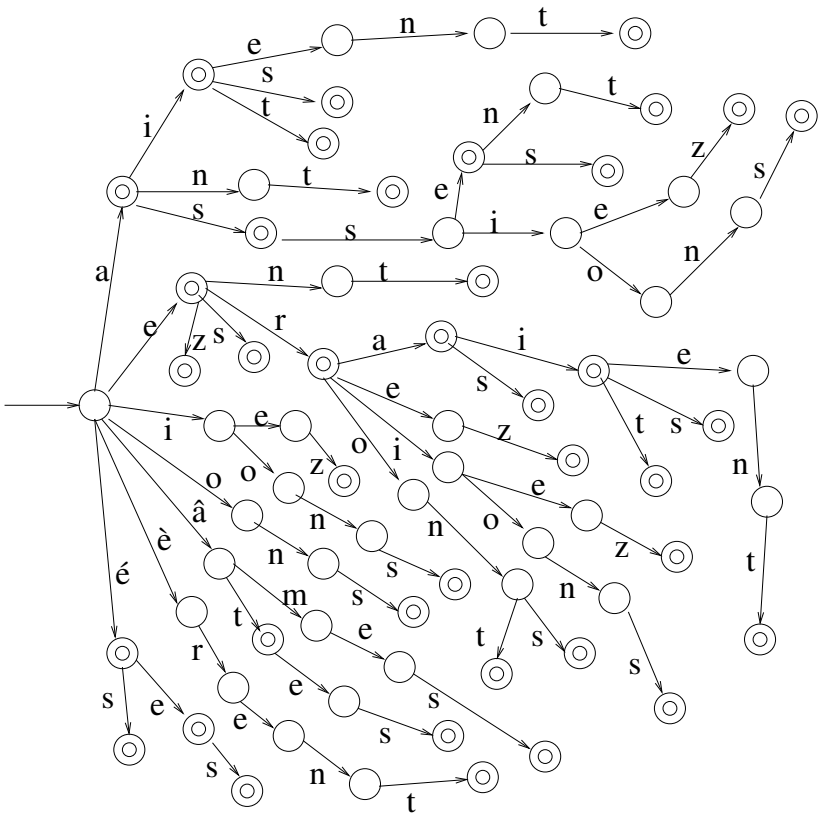

**Figure 1**
A trie whose language is the French regular endings of verbs of the first group.

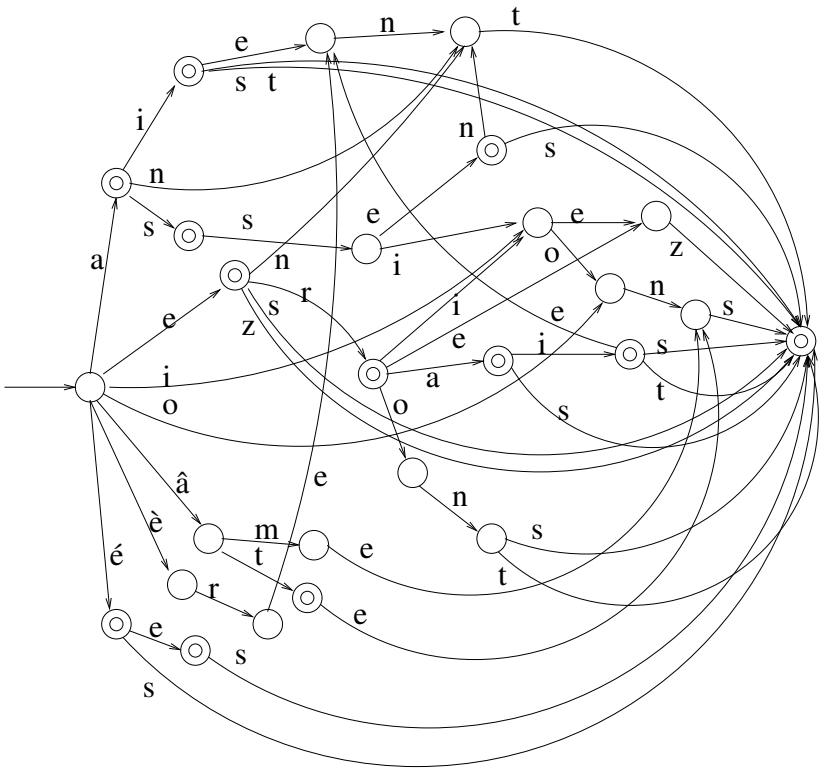


**Figure 2**
The unique minimal dictionary whose language is the French regular endings of verbs of the first group.

A trie is a dictionary with a tree-structured transition graph in which the start state is the root and all leaves are final states[2]. An example of a dictionary in a form of a trie is given in Figure 1. We can see that many subtrees in the transition graph are isomorphic. The equivalent minimal dictionary (Figure 2) is the one in which only one copy of each isomorphic subtree is kept. This means that, pointers (edges) to all isomorphic subtrees are replaced by pointers (edges) to their unique representative.

The traditional method to obtain a minimal dictionary is to first create a (not necessarily minimal) dictionary for the language and then minimize it using any one of a number of algorithms (again, see Watson (1993b; 1995) for numerous examples of such algorithms). The first stage is usually done by building a trie, for which there are fast and well understood algorithms. Dictionary minimization algorithms are quite efficient in terms of the size of their input dictionary — for some algorithms, the memory and time requirements are both linear in the number of states. Unfortunately, even such good performance is not sufficient in practice, where the intermediate dictionary (the trie) can be much larger than the available physical memory. (Some effort towards decreasing the memory requirement has been made; see Revuz (1991).) This paper presents a way to reduce these intermediate memory requirements and decrease the total construction time by constructing the minimal dictionary incrementally (word by word, maintaining an invariant of minimality), thus avoiding ever having the entire trie in memory.

The central part of most automata minimization algorithms is a classification of

2 There may also be non-leaf, in other words *interior*, nodes which are final.

states. The states of the input dictionary are partitioned such that the equivalence classes correspond to the states of the equivalent minimal automaton. Assuming the input dictionary has only reachable states (that is, $Reachable$ is *true*), we can deduce (by our alternative definition of minimality) that each state in the minimal dictionary must have a unique right language. Since this is a necessary and sufficient condition for minimality, we can use equality of right languages as the equivalence relation for our classes. Using our definition of right languages, it is easily shown that equality of right languages is an equivalence relation (it is reflexive, symmetric and transitive). We will denote two states, $p$ and $q$, belonging to the same equivalence class by $p \equiv q$ (note that $\equiv$ here is different from its use for logical equivalence of predicates). In the literature, this relation is sometimes written as $E$.

To aid in understanding, let us traverse the trie (see Figure 1) with the postorder method and see how the partitioning can be performed. For each state we encounter, we must check whether there is an equivalent state in the part of the dictionary that has already been analyzed. If so, we replace the current state with the equivalent state. If not, we put the state into a register, so that we can find it easily. It follows that the register has the following property: it contains only states which are pairwise inequivalent. We start with the (lexicographically) first leaf, moving backward through the trie toward the start state. All states up to the first **forward-branching** state (state with more than one outgoing transition) must belong to different classes and we immediately place them in the register, since there will be no need to replace them by other states. Considering the other branches, and starting from their leaves, we need to know whether or not a given state belongs to the same class as a previously registered state. For a given state $p$ (not in the register), we try to find a state $q$ in the register that would have the same right language. To do this, we do not need to compare the languages themselves — comparing sets of strings is computationally expensive. We can use our recursive definition of the right language. State $p$ belongs to the same class as $q$ if and only if:

1. they are either both final or both non-final; and
2. they have the same number of outgoing transitions; and
3. corresponding outgoing transitions have the same labels; and
4. corresponding outgoing transitions lead to states that have the same right languages.

Because the postorder method ensures that all states reachable from the states already visited are unique representatives of their classes (i.e. their right languages are unique in the visited part of the automaton), we can rewrite the last condition as:

4′ corresponding transitions lead to the same states.

If all the conditions are satisfied, the state $p$ is replaced by $q$. Replacing $p$ simply involves deleting it while redirecting all of its incoming transitions to $q$. Note that all leaf states belong to the same equivalence class. If some of the conditions are not satisfied, $p$ must be a representative of a new class and therefore must be put into the register.

To build the dictionary one word at a time, we need to merge the process of adding new words to the dictionary with the minimization process. There are two crucial questions that need to be answered. First, which states (or equivalence classes) are subject to change when new words are added? Second, is there a way to add new words to the dictionary such that we minimize the number of states that may need to be changed during the addition of a word? Looking at Figures 1 and 2, we can reproduce the same

postorder traversal of states when the input data is lexicographically sorted. (Note that in order to do this, the alphabet $\Sigma$ must be ordered, as is the case with ASCII and Unicode). To process a state, we need to know its right language. According to the method presented above, we must have the whole subtree whose root is that state. The subtree represents endings of subsequent (ordered) words. Further investigation reveals that when we add words in this order, only the states that need to be traversed to accept the previous word added to the dictionary may change when a new word is added. The rest of the dictionary remains unchanged, because a new word either

- begins with a symbol different from the first symbols of all words already in the automaton; the beginning symbol of the new word is lexicographically placed after those symbols; or
- it shares some (or even all) initial symbols of the word previously added to the dictionary; the algorithm then creates a forward branch, as the symbol on the label of the transition must be later in the alphabet than symbols on all other transitions leaving that state.

When the previous word is a prefix of the new word, the only state that is to be modified is the last state belonging to the previous word. The new word may share its ending with other words already in the dictionary, which means that we need to create links to some parts of the dictionary. Those parts, however, are not modified. This discovery leads us to the Algorithm 1, shown below.

**Algorithm 1**

*Register* := $\emptyset$;
**do** there is another word $\rightarrow$
  *Word* := next word in lexicographic order;
  *CommonPrefix* := *common_prefix(Word)*;
  *LastState* := $\delta^*(q_0, CommonPrefix)$;
  *CurrentSuffix* := *Word[length(CommonPrefix)+1...length(Word)]*;
  **if** *has_children(LastState)* $\rightarrow$
    *replace_or_register(LastState)*
  **fi**;
  *add_suffix(LastState, CurrentSuffix)*
**od**;
*replace_or_register*($q_0$)

**func** *common_prefix(Word)* $\rightarrow$
    **return** the longest prefix $w$ of *Word* such that $\delta^*(q_0, w) \neq \perp$
**cnuf**

**func** *replace_or_register(State)* $\rightarrow$
    *Child* := *last_child(State)*;
    **if** *has_children(Child)* $\rightarrow$
      *replace_or_register(Child)*
    **fi**;
    **if** $\exists_{q \in Q}(q \in Register \wedge q \equiv Child)$ $\rightarrow$
      *last_child(State)* := $q : (q \in Register \wedge q \equiv Child)$;
      *delete(Child)*

```
        else
          Register := Register ∪ {Child}
        fi
    cnuf
```

The main loop of the algorithm reads subsequent words and establishes which part of the word is already in the automaton (the *CommonPrefix*), and which is not (the *CurrentSuffix*). An important step is determining what the last state (here called *LastState*) in the path of the common prefix is. If *LastState* already has children, it means that not all states in the path of previously added word are in the path of the common prefix. In that case, by calling the function *replace_or_register*, we can let the minimization process work on those states in the path of the previously added word that are not in the common prefix path. Then we can add to the *LastState* a chain of states that would recognize the *CurrentSuffix*.

The function *common_prefix* finds the longest prefix (of the word to be added) that is a prefix of a word already in the automaton. The prefix can be empty (since $\delta^*(q, \varepsilon) = q$).

The function *add_suffix* creates a branch extending out of the dictionary, which represents the suffix of the word being added (the maximal suffix of the word which is not a prefix of any other word already in the dictionary). The last state of this branch is marked as final.

The function *last_child* returns a reference to the state reached by the lexicographically last transition that is outgoing from the argument state. Since the input data is lexicographically sorted, *last_child* returns the outgoing transition (from the state) most recently added (during the addition of the previous word). The function *replace_or_register* effectively works on the last child of the argument state. It is called with the argument that is the last state in the common prefix path (or the initial state in the last call). We need the argument state to modify its transition in those instances in which the child is to be replaced with another (equivalent) state. Firstly, the function calls itself recursively until it reaches the end of the path of the previously added word. Note that when it encounters a state with more than one child, it takes the last one, as it belongs to the previously added word. As the length of words is limited, so is the depth of recursion. Then, returning from each recursive call, it checks whether a state equivalent to the current state can be found in the register. If this is true, then the state is replaced with the equivalent state found in the register. If not, the state is registered as a representative of a new class. Note that the function *replace_or_register* processes only the states belonging to the path of the previously added word (a part, or possibly all of those created with the previous call to *add_suffix*), and that those states are never reprocessed. Finally, *has_children* returns *true* if, and only if, there are outgoing transitions from the state.

During the construction, the automaton states are either in the register or on the path for the last added word. All the states in the register are states in the resulting minimal automaton. Hence the temporary automaton built during the construction has less states than the resulting automaton plus the length of the longest word. Memory is needed for the minimized dictionary that is under construction, the call stack, and for the register of states. The memory for the dictionary is proportional to the number of states and the total number of transitions. The memory for the register of states is proportional to the number of states and can be freed once construction is complete. By choosing an appropriate implementation method one can achieve a memory complexity $\mathcal{O}(n)$ for a given alphabet, where $n$ is the number of states of the minimized automaton. This is an important advantage of our algorithm.

For each letter from the input list, the algorithm either has to make a step in the function *common_prefix* or add a state in the procedure *add_sufix*. Both operations can be performed in constant time. Each new state that has been added in the procedure *add_sufix* has to be processed exactly once in the procedure *replace_or_register*. The number of states that have to be replaced or registered is clearly smaller than the number of letters in the input list[3]. The processing of one state in the procedure consists of one register search and possibly one register insertion. The time complexity of the search is $\mathcal{O}(\log n)$,where $n$ is the number of states in the (minimized) dictionary. The time complexity of adding a state to the register is also $\mathcal{O}(\log n)$. In practice, however, by using a hash table to represent the register (and equivalence relation), the average time complexity of those operations can be made almost constant. Hence the time complexity of the whole algorithm is $\mathcal{O}(l \log n)$, where $l$ is the total number of letters in the input list.

## 4 Construction from Unsorted Data

Sometimes it is difficult or even impossible to sort the input data before constructing a dictionary. For example, there may be insufficient time or storage space to sort the data or the data originates in another program or physical source. An incremental dictionary-building algorithm would still be very useful in those situations, although unsorted data makes it more difficult to merge the trie-building and the minimization processes. We could leave the two processes disjoint, although this would lead to the traditional method of constructing a trie and minimizing it afterwards. A better solution is to minimize everything on-the-fly, possibly changing the equivalence classes of some states each time a word is added. Before actually constructing a new state in the dictionary, we first determine if it would be included in the equivalence class of a preexisting state. Similarly, we may need to change the equivalence classes of previously constructed states since their right languages may have changed. This leads to an incremental construction algorithm. Naturally, we would want to create the states for a new word in an order that would minimize the creation of new equivalence classes.

As in the algorithm for sorted data, when a new word $w$ is added, we search for the prefix of $w$ already in the dictionary. This time, however, we cannot assume that the states traversed by this common prefix will not be changed by the addition of the word. If there are any preexisting states traversed by the common prefix that are already targets of more than one in-transition (known as **confluence** states), then blindly appending another transition to the last state in this path (as we would in the sorted algorithm) would accidentally add more words than desired (see Figure 3 for an example of this).

To avoid generation of such spurious words, all states in the common prefix path from the first confluence state must be **cloned**. Cloning is the process of creating a new state that has outgoing transitions on the same labels and to the same destination states as a given state. If we compare the minimal dictionary (Figure 1) to an equivalent trie (Figure 2), we notice that a confluence state can be seen as a root of several original, isomorphic subtrees merged into one (as described in the previous section). One of the isomorphic subtrees now needs to be modified (leaving it no longer isomorphic), so it must first be separated from the others by cloning its root. The isomorphic subtrees hanging off these roots are unchanged, so the original root and its clone have the same outgoing transitions (that is, transitions on the same labels and to the same destination

3 The exact number of the states that are processed in the procedure *replace_or_register* is equal to the number of states in the trie for the input language.

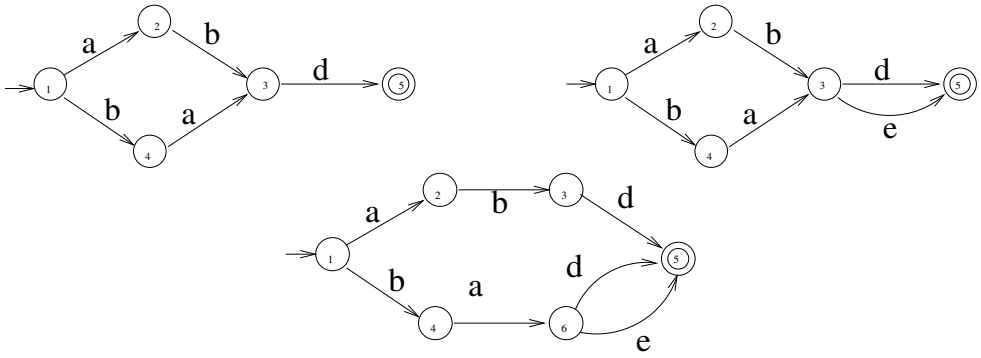


**Figure 3**
The result of blindly adding the word *bae* to a minimized dictionary (appearing on the left) containing *abd* and *bad*. The rightmost dictionary inadvertently contains *abe* as well. The lower dictionary is correct — state 3 had to be cloned.

states).

In the algorithm 1, the confluence states were never traversed during the search for the common prefix. The common prefix was not only the longest common prefix of the word to be added and all the words already in the automaton. It was also the longest common prefix of the word to be added and the last (i.e. the previous) word added to the automaton. As it was the function *replace_or_register* that created confluence states, and that function was never called on states belonging to the path of the last word added to the automaton, those states could never be found in the common prefix path.

Once the entire common prefix is traversed, the rest of the word must be appended. If there are no confluence states in the common prefix, then the method of adding the rest of the word does not differ from the method used in the algorithm for sorted data. However, we need to withdraw (from the register) the last state in the common prefix path in order not to create cycles. This is in contrast to the situation in the algorithm for sorted data where that state is not yet registered. Also, *CurrentSuffix* could be matched with a path in the automaton containing states from the common prefix path (including the last state of the prefix).

When there is a confluence state, then we need to clone some states. We start with the last state in the common prefix path, append the rest of the word to that clone and minimize it. Note that in this algorithm, we do not wait for the next word to come, so we can minimize (replace or register the states of) *CurrentSuffix* state by state as they are created. Adding and minimizing the rest of the word may create new confluence states earlier in the common prefix path, so we need to rescan the common prefix path in order not to create cycles, as illustrated in Figure 4. Then we proceed with cloning and minimizing the states on the path from the state immediately preceding the last state to the current first confluence state.

Another, less complicated but also less economical, method can be used to avoid the problem of creating cycles in presence of confluence states. In that solution, we proceed from the state immediately preceding the confluence state towards the end of the common prefix path, cloning the states on the way. But first, the state immediately preceding the first confluence state should be removed from the register. At the end of the common prefix path, we add the suffix. Then, we call *replace_or_register* with the predecessor of the state immediately preceding the first confluence state. The following should be

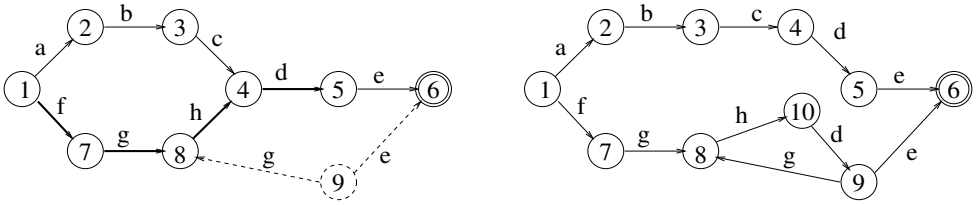


**Figure 4**
Consider an automaton (shown in solid lines on the left) accepting *abcde* and *fghde*. Suppose we want to add *fghdghde*. As the common prefix path (shown in thicker lines) contains a confluence state, we clone state 5 to obtain state 9, add the suffix to state 9, and minimize it. When we also consider the dashed lines in left figure, we see that state 8 became a new confluence state earlier in the common prefix path. The right figure shows what could happen if we did not rescan the common prefix path for confluence states. State 10 is a clone of state 4.

noted about this solution:

- memory requirements are higher, as we keep more than one isomorphic state at a time,
- the function *replace_or_register* must remain recursive (as in the sorted version), and
- the argument to *replace_or_register* must be a string, not a symbol, in order to pass subsequent symbols to children.

When the process of traversing the common prefix (up to a confluence state) and adding the suffix is complete, further modifications follow. We must recalculate the equivalence class of each state on the path of the new word. If any equivalence class changes, we must also recalculate the equivalence classes of all of the parents of all of the states in the changed class. Interestingly, this process could actually make the new dictionary smaller. For example, if we add the word *abe* to the dictionary at the bottom of Figure 3 while maintaining minimality, we obtain the dictionary shown in the right of Figure 3, which is one state smaller. The resulting algorithm is shown in Algorithm 2.

**Algorithm 2**

*Register* := $\emptyset$;
**do** there is another word $\rightarrow$
  *Word* := next word;
  *CommonPrefix* := *common_prefix*(*Word*);
  *CurrentSuffix* := *Word[length(CommonPrefix)+1...length(Word)]*;
  **if** $CurrentSuffix = \varepsilon \wedge \delta^*(q_0, CommonPrefix) \in F \rightarrow$
    **continue**
  **fi**;
  *FirstState* := *first_state*(*CommonPrefix*);
  **if** *FirstState* $= \emptyset \rightarrow$
    *LastState* := $\delta^*(q_0, CommonPrefix)$
  **else**
    *LastState* := *clone*($\delta^*(q_0, CommonPrefix)$)
  **fi**;

```
        add_suffix(LastState, CurrentSuffix);
        if FirstState ≠ ∅ →
          FirstState := first_state(CommonPrefix);
          CurrentIndex := (length(x): δ*(q0, x) = FirstState);
          for i from length(CommonPrefix) - 1 downto CurrentIndex →
              CurrentState := clone(δ*(q0, CommonPrefix[1...i]));
              δ(CurrentState, CommonPrefix[i]) := LastState;
              replace_or_register(CurrentState,Word[i+1]);
              LastState := CurrentState
          rof
        else
          CurrentIndex := length(CommonPrefix)
        fi;
        Changed := true;
        do Changed →
            CurrentIndex := CurrentIndex - 1;
            CurrentState := δ*(q0, Word[1...CurrentIndex]);
            OldState := LastState;
            if CurrentIndex > 0 →
               Register := Register - {LastState}
            fi;
            replace_or_register(CurrentState,Word[CurrentIndex+1]);
            LastState := δ(CurrentState, Word[CurrentIndex+1]);
            Changed := OldState ≠ LastState
            LastState := CurrentState
        od
        if ¬Changed ∧ CurrentIndex > 0 →
            Register := Register ∪ {CurrentState}
        fi
    od

    func replace_or_register(State, Symbol) →
          Child := δ(State, Symbol);
          if ∃q ∈ Q(q ∈ Register ∧ q ≡ Child) →
             delete(Child);
             last_child(State) := q : (q ∈ Register ∧ q ≡ Child)
          else
             Register := Register ∪{Child}
          fi
    cnuf
```

The main loop reads the words, finds the common prefix, and tries to find the first confluence state in the common prefix path. Then the remaining part of the word (*CurrentSuffix*) is added.

If a confluence state is found (i.e. *FirstState* points to a state in the automaton), all states from the first confluence state to the end of the common prefix path are cloned, and then considered for replacement or registering. Note that the inner loop (with $i$ as the control variable) begins with the penultimate state in the common prefix, because the last state has already been cloned and the function *replace_or_register* acts on a child of its argument state.

Addition of a new suffix to the last state in the common prefix changes the right languages of all states that precede that state in the common prefix path. The last part of the main loop deals with that situation. If the change resulted in such modification of the right language of a state that an equivalent state can be found somewhere else in the automaton, then the state is replaced with the equivalent one and the change propagates towards the initial state. If the replacement of a given state cannot take place, then (according to our recursive definition of the right language) there is no need to replace any preceding state.

Several changes to the functions used in the sorted algorithm are necessary to handle the general case of unsorted data. The *replace_or_register* procedure needs to be modified slightly. Since new words are added in arbitrary order, one can no longer assume that the last child (lexicographically) of the state (the one that has been added most recently) is the child whose equivalence class may have changed. However, we know the label on the transition leading to the altered child, so we use it to access that state. Also, we do not need to call the function recursively. We assume that *add_suffix* replaces or registers the states in the *CurrentSuffix* in correct order; later we process one path of states in the automaton, starting from those most distant from the initial state, proceeding towards the initial state $q_0$. So in every situation in which we call *replace_or_register*, all children of the state *Child* are already unique representatives of their equivalence classes.

Also, in the sorted algorithm, *add_suffix* is never passed $\varepsilon$ as an argument, whereas this may occur in the unsorted version of the algorithm. The effect is that the *LastState* should be marked as final since the common prefix is, in fact, the entire word. In the sorted algorithm, the chain of states created by *add_suffix* was left for further treatment until new words are added (or until the end of processing). Here, the automaton is completely minimized on the fly after adding a new word, and the function *add_suffix* can call *replace_or_register* for each state it creates (starting from the end of the suffix). Finally, the new function *first_state* simply traverses the dictionary using the given word prefix and returns the first confluence state it encounters. If no such state exists, *first_state* returns $\emptyset$.

As in the sorted case, the main loop of the unsorted algorithm executes $m$ times, where $m$ is the number of words accepted by the dictionary. The inner loops are executed at most $|w|$ times for each word. Putting a state into the register takes $\mathcal{O}(\log n)$, although it may be constant when using a hash table. The same estimation is valid for a removal from the register. In this case, the time complexity of the algorithm remains the same, but the constant changes. Similarly, hashing can be used to provide an efficient method of determining the state equivalence classes. For sorted data, only a single path through the dictionary could possibly be changed each time a new word is added. For unsorted data, however, the changes frequently fan-out and percolate all the way back to the start state, so processing each word takes more time.

### 4.1 Extending the algorithms

These new algorithms can also be used to construct transducers. The alphabet of the (transducing) automaton would be $\Sigma_1 \times \Sigma_2$, where $\Sigma_1$ and $\Sigma_2$ are the alphabet of the levels. Alternatively, elements of $\Sigma_2^*$ can be associated with the final states of the dictionary and only output once a valid word from $\Sigma_1^*$ is recognized.

## 5 Related work

An algorithm described by Revuz (1991) also constructs a dictionary from sorted data while performing a partial minimization on-the-fly. Data is sorted in reverse order and

that property is used to compress the endings of words within the dictionary as it is being built. This is called a **pseudominimization** and must be supplemented by a true minimization phase afterwards. The minimization phase still involves finding an equivalence relation over all of the states of the pseudo-minimal dictionary. It is possible to use unsorted data but it produces a much bigger dictionary in the first stage of processing. However, the time complexity of the minimization can be reduced somewhat by using knowledge of the pseudo-minimization process. Although this pseudo-minimization technique is more economic in its use of memory than traditional techniques, we are still left with a sub-minimal dictionary which can be a factor of 8 times larger than the equivalent minimal dictionary ((Revuz, 1991, page 33), reporting on the DELAF dictionary).

Recently, a semi-incremental algorithm was described by Watson (1998) at the Workshop on Implementing Automata. That algorithm requires the words to be sorted in *any* order of decreasing length (this sorting process can be done in linear time), and takes advantage of similar automata properties to those presented in this paper. In addition, the algorithm requires a *final* minimization phase after all words have been added. For this reason, it is only semi-incremental and does not maintain full minimality while adding words — although it usually maintains the automata close enough to minimality for practical applications.

## 6 Conclusions

We have presented two new methods for incrementally constructing a minimal, deterministic, acyclic finite state automaton from a finite set of words (possibly with corresponding annotations). Their main advantage is their minimal intermediate memory requirements[4]. The total construction time of these minimal dictionaries is dramatically reduced from previous algorithms. The algorithm constructing a dictionary from sorted data can be used in parallel with other algorithms that traverse or utilize the dictionary since parts of the dictionary that are already constructed are no longer subject to future change.

4 It is minimal in asymptotic terms; naturally compact data structures can also be used.

## 7 Acknowledgements

Jan Daciuk would like to express his gratitude to the Swiss Federal Scholarship Commission for providing a scholarship that made possible the work described here. Jan would also like to thank friends from ISSCO, Geneva, for their comments and suggestions on early versions of the algorithms given in this paper.

Bruce Watson and Richard Watson would like to thank Ribbit Software Systems Inc. for its continued support in these fields of applicable research.

All authors would like to thank the anonymous reviewers and Nanette Saes for their valuable comments and suggestions that led to significant improvements of the paper.

### References

Arbib, Michael A., A.J. Kfoury, and Robert N. Moll. 1988. *Introduction to Formal Language Theory*. Springer Verlag, New York, New York, USA.

Daciuk, Jan, Bruce W. Watson, and Richard E. Watson. 1998. Incremental construction of minimal acyclic finite state automata and transducers. In *International Workshop on Finite State Methods in Natural Language Processing*, Ankara, Turkey, 30 June–1 July.

Hopcroft, John E. and Jefferey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Reading, Massachusetts, USA.

Mihov, Stoyan. 1998. Direct building of minimal automaton for given list. In *Annuaire de l'Université de Sofia "St. Kl. Ohridski"*, volume 91. Faculté de Mathematique et Informatique, Sofia, Bulgaria, livre 1 edition, February.

Revuz, Dominique. 1991. *Dictionnaires et lexiques: méthodes et algorithmes*. Ph.D. thesis, Institut Blaise Pascal, Paris, France. LITP 91.44.

Watson, Bruce W. 1993a. A taxonomy of finite automata construction algorithms. Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands. Available via `www.OpenFIRE.org`.

Watson, Bruce W. 1993b. A taxonomy of finite automata minimization algorithms. Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands. Available via `www.OpenFIRE.org`.

Watson, Bruce W. 1995. *Taxonomies and Toolkits of Regular Language Algorithms*. Ph.D. thesis, Eindhoven University of Technology, the Netherlands. Available via `www.OpenFIRE.org`.

Watson, Bruce W. 1998. A fast new semi-incremental algorithm for construction of minimal acyclic DFAs. In *Third International Workshop on Implementing Automata*, Rouen, France, 17–19 September.