Incremental Construction of Minimal Acyclic Finite State Automata and Transducers

Jan Daciuk[1], Bruce W. Watson[2,3] and Richard E. Watson[3]

[1] jandac@pg.gda.pl
Technical University of Gdańsk
(Department of Applied Informatics)
Ul. G. Narutowicza 11/12
PL80-952 Gdańsk, Poland
[2] watson@cs.up.ac.za
University of Pretoria
(Department of Computer Science)
Hillcrest 0083, Pretoria, South Africa
[3] {watson, rwatson}@RibbitSoft.com
Ribbit Software Systems Inc.
(IST Technologies Research Group)
Box 24040, 297 Bernard Avenue, Kelowna
British Columbia, V1Y 1K9, Canada
www.RibbitSoft.com

Abstract. In this paper, we describe a new method for constructing minimal, deterministic, acyclic finite state automata and transducers. Traditional methods consist of two steps. The first one is to construct a trie, the second one — to perform minimization. Our approach is to construct an automaton in a single step 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 sorting of the input strings.

1 Introduction

Finite state automata are used in a variety of applications, such as natural language processing (NLP). They may store sets of words or sets of words with annotations, such as the corresponding pronunciation, lexeme, morphotactic categories, et cetera. The main reasons for the use of finite state automata in the NLP domain are their small size and very short lookup time. Of particular interest to the NLP community are deterministic, acyclic, finite state automata, which we call *dictionaries*. We refer to the set of all such dictionary automata as DAFSA.

Dictionaries can be constructed in various ways, using different data. (See Watson [3, 5] for a taxonomy of (general) finite state automata construction algorithms.) A word is simply a finite sequence of symbols over some alphabet (we do not associate them with a meaning during the construction phase). For the purpose of this article, the input data is a finite sequence of words. This is a necessary and sufficient condition for any resulting deterministic automaton to be acyclic.

The Myhill-Nerode theorem (see Hopcroft and Ullman [1]) states that among the many 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* automaton of the language.

The generalized algorithm presented in this paper has been independently developed by Jan Daciuk (he is also the sole developer of the sorted specialization of the algorithm) of the Technical University of Gdańsk and by Richard Watson and Bruce Watson of the IST Technologies Research Group at Ribbit Software Systems Inc. Jan Daciuk has made his C++ implementations of the algorithms freely available for research purposes at `www.pg.gda.pl/~jandac/fsa.html`. Ribbit's commercial C++ and Java implementations are available via `www.RibbitSoft.com`. Ribbit's implementations include several additional features such as a method to remove words from the dictionary (while maintaining minimality) and the ability to associate any type of annotation with a word in the dictionary (hence providing an efficient (p-)subsequential transducer implementation). In addition, it is possible to save a constructed dictionary and reload it on a different platform and implementation language (without endianess problems). The algorithms have been used for constructing dictionaries and transducers for spell checking, morphological analysis, two-level morphology, restoration of diacritics and perfect hashing. In addition, the algorithms have proven useful in numerous problems outside the field of NLP (for example, DNA sequence matching, computer virus recognition and document indexing).

2 Mathematical Preliminaries

Formally, 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, Σ is a finite set of symbols called the alphabet and δ is a partial mapping $\delta : Q \times \Sigma \longrightarrow Q$ denoting transitions. We can extend the δ mapping to $\delta^* : Q \times \Sigma^* \longrightarrow Q$ as in Hopcroft and Ullman [1]. 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, $|Q|$. Let the mapping $\overrightarrow{\mathcal{L}} : Q \longrightarrow \mathcal{P}(\Sigma^*)$ (where $\mathcal{P}(\Sigma^*)$ is the set of all languages over Σ) be the right language of a state q in M (the set of all strings, over Σ^*, on a path from state q to any final state of M using the extended transition relation δ^*):

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

Note that $\mathcal{L}(M) = \overrightarrow{\mathcal{L}}(q_0)$. We also define a property of an automaton specifying that all states can be reached from the start state:

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

The property of being a minimal automata is traditionally defined as follows (see Watson [3, 5]):

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

We will, however, use an alternative definition of minimality, which is shown to be equivalent (see Watson [3, 5]):

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

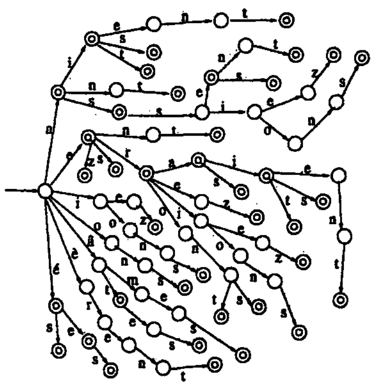

Figure 1. A trie whose language is the French regular endings of verbs of the first group.

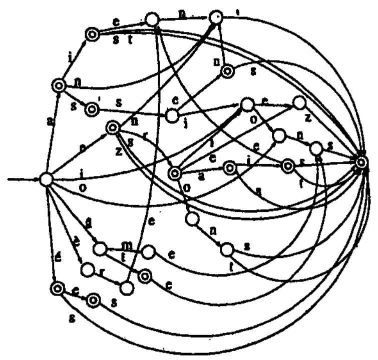

Figure 2. The unique minimal dictionary whose language is the French regular endings of verbs of the first group.

3 Construction from Sorted Data

A trie is a dictionary with a transition graph that is a tree with the start state as the root and all leaves being final. Let us picture a dictionary in a form of a trie (for example, see 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, for all isomorphic subtrees, only one copy of the tree is kept. That is, pointers (edges) to all isomorphic subtrees are replaced by pointers (edges) to their unique copy.

Traditionally, to obtain a minimal dictionary one would first create a dictionary for the language (not necessarily minimal), and then minimize it using any one of a number of algorithms (see Watson [4, 5]). Usually, the first stage is done by building a trie, for which there are fast and well understood algorithms. Although algorithms that minimize dictionaries can be fairly effective in their use of memory, they unfortunately have bad run-time performance. In addition, the size of the original dictionary can be enormous – although some effort towards decreasing its memory requirements have been reported — see Revuz [2]. 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 a trie in memory.

The central part of most automata minimization algorithms is a classification of states — see Watson [4, 5]. The states of a dictionary are partitioned into equivalence classes of which the representatives are the states of the minimal dictionary. Assuming the original dictionary does not have any useless states (that is, *Useful*(M) 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 our equivalence relation for our classes — see Watson [3, 5]. Using our definition of right languages, it is easily shown that equality of right languages is an equivalence relation (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).

Let us step through the minimization of the trie in Figure 1 using the algorithm given in Hopcroft and Ullman [1] and Watson [5]. As a first step, pairs of states where one is final and the other is not can immediately be marked as belonging to different equivalence classes (since only final states contain ε, the empty string, in their right language). Pairs of states that have a different number of outgoing transitions or the same number but with different labels can also be marked as belonging to different equivalence classes. Finally, pairs of states that have transitions labeled with the same symbols but leading to different states that have already been considered, can be marked as belonging to different equivalence classes.

Let us traverse the trie (see Figure 1) with the postorder method and see how the partition can be performed. 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. We can put them into a register of states so that we can find them easily. 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 class. The state being considered belongs to the same class as a representative of an established class if and only if:

1. they are either both final, or both non-final. If there is an annotation or some other type of information associated with each state, then states in the same equivalence class must all have equivalent information;
2. they have the same number of outgoing transitions;
3. corresponding transitions have the same labels;
4. corresponding transitions lead to the same states, and
5. states reachable via outgoing transitions are the sole representatives of their classes.

The last condition is satisfied by using the postorder method to traverse the trie. If all the conditions are satisfied, the state is replaced by the equivalent (representative) state found in the register. Replacing a state simply involves deleting the state while redirecting all of its in-transitions to the equivalent state. Note that all leaf states belong to the same equivalence class. If some of the conditions are not satisfied, the state must be a representative of a new class and therefore must be put into the register.

In order 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. Firstly, which states (or equivalence classes) are subject to change when new words are added? Secondly, 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 the Figures 1 and 2, it becomes clear that in order to reproduce the same postorder traversal of states, the input data must be lexicographically sorted. (Note that in order to do this, the alphabet Σ must be ordered). 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. All the rest of the dictionary remains unchanged. This discovery leads us to the algorithm shown in Algorithm 3.1.

Algorithm 3.1:

```
Register := $\emptyset$
do there is another word $\rightarrow$
    Word := next word;
    CommonPrefix := common_prefix(Word);
    LastState := $\delta^*(q_0, CommonPrefix)$;
    $CurrentSuffix := Word[length(CommonPrefix) + 1 \ldots length(Word)]$;
    if $has\_children(LastState)$ →
      $replace\_or\_register(LastState)$
    fi;
    $add\_suffix(LastState, CurrentSuffix)$
od;
$replace\_or\_register(q_0)$

func $common\_prefix(Word)$ →
      return $Word[1 \ldots n] : n = max\ i : \exists_{q \in Q} \delta^*(q_0, Word[1 \ldots i]) = q$
cnuf

func $replace\_or\_register(State)$ →
      $Child := last\_child(State)$;
      if not $marked\_as\_registered(Child)$ →
        if $has\_children(Child)$ →
          $replace\_or\_register(Child)$
        fi;
        if $\exists_{q \in Q}(marked\_as\_registered(q) \wedge q \equiv Child)$ →
          $delete\_branch(Child)$;
          $last\_child(State) := q$
        else
          $Register := Register \cup \{Child\}$;
          $mark\_as\_registered(Child)$
        fi
      fi
cnuf
```

□

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 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 (and an annotation associated with it, if applicable). The function *last_child* returns a (modifiable) 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). To determine which states have already been processed, each state has a marker that indicates whether or not it is already registered. Some parts of the automaton are left for further treatment (replacement or registering) until some other word is added so that those states no longer belong to the path in the automaton that accepts the new word. That marker is read with *marked_as_registered* and set with *mark_as_registered*. Finally, *has_children* returns *true* if, and only if, there are outgoing transitions from the node, and *delete_branch* deletes its argument state and all states that can be reached from it (if they are not already marked as registered).

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. Depending upon the choice of implementation method, memory may be required to maintain the equivalence relation.

The main loop of the algorithm runs m times, where m is the number of words to be accepted by the dictionary. The function *common_prefix* executes in $O(|w|)$ time, where $|w|$ is the maximum word length. The function *replace_or_register* executes recursively at most $|w|$ times for each word. In each recursive call, there is one register search and possibly one register insertion. The pessimistic time complexity of the search is $O(\log n)$, where n is the number of states in the (minimized) dictionary. The pessimistic time complexity of adding a state to the register is also $O(\log n)$. By using a hash table to represent the register (and equivalence relation), the average time complexity of those operations can be made constant. Since all children of a state are either replaced or registered, *delete_branch* executes in constant time. So the pessimistic time complexity of the entire algorithm is $O(m|w| \log n)$, while an average time complexity of $O(m|w|)$ can be achieved.

4 Construction from Unsorted Data

Sometimes it is difficult or impossible to sort the input data before constructing a dictionary. For example, when there is not enough 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 process and the minimization process. 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 a state's equivalence class 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 pre-existing state. In addition, 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 computation of the new equivalence classes.

Similar to the algorithm for sorted data, when a new word is added, we search for the common prefix 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 pre-existing 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 algoritm) would accidentally add more words than desired (see Figure 3 for an example of this).

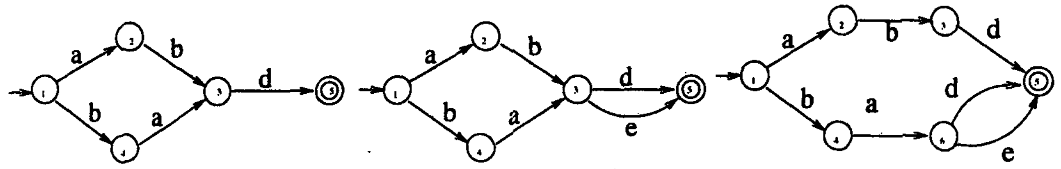

Figure 3. The result of blindly adding the word *bae* to a minimized dictionary containing *abd* and *bad*. The middle dictionary inadvertently contains *abe* as well. The rightmost dictionary is correct — state 3 had to be cloned.

To avoid generation of such spurious words, all states in the common prefix from the first state that has more than one in-transition 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 to an equivalent trie, 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 isomorphisms now needs to be modified, 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 states).

Once the entire common prefix is traversed, possibly cloning states along the way, 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. The addition of words in a lexicographical order in the sorted algorithm ensures us that we will not encounter any confluence states during the traversal on the common prefix.

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 minimal dictionary smaller. For example, if we add the word *abe* to the dictionary at the right of Figure 3 while maintaining minimality, we obtain the dictionary shown in the middle of Figure 3, which is one state smaller. The resulting algorithm is shown in Algorithm 4.1.

Algorithm 4.1:

```
$Register := \emptyset$
do there is another word $\rightarrow$
   $Word :=$ next word;
   $CommonPrefix := common\_prefix(Word)$;
   $FirstState := first\_state(CommonPrefix)$;
   if $FirstState = \emptyset \rightarrow$
      $LastState := \delta(q_0, CommonPrefix)$
   else
      $LastState := clone(\delta(q_0, CommonPrefix))$
   fi;
   $CurrentSuffix := Word[length(CommonPrefix) + 1 \dots length(Word)]$;
   $add\_suffix(LastState, CurrentSuffix)$;
   if $FirstState \neq \emptyset \rightarrow$
      $CurrentIndex := (length(x) : \delta^*(q_0, x) = FirstState)$;
      for $i$ from $length(CommonPrefix) - 1$ downto $CurrentIndex \rightarrow$
         $CurrentState := clone(\delta^*(q_0, CommonPrefix[1 \dots i]))$;
         $\delta(CurrentState, CommonPrefix[i]) := LastState$;
         $replace\_or\_register(CurrentState)$;
         $LastState := CurrentState$;
      rof
   else
      $CurrentIndex := length(CommonPrefix)$
   fi;
   $Changed := true$;
```

```
    do $Changed \rightarrow$
        $CurrentIndex := CurrentIndex - 1$;
        $CurrentState := \delta^*(q_0, Word[1 \ldots CurrentIndex])$;
        $OldState := LastState$;
        $mark\_as\_not\_registered(LastState)$;
        $Register := Register - \{LastState\}$;
        $replace\_or\_register(CurrentState)$;
        $Changed := OldState \neq LastState$
    od
od
```

□

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. Now, all children of a state must be checked; not only the most recently altered child. However, at most one child may need treatment, so the execution time is of the same order. Also, in the sorted algorithm, *add_suffix* is never passed ε 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. 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. So 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.

An algorithm described by Revuz [2] 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[4]. The minimization still involves finding an equivalence relation over all of the states of the pseudo-minimal dictionary[5]. However, the time complexity of the subset construction 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 ([2], the DELAF dictionary) than the equivalent minimal dictionary.

This new algorithm can also be used to construct transducers. The alphabet of the (transducing) automaton would be $\Sigma_1 \times \Sigma_2$, where Σ_1 and Σ_2 are the alphabet of the levels. Alternatively, as previously described, elements of Σ_2^* can be associated with the final states of the dictionary and only output once a valid word from Σ_1^* is recognized.

[4] This is called a pseudo-minimization and must be supplemented by a true minimization afterwards.

[5] It is possible to use unsorted data but it produces a much bigger dictionary in the first stage of processing.

5 Conclusions

We have presented two new methods for constructing minimal, deterministic, acyclic finite state automata whose languages are word sets (possibly with corresponding annotations). Both can be used to construct transducers as well as traditional acceptors. Their main advantage is their extremely low intermediate memory requirements which are achieved by building and minimizing the dictionaries incrementally. 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.

6 Acknowledgements

We would like to thank the anonymous referees and Nanette Saes for providing both technical and style feedback in polishing this paper. Jan Daciuk would also like to thank friends from ISSCO, Geneva, for their comments and suggestions on early versions of the algorithms given in this paper.

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

References

1. John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Co., Reading, M.A.
2. Dominique Revuz. 1991. *Dictionnaires et lexiques: méthodes et algorithmes*, Ph.D. dissertation, Institut Blaise Pascal, LITP 91.44, Paris, France.
3. Bruce W. Watson. 1993. A Taxonomy of Finite Automata Construction Algorithms. Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands. Available via `www.RibbitSoft.com/research/watson`.
4. Bruce W. Watson. 1993. A Taxonomy of Finite Automata Minimization Algorithms. Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands. Available via `www.RibbitSoft.com/research/watson`.
5. Bruce W. Watson. 1995. *Taxonomies and Toolkits of Regular Language Algorithms.* Ph.D. dissertation, Eindhoven University of Technology, The Netherlands. Available via `www.RibbitSoft.com/research/watson`.