Squibs and Discussions

Comments on "Incremental Construction and Maintenance of Minimal Finite-State Automata," by Rafael C. Carrasco and Mikel L. Forcada

Jan Daciuk*
Gdańsk University of Technology

In a recent article, Carrasco and Forcada (June 2002) presented two algorithms: one for incremental addition of strings to the language of a minimal, deterministic, cyclic automaton, and one for incremental removal of strings from the automaton. The first algorithm is a generalization of the "algorithm for unsorted data"—the second of the two incremental algorithms for construction of minimal, deterministic, acyclic automata presented in Daciuk et al. *(2000). We show that the other algorithm in the older article—the "algorithm for sorted data"—can be generalized in a similar way. The new algorithm is faster than the algorithm for addition of strings presented in Carrasco and Forcada's article, as it handles each state only once.*

1. Introduction

Carrasco and Forcada (2002) present two algorithms: one algorithm for incremental addition of strings into a minimal, cyclic, deterministic, finite-state automaton, and another for removal of strings from such an automaton. The algorithm for addition of strings can be seen as an extension to cyclic automata of the algorithm for unsorted data, the second algorithm in Daciuk et al, (2000). It turns out that not only the algorithm for unsorted data (the second algorithm in Daciuk et al. [2000]), but also the algorithm for sorted data (the first one in that article) can be extended in the same way. That extension is presented in Section 3 of this article.

Carrasco and Forcada emphasize on-line maintainance of dictionaries. Their dictionaries are constantly updated. In a different model, dictionaries are mostly consulted and are updated much less frequently. In such a model, it is more convenient to rebuild the dictionary off-line each time it is updated. By taking the process off-line, one saves much memory, as certain structures needed for construction are not needed for consultation, and other structures can be very efficiently compressed (Kowaltowski, Lucchesi, and Stolfi 1993; Daciuk, 2000). The data for dictionaries can be kept sorted; adding a few new (sorted) entries can be done in linear time. Although Carrasco and Forcada's string addition algorithm can be used in this particular model, an algorithm specialized for sorted data can perform the construction process faster than its more general equivalent.

The rest of the article is organized as follows. Section 2 introduces mathematical preliminaries. Section 3 presents an incremental algorithm for addition of sorted strings to a cyclic automaton. First, the role of a data structure called the **register** is explained in detail in Section 3.1, then necessary modifications to the algorithm in Carrasco and

* Deptartment of Knowledge Engineering, Ul. G. Narutowicza 11/12, 80-952 Gdańsk, Poland. E-mail: jandac@eti.pg.gda.pl.

Forcada (2002) are introduced in Section 3.2, and the final algorithm is presented in Section 3.3. The algorithm is then analyzed in Section 4 and evaluated in Section 5. Section 6 gives conclusions.

2. Mathematical Preliminaries

We define a deterministic finite-state automaton as $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of symbols called the alphabet, $q_0 \in Q$ is the start (or initial) state, and $F \subseteq Q$ is a set of final (accepting) states. As in Carrasco and Forcada (2002), we define $\delta : Q \times \Sigma \rightarrow Q$ as a total mapping. In other words, if the automaton is not complete, that is, if $\exists q \in Q \wedge \exists a \in \Sigma : \delta(q, a) \notin Q$, then an absorption state $\perp \notin F$ such that $\forall a \in \Sigma : \delta(\perp, a) = \perp$ must be added to Q. A complete acyclic automaton always has an absorption state. The extended mapping is defined as

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

The right language of a state q is defined as

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

The language of the automaton $\mathcal{L}(M) = \overrightarrow{\mathcal{L}}(q_0)$. The right language can be defined recursively:

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

Equality of right languages is an equivalence relation that partitions the set of states into abstraction classes (equivalence classes). The minimal automaton is the unique automaton (up to isomorphisms) that has the minimal number of states among automata recognizing the same language. It is also the automaton in which all states are useful (i.e., they are reachable from the start state, and from them a final state can be reached), and each equivalence class has exactly one member.

The length of a string $w \in \Sigma^*$ is denoted $|w|$, and the ith symbol (starting from one) in the string w is denoted w_i.

3. Incremental Addition of Sorted Strings

3.1 The Role of the Register

Carrasco and Forcada (2002) derive their algorithm for addition of strings from the union of an automaton $M = (Q, \Sigma, \delta, q_0, F)$ with a single-string automaton $M_w = (Q_w, \Sigma, \delta_w, q_{0w}, F_w)$. In a single-string automaton, $Q_w = \Pr(w) \cup \{\perp_w\}$, where $\Pr(w)$ is the set of all prefixes of w, which also serve as names of states, $\perp_w$ is the absoption state, $F_w = \{w\}$, and $q_{0w} = \epsilon$.

States in the automaton $M' = M \cup M_w$ that is the result of the union can be divided into four groups:

- **Intact** states of the form $(q, \perp_w)$ with $q \in Q - \{\perp\}$, states that are not affected by the union.

- **Cloned** states of the form (q, x) with $q \in Q - \{\perp\}$ and $x \in \Pr(w)$ such that $\delta^*(q_0, x) = q$. All other states in $(Q - \{\perp\}) \times \Pr(x)$ can be safely discarded. The new initial state (q_0, ϵ) is a cloned state.
- **Queue** states of the form $(\perp, x)$, with $x \in \Pr(w)$.
- The new **absorption** state $\perp' = (\perp, \perp_w) \notin F$. It is present only if M has an absoption state.

In Carrasco and Forcada, (2002), the algorithm for addition of strings proceeds by minimizing the queue states and cloned states, arriving at the minimal automaton. All states of M are put into a set called a register of states, which holds all unique states in the automaton. States unreachable from the new initial state are removed from the automaton and from the register. Then, starting from the states that are the most distant from the initial state, queue states and cloned states are compared against those in the register. If an equivalent state is found in the register, it replaces the state under investigation. If not, the state under investigation is added to the register.

Before we go further, we have to look at the role of the register of states in greater detail. It is explained in Daciuk et al. (2000) and omitted in Carrasco and Forcada (2002). Carrasco and Forcada do not have to examine the register closely, as they clone all states that they call **cloned states**. Incremental construction consists of two synchronized processes: One that adds new states, and another that minimizes the automaton. In minimization, it is important to check whether two states are equivalent. The Myhill-Nerode theorem tells us that two states are equivalent when they have the same right languages. Computing right languages can take much time. However, what we need to check is whether two states have the same right language, and not what that language actually is. We can use the recursive definition of the right language. If the target states of all outgoing transitions are unique in the automaton, that is, if they are already in the register, then instead of comparing their right languages, we can compare their identity (e.g., their addresses in memory). The assumption in the previous statement can be made true by enforcing a particular order in which states are compared against those in the register. When states are on a path representing a finite string, they should be processed from the end of the string toward the beginning.

The queue states should be processed in that order. If an equivalent state is found in the register, it replaces the current state. Otherwise, the current state is added to the register.

The register can be organized as a hash table. Finality of the state, the number of transitions, labels on transitions, and targets of transitions are treated together as a key—an argument to a hash function. The register does not store right languages. It stores pointers to states. If the right language of a state changes, the key of that state does not have to. Therefore, we do not need to take a state out from the register and put it back there if the key of the state does not change.

3.2 Necessary Modifications

We divide the set of cloned states into two groups: **prefix** states (up to, but excluding the first state with more than one incoming transition) and the **proper cloned** states. Proper cloned states are modified copies of other states. They are new states; they were created by adding a new string. In Carrasco and Forcada (2002), the prefix states are also cloned. However, it is usually not necessary to clone them (Carrasco and Forcada mention that on page 215). They all change their right languages as the result of adding a new string, but only the last prefix state (the most distant from the initial state) is sure to change its transitions. Therefore, it should be removed from the register *before* adding a new string. Other prefix states should be removed from the register only if they change their key features. This can happen only if the next prefix state in the path is replaced by another state. In that case, the current prefix state is removed from the register and reevaluated. If an equivalent state is found in the register, it replaces the current state, and the previous prefix state should be considered. Otherwise the state is put back into the register, and no further reevaluation is necessary.

If strings are added in an ordered way, the minimization process can be optimized in the same way as in the "sorted data algorithm," the first algorithm described in Daciuk et al. (2000). We introduce two changes to the string addition algorithm in Carrasco and Forcada (2002):

- Prefix states are not cloned when not necessary.
- States are never minimized (i.e., compared against the register and either put there or replaced by other states) more than once.

The first modification is described above. The second one requires more explanation. Let us consider an automaton in which no minimization takes place after a new string has been added. That automaton has form of a trie. If a set of strings is lexicographically sorted, then the paths in the automaton recognizing two consecutive strings w' and w share some prefix states (at least the initial state, the root of the trie). We denote the **longest common prefix** of w and w' as $lcp(w, w')$. If w' is a prefix of w, then all states in the path recognizing w' are also in the path of w. Otherwise, there will be states in the path recognizing w' that are not shared with the path recognizing w. Note that no subsequent words will have these states in the common prefix path either, as the shared initial part of paths of w' and subsequent words can only become shorter because of sorting. Therefore, the states after $lcp(w, w')$ will never change their right language, so they can be minimized without any further need of reevaluation. As soon as we add w, we know which states in the path of w' can be minimized. Instead of a trie, we keep a minimal automaton except for the path of the last string added to the automaton.

If we start from scratch and add strings in the manner just described, proper cloned states will never be created. Proper cloned states are created only when the common prefix of two words contains states with more than one incoming transition. Additional transitions coming to states are created when the states are in the register and they are found to be equivalent to some other states. But the states can be put into the register only when they are no longer in the common prefix path.

In case of a cyclic automaton, we do not start from scratch. There is an initial (minimal) automaton that contains cycles. No new cycles are created by adding mere strings one by one (as opposed to regular expressions, infinite sets of strings, etc.). As the automaton already contains some strings, and it can contain states with more than one incoming transition, proper cloned states can be created. However, no proper cloned states will be created in the common prefix path, because the path recognizing the previous string does not contain any states with more than one incoming transition.

3.3 The Algorithm

```
func build_automaton;
    $R \leftarrow Q$;
    if (fanin($q_0$) $> 0$) then
        $q_0 \leftarrow$ clone($q_0$);
    fi;
```

```
  $w' \leftarrow \epsilon$;
  **while** $((w \leftarrow \text{nextword}) \neq \epsilon)$ **do**
    $p \leftarrow \text{lcp}(w, w')$;
    $M \leftarrow \text{minim\_path}(M, w', p)$;
    $M \leftarrow \text{add\_suffix}(M, w, p)$;
    $w' \leftarrow w$;
  **end**;
  $\text{minim\_path}(M, w', q_0)$;
  **if** $\exists r \in R : \text{equiv}(r, q_0) \rightarrow$
    delete $q_0$; $q_0 \leftarrow r$;
  **fi**;
**cnuf**

**func** $\text{lcp}(M, w, w')$;
  $j \leftarrow \max(i : \forall_{k \leq j}\, w_k = w_{k'})$;
  **return** $w_1 \ldots w_j$;
**cnuf**

**func** $\text{minim\_path}(M, w, p)$;
  $q \leftarrow \delta^*(q_0, p)$;
  $i \leftarrow |p|; j \leftarrow i$;
  **while** $i \leq |w|$ **do**
    $\text{path}[i - j] \leftarrow q$;
    $q \leftarrow \delta(q, w_i)$; $i \leftarrow i + 1$;
  **end**;
  $\text{path}[i - j] \leftarrow q$;
  **while** $i > j$ **do**
    **if** $\exists r \in R : \text{equiv}(r, q)$ **then**
          $\delta(\text{path}[i - j - 1], w_{i-1}) \leftarrow r$;
          delete $q$;
    **else**
          $R \leftarrow R \cup \{q\}$;
    **fi**;
    $i \leftarrow i - 1$;
  **end**;
  **return** $M$;
**cnuf**

**func** $\text{add\_suffix}(M, w, p)$;
  $q \leftarrow \delta^*(q_0, p)$;
  $i \leftarrow |p| + 1$;
  **while** $i \leq |w|$ **and** $\delta(q, w_i) \neq \bot$ **and** $\text{fanin}(\delta(q, w_i)) \leq 1$ **do**
    $q \leftarrow \delta(q, w_i)$; $R \leftarrow R - \{q\}$; $i \leftarrow i + 1$;
  **end**;
  **while** $i \leq |w|$ **and** $\delta(q, w_i) \neq \bot$ **do**
    $\delta(q, w_i) \leftarrow \text{clone}(\delta(q, w_i))$;
    $q \leftarrow \delta(q, w_i)$; $i \leftarrow i + 1$;
  **end**;
  **while** $i < |w|$ **do**
    $\delta(q, w_i) \leftarrow \text{newstate}$;
    $q \leftarrow \delta(q, w_i)$; $i \leftarrow i + 1$;
```

```
54:   end;
55:   $F \leftarrow F \cup \{q\}$;
56:   return $M$;
57: cnuf
```

Function *fanin*(q) returns the number of incoming transition for a state q. If the initial state has more than one incoming transition, it must be cloned (lines 3–5) to prevent prepending of unwanted prefixes to words to be added. Function *nextword* simply returns the next word in lexicographical order from the input, or ϵ if there are no more words. Function *lcp* (lines 18–21) returns the longest common prefix of two words. It is called with the last string added to the automaton and the string to be added to the automaton as the arguments. For the first string, the previous string is empty. Function *minim_path* (lines 22–40) minimizes that part of the path recognizing the string previously added to the automaton that is not in the longest common prefix. This is done by going to the back of the path representing the string (lines 23–29) and checking the states one by one starting from the last state in the path (lines 30–38). The register is represented as variable R.

While function *minim_path* is not much different from an analogical function for the acyclic case, function *add_suffix* (lines 41–57) does introduce some new elements. It resembles more closely a similar function from the algorithm for unsorted data (Daciuk et al. 2000). The longest prefix common to the string to be added and the last string added to the automaton is not necessarily the same as the longest prefix common to the string to be added to the automaton and all strings already in the automaton. The latter can be longer, and the path recognizing it may contain states with more than one incoming transition. Those states have to be cloned (lines 47–50).

4. Analysis

The algorithm correctly adds new strings to the automaton, while maintaining its minimality. We assume that all states in the initial automaton are in the register, that there are no pairs of states with the same right language, that all states are reachable from the initial state, and that there is a path from every state to one of the final states. The absorption state and transitions that lead to it are not explicitly represented.

To prove that the algorithm is correct, we need to show that

1. the language of the automaton after the addition of the string contains that string;
2. no other strings are added to the automaton;
3. no strings are removed from the automaton;
4. the automaton remains minimal except for the path of the newly added string, that is, the states covered by the path of the newly added string are representatives of the only equivalence classes that may have more than one member.

It is easy to show that strings are indeed added to the language of the automaton. First, transitions with subsequent symbols from the strings are followed from the initial state. When there are no transitions with appropriate symbols, new ones are created. The state reachable with the string is made final. Minimization done by *minim_path* replaces states with other states that have the same right language. That operation does not change the language of the automaton.

If the initial state has any incoming transitions, it is cloned, and the clone becomes the new initial state. That operation does not change the language of the automaton—the right language of the new initial state is exactly the same as of the old one. The old initial state is still reachable, because it has incoming transitions from either the new initial state (the old initial state had a loop) or other states that are reachable. The cloning creates a new state that is not in the register and that is equivalent to another state in the automaton. Lines 14–16 of the algorithm check whether after addition of new strings, the new initial state is equivalent to some other state in the automaton. If it is, the new initial state is replaced with the equivalent state.

Since the automaton is deterministic, it cannot hold more than one copy of the same string. Therefore, we need only to show that no other strings are erroneously added to the automaton. Such erroneous addition could happen by creating or redirecting transitions. New transitions are created to store some suffixes of new strings that are not present in the automaton. This could lead to addition of new, superflous strings, provided the states that to which we add transitions are reentrant/confluence. However, the algorithm excludes such cases. All states in the path of the previously added string have only one incoming transition. All reentrant/confluence states not in the longest common prefix path are cloned in line 48 of function *add_suffix*. Function *minim_path* can redirect transitions only to states not in the longest common prefix path.

Since states that are deleted in line 33 in function *minim_path* (the only place in the algorithm where states are deleted) are always replaced as targets of transitions by equivalent states, strings could be deleted from the automaton only by making parts of it unreachable. However, all targets of transitions going out from a state to be deleted go to states that have more than one incoming transition—states that replaced previous targets of those transitions. This includes the case of states with no outgoing transitions.

To show that the automaton remains minimal except for the path of the newly added string, we first note that all existing states are in the register before we start adding new strings. Adding a new string creates a single chain of states not in the register. The chain is added in its entirety with function *add_suffix*, as the "previous" string for the first string is assumed to be empty. If w is the string to be added, and $\exists_{i>0}\exists_{q\in Q}\ \delta^*(q_0, w_1 \cdots w_i) \neq \bot$, then non-reentrant states not following any reentrant states in the path from q_0 to q are removed from the register, and reentrant states (and states that follow them) are cloned. For $w_{i+1} \cdots w_{|w|}$, new states and transitions are created. This concludes forming a path for the first string. That path consists entirely of states that are not in the register and that can have an equivalent state somewhere in the rest of the automaton.

When next strings are added, they are divided into two parts by function *lcp*. It divides both the previous and the next string. The first part (the longest common prefix) is shared between the previous and the next string, and it remains outside the register. This also means that for each state in that part, there may be an equivalent state in the remaining part of the automaton. The second part of the next string will form the rest of the path of states outside the register. The second part of the path of the previous string will be subject to minimization, as no further outgoing transitions will be added to any of its states in the future. Minimization replaces with their equivalent states those states in the path of the suffix of the previous string that are not unique. Since minimization is performed from the end of the string toward the longest common prefix, we can use the register and compare the states using the recursive definition of the right language, replacing right languages of target states with their addresses. At the end of the process, we have an automaton that is minimal except for the path of the last string added to it. We return to the start situation.

The algorithm has the same asymptotic complexity as the corresponding algorithms in Carrasco and Forcada (2002) and Daciuk et al. (2000). However, it is faster than algorithms for unsorted data, because it does not have to reprocess the states over and over again. Each time the original algorithm clones a state, that state is reprocessed. Cloning in the new version is limited to the part of the automaton built before addition of new strings. No state created by the algorithm is cloned afterward.

5. Evaluation

Two experiments have been performed to compare the new algorithm with the algorithm for adding strings to a minimal, deterministic, cyclic automaton presented in Carrasco and Forcada (2002). In both experiments, a cyclic automaton was created. It recognized any sequence of words from one set and any word from another set. The first set was used to construct an initial cyclic automaton recognizing any sequence of words from the first set. Then the second set was used to measure the relative speed of the algorithms being compared. In the first experiment, the first set consisted of German words beginning with Latin letters from *A* to *M*, and the second set consisted of German words beginning with letters from *N* to *Z*. This was the "easier" task, since only the initial state of the automaton had to be cloned. In the second experiment, odd-numbered German words beginning with letters *A* to *Z* formed the first set, and even-numbered ones, the second set. In this task, many paths in the automaton were shared between words from both sets. A total of 69,669 German words were used in the experiments.

In the first experiment, the new algorithm was 4.96 times faster, and in the second one, 2.53. Most of the speedup was not the result of using an algorithm optimized for sorted data—an improvement to the algorithm for adding strings in Carrasco and Forcada (2002) consisting in avoiding unnecessary cloning of prefix states (as described in section 3.2 and mentioned on page 215 in Carrasco and Forcada [2002] as a suggestion from one of Carrasco and Forcada's reviewers) was 3.12 and respectively 2.35 times faster than the original algorithm. However, the new algorithm is still the fastest.

6. Conclusions

An algorithm for adding strings to a cyclic automaton has been presented. It is faster than the algorithm for adding strings presented in Carrasco and Forcada (2002), but it operates on sorted input data. The new algorithm is a generalized version of the first algorithm presented in Daciuk et al. (2000). The relation between the algorithm presented here and the first algorithm in Daciuk et al. (2000) is the same as that between the algorithm for adding strings in Carrasco and Forcada (2002) and the second algoritm in Daciuk et al. (2000).

Acknowledgments

This research was carried out within the framework of the PIONIER Project Algorithms for Linguistic Processing, funded by NWO (Dutch Organization for Scientific Research) and the University of Groningen. The author wishes to thank the anonymous reviewers for valuable suggestions and corrections.

References

Carrasco, Rafael C. and Mikel L. Forcada. 2002. Incremental construction and maintenance of minimal finite-state automata. *Computational Linguistics*, 28(2): 207–216.

Daciuk, Jan. 2000. Experiments with automata compression. In M. Daley, M. G. Eramian, and S. Yu, editors, *Conference on Implementation and Application of Automata (CIAA'2000)*, pages 113–119, London, Ontario, Canada, July.

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

Kowaltowski, Tomasz, Cláudio L. Lucchesi, and Jorge Stolfi. 1993. Minimization of binary automata. In *First South American String Processing Workshop*, Belo Horizonte, Brazil.