Compilation Methods of Minimal Acyclic Finite-State Automata for Large Dictionaries[*]

Jorge Graña, Fco. Mario Barcala, and Miguel A. Alonso

Departamento de Computación, Facultad de Informática, Universidad de La Coruña
Campus de Elviña s/n, 15071 La Coruña, Spain
{grana,barcala,alonso}@dc.fi.udc.es

Abstract. We present a reflection on the evolution of the different methods for constructing minimal deterministic acyclic finite-state automata from a finite set of words. We outline the most important methods, including the traditional ones (which consist of the combination of two phases: insertion of words and minimization of the partial automaton) and the incremental algorithms (which add new words one by one and minimize the resulting automaton on-the-fly, being much faster and having significantly lower memory requirements). We analyze their main features in order to provide some improvements for incremental constructions, and a general architecture that is needed to implement large dictionaries in natural language processing (NLP) applications.

1 Introduction

Many applications of NLP, such as tagging or parsing a given sentence, can be too complex if we directly deal with the stream of input characters forming the sentence. Usually, a previous step of processing changes those characters into a stream of higher level items (called *tokens* and that typically are the words in the sentence), and obtains the candidate tags for these words rapidly and comfortably. This previous step is called lexical analysis or *scanning*.

The use of finite-state automata to implement efficient scanners is a well-established technique [1]. The main reasons for compressing a very large dictionary of words into a finite-state automaton are that its representation of the set of words is compact, and that looking up a word in the dictionary is very fast (proportional to the length of the word) [4]. Of particular interest for NLP are minimal acyclic finite-state automata, which recognize finite sets of words.

This kind of automata can be constructed in various ways [7]. This paper outlines the most important methods and analyzes their main features in order to propose some improvements for the algorithms of incremental construction. The motivation of this work is to build a general architecture to handle suitably two large Spanish dictionaries: the Galena lexicon (291,604 words with 354,007

[*] This work has been partially supported by the European Union (under FEDER project 1FD97-0047-C04-02), by the Spanish Government (under project TIC2000-0370-C02-01), and by the Galician Government (under project PGIDT99XI10502B).

possible taggings) and the Erial lexicon (775,621 words with 993,703 possible taggings)[1].

Section 2 describes our general model of dictionary and allows us to understand the role of the finite-state automata here. In Sect. 3, we give the formal definitions and explain how to establish a perfect hashing between the words and their positions in the dictionary, simply by assigning a *weight* to each state [5]. Section 4 recalls a minimization algorithm owing to Revuz [6], which is based on another property of the states: the *height*. Section 5 recalls the incremental construction by Daciuk [2], which performs insertions and minimizations at the same time, by storing in a *register* the states that will conform the final automaton. In Sect. 6, we combine *weights* and *heights* to improve the accesses to the *register*, and compare our implementation with the previous ones. Section 7 presents the conclusion after analysing the data obtained.

2 Compact Modeling of a Dictionary

Many words in a dictionary are manually inserted by linguists to exhaustively cover the invariant kernel of a language (articles, prepositions, conjunctions, etc.) or the terminology of a specific field. But many other words can be captured from annotated texts, making possible to obtain additional information, such as the frequency of the word or the probability with respect to each of its possible tags. This information is essential in some applications, e.g. stochastic tagging and parsing.

Therefore, our first view of a dictionary is simply a text file, with the following line format: `word tag lemma probability`. Ambiguous words use one different line for each possible tag. With no loss of generality, the words could be alphabetically ordered. Then, in the case of the `Galena` lexicon, the point in which the ambiguity of the word `sobre` appears could have this aspect[2]:

```
sobre P sobre 0.113229
sobre Scms sobre 0.00126295
sobre Vysps0 sobrar 0.0117647
```

For a later discussion, we say that the Galena lexicon has $M = 291,604$ different words, with $L = 354,007$ possible taggings. This last number is precisely the number of lines in the text file. The first tagging of `sobre` appears in the line $325,611$, but the word takes the position $268,249$ in the set of the M different lexicographically ordered words.

Of course, this is not an operative version for a dictionary. Therefore, what is important now is to provide a compiled version to compact this great amount

[1] Galena is *Generation of Natural Language Analyzers* and Erial is *Information Retrieval and Extraction Applying Linguistic Knowledge*. See `http://coleweb.dc.fi.udc.es` for more information of both projects.

[2] The tags come from the Galena tag set, which has a cardinal of $T = 373$ tags. The meanings for the tags (and for the word `sobre`) is the following: `P` is preposition (on); `Scms` is substantive, common, masculine, singular (envelope); and `Vysps0` is verb, first or third person, singular, present, subjunctive (to remain, to be superfluous).

of data, and also to guarantee an efficient access to it with the help of automata. The compiled version is shown in Fig. 1, and its main elements are:

– The `Word_to_Index` function (explained later) changes a word into its relative position in the lexicon (e.g. `sobre` into $268,249$).
– In a *mapping* array of size $M + 1$, this number is changed into the absolute position of the word (e.g. $268,249$ into $325,611$).
– This new number is used to access the arrays of *tags*, *lemmas* and *probabilities*, all of them of size L.
– The array of *tags* stores numbers, which are more compact than the names of the tags. Those names can be recover from the *tag set* array, of size T. The lexicographical ordering guarantees that the tags of a given word are adjacent, but we need to know how many they are. For this, it is enough to subtract the absolute position of the word from the value of the next cell (e.g. $325,614 - 325,611 = 3$ tags). This is also valid to correctly access the arrays of *lemmas* and *probabilities*.
– The array of *lemmas* also stores numbers. A lemma is a word that also has to be in the lexicon. The number obtained by the `Word_to_Index` function for this word is the number stored here, since it is more compact than the lemma itself. The original lemma can be recovered by the `Index_to_Word` function (explained later).
– The array of *probabilities* directly stores the probabilities. In this case, no reduction is possible.

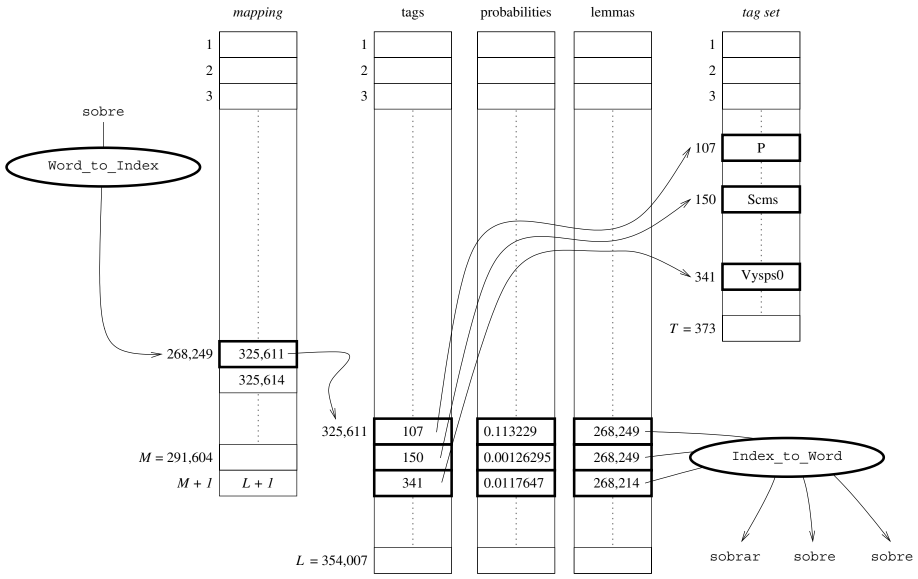

Fig. 1. Compact modeling of a dictionary

This is the most compact architecture for storing all the lexical information of the words present in a dictionary, when this information involves specific features of each word, such as the probability. Furthermore, this architecture is very flexible: it is easy to incorporate new arrays for other additional data (such as frequencies), or to remove the non-used ones (saving the corresponding space).

To complete this model, we only need the implementation of `Word_to_Index` and `Index_to_Word`. Both functions operate over a special type of automata, the numbered minimal acyclic finite-state automata described in the next section.

3 Numbered Minimal Acyclic Finite-State Automata

A *finite-state automaton* is defined by the 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states (the vertices of the underlying graph),
- Σ is a finite alphabet of the input symbols that conform the words (the labels in the transitions of the graph),
- δ is a function of $Q \times \Sigma$ into 2^Q defining the transitions of the automaton,
- q_0 is the initial state (the entrance of the graph), and
- F is the subset of final states of Q (usually marked with thicker circles).

The state or set of states reached by the transition of label a of the state q is denoted by $q.a = \delta(q, a)$. When this is only one state, i.e. when δ is a function of $Q \times \Sigma$ into Q, the automaton is *deterministic*. The notation is transitive: if w is a word then $q.w$ denotes the state reached by using the transitions labelled by each letter w_1, w_2, $\ldots$, w_n of w. A word w is accepted by the automaton if $q_0.w$ is in F. We define $\mathcal{L}(A)$, the *language recognized* by an automaton A, as the set of words w such that $q_0.w \in F$. An *acyclic* automaton is one such that the underlying graph is acyclic.

Deterministic acyclic automata are the most compact structure for recognizing finite sets of words. The ratios of compression are excellent and the recognition times are linear with respect to the length of the word to be scanned. The remaining sections of this paper will present several methods to obtain the minimal deterministic acyclic automaton for any finite set of words.

However, this is not enough for our model of dictionaries. We need a mechanism to transform every word into a univocal numeric key and viceversa. This transformation can easily be done if the automaton incorporates a *weight* for each state, this weight being the cardinal of the *right language* of the state, i.e. the number of substrings accepted from this state [5]. We refer to these automata as *numbered minimal deterministic acyclic finite-state automata*.

Figure 2 shows the numbered minimal automaton that recognizes all the forms of the English verbs `discount`, `dismount`, `recount` and `remount`[3]. The assignment of the indexing weights can be done by a simple recursive traversal of the automaton, when it has been correctly built and minimized.

Now, we can give the details of the functions that perform the hashing between the words in the lexicon and the numbers 1 to M (the size of the lexicon).

[3] The symbol `#` denotes the end of string.

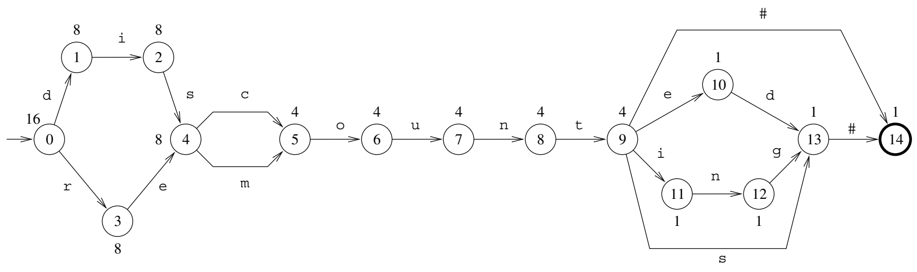

Fig. 2. Numbered minimal acyclic finite-state automaton for the forms of the verbs `discount`, `dismount`, `recount` and `remount`

The `Word_to_Index` function, shown in Fig. 5 of appendix A, starts working with an index equal to 1 and travels over the automaton using the letters of the word to scan. In every state of this path, the index is increased with the indexing weight of the target state of all transitions lexicographically preceding the transition used. If all the letters in the words have been processed and a final state is reached, the index contains the numeric key of the word. Otherwise, the function returns a value which indicates that the word is unknown.

The `Index_to_Word` function, shown in Fig. 4 of appendix A, starts working with the index and performs the analogous steps of `Word_to_Index` in order to deduce which transitions produce that index, and obtains the letters of the searched word from the labels of those transitions.

In the automaton of Fig. 2, the individual hashing of each word is:

```
 1 <-> discount    2 <-> discounted    3 <-> discounting    4 <-> discounts
 5 <-> dismount    6 <-> dismounted    7 <-> dismounting    8 <-> dismounts
 9 <-> recount    10 <-> recounted    11 <-> recounting    12 <-> recounts
13 <-> remount    14 <-> remounted    15 <-> remounting    16 <-> remounts
```

Note that M, in this case 16, is the indexing weight of the initial state and corresponds to the total number of words recognized by the automaton.

4 Minimization Based on the Height Property

In this section we start the study of the most efficient methods of building minimal acyclic automata. The first structure that we could consider to implement a scanner for a finite set of words is a *tree of letters*, which is itself an automaton where the initial state is the root and the final states are the leaves. However, the memory requirements of a tree are very high for large dictionaries[4]. Therefore, we apply a minimization process to reduce the number of states and transitions.

A minimization process can always be performed on any deterministic finite-state automaton, and the resulting automaton is equivalent, i.e. it recognizes the

[4] The Galena lexicon would need more than a million nodes (states) to recognize the 291.604 different words

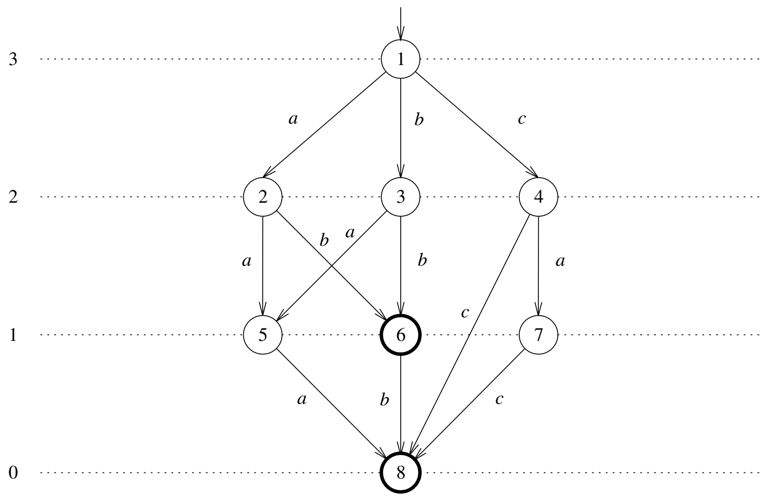

Fig. 3. A non-minimal acyclic deterministic finite-state automaton

same language as the original one [4]. Furthermore, if the automaton is acyclic, this process is simpler, as we will see through the rest of the paper.

On the other hand, and also due to the same memory requirements, it is not convenient to build a dictionary by inserting all the words in a tree and then obtaining the minimal automaton corresponding to that tree. Instead of this, it is more advisable to perform several steps of insertion and minimization[5].

In any case, to formally define the base of traditional minimization algorithms [6], we need the following definitions. Two automata are *equivalent* if they recognize the same language. Two states p and q are *equivalent* if the sub-automaton with p as initial state and the one that starts in q are equivalent. The opposite concept is that two states are non-equivalent or *distinguished.* If A is an automaton, there exists a unique automaton $\mathcal{M}$ minimal by the number of states, recognizing the same language, i.e. $\mathcal{L}(A) = \mathcal{L}(\mathcal{M})$. An automaton with no pair of equivalent states is minimal. Now, for a state s, we define its *height* $h(s) = max\,\{|w| \mid s.w \in F\}$, i.e. the height of a state s is the length of the longest path starting at s and leading to a final state. This function gives a partition Π of Q. Π_i denotes the set of states of height i. We say that the set Π_i is *distinguished* if no pair of states in Π_i is equivalent.

In Fig. 3 we show an automaton recognizing the language $\mathcal{L} = \{aaa, ab, abb, baa, bb, bbb, cac, cc\}$. States of the same height are drawn on the same dotted

[5] In [3] we describe how words must be properly inserted into an already minimized partial automaton in order to avoid inconsistencies. The basic idea is to clone conflicting states that can give rise to unintentional insertions of words not present in the original lexicon. Furthermore, we also give an empirical reasoning of the maximum size of the automaton needed to obtain a reasonable balance between the number of insertion-minimization steps and their duration.

line. This automaton is not minimal. States 2 and 3 of height 2 are equivalent. We can collapse these states by removing one of them, e.g. state 2, and replacing the target of its entering transitions by the other state, i.e. $1 \xrightarrow{a} 2$ by $1 \xrightarrow{a} 3$.

Now we can state the *height property*: if every Π_j with $j < i$ is distinguished, then two states p and q in Π_i are equivalent if and only if for any letter a in Σ the equality $p.a = q.a$ holds. The minimization algorithm by Revuz [6], in Fig. 6 of appendix A, follows from the height property. First we create a partition by height which is calculated by a standard traversal of the automaton for which the time complexity is $\mathcal{O}(t)$, where t is the number of transitions. If the automaton is not a tree, some speedup can be realized with a flag showing that the height of a state is already computed, and useless states which have no height can be eliminated during the traversal. Then, every Π_i is processed, from $i = 0$ to the height of the initial state, by sorting the states according to their transitions and collapsing equivalent states.

Using a sorting scheme with a time complexity $\mathcal{O}(f(e))$, where e is the number of elements to sort, the algorithm of Fig. 6 minimizes an acyclic automaton in

$$\mathcal{O}(t + \sum_{i=0}^{h(q_0)} f(|\Pi_i|))$$

which is less than the minimization algorithm by Hopcroft for general finite-state automata: $\mathcal{O}(n \times \log n)$, where n is the number of states [4].

This process needed 10 steps of insertion-minimization to build the minimal acyclic automaton for the Galena lexicon (11,985 states and 31,258 transitions), and took 29 seconds in a Pentium II 300 MHz. under Linux operating system.

5 Algorithms for Incremental Construction

As we have seen, traditional methods for constructing minimal acyclic automata from a finite set of words consist of two phases: the first being to construct a tree or a partial automaton, the second one being to minimize it. However, there are methods of incremental construction able to perform minimization in-line, i.e. at the same time as the words are inserted in the automaton [2]. These methods are much faster and have significantly lower memory requirements.

To build the automaton one word at a time, we need to merge the process of adding new words with the minimization process. There are two crucial questions that must be answered:

1. Which states are subject to change when new words are added?
2. Is there a way to add new words such that we minimize the number of states that may need to be changed during the addition of a word?

If the input data is lexicographically ordered, only the states that need to be traversed to accept the previous word added to the automaton may change when a new word is added. The rest of the automaton remains unchanged, because a new word either:

- begins with a symbol different from the first symbols of all words already in the automaton (in this case, the beginning symbol of the new word is lexicographically placed after those symbols); or
- it shares some initial symbols of the word previously added (in this case, the algorithm locates the last state in the path of the common prefix and creates a forward branch from that state, since the symbol on the label of the new transition must be later in the alphabet than symbols on all other transitions leaving that state).

Therefore, when the previous word is a prefix of the new word, the only states that can change are the states in the path of the previous word that are not in the path of the common prefix. The new word may share its ending with other words already inserted, which means that we need to create links to some parts of the automaton. Those parts, however, are not modified.

Now we describe the algorithm of incremental construction from a finite set of words in the lexicographical order. This algorithm, which is shown in Figs. 7 and 8 of appendix A, uses a structure called *Register* that always keeps a representative state of every equivalence class of states in the automaton. Therefore, the *Register* is itself the minimal automaton in every step.

The main loop of the algorithm reads subsequent words and establishes which part of the word is already in the automaton (the *Common_Prefix*), and which is not (the *Current_Suffix*). An important step is determining what the last state in the path of the common prefix is (the *Last_State*). If *Last_State* already has children, it means that not all states in the path of the previously added word are in the path of the common prefix. In that case, by calling the function `Replace_or_Register`, we 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 add to the *Last_State* a chain of states that recognize the *Current_Suffix*.

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 that has already been registered. 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 always takes the last one. 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 this function processes only those states belonging to the path of the previously added word, and that those states are never reprocessed.

In the same paper [2], the authors also propose an incremental construction method for unsorted sets of words, which is also based on the clonation of states that become conflicting as new words are added. The method is slower and uses more memory, but it is suitable when the sorting of the input data is complex and time-consuming.

6 Improving the Access to the Register

During the incremental 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 fewer states than the resulting automaton plus the length of the longest word. As a result of this, the space complexity is $\mathcal{O}(n)$, i.e. the amount of memory needed by the algorithm is proportional to n, the number of states in the minimal automaton. This is an important advantage of the algorithm.

With regard to the execution time, the algorithm presents two critical points which are marked with boxes in Fig. 8 of appendix A. This means that the time complexity will depend on the data structure implemented to perform the searches of equivalent states and the insertions of new representative states in the register. In [2], the authors suggest that, by using a hash table to implement the register and its equivalence relations, the time complexity of those operations can be made almost constant and equal to $\mathcal{O}(\log n)$.

Unfortunately, such a hashing structure is not described, although it can be deduced directly from the C++ implementation of the algorithm made freely available by the authors at `http://www.pg.gda.pl/~jandac/fsa.html`. This implementation took 3.4 seconds to build the minimal acyclic automaton for the Galena lexicon (11,985 states and 31,258 transitions) and 11.2 seconds to build the one for the Erial lexicon (52,861 states and 159,780 transitions), in a Pentium II 300 MHz. under Linux operating system.

Here, instead of a detailed study of that code, we prefer to detail our own implementation, since we think it automatically integrates some features that are needed in the general architecture of dictionaries presented in Sect. 2, and we have checked that is faster, as we will see later.

When a given state is subject to be replaced or registered, it must be compared with the states already present in the register. Of course, we cannot compare it with all these states, because the register becomes greater and greater as we insert new words in the automaton. Then, we have to think again: When are two states equivalent? We find the following answers for this question, each of them constituting a new filter that leaves more and more states out of the comparison process:

– Given two states, their heights have to be equal if the states are to be equivalent. The height is not specifically needed either for the incremental algorithm or for the dictionary scheme, but it nevertheless constitutes an effective filter.
  Furthermore, the height is a relatively low number (ranging between 0 and the length of the longest word), and it can be calculated in-line with no extra

traversal of the automaton (the length of a state is the maximum length of the target states of its outgoing transitions plus one).
– Given two states, the number of their outgoing transitions have to be also equal if the states are to be equivalent. This number is needed in order to construct the automaton correctly, and is also a relatively low number (ranging from 1 to the size of the alphabet used).
– Given two states, their weights have to be also equal if the states are to be equivalent. The weight is needed for the dictionary scheme, so it is a good idea to calculate it during the construction of the automaton (this is also possible since the weight of a state is the sum of the weights of the target states of its outgoing transitions).
  Of course, the range of possible values for the weight of a given state may be very high (ranging from 1 to the size of the lexicon), but empirical checks tell us that the most frequent weights are also relatively low numbers.

Therefore, our implementation of the register is a three-dimensional array which can be accessed by height, the number of outgoing transitions and weight. Each cell of this array contains the list of states that share this three features[6]. When a state is subject to being replaced or registered, we consider its features and it is only compared with the states in the corresponding list. Only then we verify the symbols of the labels of the outgoing transitions and their target states, which have to be equal if the states are to be equivalent.

When using our implementation of the incremental algorithm, the time needed to build the automaton of the Galena lexicon is reduced to 2.5 seconds. It takes an extra 4.6 seconds time to incorporate the information regarding tags, lemmas and probabilities, thus giving us a total compilation time of 7.1 seconds. In the case of the Erial lexicon, the equivalent times are 9.2 + 15.6 = 24.8 seconds.

Finally, it should be noted that the recognition speed of our automata is around 80,000 words per second. This figure is also an improvement on that obtained when using [2], which reaches 35,000 words per second.

The only explanation we can find for this improvement is that we have also managed to produce a more efficient internal architecture for automata. The description of this internal representation lies outside the scope of this paper, but any requests for further information on this subject are welcome.

7 Conclusion

Through an in-depth study of the different methods for constructing acyclic finite-state automata, we have presented two main contributions for handling suitably large sets of words in the NLP domain. The first has been to design a general architecture for dictionaries, which is able to store the great amount

[6] This is actually only true for states with weights between 1 and 15, this being empirically the most frequents. States with greater weights are stored in a separate set of lists. Nevertheless, the lists in this latter set are also ordered by weight.

of lexical data related to the words. We have shown that it is the most compact representation when we need to deal with very specific information of these words such as probabilities, this scheme being particularly appropriate for stochastic NLP applications.

In a natural way, the second contribution completes our model of dictionaries by improving the incremental methods for constructing minimal acyclic automata. In incremental constructions, since parts of the dictionary that are already constructed (i.e. the states in the *register*) are no longer subject to future change, we can use other specific features of states in parallel. These features are sometimes inspired in the working mechanisms of our architecture for dictionaries (e.g. indexing *weights*) and sometimes in the base of other algorithms (e.g. *heights*). All of them allow us to improve the access to the registered parts and check equivalences with the new states very rapidly. In consequence, the total construction time of these minimal automata is less than that of those previous algorithms.

References

[1] Aho, A. V.; Sethi, R.; Ullman, J. D. (1985). Compilers: principles, techniques and tools. *Addison-Wesley*, Reading, MA. 135

[2] Daciuk, J.; Mihov, S.; Watson, B. W.; Watson, R. E. (2000). Incremental construction of minimal acyclic finite-state automata. *Computational Linguistics*, vol. 26(1), pp. 3-16. 136, 141, 142, 143, 144

[3] Graña Gil, J. (2000). Robust parsing techniques for natural language tagging (in Spanish). *PhD. Thesis*, Departamento de Computación, Universidad de La Coruña (Spain). 140

[4] Hopcroft, J. E.; Ullman, J. D. (1979). Introduction to automata theory, languages and computations. *Addison-Wesley*, Reading, MA. 135, 140, 141

[5] Lucchesi, C. L.; Kowaltowski, T. (1993). Applications of finite automata representing large vocabularies. *Software - Practice and Experience*, vol. 23(1), pp. 15-30. 136, 138

[6] Revuz, D. (1992). Minimization of acyclic deterministic automata in linear time. *Theoretical Computer Science*, vol. 92(1), pp. 181-189. 136, 140, 141

[7] Watson, B. W. (1993). A taxonomy of finite automata construction algorithms. Computing Science Note 93/43, Eindhoven University of Technology, (The Netherlands). 135

A Pseudo-Code of the Main Algorithms

We give in this appendix the figures with the details of all the algorithms cited in the paper.

```
function $Index\_to\_Word$ ($Index$) =
    begin
        $Current\_State \leftarrow Initial\_State$;
        $Number \leftarrow Index$;
        $Word \leftarrow Empty\_Word$;
        $i \leftarrow 1$;

        repeat
            for $c \leftarrow First\_Letter$ to $Last\_Letter$ do
                if ($Valid\_Transition$ ($Current\_State, c$)) then
                    begin
                        $Auxiliar\_State \leftarrow Current\_State[c]$;
                        if ($Number > Auxiliar\_State.Number$) then
                            $Number \leftarrow Number - Auxiliar\_State.Number$
                        else
                            begin
                                $Word[i] \leftarrow c$;
                                $i \leftarrow i + 1$;
                                $Current\_State \leftarrow Auxiliar\_State$;
                                if ($Is\_Final\_State$ ($Current\_State$)) then
                                    $Number \leftarrow Number - 1$;
                                exit forloop
                            end
                    end
        until ($Number = 0$);

        return $Word$
    end;
```

Fig. 4. Pseudo-code of function `Index_to_Word`

```
function $Word\_to\_Index$ ($Word$) =
    begin
        $Index \leftarrow 1$;
        $Current\_State \leftarrow Initial\_State$;

        for $i \leftarrow 1$ to $Length$ ($Word$) do
            if ($Valid\_Transition$ ($Current\_State, Word[i]$)) then
                begin
                    for $c \leftarrow First\_Letter$ to $Predecessor$ ($Word[i]$) do
                        if ($Valid\_Transition$ ($Current\_State, c$)) then
                            $Index \leftarrow Index + Current\_State[c].Number$;
                    $Current\_State \leftarrow Current\_State[Word[i]]$;
                end
            else
                return $unknown\ word$;

        if ($Is\_Final\_State$ ($Current\_State$)) then
            return $Index$
        else
            return $unknown\ word$
    end;
```

Fig. 5. Pseudo-code of function `Word_to_Index`

```
procedure $Minimize\_Automaton$ ($Automaton$) =
    begin
        $Calculate\ \Pi$;

        for $i \leftarrow 0$ to $h(q_0)$ do
            begin
                $Sort\ the\ states\ of\ \Pi_i\ by\ their\ transitions$;
                $Collapse\ all\ equivalent\ states$
            end
    end;
```

Fig. 6. Pseudo-code of procedure `Minimize_Automaton`

```
function $Incremental\_Construction\ (Lexicon)$ =
begin
    $Register \leftarrow \emptyset$;

    while ($there\ is\ another\ word\ in\ Lexicon$) do
        begin
          $Word \leftarrow next\ word\ of\ Lexicon\ in\ lexicographic\ order$;
          $Common\_Prefix \leftarrow Common\_Prefix\ (Word)$;
          $Last\_State \leftarrow q_0.Common\_Prefix$;
          $Current\_Suffix \leftarrow Word[(Length\ (Common\_Prefix) + 1) \ldots Length\ (Word)]$;
          if ($Has\_Children\ (Last\_State)$) then
          $Register \leftarrow Replace\_or\_Register\ (Last\_State, Register)$;
          $Add\_Suffix\ (Last\_State, Current\_Suffix)$;
        end;

    $Register \leftarrow Replace\_or\_Register\ (q_0, Register)$;
    return $Register$
end;
```

Fig. 7. Pseudo-code of function `Incremental_Construction`

```
function $Replace\_or\_Register\ (State, Register)$ =
    begin
        $Child \leftarrow Last\_Child\ (State)$;
        if ($Has\_Children\ (Child)$) then
            $Register \leftarrow Replace\_or\_Register\ (Child, Register)$;
        if $\boxed{(\exists\, q \in Q : q \in Register \wedge q \equiv Child)}$ then
            begin
                $Last\_Child\ (State) \leftarrow q$;
                $Delete\ (Child)$
            end
        else
            $\boxed{Register \leftarrow Register\ \cup\ \{Child\};}$
        return $Register$
    end;
```

Fig. 8. Pseudo-code of function `Replace_or_Register`