# *On Some Applications of Finite-State Automata Theory to Natural Language Processing*

Mehryar Mohri
*AT&T Bell Laboratories*
*600 Mountain Avenue*
*Murray Hill, NJ 07974, USA*
*E-mail: mohri@research.att.com*

(*Received June 1995; Revised July 1996*†+)

**Abstract**

We describe new applications of the theory of automata to natural language processing: the representation of very large scale dictionaries and the indexation of natural language texts. They are based on new algorithms that we introduce and describe in detail. In particular, we give pseudocodes for the determinization of string to string transducers, the deterministic union of $p$-subsequential string to string transducers, and the indexation by automata. We report several experiments illustrating the applications.

## 1 Introduction

The theory of automata provides efficient and convenient tools for the representation of linguistic phenomena. Natural language processing can even be considered as one of the major fields of application of this theory (Perrin 1993). The use of finite-state machines has already been shown to be successful in various areas of computational linguistics: lexical analysis (Silberztein 1993), morphology and phonology (Koskenniemi 1985; Karttunen et al. 1992; Kaplan and Kay 1994; Mohri and Sproat 1996), local syntax (Roche 1993a; Mohri 1994d), syntax (Woods 1970; Roche 1993a; Mohri 1993; Pereira and Wright 1991), text-to-speech synthesis (Sproat 1995), speech recognition (Pereira et al. 1994; Mohri et al. 1996).

We here report new successful results of this theory in the following two applications: the representation or compilation of large scale dictionaries and the indexation of natural language texts. We have devised new algorithms that can be used as the bases for these applications. We describe in detail these new algorithms and indicate the various steps of the construction of the finite-state machines in practice, as well as the results of several experiments.

† This work was done while the author was an associate professor of computer science and computational linguistics at the Institut Gaspard Monge-LADL in Paris, France.

For the dictionary compilation experiments we carried out, we used the latest versions of the very large scale dictionaries of LADL (Laboratoire d'Automatique Documentaire et Linguistique) available in 1994. These versions of the dictionaries contain many more distinguishing morphological codes than older ones. We indicate how this affects the results. The experiments show the usefulness of $p$-subsequential transducers and their minimization in the representation of large scale dictionaries. We also experimented for the first time the compilation of the dictionary of compound words of French (DELACF) as well as that of all inflected words (simple and frozen). This allowed us to check the soundness of our programs for very large finite-state machines.

We also performed several experiments of indexation of natural language texts using transducers or automata. We mainly tested our programs on French corpora but the conclusions of our experiments are likely to apply to many other languages. Although the use of automata and transducers in indexation still needs to be optimized, experiments show it to be very promising. It also allows one to combine indexation with other text processing operations in a flexible way using classical operations such as composition of transducers.

In all these experiments, we used a large set of programs (more than 15,000 lines of code) written in C that can be used to perform efficiently many operations on automata and transducers including determinization, minimization, union, intersection, compaction, and other more complex or specific tasks. In the following, we constantly refer to these tools.

## 2 Compilation of large scale dictionaries into finite-state machines

Large dictionaries can be compiled into finite automata with distinct final states. We recall the principle of that method, the construction in practice, and the results of our experiments with large dictionaries. We then describe new methods for compiling dictionaries into $p$-subsequential transducers, indicate the results of our experiments using that method, and compare them with those obtained with automata. Notice that the methods described here both assume that the dictionaries are given as a large list of strings and not as a set of rules as considered by Kaplan and Kay (1994) for instance.

### *2.1 Representation by finite automata*

#### *2.1.1 Principle*

Large dictionaries can be efficiently represented by finite-state automata (Revuz 1991). Consider, for instance, the dictionary of inflected words of English (EDELAF), (Karlsfeld 1991). Lines of this dictionary are composed of an inflected form followed by its associated canonical form and morphological indications. The following are two typical lines of this dictionary:

done,do.V3:PP
done,done.A0

The inflected form *done* can correspond to the past participle of the verb *to do* (first line) or to the adjective *done* (second line). Inflected forms can be ambiguous, thus several canonical forms might be associated with the same inflected form. The dictionary can be factorized and put into a compact form in which canonical forms are computed from inflected forms by adding or removing some of their final letters. The above lines can be compacted into the following:

done,2.V3:PP,0.A0

Once put in this form, with each inflected form of the dictionary is associated a single code (here the code associated with *done* would be the string *2.V3:PP,0.A0*). The dictionary can then be represented by an automaton in which final states are provided with numbers referring to these codes. Figure 1 gives an example of such an automaton. It represents some of the inflected words of English.

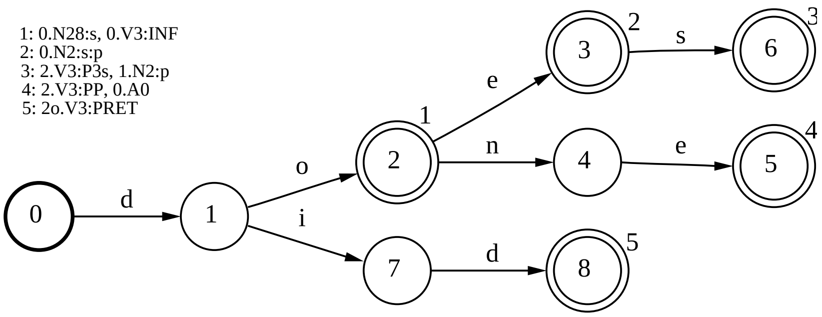


Fig. 1. Representation of dictionaries by automata.

The graph represented in Figure 1 is in fact a tree. It is generally impossible to store the set of words of a large-scale dictionary in such a deterministic tree, since this would lead to a blow-up. And, although efficient minimization algorithms for automata are available (Aho et al. 1974; Revuz 1991), one cannot apply them directly to a tree representing the whole dictionary. Therefore, the construction of the final minimal automaton requires splitting the dictionary into several smaller parts for which it is possible to construct a deterministic tree and therefore the corresponding minimal automaton, and then use several operations of union of automata to get the desired result.

### *2.1.2 Experiments*

We have fully implemented and experimented this method by considering several among the latest versions of the large dictionaries of LADL:

- the dictionary of simple inflected forms of French (FDELAF), (Courtois 1989),
- the dictionary of compound inflected forms of French (FDELAF), (Silberztein 1993),
- their union, the dictionary of all inflected forms of French (GDELAF),
- the dictionary of simple inflected forms of English (EDELAF), (Karlsfeld 1991),

- the dictionary of simple inflected forms of Italian (IDELAF).

Table 1 gives the corresponding results.

Table 1 Representation by Automata.

| | | | | | | |
|---|---|---|---|---|---|---|
| DICTIONARIES | Name | FDELAF French V.7 | FDELACF Compound | GDELAF General | EDELAF English | IDELAF Italian |
| | Nb of lines | 672,000 | 156,000 | 828,000 | 145,000 | 612,000 |
| | Initial size | 21.2 Mb | 5.6 Mb | 27.9 Mb | 3.6 Mb | 20 Mb |
| AUTOMATA | Nb of states | 84,600 | 322,800 | 389,650 | 48,770 | 78,320 |
| | Nb of transitions | 181,910 | 466,570 | 633,520 | 101,970 | 177,350 |
| | Size | 2.4 Mb | 7.6 Mb | 10 Mb | 1.2 Mb | 2.3 Mb |
| | Compacted size | 818 Kb | 1.88 Mb | 2.70 Mb | 447 Kb | 806 Kb |
| | Nb of codes | 13,200 | 1,750 | 14,950 | 1,590 | 11,190 |
| | Size of codes | 358 Kb | 445 Kb | 403 Kb | 25 Kb | 257 Kb |
| | Total final size | 1.2 Mb | 1.9 Mb | 3.1 Mb | 470 Kb | 1.1 Mb |
| TIME SPENT | Constr. (CRAY) | - | 12h40 | 18h53 | - | - |
| | Constr. (HP) | 12'30 | - | - | 4'55" | 12'30 |
| | Constr. (NEXT) | 1h18' | - | - | 17' | 1h20' |
| | Look-up (HP) | 90 w/ms | 90 w/ms | 90 w/ms | 90 w/ms | 90 w/ms |

The first lines of the table indicate the number of lines of the initial files containing these dictionaries as well as the corresponding sizes in bytes. Automata are in fact not sufficient to represent these dictionaries since one also needs to store corresponding codes. The next lines give the characteristics of the automata constructed as well as the number and total size of these codes in each case. The size of these codes depends of course on the choices made for the representation of each morphological feature and corresponds to a file simply containing their list.

The first size indicated for the automata corresponds to that of a file used to represent them. Such a representation can however be made much more compact thanks to the well-known technique proposed by Liang (1983) since the automata used in language processing are generally very sparse[1]. It is important to bear in mind that this method not only reduces the size of the automaton but also provides direct access, namely it makes the time necessary to search for a given transition at a given state constant. The compacted size indicated in table 1 corresponds to the space required to keep the automaton in memory which is equal to the one used to store it on disk when using the method just mentioned. The total final size is the amount of memory needed to represent the whole dictionary, including codes.

[1] This method has also been described by other authors in the same context of natural language processing (Liang 1983; Désarménien1986; Revuz 1991).

Our experiments were carried out using various machines (NEXT Cube 68040, 32 Mb RAM, HP/9000 755, 64 Mb RAM, CRAY II, 128 RAM). The time spent indicated for the construction of the minimal automata from the initial files should only be considered as an upper bound especially because of the presence of many other users. Not all figures are indicated in this part. Indeed, 38 Mb seemed to be insufficient[2] to apply without modification the minimization algorithm to the uncompacted automata corresponding to the dictionaries FDELACF and GDELAF. Thus, for these dictionaries and only for these, experiments were carried out on a CRAY. We did mention to be complete the time spent for the construction of the automata in the case of these dictionaries. However, those figures are not very significant due to the considerable number of programs running on that machine when the experiment was made. These experiments also helped us to check the soundness of our programs for very large automata and to construct for the first time these two very large dictionaries by minimal automata. The automaton corresponding to the GDELACF contains more than 630,000 transitions after minimization.

On the whole, the experiments show the usefulness of automata for the representation of dictionaries. Indeed, the time needed to construct the automata is short, their size is particularly compact compared to that of initial files, and they provide very fast look-up[3].

Notice that although the number of lines of the French dictionary of compound words is about the same as the number of lines of the English dictionary of simple words, the automaton corresponding to FDELACF is more than four times larger than that of EDELAF. This is mainly due to the length of compound words: although many compound words share the same prefix or suffix, the size of these shared parts is still small compared to the length of the compounds.

The representation by minimal automata of dictionaries of compound words could then seem less appropriate. One could have recourse for this type of dictionaries to other intermediate representations putting in common factors or subwords of the entries or use simple words as an alphabet for compound words. But, these methods do not necessarily improve the results. The latter for instance not only leads to a representation of FDELACF less compact (2.4 Mb, (Roche 1993b)), but also gives slower look-up because of the intermediate indexation it requires. A direct representation of FDELACF appears as both more compact and convenient. Besides, this representation offers the following advantage: the same operations then apply to FDELACF and FDELAF. So it is then natural to construct the union of these two dictionaries.

We have constructed the minimal automaton[4] corresponding to the union of these two dictionaries (GDELAF). It can be stored in 3.1 Mb of memory and contains all inflected forms of French. It allows one to lemmatize texts using a single automaton. Notice, however, that the size of this automaton is about the same as the sum of those of the dictionaries FDELAF and FDELACF.

[2] Because of independent technical reasons we could only use 38 Mb out of the 64 available in principle on the HP/9000 systems.

[3] The look-up time computed in number of words per second is only an indication. It depends on many practical parameters. It does not depend, of course, on the size of the automaton.

[4] One can efficiently construct from this machine a deterministic automaton recognizing $A^*L(GDELAF)$, where $A$ represents the French alphabet and $L(GDELAF)$ the set of all French inflected forms (Mohri 1995). This automaton is very useful in many natural language processing applications.

The experiment shows that making the union hardly reduces the number of transitions of these automata.

The main basis for the representation of dictionaries by automata is that, in general, many entries share the same codes, and that the total number of codes is small compared to the number of entries. Minimization has still then an important effect. But at worst, if all codes were distinct, then minimization would have no effect and the size of the deterministic tree representing a dictionary could not be reduced.

As a matter of fact, as dictionaries become more and more accurate the number of codes tends to increase considerably. This number is about 22,000 for the last version of FDELAF (in older versions considered by Roche (1993b) it was about 8,400). With this number of codes, the number of transitions of the automaton of FDELAF exceeds 220,000 which of course also increases the compacted size. Also, during the construction of this automaton one needs to distinguish different codes. But the space required for an efficient hashing of the codes can also become very costly. As an example, a tree used to represent these codes would have more than 100,000 nodes.

This dependency of the representation by automata on the codes seems to constitute a serious disadvantage of this method. Since the codes contain the linguistic information associated to words, their size can become very large and this can directly affect the whole representation. Thus, new versions of the dictionaries seem to suggest that the representation by automata would be less appropriate. The figures indicated in Table 1 correspond to this same version from which we have removed several substrings, and therefore some information, so as to reduce the number of codes.

### *2.2 Representation by sequential transducers*

Since the number of codes tends to increase, it becomes more appropriate to represent dictionaries by transducers. A morphological dictionary can be viewed as a list of pairs of strings of the type (inflected form, canonical form). This list defines a function which is natural to represent by a transducer. Such a representation also provides reverse look-up, namely given a canonical form to obtain the set of inflected forms associated with it. However, one needs to keep this representation efficient as in the case of deterministic automata.This led us to consider *sequential* transducers.

#### *2.2.1 definitions*

Let us recall some definitions relating to these efficient transducers.

- A transducer is said to be *sequential* when it has a deterministic input, namely at any state there is at most one transition labeled with a given element of the input alphabet.
- Sequential transducers can be extended to allow a single additional output string (*subsequential transducers*) or a finite number $p$ of output strings (*$p$-subsequential transducers*) at final states. These last transducers allow one to deal with the ambiguities in natural language processing (Mohri 1994a).
- We call *determinization* the algorithm allowing one, when possible, to obtain a $p$-subsequential transducer from a given one.

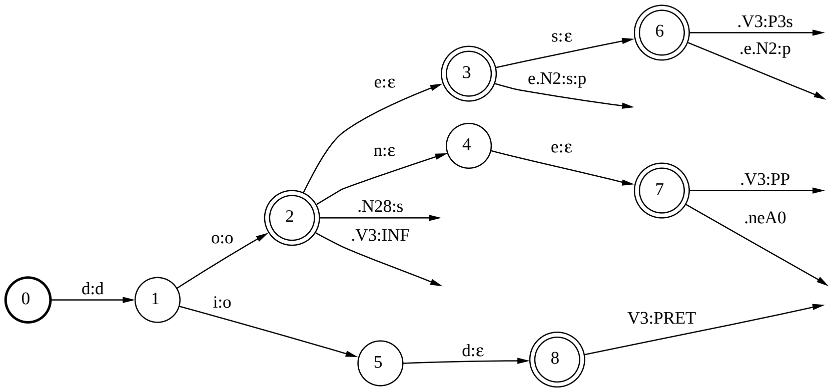


Fig. 2. Representation of dictionaries by $p$-subsequential transducers

- We also use the term *sequential* in a generic way to denote the class of all transducers with deterministic input (sequential, subsequential, $p$-subsequential) when no ambiguity arises as in the title of this section.

Like deterministic automata, sequential transducers provide a very fast look-up depending only on the length of the input string and not on the size of the machine. Entries of dictionaries are often ambiguous and one needs to associate several of them with a single form. These ambiguities can be represented using $p$-subsequential transducers. Figure 2 gives an example of $p$-subsequential transducer representing the same set of words as that of figure 1. Additional output strings are allowed at final states (arrows with no destination state).

There are several ways of adding new entries to a transducer. One method consists of adding these entries *non-deterministically*. When inserting a new pair of words in the transducer, new transitions are created when both input and output letters differ from those of existing transitions. The resulting transducer is then non-sequential since two transitions leaving the same state may have the same input labels. In many cases encountered in natural language processing, this transducer can be determinized so as to obtain a $p$-subsequential transducer.

### *2.2.2 Determinization of transducers*

The corresponding algorithm is close to the powerset construction used for determinizing automata. The difference is that here one needs to provide states of the sets with strings. These strings correspond to a delay in the emission which is due to the fact that outputs corresponding to a given input can be different. Because of this difference, only the longest common prefix of outputs can be kept and subsets are made of pairs (state, string) and not just of states. Figures 3 and 4 give an example of the application of the determinization. The subsets corresponding to the states of the subsequential transducer are indicated in 4. Notice that in this example the number of states of the determinized transducer $T_2$ is even less than with $T_1$. This is due to the fact that the transducer $T_2$ is subsequential.

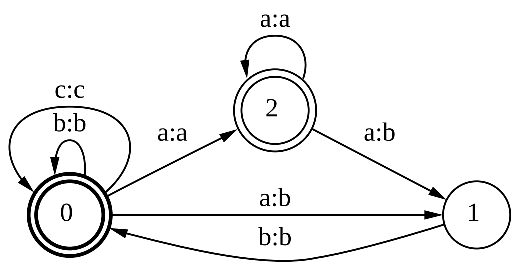


Fig. 3. Transducer $T_1$.

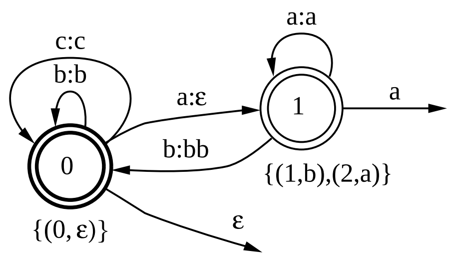


Fig. 4. Subsequential transducer $T_2$ obtained from $T_1$ by determinization.

A non-sequential transducer $T_1 = (V_1, I_1, F_1, A, B^*, \delta_1, \sigma_1)$ is a 7-tuple where:

- $V_1$ is the set of its states,
- $I_1 \subseteq V$ is the set of initial states,
- $F_1 \subseteq V$ the set of final states,
- $A$ and $B$ finite sets corresponding respectively to the input and output alphabets of the transducer,
- $\delta_1$ the state transition function mapping $V_1 \times A$ to $2^{V_1}$, the powerset of $V_1$,
- $\sigma_1$ the output function mapping $V_1 \times A \times V_1$ to $B^*$.

In the same way, a subsequential transducer $T_2 = (V_2, i_2, F_2, A, B^*, \delta_2, \sigma_2, \phi_2)$ is an 8-tuple where:

- $i_2$ is its unique initial state,
- $\delta_2$, its transition function maps $V_2 \times A$ to $V_2$,
- $\sigma_2$, its output function maps $V_2 \times A$ to $B^*$,
- $\Phi_2$, its final output function maps $F$ to $B^*$.

Given a positive integer $p$, a $p$-subsequential transducer $T_2 = (V_2, i_2, F_2, A, B^*, \delta_2, \sigma_2, \phi_2)$ can be defined in the same way except that $\Phi_2$ then maps $F$ to $(B^*)^p$.

Not all transducers can be determinized. This is because the class of subsequential transducers is a strict subclass of all transducers (Choffrut 1978). However, in most cases considered in natural language processing the transducers can be determinized. In particular, all acyclic transducers can be determinized. We here give the pseudocode of the algorithm to determinize a transducer $T_1$ when possible (figure 5). The result is a subsequential transducer $T_2$.

The notations are those just introduced above. We also denote by $x \wedge y$ the longest common prefix of two strings $x$ and $y$ and by $x^{-1}(xy)$ the string $y$ obtained by *dividing* $(xy)$ at left by $x$. We use a queue $Q$ to maintain the set of states of the resulting transducer $T_2$. Those states are equivalently subsets made of pairs $(q, w)$ of a state $q$ of $T_1$ and a string $w \in B^*$. We also use the following notations:

- $J_1(a) = \{(q, w) | \delta_1(q, a) \text{ defined and } (q, w) \in q_2\}$
- $J_2(a) = \{(q, w, q') | \delta_1(q, a) \text{ defined and } (q, w) \in q_2 \text{ and } q' \in \delta_1(q, a)\}$

to simplify the presentation of the pseudocode.

```
DETERMINIZATION_TRANSDUCER(T_1, T_2)
1   F_2 ← ∅
2   i_2 ← ⋃_{i ∈ I_1} {(i, ε)}
3   Q ← {i_2}
4   while Q ≠ ∅
5     do  q_2 ← head[Q]
6         if (there exists (q, w) ∈ q_2 such that q ∈ F_1)
7           then  F_2 ← F_2 ∪ {q_2}
8                 φ_2(q_2) ← w
9         for each a such that (q, w) ∈ q_2 and δ_1(q, a) defined
10          do    σ_2(q_2, a) ← ⋀_{(q,a) ∈ J_1(a)} [w ⋀_{q' ∈ δ_1(q,w)} σ_1(q, a, q')]
11                δ_2(q_2, a) ← ⋃_{(q,w,q') ∈ J_2(a)} {(q', [σ_2(q_2, a)]^{-1} w σ_1(q, a, q'))}
12                if (δ_2(q_2, a) is a new state)
13                  then  ENQUEUE(Q, δ_2(q_2, a))
14        DEQUEUE(Q)
```

Fig. 5. Algorithm for the determinization of a transducer $T_1$.

At each step of the algorithm a new state $q_2$ is considered (line 5). $q_2$ is a final state iff it contains a pair $(q, w)$ with $q$ final in $T_1$. In that case, $w$ is the final output at the state $q_2$. Then each input label $a$ of the transitions leaving the states of the subset $q_2$ is considered (line 10). A transition is constructed from $q_2$ to $\delta_2(q_2, a)$ with ouput $\sigma_2(q_2, a)$. $\sigma_2(q_2, a)$ is the longest common prefix of the output labels of all the transitions leaving the states $q$ of $q_2$ with input label $a$ when left concatenated with their delayed string $w$. $\delta_2(q_2, a)$ is the subset made of pairs $(q', w')$ where $q'$ is a state reached by one of the transitions with input label $a$ in $T_1$, and $w' = [\sigma_2(q_2, a)]^{-1} w\sigma_1(q, a, q')$ the delayed string that could not be output earlier in the algorithm. Notice that $[\sigma_2(q_2, a)]^{-1} w\sigma_1(q, a, q')$ is a well-defined string since $[\sigma_2(q_2, a)]$ is a prefix of all $w\sigma_1(q, a, q')$ (line 10).

Simple modifications allow one to extend the use of the algorithm to the case of transducers that can be represented by $p$-subsequential transducers. We need this extension for our applications since the dictionary transducers we consider generally admit ambiguities. The extension requires considering a function $\phi_2$ mapping $F_2$ to $(B^*)^p$, and changing lines 6-8 into the following ones:

6 **for** each $(q, w) \in q_2$ such that $q \in F_1$
7 **do** $F_2 \leftarrow F_2 \cup \{q_2\}$
8 ADD_OUTPUT$(\phi_2, q_2, w)$

where the effect of ADD_OUTPUT$(\phi_2, q_2, w)$ is to modify the function $\phi_2$ such that a new output string $w$ be added at the final state $q_2$.

### *2.2.3 Deterministic union of p-subsequential transducers*

One can also add a new entry to a transducer in a *deterministic* way. In order to add the pair $(w_1, w_2)$ to a transducer one can first insert $(w_1, \epsilon)$ in a deterministic way as with deterministic automata, and then associate to the final state reached a final output $w_2$. Proceeding this way for each new entry from the beginning of the construction one directly obtains a $p$-subsequential transducer which has the structure of a tree.

Experiments show that this method is often very efficient for constructing transducers representing large dictionaries. The construction from the dictionary file is then very fast. One disadvantage of this method is that the outputs are then pushed toward final states which creates a long delay in emission. However, $p$-subsequential transducers can be minimized (Mohri 1994a). An important characteristic of that minimization algorithm is that it pushes back ouputs as much as possible toward the initial state. Thus, it eliminates the problem just mentioned.

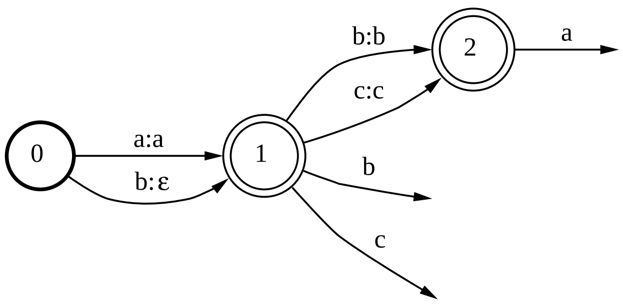


Fig. 6. $p$-Subsequential transducer $T_1$.

$p$-subsequential transducers allow very fast look-up. Minimization helps to make them also space efficient. But, as with automata, one cannot construct directly the $p$-subsequential transducer representing a large-scale dictionary. The tree construction mentioned above leads indeed to a blow up for a large number of entries.

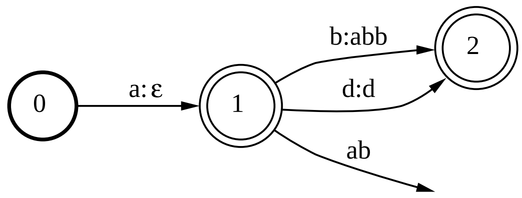


Fig. 7. $p$-Subsequential transducer $T_2$.

So, here again, one needs first to split the dictionary into several parts, construct the corresponding $p$-subsequential transducers, minimize them, and then perform the union of these transducers and reminimize the resulting one. However, one wishes the union of these transducers to be $p$-subsequential too to keep their look-up efficiency. To do so, the union of the $p$-subsequential transducers needs to be done in a *deterministic* way generating directly from two given $p$-subsequential transducers their union as a $p$-subsequential transducer.

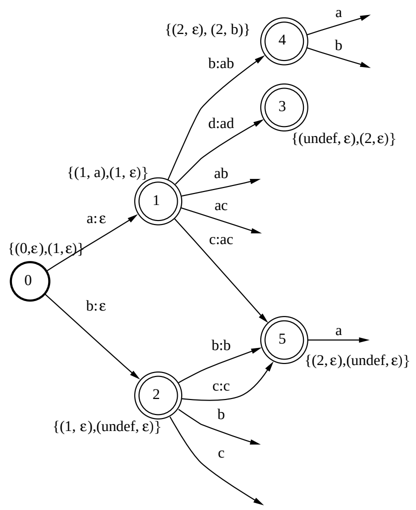


Fig. 8. $p$-Subsequential transducer $T_3$, deterministic union of $T_1$ and $T_2$.

Figures 6, 7 and 8 illustrate the deterministic union of two $p$-subsequential transducers. Only those final state outputs that are not empty strings are indicated in these figures. As in the case of automata, the corresponding union algorithm consists of reading the two transducers simultaneously and to consider pairs of states made of the states reached in each machine. However, here one also needs to associate strings to each of those states. Indeed, since for the same input label in the two machines the output labels might differ only the longest common prefix of the output labels is output. One needs to keep track of the remaining part of the strings. These are the strings associated to the states. Figure 9 gives the pseudocode of the algorithm to compute the deterministic union of two $p$-subsequential transducers.

```
UNION_p_SUBSEQUENTIAL_TRANSDUCER(T, T_1, T_2)
F ← ∅
i ← {(i_1, ε), (i_2, ε)}
Q ← {i}
while Q ≠ ∅
  do  q ← head[Q]   ▷ one can write: q = {(q_1, w_1), (q_2, w_2)}
      if (q_1 ∈ F_1 or q_2 ∈ F_2)
               then  F ← F ∪ {q}
                     for each output φ_ij(q_i) (i ∈ {1, 2}, j ≤ p)
                       do    ADD_OUTPUT(φ, q, w_i φ_ij(q_i))
      for each a such that δ_1(q_1, a) defined or δ_2(q_2, a) defined
               do    if (δ_1(q_1, a) undefined)
                       then  σ(q, a) ← w_2 σ_2(q_2, a)
                             δ(q, a) ←{(UNDEFINED,ε), (δ_2(q_2, a), ε)}
                     else if (δ_2(q_2, a) undefined)
                       then  σ(q, a) ← w_1 σ_1(q_1, a)
                             δ(q, a) ← {(δ_1(q_1, a), ε), (UNDEFINED,ε)}
                       else  σ(q, a) ← w_1 σ_1(q_1, a) ∧ w_2 σ_2(q_2, a)
                             δ(q, a) ← {(δ_1(q_1, a), [σ(q, a)]^{-1} w_1 σ_1(q_1, a)),
                                        (δ_2(q_2, a), [σ(q, a)]^{-1} w_2 σ_2(q_2, a))}
                     if (δ(q, a) is a new state)
                       then  ENQUEUE(Q, δ(q, a))
      DEQUEUE(Q)
```

Fig. 9. Algorithm for building a $p$-subsequential union $T$ of two $p$-subsequential transducers $T_1$ and $T_2$.

Given $T_1 = (V_1, i_1, F_1, A, B^*, \delta_1, \sigma_1, \phi_1)$ and $T_2 = (V_2, i_2, F_2, A, B^*, \delta_2, \sigma_2, \phi_2)$ two $p$-subsequential transducers[5], where $\phi_i$ is the output function associated with $T_i$, the algorithm constructs the union $T = (V, i, F, A, B^*, \delta, \sigma, \phi)$. We use in the algorithm a constant UNDEFINED distinct

[5] Recall that for any final state $q$, $(\phi_{ij}(q), (j \leq p)$, is the set of outputs at $q$ in $T_i$.

from all states of $T_1$ and $T_2$ and with the convention that $\delta_i(\text{UNDEFINED}, a)$ is not defined for any $a \in A$ and $i \in \{1, 2\}$.

As previously we use a queue $Q$ to maintain the states of the resulting machine. Each state $q$ of the resulting machine is considered once (line 5). It is equivalently a pair of the type $\{(q_1, w_1), (q_2, w_2)\}$ where $q_1$ is a state of $T_1$, $q_2$ a state of $q_2$ and $w_1$ and $w_2$ the delayed strings. $q$ is final iff both $q_1$ and $q_2$ are final. The set of final ouputs is then the union of all final outputs at $q_1$ and $q_2$ when left concatenated with the corresponding delayed strings (lines 6-9). Each input label $a$ of the transitions of $q_1$ and $q_2$ is then considered. A new transition from $q$ to $\delta(q, a)$ with output $\sigma(q, a)$ is then created considering the three cases:

- only $q_2$ admits a transition with input label $a$,
- only $q_1$ admits a transition with input label $a$,
- both $q_1$ and $q_2$ admit a transition with input label $a$.

In this last case only the longest common prefix of the output labels left concatenated with $w_1$ and $w_2$ can be output (line 17).

The efficiency of the implementation of these algorithms critically depends on the hashing method used to determine if the state defined is new (figure 5 line 12, figure 9 line 19). All other operations are less time consuming. We defined and used a new dynamic double hashing method based on classical methods (Aho et al. 1986; Cormen et al. 1992)that ensures both very fast look-up and uses few extra space. Both the determinization and the union algorithms are then very fast in practice.

### *2.2.4 Experiments*

We have fully implemented the algorithms and methods described in the two previous sections and experimented them with several large dictionaries. Table 2 illustrates some of these results[6]. The first lines of the table recall the size of the considered dictionaries and give indications about the number of transitions and states of the obtained transducers. The output labels of these transducers are not just letters. We have indicated in the following lines the number of elements and the size in memory of the output alphabet.

The final size corresponds to the space in memory or on disk required by the whole transducer when compacted. Since the transducer is sequential a compaction method close to the one used with automata can be used here (Liang 1983). It helps to reduce the size of the transducer and allows constant look-up time.

The experiments were carried out on two machines. The time spent indicated corresponds to the whole process of the construction of the transducer from the initial file containing the dictionary[7]. The look-up time should only be considered as an indication.

[6] The DELAPF dictionary consists of the list of inflected forms of French with their corresponding set of pronunciations (Laporte 1988).

[7] Notice that one does not need here to sort the initial dictionary file, or factorize it and compact it as in the case of the representation by automata.

Table 2. Representation by $p$-subsequential transducers.

| | | | | | |
|---|---|---|---|---|---|
| DICTIONARIES | Name | DELAPF Phonetic V.3 | FDELAF French V.7 | EDELAF English | IDELAF Italian |
| | Nb of lines | 472,000 | 672,000 | 145,000 | 612,000 |
| | Initial size | 9.6 Mb | 21.2 Mb | 3.6 Mb | 20 Mb |
| TRANSDUCERS | Nb of states | 46,750 | 67,000 | 47,540 | 64,390 |
| | Nb of transitions | 130,125 | 191,300 | 115,450 | 194,606 |
| | $p$ | 4 | 7 | 8 | 8 |
| | Size of the alph. (Nb) | 13,490 | 22,790 | 14,150 | 21,870 |
| | Size of the alph. (Space) | 85 Kb | 116 Kb | 154 Kb | 109 Kb |
| | Uncompacted size | 2.1 Mb | 3.6 Mb | 2 Mb | 3.5 Mb |
| | Final size | 870 Kb | 1.3 Mb | 790 Kb | 1.3 Mb |
| TIME SPENT | Construction (HP) | 9'30 | 20' | 11'35" | 19' |
| | Construction (NEXT) | 38' | 1h21 | 30' | 1h35' |
| | Look-up (HP) | 80 w/ms | 80 w/ms | 80 w/ms | 80 w/ms |

Since the transducers we use here are sequential, the look-up time only depends on the length of the input word and not on the size of the dictionary. Also, the minimization algorithm we use for transducers not only helps to reduce the number of states without losing the advantage of look-up efficiency, but also provides a quasi-determinization of the output side of the transducer (Mohri 1994a). Thus, the average time for the inverse look-up (from canonical forms to the inflected forms) is also very low and comparable to the one indicated in the table.

The time spent to construct the minimal $p$-subsequential transducers is longer than that of the construction of corresponding automata (sometimes only slightly longer), but it is still quite short and the whole process of construction of the transducers can be considered as very fast. The experiments also show that the size of the output alphabet remains small and that its contribution to the final size of the transducer is negligible.

The final transducers obtained are very compact. Compare for instance the size of the $p$-subsequential transducer representing the French dictionary FDELAF and that of the corresponding automaton. Both sizes are roughly the same (1.2 Mb for the automaton, 1.3 for the transducer). Experiments also suggest that with the increase of the number of codes, the size of the automaton could exceed the size of the minimal transducer. Besides, the transducer allows look-up's from both sides whereas the automaton does not.

One can also compose the transducer with an automaton using the output of the transducer to match the labels of the automaton. This of course cannot be done using the representation by automata. Also, the use of the minimal transducers makes the finite-state machine more independent of the choice of the codes. Although the order in which morphological features are indicated is still important and could change the number of transitions or states of the transducer, the prefixation stage of the minimization algorithm makes the transducer independent of the precise notations for tenses or moods for instance. It is also worthwhile to point out that the $p$-subsequential transducers obtained here both provide a fast look-up time and are more compact than the non-sequential transducers used to represent dictionaries. The size of the non sequential transducer representing FDELAF constructed by Roche (1993b) is about 1.5 Mb (> 1.3 Mb), and the size of the DELAPF transducer 6.9 Mb (>> 870 Kb here). Notice that the time spent for the construction of these transducers is about the same in both cases.

These results of our experiments tend to confirm the efficiency of the representation of dictionaries by $p$-subsequential transducers. They provide fast look-up time, double side look-up, and compactness. Several extensions of these results can be considered. Practical results can be improved using various optimization heuristics to reduce sizes or to the time spent for the construction of the machines. From a theoretical point of view, one can consider the extension of these methods to the case of polysequential transducers (Schutzenberger 1987) and to that of large weighted finite-state machines (Mohri et al. 1996).

## 3 Indexation with finite-state machines

Finite-state machines can also be used in indexation of texts. The algorithm devised by Crochemore (1986) allows one to obtain in linear time and linear space a subsequential transducer representing the general index of a text, namely the position of all factors of a text[8]. An example of such a transducer for a text reduced to $t = aabba$ is represented in figure 10.

### *3.1 Indexation with subsequential string to weight transducers*

The input automaton associated with this transducer represents exactly the set of suffixes of the text $t$. Since any factor of a text is a prefix of a suffix of this text, this input automaton also allows one to recognize the factors of $t$. The transducer associates a number to each input string. This number is obtained by adding the output integers of the labels during the recognition. For instance, when applied to the factor $b$ the transducer outputs the single integer 2. This integer corresponds to the first position of the factor $b$ in the text. All other possible positions are obtained by searching the rest of the automaton from the state 5. Any path leading from this state to a final state corresponds to a position of $b$. Here, two paths (5-4-6 and 5-6) can be found and give the following integers: $0 + 0 = 0$ and 1. Therefore the set of the positions of $b$ in $t$ is $\{2 + 0 = 2, 2 + 1 = 3\}$. In the same way, the transducer can be used to find the three positions of $a$ in $t$: $0 + (4) = 4$ (path 1), $0 + (1 + 0 + 0) = 1$ (path 1-3-4-6), $0 + (0 + 0 + 0 + 0) = 0$ (path 1-2-3-4-6).

The subsequential transducer allows one to find the set of positions of any factor but the complexity of this operation depends on the size of the text. There exists a representation of this subsequential transducer which allows one to obtain the set of positions of a given word $w$ in

[8] The input automaton associated with this transducer is minimal. A fortiori, the transducer is minimal in the sense defined by Mohri (1994b).

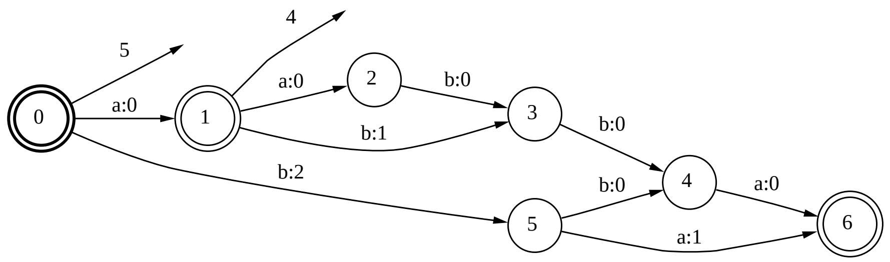


Fig. 10. Indexation by a subsequential transducer.

linear time $O(|w|)$. In that representation, input labels of transitions are strings. However, such a representation is not convenient when dealing with natural language processing. Indeed, typical use of this index would consists of searching for the existence and potential positions of a set of words described by a regular expression. The regular expression can also be represented by an automaton, but given the representation of the transducer its composition with the automaton is made complex and time consuming.

### 3.2 Indexation with automata

We have devised an algorithm for constructing an automaton giving the general index of a text, in which labels are alphabet letters. Using this automaton, once a given word is read, the set of its positions can be directly obtained from the list of integers associated with the reached state.

The following figure (figure 11) illustrates this representation. States are here provided with positions' lists. Each of these lists corresponds to the set of ending positions of any word reaching this state when read from the initial state. Hence, to obtain the set of positions of a string ending at state $q$, one just needs to subtract the length of this string from each integer of the list of $q$. For instance the positions' list of state 5 gives the set of positions of $b$: $\{3 - 1 = 2, 4 - 1 = 3\}$. Thus, such indexing automata allow one to obtain directly the set of positions of a word. The complexity of the operation is here clearly linear in the length of the searched word.

Our algorithm is close to that of Crochemore (1986). It is based one an equivalence relation $R$ over factors of a given text $t$. Two factors are equivalent according to $R$ iff they have the same contexts, namely if they have the same set of ending positions in $t$. Two equivalent factors correspond to the same state in this algorithm. Reading these strings from the initial state one reaches that state. At every step of the algorithm, a failure function $s$ is defined. For a given factor $u$ ending at the state $q$ when read from the initial state, $s[q]$ allows one to determine the set of suffixes of $u$ that are not equivalent to $u$. So those suffixes lead to states distinct from $q$. In order to construct the automaton we are interested in, one needs to indicate the ending positions of $u$ at the last state $q$ and also at all other states corresponding to suffixes not equivalent to $u$. This is the main idea the construction of the indexing automaton is based upon.

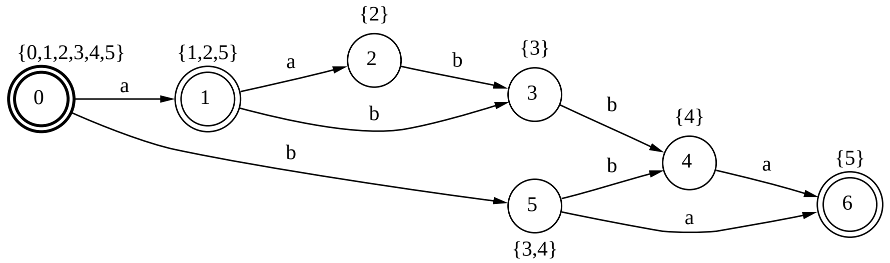


Fig. 11. Indexation by an automaton.

A complete description of the algorithm is given below. It describes the construction of a deterministic automaton $G = (V, i, F, A, \delta)$ indexing a text $t$, with:

- $V$ the set of states of $G$,
- $i \in V$ its initial state,
- $F \subseteq V$ its final states,
- $A$ the alphabet of the automaton,
- $\delta$ its transition function which maps $V \times A$ to $V$.

The function ADD_LISTP is used to add a new position to the list of a particular state. We have mainly used the same notations as those of (Crochemore 1986). In particular, the function $s$ corresponds to the definition recalled above and the function $l$ is such that $l[q]$ is the longest word leading from the initial state to the state $q$: $l[q] = max\{|u|/u \in A^*$ and $\delta(i, u) = q\}$.

The automaton constructed this way is the minimal automaton[9] recognizing the set of suffixes of a given text $t$ (Blumer et al. 1987). The complexity of this algorithm is quadratic, but it leads to a very efficient full indexation of a text. In practice, the construction is very fast. The average time spent to construct the automaton from a text of 50 Kb is about one second, using an HP/9000 755. The corresponding automaton has about 80,000 states. Such an automaton can be put in a very compact form, but one also needs to store the set of positions of the text. Experiments with natural language texts show that the lists of positions of the states of the automaton contain very few elements unless the state corresponds to a short word. Indeed, short words such as articles or prepositions have generally many occurrences in a text.

One can reduce the space necessary for the construction of the indexing automaton by storing a new position at a given state $q$ only if $l[q] > \alpha$ where $\alpha$ is a chosen parameter. Indeed, even if ending positions are not given at a state $q$, it is still possible to find them by examining every path starting at $q$.

[9] We do not need to define in this algorithm the final states, since the corresponding notion is no more necessary for the purpose of indexation.

```
AUTOMATON_INDEXATION(G, t)
begin
  create new state art; create new state init
  l[init] ← 0; s[init] ← art; p ← init
  for i ← 0 to length[t] − 1
    do  a ← t[i]
        create new state q
        l[q] ← l[p] + 1
        while p ≠ init and δ(p, a) undefined
          do    δ(p, a) ← q
                ADD_LISTP(p, i)
                p ← s[p]
        if (δ(p, a) undefined)
          then  δ(init, a) ← q
                s[q] ← init
        else if (l[p] + 1 = l[δ(p, a)])
          then  s[q] ← δ(p, a)
                while p ≠ init
                  do    ADD_LISTP(p, i)
                        p ← s[p]
          else  create copy r of δ(p, a) ▷ with same transitions and list of positions
                s[r] ← s[δ(p, a)]
                l[r] ← l[p] + 1
                s[q] ← s[δ(p, a)] ← r
                while p ≠ art and l[δ(p, a)] ≥ l[r]
                  do    δ(p, a) ← r
                        ADD_LISTP(p, i)
                        p ← s[p]
                if (p ≠ art)
                  then  while p ≠ init
                          do  ADD_LISTP(p, i)
                              p ← s[p]
        p ← q
  while p ≠ init
    do  ADD_LISTP(p, i)
        p ← s[p]
end.
```

Fig. 12. Algorithm for the indexation by an automaton $G$ of a text $t$.

The search can be stopped for each path when a state with a non empty list of positions is found. Subtracting the distance from $q$ to such a state from the positions indicated at this state gives the ending positions of the searched word. Suppose for instance that the list of positions of the state 5 were not available. Then considering all leaving paths from state 5 and reaching states with non empty lists, one would obtain the states 4 and 6, and the desired list of ending positions of $b$, $\{4-1=3, 5-1=4\}$. Experimental results show that this method allows one to reduce considerably the number of positions to store and that its cost in the search of the positions of a word is low enough to be negligible when $\alpha$ is chosen to be about 5.

Indexation of very large texts can also be performed this way by splitting the text into several smaller sections. The search for the positions can then be made by considering the automaton associated with each of these sections or equivalently by considering the union of the automata of all sections.

The results related to the indexation by automata still need to be improved considerably. They provide a very efficient indexation, but the amount of storage they demand is still too important compared to other indexing methods. However, the use of automata allows one to combine indexation with other transducers containing linguistic information to refine the search in a very flexible way.

The indexation automaton can for instance be composed with a morphological transducer. When searching for a canonical form one then directly obtains the positions associated with all the corresponding inflected forms. Similarly the indexation automaton can be composed with a finite-state grammar (weighted or not) and a morphological transducer to restrict the search. In all these cases, the result of the search is given as an automaton. Moreover, the composition can be made on-the-fly (Mohri et al. 1996).

## 4 Conclusion

We described new algorithms that can be used in various natural language processing applications to improve the space and time efficiency. The experiments we reported tend to confirm their usefulness in computational linguistics. We gave detailed indications about various constructions and mentioned the pseudocodes of our algorithms. They should help one to make those experiments without much difficulty.

The algorithms relating to sequential transducers (determinization and union) could be used in many other areas to improve the use of transducers when possible. The indexation algorithm suggests possible combinations with other finite-state tools to perform complex natural language indexation of corpora. The efficient and natural operations such as composition, and union of automata and transducers that it allows are the fundamental basis for syntactic pattern recognition in texts.

The algorithms that we introduced and described are the result of work in the theory of automata. We could not indicate here all the possibilities offered by that theory. It admits interesting applications in most areas of natural language processing and often provides a full and coherent picture of deep mechanisms with the appropriate level of abstraction. It also helps to make clearer the connections between natural language processing and other areas such as combinatorial pattern matching and computational biology.

## References


Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The design and analysis of computer algorithms.* Addison Wesley: Reading, MA.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers, Principles, Techniques and Tools.* Addison Wesley: Reading, MA.

A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnell. 1987. Complete inverted files for efficient text retrieval and analysis. *Journal of ACM*, 34.

Christian Choffrut. 1978. *Contributions à l'étude de quelques familles remarquables de fonctions rationnelles.* Ph.D. thesis, (thèse de doctorat d'Etat), Université Paris 7, LITP: Paris, France.

T. Cormen, C. Leiserson, and R. Rivest. 1992. *Introduction to Algorithms.* The MIT Press: Cambridge, MA.

Blandine Courtois. 1989. Delas: Dictionnaire électronique du LADL pour les mots simples du français. Technical report, LADL.

Maxime Crochemore. 1986. Transducers and repetitions. *Theoretical Computer Science*, 45.

Jacques Désarménien. 1986. La division par ordinateur des mots français: application à TEX. *Technique et Science Informatiques*, 5(4).

Maurice Gross. 1989. The use of finite automata in the lexical representation of natural language. *Lecture Notes in Computer Science*, 377.

Ronald M. Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3).

Gaby Karlsfeld. 1991. Dictionnaire morphologique de l'anglais. Technical report, LADL.

Lauri Karttunen, Ronald M. Kaplan, and Annie Zaenen. 1992. Two-level morphology with composition. In *Proceedings of the fifteenth International Conference on Computational Linguistics (COLING'92), Nantes, France.* COLING.

Lauri Karttunen. 1993. Finite-state lexicon compiler. Technical Report Xerox PARC P93-00077, Xerox PARC.

Kimmo Koskenniemi. 1985. Compilation of automata from morphological two-level rules. In *Proceedings of the Fifth Scandinavian Conference of Computational Linguistics, Helsinki, Finland.*

Eric Laporte. 1988. *Méthodes algorithmiques et lexicales de phonétisation de textes.* Ph.D. thesis, Université Paris 7: Paris, France.

Franklin Mark Liang. 1983. *Word Hy-phen-a-tion by Comput-er.* Ph.D. thesis, Stanford University, Stanford.

Mehryar Mohri and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *34th Meeting of the Association for Computational Linguistics (ACL 96), Proceedings of the Conference, Santa Cruz, California.* ACL.

Mehryar Mohri, Fernando C. N Pereira, and Michael Riley. 1996. Weighted automata in text and speech processing. In *ECAI-96 Workshop, Budapest, Hungary.* ECAI.

Mehryar Mohri. 1993. *Analyse et représentation par automates de structures syntaxiques composées.* Ph.D. thesis, Université Paris 7: Paris, France.

Mehryar Mohri. 1994a. Compact representations by finite-state transducers. In $32^{nd}$ *Meeting of the Association for Computational Linguistics (ACL 94), Proceedings of the Conference, Las Cruces, New Mexico.* ACL.

Mehryar Mohri. 1994b. Minimization of sequential transducers. *Lecture Notes in Computer Science*, 807.

Mehryar Mohri. 1994d. Syntactic analysis by local grammars automata: an efficient algori thm. In *Proceedings of the International Conference on Computational L exicography (COMPLEX 94)*. Linguistic Institute, Hungarian Academy of Science: Budapest, Hungary.

Mehryar Mohri. 1995. Matching patterns of an automaton. *Lecture Notes in Computer Science*, 937.

Fernando C. N. Pereira and Rebecca N. Wright. 1991. Finite-state approximation of phrase structure grammars. In 29*th Annual Meeting of the Association for Computational Lin guistics (ACL 94), Proceedings of the Conference, Berkeley, California*. ACL.

Fernando C. N Pereira, Michael Riley, and Richard Sproat. 1994. Weighted rational transductions and their application to human language processing. In *ARPA Workshop on Human Language Technology*. Advanced Research Projects Agency.

Dominique Perrin. 1990. Finite automata. In J. Van Leuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 1–57. Elsevier, Amsterdam.

Dominique Perrin. 1993. Les débuts de la théorie des automates. Technical Report LITP 93.04, LITP.

Dominique Revuz. 1991. *Dictionnaires et lexiques, méthodes et algorithmes*. Ph.D. thesis, Université Paris 7: Paris, France.

Emmanuel Roche. 1993a. *Analyse syntaxique transformationnelle du français par transducteur et lexique-grammaire*. Ph.D. thesis, Université Paris 7: Paris, France.

Emmanuel Roche. 1993b. Dictionary compression experiments. Technical Report IGM 93-5, Institut Gaspard Monge, Noisy-le-Grand.

Marcel Paul Schutzenberger and Christophe Reutenauer. 1991. Minimization of rational word functions. *SIAM Journal of Computing*, 20(4).

Marcel Paul Schutzenberger. 1987. Polynomial decomposition of rational functions. In *Lecture Notes in Computer Science*. Lecture Notes in Computer Science, Springer-Verlag: Berlin Heidelberg New York.

Max Silberztein. 1993. *Dictionnaires électroniques et analyse automatique de textes: le système INTEX*. Masson: Paris, France.

Richard Sproat. 1995. A finite-state architecture for tokenization and grapheme-to-phoneme conversion in multilingual text analysis. In *Proceedings of the ACL SIGDAT Workshop, Dublin, Ireland*. ACL.

W.A. Woods. 1970. Transition network grammars for natural language analysis. *Communications of the Association for the Computational Machinery*, 13(10).