MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

Deterministic Part-of-Speech Tagging with Finite State Transducers

Emmanuel Roche, Yves Schabes

TR94-07   December 1994

Abstract

Stochastic approaches to natural language processing have often been preferred to rule-based approaches because of their robustness and their automatic training capabilities. This was the case for part-of-speech tagging until Brill showed how state of the art part-of-speech tagging can be achieved by inferring a rule-based part-of-speech tagger from a training corpus. However current implementations of Brillś tagger run more slowly than previous approaches. In this paper, we present a finite-state tagger inspired by Brillś work which operates in optimal time in the sense that the time to assign tags to a sentence corresponds to the time required to deterministically follow a single path in a deterministic finite state machine. This result is achieved by encoding the application of the rules found in Brillś tagger as a non-deterministic finite state transducer and then turning it into a deterministic transducer. The resulting deterministic transducer yields a part-of-speech tagger whose speed is dominated by the access time of mass storage devices.

Computational Linguistics, March 1995

Revisions history.

1. Version 1.0, May 2nd 1994.
2. Version 1.1, June 16th 1994.
3. Version 1.2, June 22nd 1994.
4. Version 1.3, July 27th 1994.
5. Version 1.4, July 1994.
6. Version 2.0, December 9th 1994.
7. This version is Revision 3.0 of Date: 95/03 .

1 Introduction

Finite-state devices have important applications to many areas of computer science, including pattern matching, databases and compiler technology. Although their linguistic adequacy to natural language processing has been questioned in the past (Chomsky, 1964), there has recently been a dramatic renewal of interest in the application of finite-state devices to several aspects of natural language processing. This renewal of interest is due to the speed and the compactness of finite-state representations. This efficiency is explained by two properties: finite-state devices can be made deterministic, and they can be turned into a minimal form. Such representations have been successfully applied to different aspects of natural language processing, such as morphological analysis and generation (Karttunen, Kaplan, and Zaenen, 1992; Clemenceau, 1993), parsing (Roche, 1993; Tapanainen and Voutilainen, 1993), phonology (Laporte, 1993; Kaplan and Kay, 1994) and speech recognition (Pereira, Riley, and Sproat, 1994). Although finite-state machines have been used for part-of-speech tagging (Tapanainen and Voutilainen, 1993; Silberztein, 1993), none of these approaches has the same flexibility as stochastic techniques. Unlike stochastic approaches to part-of-speech tagging (Church, 1988; Kupiec, 1992; Cutting et al., 1992; Merialdo, 1990; DeRose, 1988; Weischedel et al., 1993), up to now the knowledge found in finite-state taggers has been handcrafted and cannot be automatically acquired.

Recently, Brill (1992) described a rule-based tagger which performs as well as taggers based upon probabilistic models and which overcomes the limitations common in rule-based approaches to language processing: it is robust and the rules are automatically acquired. In addition, the tagger requires drastically less space than stochastic taggers. However, current implementations of Brill's tagger are considerably slower than the ones based on probabilistic models since it may require RCn elementary steps to tag an input of n words with R rules requiring at most C tokens of context.

Although the speed of current part-of-speech taggers is acceptable for interactive systems where a sentence at a time is being processed, it is not adequate for applications where large bodies of text need to be tagged, such as in information retrieval, indexing applications and grammar checking systems. Furthermore, the space required for part-of-speech taggers is also an issue in commercial personal computer applications such as grammar checking systems. In addition, part-of-speech taggers are often being coupled with a syntactic analysis module. Usually these two modules are written in different frameworks, making it very difficult to integrate interactions between the two modules.

In this paper, we design a tagger that requires n steps to tag a sentence of length n, independent of the number of rules and the length of the context they require. The tagger is represented by a finite-state transducer, a framework which can also be the basis for syntactic analysis. This finite-state tagger will also be found useful combined with other language components since it can be naturally extended by composing it with finite-state transducers which could encode other aspects of natural language syntax.

Relying on algorithms and formal characterization described in later sections, we explain how each rule in Brill's tagger can be viewed as a non-deterministic finite-state transducer. We also show how the application of all rules in Brill's tagger is achieved by composing each of these non-deterministic transducers and why non-determinism arises in this transducer. We then prove the correctness of the general algorithm for determinizing (whenever possible) finite-state transducers and we successfully apply this algorithm to the previously obtained non-deterministic transducer. The resulting deterministic transducer yields a part-of-speech tagger which operates in optimal time in the sense that the time to assign tags to a sentence corresponds to the time required to follow a single path in this deterministic finite-state machine. We also show how the lexicon used by the tagger can be optimally encoded using a finite-state machine.

The techniques used for the construction of the finite-state tagger are then formalized and mathematically proven correct. We introduce a proof of soundness and completeness with a worst case complexity analysis for an algorithm for determinizing finite-state transducers.

We conclude by proving how the method can be applied to the class of transformation-based error-driven systems.

2 Overview of Brill's Tagger

Brill's tagger is comprised of three parts, each of which is inferred from a training corpus: a lexical tagger, an unknown word tagger and a contextual tagger. For purposes of exposition, we will postpone the discussion of the unknown word tagger and focus mainly on the contextual rule tagger, which is the core of the tagger.

The lexical tagger initially tags each word with its most likely tag, estimated by examining a large tagged corpus, without regard to context. For example, assuming that *vbn* is the most likely tag for the word “killed” and *vbd* for “shot”, the lexical tagger might assign the following part-of-speech tags:[1]

(1) Chapman/*np* killed/*vbn* John/*np* Lennon/*np*
(2) John/*np* Lennon/*np* was/*bedz* shot/*vbd* by/*by* Chapman/*np*
(3) He/*pps* witnessed/*vbd* Lennon/*np* killed/*vbn* by/*by* Chapman/*np*

Since the lexical tagger does not use any contextual information, many words can be tagged incorrectly. For example, in (1), the word “killed” is erroneously tagged as a verb in past participle form, and in (2), “shot” is incorrectly tagged as a verb in past tense.

Given the initial tagging obtained by the lexical tagger, the contextual tagger applies a sequence of rules in order and attempts to remedy the errors made by the initial tagging. For example, the rules in Figure 1 might be found in a contextual tagger.

1. *vbn vbd* PREVTAG *np*
2. *vbd vbn* NEXTTAG *by*

Figure 1: Sample rules

The first rule says to change tag *vbn* to *vbd* if the previous tag is *np*. The second rule says to change *vbd* to tag *vbn* if the next tag is *by*. Once the first rule is applied, the tag for “killed” in (1) and (3) is changed from *vbn* to *vbd* and the following tagged sentences are obtained:

(4) Chapman/*np* killed/*vbd* John/*np* Lennon/*np*

[1] The notation for part-of-speech tags is adapted from the one used in the Brown Corpus (Francis and Kučera, 1982): *pps* stands for third singular nominative pronoun, *vbd* for verb in past tense, *np* for proper noun, *vbn* for verb in past participle form, *by* for the word “by”, *at* for determiner, *nn* for singular noun and *bedz* for the word “was”.

(5) John/*np* Lennon/*np* was/*bedz* shot/*vbd* by/*by* Chapman/*np*
(6) He/*pps* witnessed/*vbd* Lennon/*np* killed/*vbd* by/*by* Chapman/*np*

And once the second rule is applied, the tag for "shot" in (5) is changed from *vbd* to *vbn*, resulting in (8) and the tag for "killed" in (6) is changed back from *vbd* to *vbn*, resulting in (9):

(7) Chapman/*np* killed/*vbd* John/*np* Lennon/*np*
(8) John/*np* Lennon/*np* was/*bedz* shot/*vbn* by/*by* Chapman/*np*
(9) He/*pps* witnessed/*vbd* Lennon/*np* killed/*vbn* by/*by* Chapman/*np*

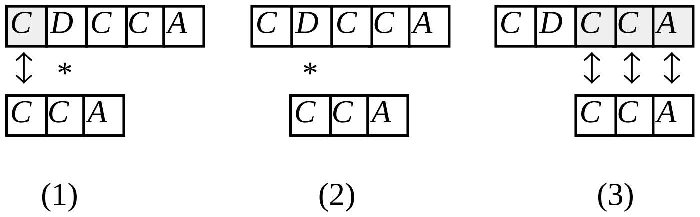
It is relevant to our following discussion to note that the application of the NEXTTAG rule must look ahead one token in the sentence before it can be applied and that the application of two rules may perform a series of operations resulting in no net change. As we will see in the next section, these two aspects are the source of local non-determinism in Brill's tagger.

The sequence of contextual rules is automatically inferred from a training corpus. A list of tagging errors (with their counts) is compiled by comparing the output of the lexical tagger to the correct part-of-speech assignment. Then, for each error, it is determined which instantiation of a set of rule templates results in the greatest error reduction. Then the set of new errors caused by applying the rule is computed and the process is repeated until the error reduction drops below a given threshold.

After training on the Brown Corpus, using the set of contextual rule templates shown in Figure 2, 280 contextual rules are obtained. The resulting rule-based tagger performs as well as the state-of-the-art taggers based upon probabilistic models. It also overcomes the limitations common in rule-based approaches to language processing: it is robust, and the rules are automatically acquired. In addition, the tagger requires drastically less space than stochastic taggers. However, as we will see in the next section, Brill's tagger is inherently slow.

3 Complexity of Brill's Tagger

Once the lexical assignment is performed, in Brill's algorithm, each contextual rule acquired during the training phase is applied to each sentence to be



A	B	PREVTAG	C	change A to B if previous tag is C
A	B	PREV1OR2OR3TAG	C	change A to B if previous one or two or three tag is C
A	B	PREV1OR2TAG	C	change A to B ifprevious one or two tag is C
A	B	NEXT1OR2TAG	C	change A to B if next one or two tag is C
A	B	NEXTTAG	C	change A to B if next tag is C
A	B	SURROUNDTAG	C D	change A to B if surrounding tags are C and D
A	B	NEXTBIGRAM	C D	change A to B if next bigram tag is C D
A	B	PREVBIGRAM	C D	change A to B if previous bigram tag is C D

Figure 2: Contextual Rule Templates

tagged. For each individual rule, the algorithm scans the input from left to right while attempting to match the rule.

This simple algorithm is computationally inefficient for two reasons. The first reason for inefficiency is the fact that an individual rule is matched at each token of the input, regardless of the fact that some of the current tokens may have been previously examined when matching the same rule at a previous position. The algorithm treats each rule as a template of tags and slides it along the input, one word at a time. Consider, for example, the rule *A B PREVBIGRAM C C* that changes tag A to tag B if the previous two tags are C.

Figure 3: Partial matches of *A B PREVBIGRAM C C* on the input C D C C A.

When applied to the input $CDCCA$, the pattern CCA is matched three times, as shown in Figure 3. At each step no record of previous partial matches or mismatches is remembered. In this example, C is compared with the second input token D during the first and second steps, and therefore, the second step could have been skipped by remembering the comparisons from the first step. This method is similar to a naive pattern matching algorithm.

The second reason for inefficiency is the potential interaction between rules. For example, when the rules in Figure 1 are applied to sentence (3), the first rule results in a change (6) which is undone by the second rule as shown in (9). The algorithm may therefore perform unnecessary computation.

In summary, Brill's algorithm for implementing the contextual tagger may require RCn elementary steps to tag an input of n words with R contextual rules requiring at most C tokens of context.

4 Construction of the Finite-State Tagger

We show how the function represented by each contextual rule can be represented as a non-deterministic finite-state transducer and how the sequential application of each contextual rule also corresponds to a non-deterministic finite-state transducer being the result of the composition of each individual transducer. We will then turn the non-deterministic transducer into a deterministic transducer. The resulting part-of-speech tagger operates in linear time independent of the number of rules and the length of the context. The new tagger operates in optimal time in the sense that the time to assign tags to a sentence corresponds to the time required to follow a single path in the resulting deterministic finite-state machine.

Our work relies on two central notions: the notion of a finite-state transducer and the notion of a subsequential transducer. Informally speaking, a finite-state transducer is a finite-state automaton whose transitions are labeled by pairs of symbols. The first symbol is the input and the second is the output. Applying a finite-state transducer to an input consists of following a path according to the input symbols while storing the output symbols, the result being the sequence of output symbols stored. Section 8.1 formally defines the notion of transducer.

Finite-state transducers can be composed, intersected, merged with the union operation and sometimes determinized. Basically, one can manipulate finite-state transducers as easily as finite-state automata. However, whereas every finite-state automaton is equivalent to some deterministic finite-state automaton, there are finite-state transducers that are not equivalent to any deterministic finite-state transducer. Transductions that can be computed by some deterministic finite-state transducer are called *subsequential functions*. We will see that the final step of the compilation of our tagger consists of transforming a finite-state transducer into an equivalent subsequential transducer.

We will use the following notation when pictorially describing a finite-state transducer: final states are depicted with two concentric circles; ϵ represents the empty string; on a transition from state i to state j, a/b indicates a transition on input symbol a and output symbol(s) b;[2] a question mark (?) on an arc transition (for example labeled $?/b$) originating at state i stands for any input symbol that does not appear as an input symbol on any other outgoing arc from i. In this document, each depicted finite-state transducer will be assumed to have a single initial state, namely the leftmost state appearing in the figures (usually labeled 0).

We are now ready to construct the tagger. Given a set of rules, the tagger is constructed in four steps.

The first step consists of turning each contextual rule found in Brill's tagger into a finite-state transducer. Following the example discussed in Section 2, the functionality of the rule *vbn vbd PREVTAG np* is represented by the transducer shown in Figure 4 on the left.

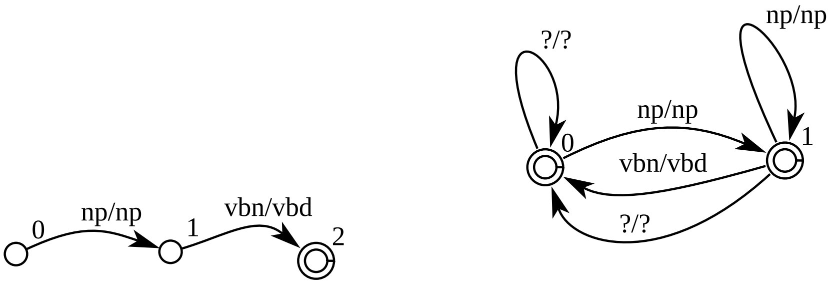


Figure 4: *Left:* transducer T_1 representing the contextual rule *vbn vbd PREVTAG np*. *Right:* local extension $LocExt(T_1)$ of T_1

[2] When multiple output symbols are emitted, a comma symbolizes the concatenation of the output symbols.

Each of the contextual rules is defined locally; that is, the transformation it describes must be applied at each position of the input sequence. For instance, the rule *A B PREV1OR2TAG C*, that changes A into B if the previous tag or the one before is C, must be applied twice on $C\ A\ A$ (resulting in the output $C\ B\ B$). As we have seen in the previous section, this method is not efficient.

The second step consists of turning the transducers produced by the preceding step into transducers that operate globally on the input in one pass. This transformation is performed for each transducer associated with each rule. Given a function f_1 that transforms, say, a into b (i.e. $f_1(a) = b$), we want to extend it to a function f_2 such that $f_2(w) = w'$ where w' is the word built from the word w where each occurrence of a has been replaced by b. We say that f_2 is the *local extension*[3] of f_1 and we write $f_2 = LocExt(f_1)$. Section 8.2 formally defines this notion and gives an algorithm for computing the local extension.

Referring to the example of Section 2, the local extension of the transducer for the rule *vbn vbd PREVTAG np* is shown to the right of Figure 4. Similarly, the transducer for the contextual rule *vbd vbn NEXTTAG by* and its local extension are shown in Figure 5.

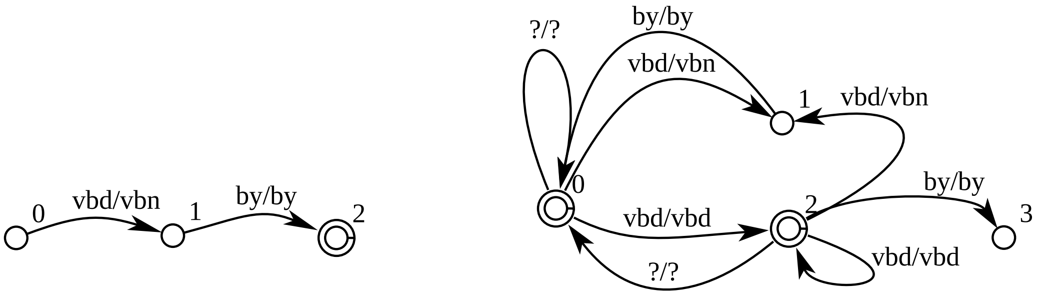


Figure 5: *Left:* transducer T_2 representing *vbd vbn NEXTTAG by*. *Right:* local extension $LocExt(T_2)$ of T_2

The transducers obtained in the previous step still need to be applied one after the other. The third step combines all transducers into one single

[3] This notion was introduced by Roche (1993).

transducer. This corresponds to the formal operation of composition defined on transducers. The formalization of this notion and an algorithm for computing the composed transducer are well-known and are described originally by Elgot and Mezei (1965).

Returning to our running example of Section 2, the transducer obtained by composing the local extension of T_2 (right in Figure 5) with the local extension of T_1 (right in Figure 4) is shown in Figure 6.

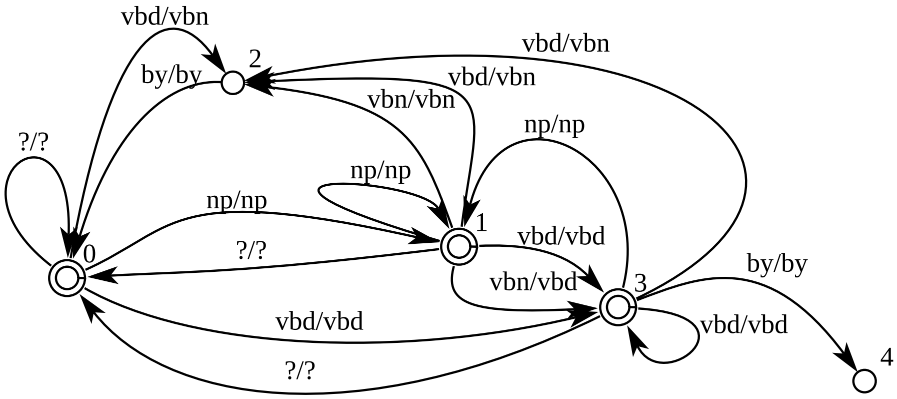


Figure 6: Composition $T_3 = LocExt(T_1) \circ LocExt(T_2)$

The fourth and final step consists of transforming the finite-state transducer obtained in the previous step into an equivalent subsequential (deterministic) transducer. The transducer obtained in the previous step may contain some non-determinism. The fourth step tries to turn it into a deterministic machine. This determinization is not always possible for any given finite-state transducer. For example, the transducer shown in Figure 7 is not equivalent to any subsequential transducer. Intuitively speaking, such a transducer has to look ahead an unbounded distance in order to correctly generate the output. This intuition will be formalized in Section 9.2.

However, as proven in Section 10, the rules inferred in Brill's tagger can always be turned into a deterministic machine. Section 9.1 describes an algorithm for determinizing finite-state transducers. This algorithm will not terminate when applied to transducers representing non-subsequential functions.

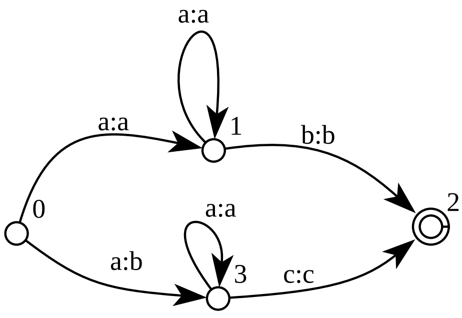


Figure 7: Example of a transducer not equivalent to any subsequential transducer.

In our running example, the transducer in Figure 6 has some non-deterministic paths. For example, from state 0 on input symbol *vbd*, two possible emissions are possible: *vbn* (from 0 to 2) and *vbd* (from 0 to 3). This non-determinism is due to the rule *vbd vbd NEXTTAG by*, since this rule has to read the second symbol before it can know which symbol must be emitted. The deterministic version of the transducer T_3 is shown in Figure 8. Whenever non-determinism arises in T_3, the deterministic machine emits the empty symbol ϵ, and postpones the emission of the output symbol. For example, from the start state 0, the empty string is emitted on input *vbd*, while the current state is set to 2. If the following word is *by*, the two token string *vbn by* is emitted (from 2 to 0), otherwise *vbd* is emitted (depending on the input from 2 to 2 or from 2 to 0).

Using an appropriate implementation for finite-state transducers (see Section 11), the resulting part-of-speech tagger operates in linear time, independent of the number of rules and the length of the context. The new tagger therefore operates in optimal time.

We have shown how the contextual rules can be implemented very efficiently. We now turn our attention to lexical assignment, the step that precedes the application of the contextual transducer. This step can also be made very efficient.

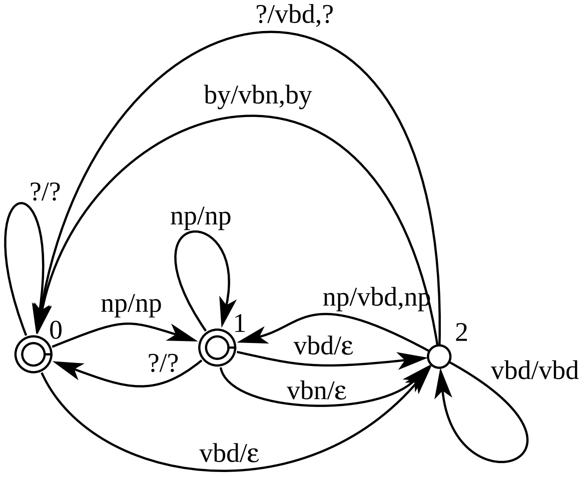


Figure 8: Subsequential form for T_3

5 Lexical Tagger

The first step of the tagging process consists of looking up each word in a dictionary. Since the dictionary is the largest part of the tagger in terms of space, a compact representation is crucial. Moreover, the lookup process has to be very fast too, otherwise the improvement in speed of the contextual manipulations would be of little practical interest.

To achieve high speed for this procedure, the dictionary is represented by a deterministic finite-state automaton with both low access time and small storage space. Suppose one wants to encode the sample dictionary of Figure 9. The algorithm, as described in (Revuz, 1991), consists of first building a tree whose branches are labeled by letters and whose leaves are labeled by a list of tags (such as *nn vb*) , and then minimizing it into a directed acyclic graph (DAG). The result of applying this procedure to the sample dictionary of Figure 9 is the DAG of Figure 10. When a dictionary is represented as a DAG, looking up a word in it consists simply of following one path in the DAG. The complexity of the lookup procedure depends only on the length of the word; in particular, it is independent of the size of the dictionary.

ads	*nns*
bag	*nn vb*
bagged	*vbn vbd*
bayed	*vbn vbd*
bids	*nns*

Figure 9: Sample Dictionary

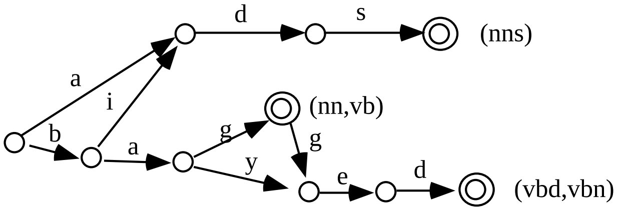


Figure 10: DAG representation of the dictionary found in Figure 9.

The lexicon used in our system encodes 54,000 words. The corresponding DAG takes 360 Kbytes of space and it provides an access time of 12,000 words per second.[4]

6 Tagging unknown words

The rule-based system described by Brill (1992) contains a module that operates after all the known words — that is, words listed in the dictionary — have been tagged with their most frequent tag, and before the set of contextual rules are applied. This module guesses a tag for a word according to its suffix (e.g. a word with an "ing" suffix is likely to be a verb), its prefix (e.g. a word starting with an uppercase character is likely to be a proper noun) and other relevant properties.

This module basically follows the same techniques as the ones used to implement the lexicon. Due to the similarity of the methods used, we do not provide further details about this module.

[4] The size of the dictionary in ASCII form is 742KB.

7 Empirical Evaluation

The tagger we constructed has an accuracy identical[5] to Brill's tagger or the one of statistical-based methods, however it runs at a much higher speed. The tagger runs nearly ten times faster than the fastest of the other systems. Moreover, the finite-state tagger inherits from the rule-based system its compactness compared to a stochastic tagger. In fact, whereas stochastic taggers have to store word-tag, bigram and trigram probabilities, the rule-based tagger and therefore the finite-state one only have to encode a small number of rules (between 200 and 300).

We empirically compared our tagger with Eric Brill's implementation of his tagger, and with our implementation of a trigram tagger adapted from the work of Church (1988) that we previously implemented for another purpose. We ran the three programs on large files and piped their output into a file. In the times reported, we included the time spent reading the input and writing the output. Figure 11 summarizes the results. All taggers were trained on a portion of the Brown corpus. The experiments were run on an HP720 with 32Mbytes of memory. In order to conduct a fair comparison, the dictionary lookup part of the stochastic tagger has also been implemented using the techniques described in Section 5. All three taggers have approximately the same precision (95% of the tags are correct)[6]. By design, the finite-state tagger produces the same output as the rule-based tagger. The rule-based tagger — and the finite-state tagger — do not always produce the exact same tagging as the stochastic tagger (they don't make the same errors); however, no significant difference in performance between the systems was detected.[7]

Independently, Cutting et al. (1992) quote a performance of 800 words/second for their part-of-speech tagger based on hidden Markov models.

The space required by the finite-state tagger (815KB) is decomposed as follows: 363KB for the dictionary, 440KB for the subsequential transducer and 12KB for the module for unknown words.

[5] Our current implementation is functionally equivalent to the tagger as described by Brill (1992). However, the tagger could be extended to include recent improvements described in more recent papers (Brill, 1994).

[6] For evaluation purposes, we randomly selected 90% of the Brown corpus for training purposes and 10% for testing. We used the Brown corpus set of part-of-speech tags.

[7] An extended discussion of the precision of the rule-based tagger can be found in (Brill, 1992).

	Stochastic Tagger	Rule-Based Tagger	Finite-State Tagger
Speed	1,200 w/s	500 w/s	10,800 w/s
Space	2,158KB	379KB	815KB

Figure 11: Overall performance comparison.

The speed of our system is decomposed in Figure 12.[8]

	dictionary lookup	unknown words	contextual
Speed	12,800 w/s	16,600 w/s	125,100 w/s
Percent of the time	85%	6.5%	8.5%

Figure 12: Speed of the different parts of the program

Our system reaches a performance level in speed for which other very low level factors (such as storage access) may dominate the computation. At such speeds, the time spent reading the input file, breaking the file into sentences, and sentences into words, and writing the result into a file is no longer negligible.

8 Finite-State Transducers

The methods used in the construction of the finite-state tagger described in the previous sections were described informally. In the following section, the notions of finite-state transducers and the notion of local extension are defined. We also provide an algorithm for computing the local extension of a finite-state transducer. Issues related to the determinization of finite-state transducers are discussed in the section following this one.

[8] In Figure 12, the dictionary lookup includes reading the file, splitting it into sentences, looking up each word in the dictionary and writing the final result to a file. The dictionary lookup and the tagging of unknown words take roughly the same amount of time, but since the second procedure only applies on unknown words (around 10% in our experiments) the percentage of time it takes is much smaller.

8.1 Definition of Finite-State Transducers

A *finite-State transducer* T is a 5-tuple (Σ, Q, i, F, E) where: Σ is a finite alphabet; Q is the set of states or vertices; $i \in Q$ is the initial state; $F \subseteq Q$ is the set of final states; $E \subseteq Q \times \Sigma \cup \{\epsilon\} \times \Sigma^* \times Q$ is the set of edges or transitions.

For instance, Figure 13 is the graphical representation of the transducer:

$$T_1 = (\{a, b, c, d, e\}, \{0, 1, 2, 3\}, 0, \{3\}, \{(0, a, b, 1), (0, a, c, 2), (1, d, d, 3), (2, e, e, 3)\}).$$

A finite-state transducer T also defines a function on words in the following way: the extended set of edges $\hat{E}$, the transitive closure of E, is defined by the following recursive relation:

- if $e \in E$ then $e \in \hat{E}$
- if $(q, a, b, q'), (q', a', b', q'') \in \hat{E}$ then $(q, aa', bb', q'') \in \hat{E}$.

Then the *function* f from Σ^* to Σ^* defined by $f(w) = w'$ iff $\exists q \in F$ such that $(i, w, w', q) \in \hat{E}$ is the function defined by T. One says that T represents f and writes $f = |T|$. The functions on words that are represented by finite-state transducers are called *rational functions*. If, for some input w, more than one output is allowed (e.g. $f(w) = \{w_1, w_2, \cdots\}$) then f is called a *rational transduction*.

In the example of Figure 13, T_1 is defined by $|T_1|(ad) = bd$ and $|T_1|(ae) = ce$.

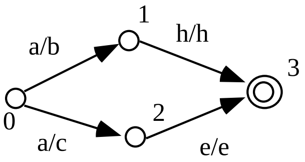


Figure 13: T_1: Example of Finite-State Transducer

Given a finite-state transducer $T = (\Sigma, Q, i, F, E)$, the following additional notions are useful: its state *transition function* d that maps $Q \times \Sigma \cup \{\epsilon\}$ into 2^Q defined by $d(q, ag) = \{q' \in Q | \exists w' \in \Sigma^*$ and $(q, a, w', q') \in E\}$; and its *emission function* δ that maps $Q \times \Sigma \cup \{\epsilon\} \times Q$ into 2^{Σ^*} defined by $\delta(q, a, q') = \{w' \in \Sigma^* | (q, a, w,', q') \in E\}$.

A finite-state transducer could be seen as a finite-state automaton, each of whose label is a pair. In this respect, T_1 would be deterministic; however, since transducers are generally used to compute a function, a more relevant definition of determinism consists of saying that both the transition function d and the emission function δ lead to sets containing at most one element, that is, $|d(q,a)| \leq 1$ and $|\delta(q,a,q')| \leq 1$. With this notion, if a finite-state transducer is deterministic, one can apply the function to a given word by deterministically following a single path in the transducer. Deterministic transducers are called *subsequential transducers* (Schűtzenberger, 1977)[9]. Given a deterministic transducer, we can define the partial functions $q \otimes a = q'$ iff $d(q,a) = \{q'\}$ and $q * a = w'$ iff $\exists q' \in Q$ such that $q \otimes a = q'$ and $\delta(q,a,q') = \{w'\}$. This leads to the definition of *subsequential transducers*: a subsequential transducer T' is a 7-tuple $(\Sigma, Q, i, F, \otimes, *, \rho)$ where: Σ, Q, i, F are defined as above; $\otimes$ is the deterministic state transition function that maps $Q \times \Sigma$ on Q, one writes $q \otimes a = q'$; $*$ is the deterministic emission function that maps $Q \times \Sigma$ on Σ^*, one writes $q * a = w'$; and the final emission function ρ maps F on Σ^*, one writes $\rho(q) = w$.

For instance, T_1 is not deterministic because $d(1,a) = \{a,b\}$, but it is equivalent to T_2 represented Figure 14 in the sense that they represent the same function, i.e $|T_1| = |T_2|$. T_2 is defined by $T_2 = (\{a,b,c,h,e\}, \{0,1,2\}, 0, \{2\}, \otimes, *, \rho)$ where $0 \otimes a = 1$, $0 * a = \epsilon$, $1 \otimes h = 2$, $1 * h = bh$, $1 \otimes e = 2$, $1 * e = ce$ and where $\rho(2) = \epsilon$.

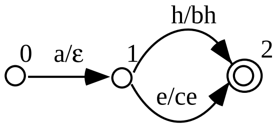


Figure 14: Subsequential Transducer T_2

8.2 Local Extension

In this section, we will see how a function which needs to be applied at all input positions can be transformed into a global function that needs to be applied once on the input. For instance, consider T_3 of Figure 15. It

[9] A *sequential transducer* is a deterministic transducer for which all states are final. Sequential transducers are also called *generalized sequential machines* (Eilenberg, 1974).

represents the function $f_3 = |T_3|$ such that $f_3(ab) = bc$ and $f_3(bca) = dca$. We want to build the function that, given a word w, each time w contains ab (i.e. ab is a factor of the word) (resp. bca), this factor is transformed into its image bc (resp. dca). Suppose for instance that the input word is $w = aabcab$, as shown on Figure 16, and that the factors that are in $dom(f_3)$[10] can be found according to two different factorizations: i.e. $w_1 = a \cdot w_2 \cdot c \cdot w_2$[11] where $w_2 = ab$ and $w_1 = aa \cdot w_3 \cdot b$ where $w_3 = bca$. The *local extension* of f_3 will be the function that takes each possible factorization and transforms each factor according to f_3, i.e. $f_3(w_2) = bc$ and $f_3(w_3) = dca$, and leaves the other parts unchanged; here this leads to two outputs: $abccbc$ according to the first factorization, and $aadcab$ according to the second factorization.

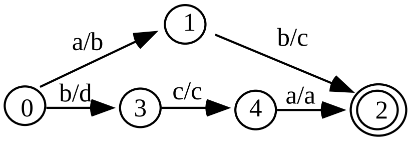


Figure 15: T_3: a finite-state transducer to be extended

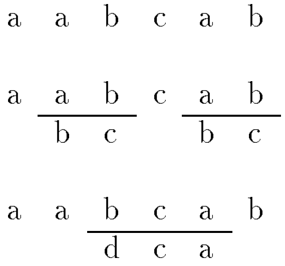


Figure 16: *Top:* input *Middle*: first factorization *Bottom*: second factorization

The notion of local extension is formalized through the following definition.

[10] $dom(f)$ denotes the *domain* of f, that is, the set of words that have at least one output through f.

[11] If $w_1, w_2 \in \Sigma^*$, $w_1 \cdot w_2$ denotes the concatenation of w_1 and w_2. It can also be written $w_1 w_2$.

Definition 1 *If f is a rational transduction from Σ^* to Σ^*, the* local extension *$F = LocExt(f)$ is the rational transduction from Σ^* on Σ^* defined in the following way: if $u = a_1b_1a_2b_2 \cdots a_nb_na_{n+1} \in \Sigma^*$ then $v = a_1b'_1a_2b'_2 \cdots a_nb'_na_{n+1} \in F(u)$ if $a_i \in \Sigma^* - (\Sigma^* \cdot dom(f) \cdot \Sigma^*)$, $b_i \in dom(f)$ and $b'_i \in f(b_i)$.*[12]

Intuitively, if $F = LocExt(f)$ and $w \in \Sigma^*$, each factor of w in $dom(f)$ is transformed into its image by f and the remaining part of w is left unchanged. If f is represented by a finite-state transducer T and $LocExt(f)$ is represented by a finite-state transducer T', one writes $T' = LocExt(T)$.

It could also be seen that if γ_T is the identity function on $\Sigma^* - (\Sigma^* \cdot dom(T) \cdot \Sigma^*)$, then $LocExt(T) = \gamma_T \cdot (T \cdot \gamma_T)^*$.[13] Figure 20 gives an algorithm that computes the local extension directly.

The idea is that an input word is processed non-deterministically from left to right. Suppose for instance that we have the initial transducer T_4 of Figure 17 and that we want to build its local extension T_5 of Figure 18. When the input is read, if a current input letter cannot be transformed at the first state of T_4 (the letter c for instance), it is left unchanged: this is expressed by the looping transition on the initial state 0 of T_5 labeled ?/?.[14] On the other hand, if the input symbol, say a, can be processed at the initial state of T_4, one doesn't know yet whether a will be the beginning of a word that can be transformed (e.g. ab) or whether it will be followed by a sequence which makes it impossible to apply the transformation (e.g. ac). Hence one has to entertain two possibilities, namely (1) we are processing the input according to T_4 and the transitions should be a/b, or (2) we are within the identity and the transition should be a/a. This leads to two kind of states: the transduction states (marked *transduction* in the algorithm) and the identity states (marked *identity* in the algorithm). It can be seen in Figure 18 that this leads to a transducer that has a copy of the initial transducer and an additional part that processes the identity while making sure it could not

[12] The dot '$\cdot$' stands for the concatenation operation on strings.

[13] In this last formula, the concatenation $\cdot$ stands for the concatenation of the graph of the function, that is for the concatenation of the transducers viewed as automata whose labels are of the form a/b.

[14] As explained before, a transition labeled by the symbol ? stands for all the transitions labeled with a letter that doesn't appear on any outgoing arc from this state. A transition labeled ?/? stands for all the diagonal pairs (a, a) s.t. a is not an input symbol on any outgoing arc from this state.

have been transformed. In other words, the algorithm consists of building a copy of the original transducer and at the same time the identity function that operates on $\Sigma^* - \Sigma^* \cdot dom(T) \cdot \Sigma^*$.

Let us now see how the algorithm of Figure 20 applies step by step to the transducer T_4 of Figure 17, producing the transducer T_5 of Figure 18.

In Figure 20, $C'[0] = (\{i\}, identity)$ of line 1 states that the state 0 of the transducer to be built is of type *identity* and refers to the initial state $i = 0$ of T_4. q represents the current state and n the current number of states. In the loop $do\{\cdots\}while(q < n)$, one builds the transitions of each state one after the other: if the transition points to a state not already built, a new state is added, thus incrementing n. The program stops when all states have been inspected and when no additional state is created. The number of iterations is bounded by $2^{\|T\|*2}$, where $\|T\| = |Q|$ is the number of states of the original transducer[15]. Line 3 says that the current state within the loop will be q and that this state refers to the set of states S and is marked by the type *type*. In our example, at the first occurrence of this line, S is instantiated to $\{0\}$ and $type = identity$. Line 5 adds the current identity state to the set of final states and a transition to the initial state for all letters that do not appear on any outgoing arc from this state. Lines 6 to 11 build the transitions from and to the identity states, keeping track of where this leads in the original transducer. For instance, a is a label that verifies the conditions of line 6. Thus a transition a/a is to be added to the *identity* state 2 which refers to 1 (because of the transition a/b of T_4) and to $i = 0$ (because it is possible to start the transduction T_4 from any place of the identity). Line 7 checks that this state doesn't already exist and adds it if necessary. $e = n++$ means that the arrival state for this transition, i.e. $d(q, w)$, will be the last added state and that the number of states being built has to be incremented. Line 11 actually builds the transition between 0 and $e = 2$ labeled a/a. Line 12 through 17 describe the fact that it is possible to start a transduction from any *identity* state. Here one transition is added to one new state, i.e. a/b to 3. The next state to be considered is 2 and it is built like state 0 except that the symbol b should block the current output. In fact, the state 1 means that we already read a with a as output, thus if one reads b, this means that ab is at the current point, and since ab should be transformed into bc, the current identity transformation (that is $a \rightarrow a$) should be blocked: this is expressed

[15] In fact, $Q' \subset 2^{Q \times \{transduction, identity\}}$. Thus, $q \leq 2^{2|Q|}$.

by the transition b/b that leads to state 1 (this state is a “trash” state; that is, it has no outgoing transition and it is not final).

The following state is 3, which is marked as being of type *transduction*, which means that lines 19 through 27 should be applied. This consists simply of copying the transitions of the original transducer. If the original state was final, as for $4 = (\{2\}, transduction)$, an ϵ/ϵ transition to the original state is added (to get the behavior of T^+).

The transducer $T_6 = LocExt(T_3)$ of Figure 19 gives a more complete (and slightly more complex) example of applying this algorithm.

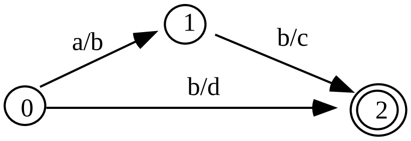


Figure 17: Sample Transducer T_4

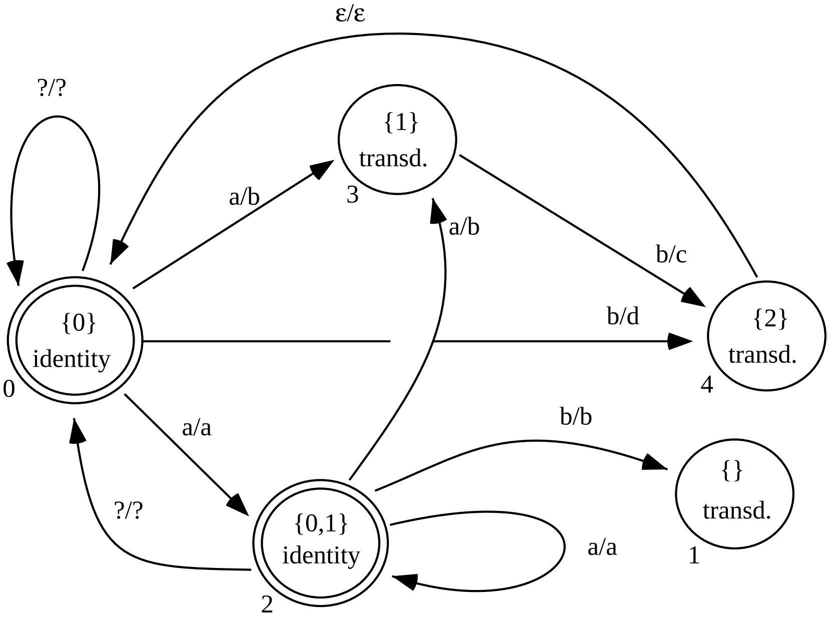


Figure 18: Local Extension T_5 of T_4: $T_5 = LocExt(T_4)$

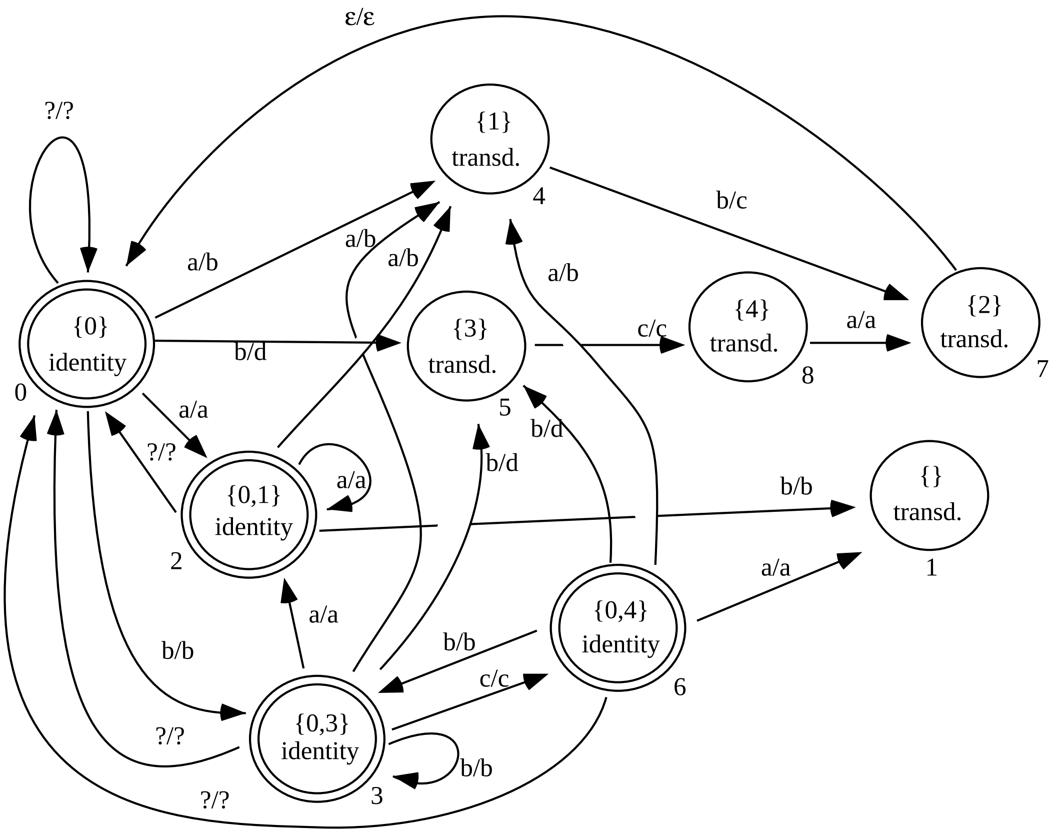


Figure 19: Local Extension T_6 of T_3: $T_6 = LocExt(T_3)$

```
$LocalExtension(T' = (\Sigma, Q', i', F', E'), T = (\Sigma, Q, i, F, E))$
  $C'[0] = (\{i\}, identity); q = 0; i' = 0; F' = \emptyset; E' = \emptyset; Q' = \emptyset; C'[1] = (\emptyset, transduction); n = 2;$
  do {
      $(S, type) = C'[q]; Q' = Q' \cup \{q\};$
      if $(type == identity)$
          $F' = F' \cup \{q\}; E' = E' \cup \{(q, ?, ?, i')\};$
          for each $w \in \Sigma \cup \{\epsilon\}$ s.t. $\exists x \in S,\ d(x, w) \neq \emptyset$ and $\forall y \in S,\ d(y, w) \cap F = \emptyset$
              if $(\exists r \in [0, n-1]$ such that $C'[r] == (\{i\} \cup \bigcup_{x \in S} d(x, w), identity)$
                $e = r;$
              else
                $C'[e = n++] = (\{i\} \cup \bigcup_{x \in S} d(x, w), identity);$
              $E' = E' \cup \{(q, w, w, e)\};$
          for each $(i, w, w', x) \in E$
              if $(\exists r \in [0, n-1]$ such that $C'[r] == (\{x\}, transduction)$
                $e = r;$
              else
                $C'[e = n++] = (\{x\}, transduction);$
              $E' = E' \cup \{(q, w, w', e)\};$
          for each $w \in \Sigma \cup \{\epsilon\}$ s.t. $\exists x \in S\ d(x, w) \cap F \neq \emptyset$ then $E' = E' \cup \{(q, w, w, 1)\};$
      else if $(type == transduction)$
          if $\exists x_1 \in Q$ s.t. $S == \{x_1\}$
              if $(x_1 \in F)$ then $E' = E' \cup \{(q, \epsilon, \epsilon, 0)\};$
              for each $(x_1, w, w', y) \in E$
                if $(\exists r \in [0, n-1]$ such that $C'[r] == (\{y\}, transduction)$
                  $e = r;$
                else
                  $C'[e = n++] = (\{y\}, transduction);$
                $E' = E' \cup \{(q, w, w', e)\};$
      q++;
  }while$(q < n);$
```

Figure 20: Local Extension Algorithm.

9 Determinization

The basic idea behind the determinization algorithm comes from Mehryar Mohri[16]. In this section, after giving a formalization of the algorithm, we introduce a proof of soundness and completeness with its worst case complexity analysis.

9.1 Determinization Algorithm

In the following, for $w_1, w_2 \in \Sigma^*$, $w_1 \wedge w_2$ denotes the longest common prefix of w_1 and w_2.

The finite-state transducers we use in our system have the property that they can be made deterministic; that is, there exists a subsequential transducer that represents the same function[17]. If $T = (\Sigma, Q, i, F, E)$ is such a finite-state transducer, the subsequential transducer $T' = (\Sigma, Q', i', F', \otimes, *, \rho)$ defined as follows will be later proved equivalent to T:

- $Q' \subset 2^{Q \times \Sigma^*}$. In fact the determinization of the transducer is related to the determinization of FSAs in the sense that it also involves a power set construction. The difference is that one has to keep track of the set of states of the original transducer one might be in and also of the words whose emission have been postponed. For instance, a state $\{(q_1, w_1), (q_2, w_2)\}$ means that this state corresponds to a path that leads to q_1 and q_2 in the original transducer and that the emission of w_1 (resp. w_2) was delayed for q_1 (resp. q_2).
- $i' = \{(i, \epsilon)\}$. There is no postponed emission at the initial state.
- the emission function is defined by:

  $$S * a = \bigwedge_{(q,u)\in S} \bigwedge_{q' \in d(q,a)} u \cdot \delta(q, a, q')$$

  This means that, for a given symbol, the set of possible emissions is obtained by concatenating the postponed emissions with the emission

[16] Mohri (1994b) also gives a formalization of the algorithm.

[17] As opposed to automata, a large class of finite-state transducers, don't have any deterministic representation; they can't be determinized.

at the current state. Since one wants the transition to be deterministic, the actual emission is the longest common prefix of this set.

- the state transition function is defined by:

$$S \otimes a = \bigcup_{(q,u)\in S} \bigcup_{q'\in d(q,a)} \{(q', (S * a)^{-1} \cdot u \cdot \delta(q, a, q'))\}$$

  Given $u, v \in \Sigma^*$, $u \cdot v$ denotes the concatenation of u and v and $u^{-1} \cdot v = w$, if w is such that $u \cdot w = v$, $u^{-1} \cdot v = \emptyset$ if no such w exists.

- $F' = \{S \in Q' | \exists (q, u) \in S \text{ and } q \in F\}$

- if $S \in F'$, $\rho(S) = u$ s.t. $\exists q \in F$ s.t. $(q, u) \in S$. We will see in the proof of correctness that ρ is properly defined.

The determinization algorithm of Figure 22 computes the above subsequential transducer.

Let us now apply the determinization algorithm of Figure 22 on the finite-state transducer T_1 of Figure 13 and show how it builds the subsequential transducer T_5 of Figure 21. Line 1 of the algorithm builds the first state and instantiates it with the pair $\{(0, \epsilon)\}$. q and n respectively denote the current state and the number of states having been built so far. At line 5, one takes all the possible input symbols w; here only a is possible. w' of line 6 is the output symbol, $w' = \epsilon \cdot (\bigwedge_{\overline{q}' \in \{1,2\}} \delta(0, a, \overline{q}'))$, thus $w' = \delta(0, a, 1) \wedge \delta(0, a, 2) = b \wedge c = \epsilon$. Line 8 is then computed as follows: $S' = \bigcup_{\overline{q} \in \{0\}} \bigcup_{\overline{q}' \in \{1,2\}} \{\overline{q}', \epsilon^{-1} \cdot \delta(0, a, \overline{q}')\}$, thus $S' = \{(1, \delta(0, a, 1))\} \cup \{(2, \delta(0, a, 2)\} = \{(1, b), (2, c)\}$. Since no r verifies the condition on line 9, a new state e is created to which the transition labeled $a/w = a/\epsilon$ points and n is incremented. On line 15, the program goes to the construction of the transitions of state 1. On line 5, d and e are then two possible symbols. The first symbol, h, at line 6, is such that w' is $w' = \bigwedge_{\overline{q}' \in d(1,h) = \{2\}} b \cdot \delta(1, h, \overline{q}')) = bh$. Henceforth, the computation of line 8 leads to $S' = \bigcup_{\overline{q} \in \{1\}} \bigcup_{\overline{q}' \in \{2\}} \{(\overline{q}', (bh)^{-1} \cdot b \cdot h)\} = \{(2, \epsilon)\}$. State 2 labeled $\{(2, \epsilon)\}$ is thus added and a transition labeled h/bh that points to state 2 is also added. The transition for the input symbol e is computed the same way.

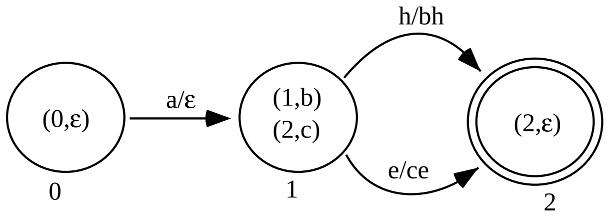


Figure 21: Subsequential transducer T_5 such that $|T_5| = |T_1|$

```
$DeterminizeTransducer(T' = (\Sigma, Q', i', F', \otimes, *, \rho), T = (\Sigma, Q, i, F, E))$
    $i' = 0; q = 0; n = 1; C'[0] = \{(0, \epsilon)\}; F' = \emptyset; Q' = \emptyset;$
    do {
        $S = C'[q]; Q' = Q' \cup \{q\};$
        if $\exists(\overline{q}, u) \in S$ s.t. $\overline{q} \in F$ then $F' = F' \cup \{q\}$ and $\rho(q) = u$;
        foreach $w$ such that $\exists(\overline{q}, u) \in S)$ and $d(\overline{q}, w) \neq \emptyset$ {
                $w' = \bigwedge_{(\overline{q},u) \in S} \bigwedge_{\overline{q}' \in d(\overline{q},w)} u \cdot \delta(\overline{q}, w, \overline{q}')$
                $q * w = w';$
                $S' = \bigcup_{(\overline{q},u) \in S} \bigcup_{\overline{q}' \in d(\overline{q},w)} \{(\overline{q}', w'^{-1} \cdot u \cdot \delta(\overline{q}, w, \overline{q}'))\};$
                if $\exists r \in [0, n-1]$ such that $C'[r] == S'$
                        $e = r;$
                else
                        $C'[e = n++] = S';$
                $q \otimes w = e;$
        }
        $q++;$
    }while($q < n$);
```

Figure 22: Determinization Algorithm

The subsequential transducer generated by this algorithm could in turn be minimized by an algorithm described in (Mohri, 1994a). However, in the case of the part-of-speech tagger, the transducer is nearly minimal.

9.2 Proof of Correctness

Although it is decidable whether a function is subsequential or not (Choffrut, 1977), the determinization algorithm described in the previous section does not terminate when run on a non-subsequential function.

Two issues are addressed in this section. First, the proof of soundness: the fact that if the algorithm terminates, then the output transducer is deterministic and represents the same function. Second, the proof of completeness: the algorithm terminates in the case of subsequential functions.

Soundness and completeness are a consequence of the main proposition which states that if a transducer T represents a subsequential function f, then the algorithm *DeterminizeTransducer* described in the previous section applied on T computes a subsequential transducer representing the same function.

In order to simplify the proofs, we will only consider transducers that do not have ϵ input transitions, that is $E \subseteq Q \times \Sigma \times \Sigma^* \times Q$, and also without loss of generality, transducers that are reduced and that are deterministic in the sense of finite-state automata[18].

In order to prove this proposition, we need to establish some preliminary notations and lemmas.

First we extend the definition of the transition function d, the emission function δ, the deterministic transition function $\otimes$ and the deterministic emission function $*$ on words in the classical way. We then have the following properties:

$$\begin{aligned}
d(q, ab) &= \bigcup_{q' \in d(q,a)} d(q', b) \\
\delta(q_1, ab, q_2) &= \bigcup_{\{q' \in d(q_1,a) | q_2 \in d(q',b)\}} \delta(q_1, a, q') \cdot \delta(q', b, q_2) \\
q \otimes ab &= (q \otimes a) \otimes b \\
q * ab &= (q * a) \cdot (q \otimes a) * b
\end{aligned}$$

[18] A transducer defines an automaton whose labels are the pairs "input/output"; this automaton is assumed to be deterministic.

For the following, it useful to note that if $|T|$ is a function, then δ is a function too.

The following lemma states an invariant that holds for each state S built within the algorithm. The lemma will later be used for the proof of soundness.

Lemma 1 *Let $I = C'[0]$ be the initial state. At each iteration of the "do" loop in* DeterminizeTransducer*, for each $S = C'[q]$ and for each $w \in \Sigma^*$ such that $I \otimes w = S$, the following holds:*

(i) $I * w = \bigwedge_{q \in d(i,w)} \delta(i, w, q)$

(ii) $S = I \otimes w = \{(q, u) | q \in d(i, w) \text{ and } u = (I * w)^{-1} \cdot \delta(i, w, q)\}$

Proof. (i) and (ii) are obviously true for $S = I$ (since $d(i, \epsilon) = i$ and $\delta(i, \epsilon, i) = \epsilon$) and we will show that given some $w \in \Sigma^*$ if it is true for $S = I \otimes w$ then it is also true for $S_1 = S \otimes a = I \otimes wa$ for all $a \in \Sigma$.

Assuming that (i) and (ii) hold for S and w, then for each $a \in \Sigma$:

$$\begin{aligned}
\bigwedge_{q \in d(i,w), q' \in d(q,a)} \delta(i, w, q) \cdot \delta(q, a, q') &= (I * w) \cdot \bigwedge_{q \in d(i,w), q' \in d(q,a)} ((I * w)^{-1} \cdot \delta(i, w, q)) \cdot \delta(q, a, q') \\
&= (I * w) \cdot \bigwedge_{(q,u) \in S = I \otimes w, q' \in d(q,a)} u \cdot \delta(q, a, q') \\
&= (I * w) \cdot (S * a) \\
&= I * w \cdot (I \otimes w) * a \\
&= I * wa
\end{aligned}$$

This proves (i).

We now turn to (ii). Assuming that (i) and (ii) hold for S and w, then for each $a \in \Sigma$, let $S_1 = S \otimes a$; the algorithm (line 8) is such that

$$S_1 = \{(q', u') | \exists (q, u) \in S, q' \in d(q, a) \text{ and } u' = (S * a)^{-1} \cdot u \cdot \delta(q, a, q')\}$$

Let

$$S_2 = \{(q', u') | q' \in d(i, wa) \text{ and } u' = (I * wa)^{-1} \cdot \delta(i, wa, q')\}$$

We show that $S_1 \subset S_2$. Let $(q', u') \in S_1$, then $\exists (q, u) \in S$ s.t. $q' \in d(q, a)$ and $u' = (S * a)^{-1} \cdot u \cdot \delta(q, a, q')$. Since $u = (I * w)^{-1} \cdot \delta(i, w, q)$, then

$u' = (S*a)^{-1} \cdot (I*w)^{-1} \cdot \delta(i,w,q) \cdot \delta(q,a,q')$, that is, $u' = (I*wa)^{-1} \cdot \delta(i,wa,q')$. Thus $(q',u') \in S_2$. Hence $S_1 \subset S_2$.

We now show that $S_2 \subset S_1$. Let $(q',u') \in S_2$, and let $q \in d(i,w)$ be s.t. $q' \in d(q,a)$ and $u = (I*w)^{-1} \cdot \delta(i,w,q)$ then $(q,u) \in S$ and since $u' = (I*wa)^{-1} \cdot \delta(i,wa,q') = (S*a)^{-1} \cdot u \cdot \delta(q,a,q')$, $(q',u') \in S_1$

This concludes the proof of (ii). □

The following lemma states a common property of the state S, which will be used in the complexity analysis of the algorithm.

Lemma 2 *Each $S = C'[q]$ built within the "do" loop is s.t. $\forall q \in Q$, there is at most one pair $(q,w) \in S$ with q as first element.*

Proof. Suppose $(q,w_1) \in S$ and $(q,w_2) \in S$, and let w be s.t. $I \otimes w = S$. Then $w_1 = (I*w)^{-1} \cdot \delta(i,w,q)$ and $w_2 = (I*w)^{-1} \cdot \delta(i,w,q)$. Thus $w_1 = w_2$. □

The following lemma will also be used for soundness. It states that the final state emission function is indeed a function.

Lemma 3 *For each S built in the algorithm, if $(q,u),(q',u') \in S$, then $q,q' \in F \Rightarrow u = u'$*

Proof. Let S be one state set built in line 8 of the algorithm. Suppose $(q,u),(q',u') \in S$ and $q,\ q' \in F$. According to (ii) of lemma 1, $u = (I*w)^{-1} \cdot \delta(i,w,q)$ and $u' = (I*w)^{-1} \cdot \delta(i,w,q')$. Since $|T|$ is a function and $\{\delta(i,w,q), \delta(i,w,q')\} \in |T|(w)$ then $\delta(i,w,q) = \delta(i,w,q')$, therefore $u = u'$. □

The following lemma will be used for completeness.

Lemma 4 *Given a transducer T representing a subsequential function, there exists a bound M s.t. for each S built at line 8, for each $(q,u) \in S$, $|u| \leq M$.*

We rely on the following theorem proven by Choffrut (1978):

Theorem 1 *A function f on Σ^* is subsequential iff it has bounded variations and for any rational language $L \subset \Sigma^*$, $f^{-1}(L)$ is also rational.*

with the following two definitions:

Definition 2 *The left distance between two strings u and v is*

$$\|u, v\| = |u| + |v| - 2|u \wedge v|$$

Definition 3 *A function f on Σ^* has bounded variations iff for all $k \geq 0$, there exists $K \geq 0$ s.t. $u, v \in dom(f)$, $\|u, v\| \leq k \Rightarrow \|f(u), f(v)\| \leq K$*

Proof of lemma 4: Let $f = |T|$. For each $q \in Q$ let $c(q)$ be a string w s.t. $d(q, w) \cap F \neq \emptyset$ and s.t. $|w|$ is minimal among such strings. Note that $|c(q)| \leq \|T\|$ where $\|T\|$ is the number of states in T. For each $q \in Q$ let $s(q) \in Q$ be a state s.t. $s(q) \in d(q, c(q)) \cap F$. Let us further define

$$\begin{aligned} M_1 &= max_{q \in Q}|\delta(q, c(q), s(q))| \\ M_2 &= max_{q \in Q}|c(q)| \end{aligned}$$

Since f is subsequential, it is of bounded variations, therefore there exists K s.t. if $\|u, v\| \leq 2M_2$ then $\|f(u), f(v)\| \leq K$. Let $M = K + 2M_1$.

Let S be a state set built at line 8 , let w be s.t. $I \otimes w = S$ and $\lambda = I * w$. Let $(q_1, u) \in S$. Let $(q_2, v) \in S$ be s.t. $u \wedge v = \epsilon$. Such a pair always exists, since if not

$$\begin{aligned} |\bigwedge_{(q', u') \in S} u'| &> 0 \\ \text{thus } |\lambda \cdot \bigwedge_{(q', u') \in S} u'| &= |\bigwedge_{(q', u') \in S} \lambda \cdot u'| > |\lambda| \end{aligned}$$

Thus, because of (ii) in lemma 1,

$$|\bigwedge_{q' \in d(i, w)} \delta(i, w, q')| > |I * w|$$

which contradicts (i) in lemma 1.

Let $\omega = \delta(q_1, c(q_1), s(q_1))$ and $\omega' = \delta(q_2, c(q_2), s(q_2))$.

Moreover, for any a,b,c,$d \in \Sigma^*$, $\|a, c\| \leq \|ab, cd\| + |b| + |d|$. In fact, $\|ab, cd\| = |ab| + |cd| - 2|ab \wedge cd| = |a| + |c| + |b| + |d| - 2|ab \wedge cd| = \|a, c\| + 2|a \wedge c| + |b| + |d| - 2|ab \wedge cd|$ but $|ab \wedge cd| \leq |a \wedge c| + |b| + |d|$ and since $\|ab, cd\| = \|a, c\| - 2(|ab \wedge cd| - |a \wedge c| - |b| - |d|) - |b| - |d|$ one has $\|a, c\| \leq \|ab, cd\| + |b| + |d|$.

Therefore, in particular, $|u| \leq \|\lambda u, \lambda v\| \leq \|\lambda u \omega, \lambda v \omega'\| + |\omega| + |\omega'|$, thus $|u| \leq \|f(w \cdot c(q_1)), f(w \cdot c(q_2))\| + 2M_1$. But $\|w \cdot c(q_1), w \cdot c(q_2)\| \leq |c(q_1)| +$

$|c(q_2)| \leq 2M_2$, thus $\|f(w \cdot c(q_1)), f(w \cdot c(q_2))\| \leq K$ and therefore $|u| \leq K + 2M_1 = M$. □

The time is now ripe for the main proposition which proves soundness and completeness.

Proposition 1 *If a transducer T represents a subsequential function f, then the algorithm* DeterminizeTransducer *described in the previous section applied on T computes a subsequential transducer τ representing the same function.*

Proof. The lemma 4 shows that the algorithm always terminates if $|T|$ is subsequential.

Let us show that $dom(|\tau|) \subset dom(|T|)$. Let $w \in \Sigma^*$ s.t. w is not in $dom(|T|)$, then $d(i, w) \cap F = \emptyset$. Thus, according to (ii) of lemma 1, for all $(q, u) \in I \otimes w$, q is not in F, thus $I \otimes w$ is not terminal and therefore w is not in $dom(\tau)$.

Conversely, let $w \in dom(|T|)$. There exists a unique $q_f \in F$ s.t. $|T|(w) = \delta(i, w, q_f)$ and s.t. $q_f \in d(i, w)$. Therefore $|T|(w) = (I * w) \cdot ((I * w)^{-1} \cdot \delta(i, w, q_f))$ and according to (ii) of lemma 1 $(q_f, (I * w)^{-1} \cdot \delta(i, w, q_f)) \in I \otimes w$ and since $q_f \in F$, lemma 3 shows that $\rho(I \otimes w) = (I * w)^{-1} \cdot \delta(i, w, q_f)$, thus $|T|(w) = (I * w) \cdot \rho(I \otimes w) = |\tau|(w)$. □

9.3 Worst-case complexity

In this section we give a worst-case upper bound of the size of the subsequential transducer in term of the size of the input transducer.

Let $L = \{w \in \Sigma^*$ s.t. $|w| \leq M\}$ where M is the bound defined in the proof of lemma 4. Since, according to lemma 2, for each state set Q', for each $q \in Q$, Q' contains at most one pair (q, w), the maximal number N of states built in the algorithm is smaller than the sum of the number of functions from states to strings in L for each state set, that is

$$N \leq \sum_{Q' \in 2^Q} |L|^{|Q'|}$$

we thus have $N \leq 2^{|Q|} \times |L|^{|Q|} = 2^{|Q|} \times 2^{|Q| \times \log_2 |L|}$ and therefore $N \leq 2^{|Q|(1+\log |L|)}$.

Moreover,

$$|L| = 1 + |\Sigma| + \cdots + |\Sigma|^M = \frac{|\Sigma|^{M+1} - 1}{|\Sigma| - 1} \text{ if } |\Sigma| > 1$$

and $|L| = M + 1$ if $|\Sigma| = 1$. In this last formula, $M = K + 2M_1$ as described in lemma 4. Note that if $P = max_{a \in \Sigma} |\delta(q, a, q')|$ is the maximal length of the simple transitions emissions, $M_1 \leq |Q| \times P$, thus $M \leq K + 2 \times |Q| \times P$.

Therefore, if $|\Sigma| > 1$, the number of states N is bounded: $N \leq 2^{|Q| \times (1 + \log \frac{|\Sigma|^{(K+2\times|Q|\times P+1} - 1}{|\Sigma| - 1})}$ and if $|\Sigma| = 1$, $N \leq 2^{|Q| \times (1 + \log (K + 2 \times |Q| \times P + 1))}$.

10 Subsequentiality of Transformation-Based Systems

The proof of correctness of the determinization algorithm and the fact that the algorithm terminates on the transducer encoding Brill's tagger show that the final function is subsequential and equivalent to Brill's original tagger.

In this section, we prove in general that any transformation-based system, such as those used by Brill, is a subsequential function. In other words, any transformation-based system can be turned into a deterministic finite-state transducer.

We define transformation-based systems as follows.

Definition 4 *A transformation-based system is a finite sequence* $(f_1, \cdots, f_n)$ *of subsequential functions whose domains are bounded.*

Applying a transformation-based system consists of taking the functions f_i, one after the other, and for each of them, one looks for the first position in the input at which it applies, and for the longest string starting at that position, transforms this string, go to the end of this string, and iterate until the end of the input.

It is not true that, in general, the local extension of a subsequential function is subsequential[19]. For instance, consider the function f_a of Figure 23.

The local extension of the function f_a is not a function. In fact, consider the input string *daaaad*, it can be decomposed either into $d \cdot aaa \cdot ad$ or into

[19] However, the local extensions of the functions we had to compute *were* subsequential.

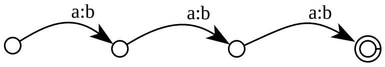


Figure 23: Function f_a

$da \cdot aaa \cdot d$. The first decomposition leads to the output $dbbbad$ and the second one to the output $dabbbd$.

The intended use of the rules in the tagger defined by Brill is to apply each function from left to right. In addition, if several decompositions are possible, the one that occurs first is the one chosen. In our previous example, it means that only the output $dbbbad$ is generated.

This notion is now defined precisely.

Let α be the rational function defined by $\alpha(a) = a$ for $a \in \Sigma$, $\alpha([) = \alpha(]) = \epsilon$ on the additional symbols '[' and ']' with α such that $\alpha(u \cdot v) = \alpha(u) \cdot \alpha(v)$.

Definition 5 *Let $Y \subset \Sigma^*$ and $X = \Sigma^* - \Sigma^* \cdot Y \cdot \Sigma^*$, a Y-decomposition of x is a string $y \in X \cdot ([\cdot Y \cdot] \cdot X)^*$ s.t. $\alpha(y) = x$*

For instance, if $Y = dom(f_a) = \{aaa\}$, the set of Y-decompositions of $x = daaad$ is $\{d[aaa]ad, da[aaa]d\}$.

Definition 6 *Let $<$ be a total order on Σ and let $\overline{\Sigma} = \Sigma \cup \{[,]\}$ be the alphabet Σ with the two additional symbols '[' and ']'. Let extend the order $>$ to $\overline{\Sigma}$ by $\forall a \in \Sigma$, '['$< a$ and $a <$ ']'. $<$ defines a lexicographic order on $\overline{\Sigma}^*$ that we also denote $<$. Let $Y \subset \Sigma^*$ and $x \in \Sigma^*$, the* minimal Y-decomposition *of x is the Y-decomposition which is minimal in $(\overline{\Sigma}^*, <)$.*

For instance, the minimal $dom(f_a)$-decomposition of $daaaad$ is $d[aaa]ad$. In fact, $d[aaa]ad < da[aaa]d$.

Proposition 2 *Given $Y \subset \Sigma^+$ finite, the function md_Y that to each $x \in \Sigma^*$ associates its minimal Y-decomposition, is subsequential and total.*

Proof. Let dec be defined by $dec(w) = u \cdot [\cdot v \cdot] \cdot dec((uv)^{-1} \cdot w)$ where $u, v \in \Sigma^*$ are s.t. $v \in Y$, $\exists v' \in \Sigma^*$ with $w = uvv'$ and $|u|$ is minimal among such strings. The function md_Y is total because the function dec always returns an output which is a Y-decomposition of w.

We shall now prove that the function is rational and then that it has bounded variations; this will prove according to theorem 1 that the function is subsequential. In the following $X = \Sigma^* - \Sigma^* \cdot Y \cdot \Sigma^*$. The transduction T_Y that generates the set of Y-decompositions is defined by

$T_Y = \mathrm{Id}_X \cdot (\epsilon/[\cdot \mathrm{Id}_Y \cdot \epsilon/] \cdot \mathrm{Id}_X)^*$

where Id_X (resp. Id_Y) stands for the identity function on X (resp. Y). Furthermore, the transduction $T_{\overline{\Sigma},>}$ that to each string $w \in \overline{\Sigma}^*$ associates the set of strings strictly greater than w, that is $T_{\overline{\Sigma},>}(w) = \{w' \in \overline{\Sigma}^* | w < w'\}$, is defined by the transducer of Figure 24 in which $A = \{(x,x) | x \in \overline{\Sigma}\}$, $B = \{(x,y) \in \overline{\Sigma}^2 | x < y\}$, $C = \overline{\Sigma}^2$, $D = \{\epsilon\} \times \overline{\Sigma}$ and $E = \overline{\Sigma} \times \{\epsilon\}$.[20]

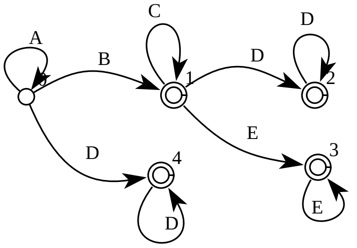


Figure 24: Transduction $T_{\overline{\Sigma},>}$

Therefore, the right-minimal Y-decomposition function md_Y is defined by $md_Y = T_Y - (T_{\overline{\Sigma},>} \circ T_Y)$ which proves that md_Y is rational.

Let $k > 0$. Let $K = 6 \times k + 6 \times M$ where $M = max_{x \in Y}|x|$. Let u, $v \in \Sigma^*$ be s.t. $\|u, v\| \leq k$. Let us consider two cases: (i) $|u \wedge v| \leq M$ and (ii) $|u \wedge v| > M$.

(i): $|u \wedge v| \leq M$, thus $|u|,|v| \leq |u \wedge v| + \|u, v\| \leq M + k$. Moreover, for each $w \in \Sigma^*$, for each Y-decomposition w' of w, $|w'| \leq 3 \times |w|$. In fact, Y doesn't contain ϵ, thus the number of [(resp.]) in w' is smaller than $|w|$. Therefore, $|md_Y(u)|, |md_Y(v)| \leq 3 \times (M + k)$ thus $\|md_Y(u), md_Y(v)\| \leq K$.

(ii): $u \wedge v = \lambda \cdot \omega$ with $|\omega| = M$. Let μ, ν be s.t. $u = \lambda\omega\mu$ and $v = \lambda\omega\nu$. Let λ', ω', μ', λ'', ω'' and ν'' be s.t. $md_Y(u) = \lambda'\omega'\mu'$, $md_Y(v) = \lambda''\omega''\nu''$, $\alpha(\lambda') = \alpha(\lambda'') = \lambda$, $\alpha(\omega') = \alpha(\omega'') = \omega$, $\alpha(\mu') = \mu$ and $\alpha(\nu'') = \nu$. Suppose that $\lambda' \neq \lambda''$, for instance $\lambda' < \lambda''$. Let i be the first indice s.t. $(\lambda')_i < (\lambda'')_i$.[21]

[20] This construction is similar to the transduction built within the proof of Eilenberg's cross section theorem (Eilenberg, 1974).

[21] $(w)_i$ refers to the i^{th} letter in w.

We have two possible situations: (ii.1) $(\lambda')_i = [$ and $\lambda'' \in \Sigma$ or $(\lambda'')_i =]$. In that case, since the length of the elements in Y is smaller than $M = |\omega'|$, one has $\lambda'\omega' = \lambda_1[\lambda_2]\lambda_3$ with $|\lambda_1| = i$, $\lambda_2 \in Y$ and $\lambda_3 \in \overline{\Sigma}^*$. We also have $\lambda''\omega'' = \lambda_1\lambda_2'\lambda_3'$ with $\alpha(\lambda_2') = \alpha(\lambda_2)$ and the first letter of λ_2' is different from [. Let λ_4 be a Y-decomposition of $\alpha(\lambda_3'\nu'')$, then $\lambda_1[\lambda_2]\lambda_4$ is a Y-decomposition of v strictly smaller than $\lambda_1\lambda_2'\lambda_3'\nu'' = md_Y(v)$ which contradicts the minimality of $md_Y(v)$. The second situation is (ii.2): $(\lambda')_i \in \Sigma$ and $(\lambda'')_i =]$, then we have $\lambda'\omega' = \lambda_1[\lambda_2\lambda_3]\lambda_4$ s.t. $|\lambda_1[\lambda_2| = i$ and $\lambda''\omega'' = \lambda_1[\lambda_2]\lambda_3'\lambda_4'$ s.t. $\alpha(\lambda_3') = \alpha(\lambda_3)$ and $\alpha(\lambda_4') = \alpha(\lambda_4)$. Let λ_5 be a Y-decomposition of $\lambda_4'\nu''$ then $\lambda_1[\lambda_2\lambda_3]\lambda_5$ is a Y-decomposition of v strictly smaller than $\lambda''\omega''\nu''$ which leads to the same contradiction. Therefore, $\lambda' = \lambda''$ and since $|\mu'| + |\nu''| \leq 3 \times (|\mu| + |\nu|) = 3\times\|u,v\| \leq 3\times k$, $\|md_Y(u), md_Y(v)\| \leq |\omega'|+|\omega''|+|\mu'|+|\nu''| \leq 2\times M+3\times k \leq K$. This proves that md_Y has bounded variations and therefore that it is subsequential. □

We can now define precisely what is the effect of a function when one applies it from left to right, as was done in the original tagger.

Definition 7 *If f is a rational function, $Y = dom(f) \subset \Sigma^+$, the right-minimal local extension of f, denoted RmLocExt(f), is the composition of a right-minimal Y-decomposition md_Y with $Id_{\Sigma^*} \cdot ([/\epsilon \cdot f \cdot\ ]/\epsilon \cdot Id_{\Sigma^*})^*$.*

RmLocExt being the composition of two subsequential functions, it is itself subsequential, this proves the following final proposition which states that given a rule-based system similar to Brill's system, one can build a subsequential transducer that represents it:

Proposition 3 *If $(f_1, \cdots, f_n)$ is a sequence of subsequential functions with bounded domains then*
$RmLocExt(f_1) \circ \cdots \circ RmLocExt(f_n)$ is subsequential.

We have proven in this section that our techniques apply to the class of transformation-based systems. We now turn our attention to the implementation of finite-state transducers.

11 Implementation of Finite-state Transducers

Once the final finite-state transducer is computed, applying it to an input is straightforward: it consists of following a unique path in the transducer whose left labels correspond to the input. However, in order to have a complexity fully independent of the size of the grammar and in particular, independent of the number of transitions at each state, one should carefully choose an appropriate representation for the transducer. In our implementation, the transitions can be accessed randomly. The transducer is first represented by a two-dimensional table whose rows are indexed by the states and whose columns are indexed by the alphabet of all possible input letters. The content of the table at line q and at column a is the word w such that the transition from q with the input label a outputs w. Since only a few transitions are allowed from many states, this table is very sparse and can be compressed. This compression is achieved using a procedure for sparse data tables following the method given by Tarjan and Yao (1979).

12 Acknowledgments

We thank Eric Brill for providing us with the code of his tagger and for many useful discussions. We also thank Aravind K. Joshi, Mark Liberman and Mehryar Mohri for valuable discussions. We thank the anonymous reviewers for many helpful comments that led to improvements in both the content and the presentation of this paper.

13 Conclusion

The techniques described in this paper are more general than the problem of part-of-speech tagging and are applicable to the class of problems dealing with local transformation rules.

We showed that any transformation based program can be transformed into a deterministic finite-state transducer. This yields to optimal time implementations of transformation based programs.

As a case study, we applied these techniques to the problem of part-of-speech tagging and presented a finite-state tagger that requires n steps to tag a sentence of length n, independent of the number of rules and the length of the context they require. We achieved this result by representing the rules acquired for Brill's tagger as non-deterministic finite-state transducers. We composed each of these non-deterministic transducers and turned the resulting transducer into a deterministic transducer. The resulting deterministic transducer yields a part-of-speech tagger which operates in optimal time in the sense that the time to assign tags to a sentence corresponds to the time required to follow a single path in this deterministic finite-state machine. The tagger outperforms in speed both Brill's tagger and trigram-based taggers. Moreover, the finite-state tagger inherits from the rule-based system its compactness compared to a stochastic tagger. We also proved the correctness and the generality of the methods.

We believe that this finite-state tagger will also be found useful combined with other language components, since it can be naturally extended by composing it with finite-state transducers which could encode other aspects of natural language syntax.

Bibliography

Brill, Eric. 1992. A simple rule-based part of speech tagger. In *Third Conference on Applied Natural Language Processing*, pages 152–155, Trento, Italy.

Brill, Eric. 1994. A report of recent progress in transformation error-driven learning. In *AAAI'94, Tenth National Conference on Artificial Intelligence.*

Choffrut, Christian. 1977. Une caractérisation des fonctions séquentielles et des fonctions sous-séquentielles en tant que relations rationnelles. *Theoretical Computer Science*, 5:325–338.

Choffrut, Christian. 1978. *Contribution à l'étude de quelques familles remarquables de fonctions rationnelles.* Ph.D. thesis, Université Paris VII (Thèse d'Etat).

Chomsky, N. 1964. *Syntactic Structures.* Mouton and Co., The Hague.

Church, Kenneth Ward. 1988. A stochastic parts program and noun phrase parser for unrestricted text. In *Second Conference on Applied Natural Language Processing*, Austin, Texas.

Clemenceau, David. 1993. *Structuration du Lexique et Reconnaissance de Mots Dérivés*. Ph.D. thesis, Université Paris 7.

Cutting, Doug, Julian Kupiec, Jan Pederson, and Penelope Sibun. 1992. A practical part-of-speech tagger. In *Third Conference on Applied Natural Language Processing*, pages 133–140, Trento, Italy.

DeRose, S.J. 1988. Grammatical category disambiguation by statistical optimization. *Computational Linguistics*, 14:31–39.

Eilenberg, Samuel. 1974. *Automata, languages, and machines.* Academic Press, New York.

Elgot, C. C. and J. E. Mezei. 1965. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, 9:47–65, January.

Francis, W. Nelson and Henry Kučera. 1982. *Frequency Analysis of English Usage*. Houghton Mifflin, Boston.

Kaplan, Ronald M. and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

Karttunen, Lauri, Ronald M. Kaplan, and Annie Zaenen. 1992. Two-level morphology with composition. In *Proceedings of the* 14^{th} *International Conference on Computational Linguistics (COLING'92).*

Kupiec, J. M. 1992. Robust part-of-speech tagging using a hidden Markov model. *Computer Speech and Language*, 6:225–242.

Laporte, Eric. 1993. Phonétique et transducteurs. Technical report, Université Paris 7, June.

Merialdo, Bernard. 1990. Tagging text with a probabilistic model. Technical Report RC 15972, IBM Research Division.

Mohri, Mehryar. 1994a. Minimisation of sequential transducers. In *Proceedings of the Conference on Computational Pattern Matching 1994.*

Mohri, Mehryar. 1994b. On some applications of finite-state automata theory to natural language processing. Technical report, Institut Gaspard Monge.

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

Revuz, Dominique. 1991. *Dictionnaires et Lexiques, Méthodes et Algorithmes*. Ph.D. thesis, Université Paris 7.

Roche, Emmanuel. 1993. *Analyse Syntaxique Transformationelle du Français par Transducteurs et Lexique-Grammaire*. Ph.D. thesis, Université Paris 7, January.

Schűtzenberger, Marcel Paul. 1977. Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, 4:47–57.

Silberztein, Max. 1993. *Dictionnaires Electroniques et Analyse Lexicale du Français — Le Système INTEX*. Masson.

Tapanainen, Pasi and Atro Voutilainen. 1993. Ambiguity resolution in a reductionistic parser. In *Sixth Conference of the European Chapter of the ACL, Proceedings of the Conference*, Utrecht, April.

Tarjan, Robert Endre and Andrew Chi-Chih Yao. 1979. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November.

Weischedel, Ralph, Marie Meteer, Richard Schwartz, Lance Ramshaw, and Jeff Palmucci. 1993. Coping with ambiguity and unknown words through probabilistic models. *Computational Linguistics*, 19(2):359–382, June.