Natural Language Engineering **1** (1): 1-16. Printed in the United Kingdom

Finite State Methods for Hyphenation

Gosse Bouma
Alfa-informatica, Rijksuniversiteit Groningen
Groningen, The Netherlands

(*Received 30 March 2002; revised 30 September 2002*)

Abstract

Hyphenation is the task of identifying potential hyphenation points in words. In this paper, three finite-state hyphenation methods for Dutch are presented and compared in terms of accuracy and size of the resulting automata.

1 Introduction

A typesetting program may have to hyphenate words to produce justified paragraphs. Hyphenation can be carried out using a word list with hyphenation points (usually derived from an electronic dictionary). The major drawback of this method is that it will only cover words explicitly listed in the dictionary.

Pattern-based hyphenation methods use patterns of a sequence of two or more characters, for which valid (or invalid) hyphenation positions are specified. A pattern can be applied to all words containing the pattern as a substring. A well-known pattern-based hyphenation-method, used in the typesetting package TEX, is that of Liang (1983). Patterns are usually derived from a dictionary, but since patterns represent substrings of words, they also apply to words not in the original list.

Rule-based hyphenation methods rely on syllable and morpheme structure. The approach described in the next section uses syllable structure to implement a hyphenation procedure. No attempt was made, however, to deal accurately with morpheme boundaries, and therefore the system achieves only modest accuracy. Rule-based methods use dictionary information only indirectly and are therefore not restricted to a fixed set of words.

It has been observed that rule-based and pattern-based hyphenation can be carried out by a finite-state transducer (Kaplan and Karttunen, 1998). Such a transducer would take a character string as input and return a string with hyphenation points. A transducer of this form can be constructed by translating hyphenation rules or patterns into a finite state transducer or sequence of transducers. In this paper, we investigate the feasibility of actually developing an accurate finite-state hyphenator for Dutch.

Below, we consider three finite-state hyphenation methods. The core rule of Dutch hyphenation is that hyphenation points fall between syllables, where words can be

divided into syllables using a maximal onset rule. The first method implements this rule as a deterministic finite state transducer. When evaluated on word list derived from Celex (Baayen, Piepenbrock, and van Rijn, 1993), this simple system achieves a hyphen accuracy of 94.5%.

The second system builds on the results of the first, but uses transformation-based learning (TBL) (Brill, 1995) to derive rules which correct the errors of the first system. The system is trained on the Celex word list. Induced rules typically help to correct mistakes which are due to the fact that the first system ignores morpheme structure. The hyphen accuracy after applying TBL is 99.35%. The rules induced by TBL can be interpreted as a cascade of finite-state transducers (Roche and Schabes, 1997), . We present experimental results which suggest that composition of large numbers of rules may be of limited practical use.

Finally, we present results for the pattern-based hyphenation method of Liang (1983), which is used in TEX. The hyphenation patterns for Dutch (more than 8,000) are again derived from Celex. We compared the performance of TEX and the TBL-system on running text. Hyphen accuracy of TEX is estimated to be 99.8%, whereas the TBL-system achieves approximately 99.1% accuracy. We also present a method for compiling hyphenation-patterns into a single FST.[1]

2 Hyphenation rules for Dutch

Hyphenation in Dutch is determined by both morpheme and syllable structure. Hyphenation points fall between morpheme boundaries, and, within a morpheme, between syllables. In particular, Brandt Corstius (1978) describes a hyphenation procedure which first inserts hyphenation points (in morphologically complex words) between word and (certain) morpheme boundaries, and next inserts hyphenation points (at the morpheme level) between syllables, while maximizing onsets. The procedure can be illustrated using with the following example. The word *drugspanden* (*drug-houses*) is a plural compound noun composed of *drugs* and *panden*. Thus, a hyphenation point is inserted between *s* and *p*. Next, *panden* can be segmented into syllables as *pan-den* or *pand-en*. The segmentation *pan-den* maximizes the second onset, and thus is the correct one. Note that the first step must identify compounds and derivational affixes, but not all morpheme structure. The word *panden* is a plural noun consisting of the noun *pand* and the plural suffix *-en*, yet it is segmented as *pan-den*. There are only a few cases which are not covered by the procedure above.[2]

[1] One reviewer points points out that patterns can be used as input for a fourth finite-state hyphenation method. E (*examples*) is a list of substrings of correctly hyphenated words. C (*counterexamples*) is a list of the same substrings with an incorrectly placed hyphen. A correctly hyphenated word may not contain a member of C-E. This constraint can be checked by a FSA defined in terms of C and E. The accuracy and practical feasiblity of this method depends on the length of the substrings considered. We have not evaluated this method.

[2] In some cases the spelling of a hyphenated word differs from that of its non-hyphenated counterpart. The word *extraatje* (*extra* + dimunitive suffix *tje*) is hyphenated as *extra-tje*. This phenomenon is known in TEX as *discretionary hyphenation* (Sojka, 1995). Another complication arises in cases where a word has an ambiguous morphological structure, and hyphenation depends on which analysis is chosen. Daelemans and van

`[]`	the empty string
`[`$R_1, \ldots, R_n$`]`	concatenation
`{`$R_1, \ldots, R_n$`}`	disjunction
`R^`	optionality
`R*`	zero or more occurrences of R
`A:B`	the transducer which maps strings of A onto strings of B
`id(A)`	identity: the transducer which maps each element in A onto itself.
`T o U`	composition of the transducers T and U.
`macro(Term,R)`	use Term as an abbreviation for R.

Fig. 1. FSA regular expression syntax used in this paper. A and B are regular expressions denoting recognizers, T and U transducers, and R can be either.

3 Syllable-based hyphenation

In this section, we present a finite-state transducer which inserts hyphens between well-formed syllables, while maximizing onsets. The accuracy of the system is limited, as it ignores the fact that compounds and derived words may require hyphenation patterns conflicting with the maximal onset rule.

Finite state automata will be defined using regular expressions, as defined in figure 1. Regular expressions are compiled into automata by the FSA utilities toolkit (van Noord, 1997; van Noord and Gerdemann, 2001).[3]

3.1 Implementation

Consider the problem of segmenting a word into a sequence of syllables. Given a suitable definition of `syllable`, one might consider the following as a first attempt:

(1) `[ [ syllable, []:- ]*, syllable ]`

This regular expression defines a transducer that accepts non-empty sequences of syllables as input and outputs the same sequence, with a hyphen inserted after every syllable except for the last. Given an input such as *alfabet* (*alphabet*), it will produce *al-fa-bet, alf-a-bet, alf-ab-et*, or *al-fab-et*, although in fact only the first is correct. This transducer is non-deterministic and does not maximize onsets. A second problem with this definition is that, in general, it will fail to give the right results for syllables containing a nucleus represented by more than a single character. I.e. for a word such as *waait* (*blows*), (1) would produce, among others, an output with three hyphenation points (*wa-a-it*), although in fact *waait* is a monosyllabic word. The problem is caused by the fact that the characters *a,a,* and *i* represent a diphtong but can also each occur independently as a nucleus, and form a syllable. Thus, it seems that apart from a maximal onset rule, there is also something like a *maximal nucleus* rule.

den Bosch (1992) mention *kwartslagen* as an example, which can be hyphenated as *kwart-slagen* (*quarter turns*) or *kwarts-lagen* (*quartz layers*).

[3] The FSA utilities toolkit is available from `www.let.rug.nl/~vannoord/fsa`.

A more accurate definition of hyphenation needs to define a deterministic automaton, which, given multiple ways of segmenting a string into valid syllables, returns only the output in which both nuclei and onsets are maximal. The option adopted below, also proposed in Karttunen (2001) , is to use the *replace*-operator of Kaplan and Kay (1994) and Karttunen (1995), which supports *longest match* replacements directly.[4]

In the FSA-notation of Gerdemann and van Noord (1999) a regular expression `replace(Target, LeftContext, RightContext)`, where `Target` is a transducer and `LeftContext` and `RightContext` are recognizers, defines a transducer which replaces all occurrences of the domain of `Target` between `LeftContext` and `RightContext` by strings in the range of `Target`. Furthermore, `replace` performs left-most, longest match, replacement, i.e. it operates as if moving through the string from left to right, at each point identifying the longest possible replacement target.

The regular expression for hyphenation given above can be rephrased as a replace statement, which inserts hyphens between syllables:

(2) `replace([]:-, syllable, syllable)`

Again, given an adequate definition of `syllable`, this transducer will insert hyphens between syllable strings. Note, however, that the left-to-right mode of application of `replace` ensures that hyphens are inserted as early as possible, thus implicitly ensuring that onsets are maximal. The word *alfabet* is hyphenated only as *al-fa-bet* by this expression, and not as any of the alternatives mentioned above.

Implementing the *maximal nucleus* rule required for the correct hyphenation of words containing multiple character nuclei, can be achieved using the longest match property of `replace` explicitly:

(3) `replace([ []:@, identity(nucleus), []:@ ], [], [] )`

This transducer inserts the marker '@' before and after nuclei. As the target for a replacement is determined using longest match, multiple character strings corresponding to a possible nucleus are always marked as a single nucleus. The word *waait* is marked as *w@aai@t* only. Left-to-right, longest match, identification of nuclei appears to be very accurate. [5]

A syllable-based finite-state hyphenation method is now simply the composition of the transducers defined in (3) and (2), given a definition of syllable which incorporates the '@'-marker:

(4)
```
replace([[]:@, identity(nucleus), []:@],[],[])
                          o
     replace( []:-, syllable, syllable )
```

[4] An alternative approach would be to formulate the problem in terms of (finite-state) Optimality Theory, as described in Karttunen (1998) and Gerdemann and van Noord (2000) in such a way that maximal onsets and nuclei are preferred.

[5] One rare case where it fails is the word *dieet* (*diet*), which contains the nucleus *i* followed *ee*, but longest match recognizes *ie* and *e*.

```
macro(hyphenate,
       replace([q,u]:Q, [], [])                    %% qu -> Q
     o replace([ []:@, id(nucleus),[]:@ ],[],[])   %% mark nucleus
     o replace([e,@,@]:[@,@,e],i,{e,u})            %% d@ie@@e@t -> d@i@@ee@t
     o replace( []:-, [@, cons *], [onset^ , @] ) %% insert hyphens
     o replace(-:[], x, [@,{a,e,u,i,o,y}])         %% remove - after x
     o replace(Q:[q,u], [], [])                    %% Q -> qu
     o replace(@:[], [], []) ).                      %% remove markers


macro(nucleus,{ a, [a,i], [a,a,i], ..., u, [u,i], y,
                [{[a,a],[o,o],[u,u]}, cons ] } ).


macro(onset, { b, [b,l], [b,r], ..., z, [z,w] } ).
```

Fig. 2. Finite state syllable-based hyphenation of Dutch

The actual implementation, given in figure 2, imposes slightly weaker conditions on the insertion of hyphens:

(5) `replace( []:-, [@, consonant *], [onset^, @] )`

Instead of specifying a full syllable as left and right context, this definition only requires a left context consisting of a marker followed by an arbitrary number of consonants (corresponding to the coda of the preceding syllable), and a right context consisting of an (optional) onset followed by a marker. It imposes no constraints on codas, other than that they must form a sequence of consonants. While this does not lead to overgeneration, it does account for the hyphenation of loan-words containing codas which do not follow Dutch spelling (i.e. the words *checklist, arctisch* (*arctic*), and *pizza* are hyphenated as *check-list, arc-tisch*, and *piz-za* in spite of the fact that *ck, rc* and *z* are normally not used as coda).

The accuracy of the hyphenation method defined above can be further improved by making a number of adjustments for irregular hyphenations of words containing the bigram *qu* (which is treated as a consonant), the character *x* (which is never followed by a hyphen), and words containing the bigrams *aa, ee, oo* and *uu*, which indicate long vowels in closed syllables, and therefore can never be followed by a hyphen. Some cases (such as *dieet*) where the longest match strategy for marking the nucleus fails can also be corrected easily.

3.2 Evaluation and Error-analysis

The deterministic FST for the regular expression for `hyphenation` as given in figure 2 has 89 states and 826 transitions.[6] For evaluation, we used the Celex (Baayen,

[6] Transitions include predicates as described in (van Noord and Gerdemann2001). A predicate ranges a set of symbols, and thus, the number of transitions is in general (much) smaller than would normally have been the case.

Piepenbrock, and van Rijn1993) DOW-list (*dutch orthograhpy words*), which provides hyphenation patterns for over 330,000 words. All duplicates and all items containing capitals and diacritics were removed. This leaves approximately 290,000 items. The average word length is 10.85 characters and there are approximately 2.5 hyphens per word. On this list, `hyphenate` achieves a word accurary of 86.1% and a hyphen accuracy of 94.5%.

A 10% subset of the data, was inspected for error analysis, where errors were counted and classified using the alignment method described in the next section. As the system implements the maximal onset rule, but has no way of recognizing compounds or derivational affixes, errors are expected to occur typically in situations where a word or affix boundary conflicts with the maximal onset rule. This is confirmed by the fact that 87.5% of the errors are cases where a hyphen is displaced one position to the left (i.e. *drug-span-den* should be *drugs-pan-den*) and 1.1% of the errors are cases where a hyphen is displaced two positions to the left (*ang-staan-ja-gend* (*scary*, lit. *scare-on-making*) should be *angst-aan-ja-gend*). A hyphen was displaced one position to the right in 4.9% of the errors. These are all caused by the fact that our list of onsets is not exhaustive. For instance, the system produces *ar-tis-jok* (*artichoke* which should be hyphenated as *ar-ti-sjok*) because it does not contain the onset *sj*. In 5.5% of the errors the system has produced a spurious hyphen. These typically occur in loan words containing a nucleus not included in the syllable-based system (*coach* is hyphenated as *co-ach*) . The system missed a hyphen in 1.0% of the errors. Such errors are mostly caused by mistakes in the identification of a nucleus. The word *be-ij-ver* (*work towards*) is hyphenated as *beij-ver* because *ei* is a valid nucleus. This is a case, therefore, where the longest match strategy for recognizing the nucleus fails and which was left unaccounted for in the definition of `hyphenate`.

4 Improving accuracy using TBL

Transformation-based learning (TBL) (Brill, 1995) can be used to improve the accuracy of the system outlined above. Given a word list hyphenated by the base system, aligned with the correct hyphenation patterns, TBL will attempt to induce a rule which corrects the maximal number of errors while introducing a minimum of new errors. This rule is applied to the training data. This process iterates until no new rules with a score (i.e. the number of corrections minus the number of errors introduced by a rule) above a certain threshold can be found. The fact that most rules are not 100% accurate (i.e. introduce new errors besides correcting existing errors) is not necessarily a problem, as more specific rules can be learned later which correct the newly introduced errors. The result of TBL can be tested on a data-set by applying the induced rules in the order in which they have been induced.

4.1 Alignment

TBL requires aligned data for training and testing. Hyphenation can be seen as a classification task, which decides, for instance, for each character whether it is

preceded by a hyphen or not. Thus, the result of hyphenating a word list using the base system can be encoded as a character string, where each character is aligned with a 1 if it is preceded by a hyphen, and with a 0 otherwise. Furthermore, as some rules should apply only to the beginning or end of a word, boundary markers are added. The correct hyphenation can be aligned with this string as a similar list of 0's and 1's:

(6)

word	*potato*	+	a	a	r	d	a	p	p	e	l	+
system	aar-dap-pel	0	0	0	0	1	0	0	1	0	0	0
correct	aard-ap-pel	0	0	0	0	0	1	0	1	0	0	0

For TBL, the encoding in (6) has the disadvantage that correcting a single error (i.e. *aar-dap-pel* → *aard-ap-pel*) requires learning two error-correction rules, one changing a 1 into a 0 when aligned with a *d* in a suitable context, and one changing a 0 into a 1 when aligned with an *a* in a similar context. Obviously, these two rules are closely related. Therefore, more effective error correction can take place if a single rule could be learned to correct the error.

We therefore adopted a slightly more involved alignment procedure, where the correct output can be marked with 0 and 1 as before, but also with 2 (the character is followed by a hyphen), 3 (the next character is followed by a hyphen) or 9 (the preceding character was preceded by a hyphen). Examples of the new alignment method are given in figure 3.

Note that the system output is still given as a sequence of 0's and 1's, and thus, that the aligned system output does not contain information which points to the correct alignment. Alignment of the correct answer with the system data requires a procedure which codes the correct answer relative to the system output. As the number of hyphens in the system output and the correct output tends to be equal (but see below), a procedure which determines for each correct hyphen where it is located relative to the corresponding system hyphen gives good results. If no corresponding system hyphen can be found in the two proceeding positions or in the next position, a 0/1 alignment is introduced. Similarly, if a system hyphen cannot be aligned with any hyphen in the correct answer, a 1/0 alignment is introduced.

4.2 Experiments

Training and test data consisted of the Celex word list described in the previous section. The list was divided into 10 sections (selecting every 10th word, with an offset of 0-9), hyphenated by the base system, and the result was aligned with the correct hyphenation as provided by Celex.

For training and testing we used the fnTBL toolkit[7] (Ngai and Florian, 2001) which implements an efficient version of Brill's original algorithm. Training took between a few minutes and 3 hours, depending on the amount of data used and the complexity of the rule templates.

Rule templates for TBL were provided which change the value of one cell in the

[7] `http://nlp.cs.jhu.edu/~rflorian/fntbl/index.html`

word	(*potato*)	+	a	a	r	d	a	p	p	e	l	+					
system	aar-dap-pel	0	0	0	0	1	0	0	1	0	0	0					
correct	aard-ap-pel	0	0	0	0	2	0	0	1	0	0	0					
word	(*coach*)	+	c	o	a	c	h	+									
system	co-ach	0	0	0	1	0	0	0									
correct	coach	0	0	0	0	0	0	0									
word	*artichoke*)	+	a	r	t	i	s	j	o	k	+						
system	ar-tis-jok	0	0	0	1	0	0	1	0	0	0						
correct	ar-ti-sjok	0	0	0	1	0	0	9	0	0	0						
word	(*scary*)	+	a	n	g	s	t	a	a	n	j	a	g	e	n	d	+
system	ang-staan-ja-gend	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0
correct	angst-aan-ja-gend	0	0	0	0	3	0	0	0	0	1	0	1	0	0	0	0
word	(*with make up on*)	+	g	e	s	c	h	m	i	n	k	t	+				
sys	gesch-minkt	0	0	0	0	0	0	1	0	0	0	0	0				
cor	ge-schminkt	0	0	0	1	0	0	0	0	0	0	0	0				

Fig. 3. Improved alignment

system output (i.e. change 1 into 2), using a surrounding window of maximally 5 characters as context to constrain the rule. For instance, to correct *aar-dap-pel* into *aard-ap-pel*, the system might learn the following rule:

(7)

a	a	r	d	a
			1	
			↓	
			2	

An overview of some of the rules learned using 90% of the data for training is given in figure 4. Half of the rules in the top-10 (1, 3, 4, 5, 8) illustrate that the character '*s*' is problematic for hyphenation, as it can be the start of many different onsets, but also can be the final character in many codas. The second rule correctly hyphenates words with the suffix-morpheme *-achtig*, which counts as introducing a boundary for the compound rule. The majority of induced rules are of the 1→2 type. Rule 16 is the highest-ranked rule which makes a different correction. It is of the 2→1 type and corrects the effect of an earlier rule (rule 8 in particular). Rule 27 is the second rule which shifts a hyphen leftward. It recognizes the prefix *ge-*. Note that according to the maximal onset rule, the hyphen would have been placed in front of the *s* to begin with. Thus, this rule also corrects some of the errors introduced by rule 8 (and possibly by other preceding rules).

We performed experiments on various portions of the Celex data. The results are given in figure 5. When trained on only 10% of the available data, TBL learns a relatively small number of highly effective rules. The error rate can be further reduced by using 90% of the data, although the number of induced rules also increases substantially in that case.

The results in figure 5 improve on previous work. The best result of Daelemans and van den Bosch (1992) (98.3% hyphen accuracy) used an exemplar-based learning method, trained on a word list of 19,000 words and using a fixed context window of 7 (i.e. the window only refers to the target character and the three preceding and

Rank	Good	Bad	Score	Rule
1	2918	314	2604	i-st → is-t
2	1890	103	1787	-?achti → ?-achti
3	1666	101	1565	a-st → as-t
4	1411	135	1276	ing-s → ings-
5	714	37	677	u-st → us-t
8	1333	703	630	e-ste → es-te
16	320	26	294	bes-t → be-st
1408	3	0	3	-??eid → ?-?eid

Fig. 4. Rules learned by TBL, trained on 90% of the data. '?' represents an arbitrary character, '+' is the word boundary symbol.

	initial	10%	20%	30%	60%	90%
Number of Induced Rules		264	507	737	1,139	1,409
Hyphen Accuracy	94.16	98.15	98.60	98.82	99.04	99.27
Word Accuracy	85.30	95.34	96.49	97.02	97.59	98.17

Fig. 5. Results of learning hyphenation rules using 10%-90% of the data.

following characters). Vosse (1994) trains a Hidden-Markov Model and a pattern-based hyphenator similar to the system of Liang (1983) on a word list of 190,000 words and achieves 97.8% and 98.2% hyphen accuracy for the Markov Model and pattern-based system respectively.

The results given in figure 5 can be improved upon only slightly by using more context. Using 90% of the data for training, a context of 7 gives a hyphen accuracy of 99.35% (using 1,484 rules) and including information about the absence or presence of other hyphens in the context gives an accuracy of 99.32% (using 1,404 rules).

4.3 Compilation of TBL rules to a FST

TBL rules can be interpreted as finite state transducers (Roche and Schabes, 1997). A single rule corresponds to a transducer which interchanges characters and hyphens or, alternatively, changes digits. Rules learned by TBL are applied to the data in the order in which they are learned. A sequence of TBL rules therefore corresponds to the composition of the individual rule transducers.

The syllable-based hyphenation system transduces an input string into a hyphenated string. TBL rules can be interpreted as rules which correct errors of the base-system by shifting, inserting, or deleting hyphens in specific contexts. Recall that, given the alignment-method used for TBL, changing a '1' into a '2' means that a hyphen has to follow rather than precede the corresponding character. Thus, the first rule induced by TBL corresponds to the following regular expression:

(8) `replace([-,s]:[s,-], [i], [])`

The base system hyphenates *communisme* (*comunism*) as *com-mu-ni-sme*. The regular expression above corrects this to *com-mu-nis-me*. Similar regular expressions

can be given for rules which shift a hyphen two positions rightwards, which shift a hyphen leftward, or which insert or delete a hyphen.

A second method for interpreting the TBL-rules makes more direct use of the alignment-method. The output of the syllable-based system can be transduced easily into a string where each character is preceded by a '0' or a '1', indicating the absence or presence of a hyphen in that position. Thus, *com-mu-ni-sme* would be represented as (9). TBL-rules can now be interpreted as regular expressions for replacing a single digit. Thus, the first rule learned by TBL would correspond to the regular expression in (10).

(9) 0c0o0m1m0u1n0i1s0m0e

(10) `replace(1:2, [i], [s])`

The output of a cascade of such rules is a character string interspersed with digits. The corresponding hyphenated string is obtained by a transducer which deletes `0`, translates a `1` into a hyphen, `[2,C]` into `[C,-]` (for any character `C`), etc.

A finite state transducer implementing the base system and the result of TBL can now be conceptualized as follows:

(11) `hyphenate o introduce_digits o apply_rules o interpret_digits`

The advantage of the second method is that TBL-rules correspond to one character substitutions, whereas the first method introduces more complicated replacement-targets. In practice, we observed that the second method is also less computationally demanding (both in terms of memory required during compilation and in terms of the size of the resulting automaton).

A disadvantage of the second method is that it requires that contexts must refer both to characters and digits. That is, the fourth rule learned by TBL does not correspond to (13) but to (14), where `digit` is the disjunction defined in (12).

(12) `macro(digit, {0, 1 ,2, 3, 9})`

(13) `replace(1:2, [i,n,g], [s])`

(14) `replace(1:2, [i,digit,n,digit,g], [s])`

Rule contexts therefore become very large (i.e. a 5 character context gives rise to a regular expression with a context of length 10). In practice, compilation of such rules is difficult. We therefore also experimented with a rule templates which allowed digits to be included in the rules. The effect of this is that instead of having to insert `digit` in contexts, we now obtain rules where the value of each digit in the context is known (usually, 0). For instance, instead of learning the rule in (14), the system now learns:

(15) `replace(1:2, [i,0,n,0,g], [s])`

A second alternative we expermimented with is treating pairs of digits and characters as a single symbol. This has the advantage that contexts are reduced even further, but has the disadvantage that the alphabet increases to approximately 5 x 26 (i.e. all combinations of a digit used in the alignment and a character).

Rules	no digits		digits		combined	
	S	T	S	T	S	T
25	137	1155	112	704	48	832
50	355	4159	261	2184	115	3163
100	783	9196	611	5628	219	7673
200	1,846	24,019	1,206	11,748	486	20,388

Fig. 6. Number of states S and transitions T for the FST consisting of the composition of N TBL rules using various methods.

Some results for composing N FST's for individual TBL-rules into a single transducer for the initial system, the system with digits in contexts, and the system with combined characters, are given in figure 6. Although the size of the transducer grows approximately linearly with the number of incorporated rules, the overall size of the transducer is nevertheless too large for incorporation of all 1,400 rules induced by TBL. Using the Prolog-based FSA implementation on a 64-bit machine with 1 Gb of memory, we managed to compile maximally 400 rules into a single transducer. To apply the full set of induced rules to new data, the best one can do therefore is compose FST's for up to 400 rules, and use a pipeline architecture to pass the output of one transducer as input to the next.

These results are less encouraging than those of Roche and Schabes (1997). Note, however, that the rule-set they experimented with were the result of applying TBL for part-of-speech tagging (as in Brill (1995)). Their rule set consisted of 280 rules with a context of at most three symbols. If the rule set is much larger, and contains lengthy contexts, compilation may not be feasible in practice.

5 A comparison with TEX

The pattern-based hyphenation method of Liang (1983), incorporated in TEX , uses a word list to derive patterns which indicate legal and illegal hyphenation points. The extraction of patterns is very similar to TBL in that it tries to find the patterns which introduce most legal (or illegal) hyphenation points, while introducing a minimal number of errors. In this section, we present two results which shed light on the relationship between the hyphenation methods presented above and the pattern-based method. We evaluate the accuracy of TEX and the TBL system for Dutch on running text. Next, we present a method for compiling hyphenation patterns into a single FST. We start with an overview of the pattern-based method.

5.1 Hyphenation in TEX

Hyphenation in TEX uses five levels of patterns to determine legal and illegal hyphenation points. Level 1 contains patterns which insert the digit 1, level 2 patterns introduce 2, etc. In the resulting string, odd numbers stand for potential hyphenation points, while even numbers indicate illegal hyphenation points. Tutelaers (1999) illustrates the method with the example in figure 7. First, word boundary markers (.) are added to the word to be hyphenated. Next, level 1 patterns are

Level	Patterns	String
(mark boundaries)		.pijnappel.
1	jn$_1$a $_1$na	.pij$_1$n$_1$appel.
2	i$_2$j j$_2$na $_2$nap	.pi$_2$j$_2$n$_1$appel.
3	$_3$pijn p$_3$p	.$_3$pi$_2$j$_2$n$_1$ap$_3$pel.
4	.p$_4$ jna$_4$ $_4$pp pe$_4$l $_4$l.	.$_3$p$_4$i$_2$j$_2$n$_1$a$_4$p$_3$pe$_4$l.
5	p$_5$p	.$_3$p$_4$i$_2$j$_2$n$_1$a$_4$p$_5$pe$_4$l.
(interpret result)		pijn-ap-pel

Fig. 7. Pattern-based Hyphenation in TEX

applied. A pattern matches if it contains a character-string which matches some part of the word. Application of the pattern means that the corresponding marker is added to the word string. In this case, two patterns apply, introducing two markers. Next, level 2 patterns are applied. Higher level rules override the effects of lower rules, and thus the j$_2$na pattern overwrites the effect of the $_1$na pattern. In the final step, odd numbers are realized as hyphens, while even numbers can simply be discarded. TEX adopts the typographic convention of never hyphenating words after the first or before the penultimate and last character.

The acquisition of patterns follows a procedure which is strikingly similar to TBL. In a first round, patterns are collected which identify a high number of potential hyphenation points, while overgeneralizing minimally. In a second round, patterns are learned which block the erroneuos hyphenation points which are the result of applying the level-1 rules. This process iterates three more times, giving rise to five levels of rules in total.

There are also a few differences between this method and TBL. First, all patterns on a given level can be applied simultaneously to an input string. Ordering of patterns is only relevant between levels. TBL rules, on the other hand, should be applied in the order in which the are acquired. Second, the score of each rule must be above a certain threshold T, where the score of a pattern is computed as follows:

(16) $Score = Good \times G - Bad \times B$

Good and *Bad* are the counts for the number of correct and wrong applications of the pattern to the data, G and B are weights which determine the accuracy of the rule (i.e. by making B much higher than G, accurate rules will score higher than less accurate rules which might correct a larger number of errors). The value of G, B and the threshold T may vary according to level. Furthermore, the maximal length of the patterns considered may vary per level. A typical set-up appears to be one where the context is short and the threshold is high initially, while for higher levels, the context is longer, the threshold is lower, and the value of B is much higher than that of G. For TBL, there is no fixed number of levels, and parameters can only be set globally. Finally, Liang's method learns both legal and illegal hyphenation points. The TBL method combines the effect of identifying legal and illegal hyphenation points by learning rules which shift a hyphen. Shifting a hyphen implicitly classifies the original position as an illegal hyphenation point and the target position as a legal hyphenation point.

	TEX	TBL
Mistake	3	36
Missing hyphen	12	25
Total	15	61
Estimated Hyphen Accuracy	99.8	99.1
Estimated Word Accuracy	99.8	99.5

Fig. 8. Comparing TEX and TBL on 11,641 words (containing 7,024 potential hyphenation points) of running text.

Hyphenation patterns for Dutch were created on the basis of the same Celex word list used in the previous sections. As a consequence of a spelling reform in 1996[8] the construction of patterns has been redone recently (Tutelaers, 1999). The new Dutch pattern file contains a total of 8,870 patterns, where patterns are maximally 8 characters long.

5.2 Hyphen accuracy on running text

Training and testing on word lists for which the correct hyphenation is known, is convenient but also artificial, as the characteristics of running text differ sharply from that of a word list. The average word length for the Celex list is 10.85 while it is approximately 4.5 for Dutch running text. The average number of hyphenation points per word is 2.5 for Celex, but only 0.6 for the fragment of running text described below. Evaluation on running text may therefore give results which differ from the results for a word list. Note also that, as the TEX patterns for Dutch were derived using 100% of the Celex list, evaluation on data held out from the word list is impossible.

A test set was created consisting of 1,000 sentences of newspaper text (selected from the CD-ROM version of *de Volkskrant*, 1997). This set contained a total of 11,641 words. Approximately 10% of the word types in the test set were not included in Celex. 4,219 of the words were hyphenated by TEX when forced to produce all hyphenation points, giving rise to a total of 7,024 hyphens. Instead of checking all results, we only inspected those cases where TEX and the TBL system disagreed. This was the case for 83 words, 9 of which were typo's which were discarded. The results of the comparison are given in figure 8.

Both systems make very few mistakes, but TEX does considerably better than the TBL system. The difference seems to be due mostly to the fact that TEX uses well over 8,000 patterns, where the TBL system uses only 1,400 patterns. On the other hand, the TBL system only corrects the output of the syllable-based system, and thus is expected to require less rules. Both systems used the same data for acquisition of rules or patterns, but the TEX system uses rules with more context (up to 8 characters). Furthermore, it seems the careful tuning of parameters guiding

[8] See `www.minocw.nl/spelling`.

the acquisition of patterns at each level allows the pattern-based system to acquire more effective patterns than the generic TBL method.

It is tempting to think of the results in figure 8 as indications of accuracy. Note, however, that, apart from the fact that the test set was small, the systems have only been compared on words where there was disagreement. It is possible therefore that the test results contain errors which have gone unnoticed because both systems made the same mistake.

5.3 Compilation of patterns

Although Liang (1983) uses a finite-state method to store and apply patterns efficiently, patterns are not actually compiled into a transducer which takes a word as input and produces the hyphenation according to the patterns as output. The construction of such a transducer is proposed in Kaplan and Karttunen (1998).

A hyphenating FST for hyphenation patterns can be constructed as the composition of the following sequence of transducers. First, 0's are introduced between all characters in the input string. Next, level 1 rules are applied. Applying a level 1 rule means that a 0 is replaced by a 1 in the relevant context. For instance, the pattern in (17) corresponds to the regular expression in (18). Note that contexts must be interspersed with digits up to the corresponding level of rule application. Application of all level 1 rules can be achieved by composing (in arbitrary order) all regular expressions for the individual patterns into a single transducer. Higher level rules must be able to overwrite the effect of earlier, lower level rules. This can be achieved by interpreting a level 2 pattern such as (19) as the regular expression in (20). This pattern substitutes both 0's and 1's by 2's in the relevant context. The level 1 transducer is composed with the transducer for all level 2 rules, etc. Finally, odd numbers are realized as hyphens and even numbers are discarded. Schematically, we have the FST defined by the regular expression in (21).

(17) jn_1a

(18) `replace(0:1,[j,{0,1},n],[a])`

(19) j_2na

(20) `replace({0,1}:2,[j],[n,{0,1,2},a])`

(21) `insert_digits(0) o patterns(1) o patterns(2) o patterns(3) o patterns(4) o patterns(5) o digits2hyphens`

The construction of an FST for hyphenation patterns is not unlike the construction of an FST for TBL-rules. Yet, compilation of more than 8,000 patterns into a single transducer turned out to be feasible. The result has over 600,000 transitions, and gives rise to a binary of 15 Mb.[9]

[9] One reviewer points out that a similar compilation was carried out for English (4753 patterns) by Lauri Karttunen, using the Xerox finite-state tools. The resulting FST contains 39.875 states and 683.440 arcs.

It is not clear why compilation of hyphenation patterns is 'easier' than compilation of TBL patterns for the same task. One possible explanation could be the fact that hyphenation patterns are ordered in blocks, where the application of rules within a block is not ordered, whereas TBL-rules have to be applied in order of acquisition. The latter suggests more interaction between rules than the former, which might have an effect on the complexity of the corresponding automaton. Strict ordering of TBL-rules is necessary because some rule may provide input for a later rule (*feeding*), or because some rule blocks the application of a later rule (*bleeding*). If no feeding or bleeding relationship exists between rules, they can be applied in any order. We computed these relationships for the hyphenation rules learned by TBL, and concluded that 19 blocks of rules (where the order of application within a block is irrelevant) are needed to account for all feeding and bleeding relationships. Thus, it seems that the interaction between rules is indeed more complex in the TBL system than in the pattern-based system.

6 Conclusions

In this paper we have presented two finite-state methods for hyphenation as well as a method for compiling an existing method into a finite-state transducer. The use of the *replace*-operator was crucial in all methods. In particular, the syllable-based method capitalizes both on the fact that *replace* performs a longest-match replacement and on the fact that it preforms replacements from left to right. The method for compiling TBL-rules into a single transducer proposed by Roche and Schabes (1997) turned out to be impractical for the large number of rules required for accurate hyphenation. Hyphenation patterns as used by TEX on the other hand, proved to be compilable into a single FST, albeit a large one. It is unclear why TBL-rules are 'harder' in this respect than patterns.

Accuracy of hyphenation after applying TBL turned out to be higher than that of previous systems for Dutch, that were trained and evaluated on word lists. The numerous hyphenation patterns for Dutch used by TEX are even more accurate. The acquisition of TBL-rules and hyphenation patterns is strikingly similar. By using larger context windows for patterns, and by tuning the acquisition of patterns carefully to the hyphenation problem, TEX is able to induce a much larger set of patterns, which turns out to have a positive effect on accuracy.

Acknowledgements

This paper has benefitted from comments from reviewers and participants of the workshop on Finite State Methods for Natural Language Processing 2001, held during ESSLLI XII (Helsinki, 2001). Following suggestions by Lauri Karttunen en Theo Jansen, the method for compiling TEX patterns into a FST was implemented together with Gertjan van Noord, partially during the workshop. Gertjan van Noord also provided useful feedback on all other aspects of the paper.

References

Baayen, R. H., R. Piepenbrock, and H. van Rijn. 1993. *The CELEX Lexical Database (CD-ROM)*. UPenn, Philadelphia, PA: Linguistic Data Consortium.

Brandt Corstius, H. 1978. *Computer-taalkunde*. Muiderberg: Coutinho.

Brill, Eric. 1995. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21:543–566.

Daelemans, Walter and Antal van den Bosch. 1992. Generalization performance of backpropagation learning on a syllabification task. In *Connectionism and Natural Language Processing. Proceedings Third Twente Workshop on Language Technology*, pages 27–38.

Gerdemann, Dale and Gertjan van Noord. 1999. Transducers from rewrite rules with backreferences. In *Proceedings of the Ninth Conference of the European Chapter of the Association for Computational Linguistics*, pages 126–133, Bergen.

Gerdemann, Dale and Gertjan van Noord. 2000. Approximation and exactness in finite state optimality theory. In Jason Eisner, Lauri Karttunen, and Alain Thériault, editors, *Finite-State Phonology. Proceedings of the Fifth Workshop of the ACL SPecial Interest Group in Computational Phonology*, pages 34–45, Luxembourg.

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

Kaplan, Ronald M. and Lauri J. Karttunen. 1998. Finite-state encoding system for hyphenation rules. United States Patent 5,737,621.

Karttunen, Lauri. 1995. The replace operator. In *33th Annual Meeting of the Association for Computational Linguistics*, pages 16–23, Boston, Massachusetts.

Karttunen, Lauri. 1998. The proper treatment of optimality in computational phonology. In Lauri Karttunen, editor, *FSMNLP'98: International Workshop on Finite State Methods in Natural Language Processing*. Association for Computational Linguistics, Somerset, New Jersey, pages 1–12.

Karttunen, Lauri. 2001. Applications of finite-state technology in natural language processing. In S. Yu and A. Paun, editors, *Implementation and Application of Automata*, volume 2088 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, Germany, pages 34–46.

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

Ngai, Grace and Radu Florian. 2001. Transformation-based learning in the fast lane. In *Proceedings of the second conference of the North American chapter of the ACL*, pages 40–47, Pittsburgh.

Roche, Emmanuel and Yves Schabes. 1997. Deterministic part-of-speech tagging with finite-state transducers. In Emmanuel Roche and Yves Schabes, editors, *Finite state language processing*. MIT Press, Cambridge, Mass., pages 205–239.

Sojka, Petr. 1995. Notes on compound word hyphenation in TEX. In *TUGboat 1995*. Proceedings of the 16th annual meeting of the Tex users group.

Tutelaers, P. T. H. 1999. Afbreken in TEX, hoe werkt dat nou? available at ftp://ftp.tue.nl/pub/tex/afbreken.

van Noord, Gertjan. 1997. FSA Utilities: A toolbox to manipulate finite-state automata. In Darrell Raymond, Derick Wood, and Sheng Yu, editors, *Automata Implementation*. Springer Verlag. Lecture Notes in Computer Science 1260.

van Noord, Gertjan and Dale Gerdemann. 2001. Finite state transducers with predicates and identity. *Grammars*, 4(3).

Vosse, Theo. 1994. *The Word Connection*. Ph.D. thesis, Rijksuniversiteit Leiden.