Automatic Acquisition of Two-Level Morphological Rules

Pieter Theron
Dept. of Computer Science
Stellenbosch University
Stellenbosch 7600, South Africa
ptheron@cs.sun.ac.za

Ian Cloete
Dept. of Computer Science
Stellenbosch University
Stellenbosch 7600, South Africa
ian@cs.sun.ac.za

Abstract

We describe and experimentally evaluate a complete method for the automatic acquisition of two-level rules for morphological analyzers/generators. The input to the system is sets of source-target word pairs, where the target is an inflected form of the source. There are two phases in the acquisition process: (1) segmentation of the target into morphemes and (2) determination of the optimal two-level rule set with minimal discerning contexts. In phase one, a minimal acyclic finite state automaton (AFSA) is constructed from string edit sequences of the input pairs. *Segmentation* of the words into morphemes is achieved through viewing the AFSA as a directed acyclic graph (DAG) and applying heuristics using properties of the DAG as well as the elementary edit operations. For phase two, the determination of the *optimal rule set* is made possible with a novel representation of rule contexts, with morpheme boundaries added, in a new DAG. We introduce the notion of a *delimiter edge*. Delimiter edges are used to select the correct two-level *rule type* as well as to extract minimal discerning rule contexts from the DAG. Results are presented for English adjectives, Xhosa noun locatives and Afrikaans noun plurals.

1 Introduction

Computational systems based on the two-level model of morphology (Koskenniemi, 1983) have been remarkably successful for many languages (Sproat, 1992). The language specific information of such a system is stored as

1. a morphotactic description of the words to be processed as well as
2. a set of two-level morphonological (or spelling) rules.

Up to now, these two components had to be coded largely by hand, since no automated method existed to acquire a set of two-level rules for input source-target word pairs. To hand-code a 100% correct rule set from word pairs becomes almost impossible when a few hundred pairs are involved. Furthermore, there is no guarantee that such a hand coded lexicon does not contain redundant rules or rules with too large contexts. The usual approach is rather to construct general rules from small subsets of the input pairs. However, these general rules usually allow overrecognition and overgeneration — even on the subsets from which they were inferred.

Simons (Simons, 1988) describes methods for studying morphophonemic alternations (using *annotated* interlinear text) and Grimes (Grimes, 1983) presents a program for discovering affix positions and cooccurrence restrictions. Koskenniemi (Koskenniemi, 1990) provides a sketch of a discovery procedure for phonological two-level rules. Golding and Thompson (Golding and Thompson, 1985) and Wothke (Wothke, 1986) present systems to automatically calculate a set of word-formation rules. These rules are, however, ordered one-level rewrite rules and not unordered two-level rules, as in our system. Kuusik (Kuusik, 1996) also acquires ordered one-level rewrite rules, for stem sound changes in Estonian. Daelemans *et al.* (Daelemans *et al.*, 1996) use a general symbolic machine learning program to acquire a decision tree for matching Dutch nouns to their correct diminutive suffixes. The input to their process is the syllable structure of the nouns and a given set of five suffix allomorphs. They do not learn rules for possible sound changes. Our process automatically acquires the necessary two-level sound changing rules for prefix and suffix allomorphs, as well as the rules for stem sound changes. Connectionist work on the acquisition of morphology has been more concerned with implementing psychologically motivated models, than with acquisition of rules for a practical system ((Sproat, 1992, p.216) and (Gasser, 1994)).

The contribution of this paper is to present a complete method for the automatic acquisition of an optimal set of two-level rules (i.e. the second component above) for source-target word pairs. It is assumed that the target word is formed from the source through the addition of a prefix and/or a suffix[1]. Furthermore, we show how a partial acquisition of the morphotactic description (component one) results as a by-product of the rule-acquisition process. For example, the morphotactic description of the target word in the input pair

[1]

Source	Target
happy	*happier*

is computed as

[2]

$$happier \quad = \quad happy + er$$

The right-hand side of this morphotactic description is then mapped on the left-hand side,

[3]

$$\begin{array}{cccccccc} h & a & p & p & y & + & e & r \\ h & a & p & p & i & 0 & e & r \end{array}$$

For this example the two-level rule

[4]

$$y{:}i \Leftrightarrow p{:}p \ \_$$

can be derived. These processes are described in detail in the rest of the paper: Section 2 provides an overview of the two-level rule formalism, Section 3 describes the acquisition of morphotactics through segmentation and Section 4 presents the method for computing the optimal two-level rules. Section 5 evaluates the experimental results and Section 6 summarizes.

2 Two-level Rule Formalism

Two-level rules view a word as having a *lexical* and a *surface* representation, with a correspondence between them (Antworth, 1990), e.g.:

[5]

$$\begin{array}{lcccccccc} \text{Lexical:} & h & a & p & p & y & + & e & r \\ \text{Surface:} & h & a & p & p & i & 0 & e & r \end{array}$$

Each pair of lexical and surface characters is called a *feasible pair*. A feasible pair can be written as *lexical-character:surface-character*. Such a pair is called a *default pair* when the lexical character and surface character are identical (e.g. *h:h*). When the lexical and surface character differ, it is called a *special pair* (e.g. *y:i*). The null character (0) may appear as either a lexical character (as in *+:0*) or a surface character, but not as both.

[1]Non-linear operations (such as infixation) are not considered here, since the basic two-level model deals with it in a round-about way. We can note that extensions to the basic two-level model have been proposed to handle non-linear morphology (Kiraz, 1996).

Two-level rules have the following syntax (Sproat, 1992, p.145):

[6]

$$CP \ \mathbf{op} \ LC \ \_ \ RC$$

CP (*correspondence part*), LC (*left context*) and RC (*right context*) are regular expressions over the alphabet of feasible pairs. In most, if not all, implementations based on the two-level model, the correspondence part consists of a single special pair. We also consider only single pair CPs in this paper. The operator **op** is one of four types:

1. Exclusion rule: $a{:}b \ /\!\Leftarrow \ LC \ \_ \ RC$
2. Context restriction rule: $a{:}b \Rightarrow LC \ \_ \ RC$
3. Surface coercion rule: $a{:}b \Leftarrow LC \ \_ \ RC$
4. Composite rule: $a{:}b \Leftrightarrow LC \ \_ \ RC$

The exclusion rule ($/\!\Leftarrow$) is used to prohibit the application of another, too general rule, in a particular subcontext. Since our method does not overgeneralize, we will consider only the $\Rightarrow$, $\Leftarrow$ and $\Leftrightarrow$ rule types.

3 Acquisition of Morphotactics

The morphotactics of the input words are acquired by (1) computing the string edit difference between each source-target pair and (2) merging the edit sequences as a minimal acyclic finite state automaton. The automaton, viewed as a DAG, is used to segment the target word into its constituent morphemes.

3.1 Determining String Edit Sequences

A string edit sequence is a sequence of *elementary operations* which change a source string into a target string (Sankoff and Kruskal, 1983, Chapter 1). The elementary operations used in this paper are single character *deletion* (DELETE), *insertion* (INSERT) and *replacement* (REPLACE). We indicate the copying of a character by NOCHANGE. A cost is associated with each elementary operation. Typically, INSERT and DELETE have the same (positive) cost and NOCHANGE has a cost of zero. REPLACE could have the same or a higher cost than INSERT or DELETE. Edit sequences can be ranked by the sum of the costs of the elementary operations that appear in them. The interesting edit sequences are those with the lowest total cost. For most word pairs, there are more than one edit sequence (or mapping) possible which have the same minimal total cost. To select a single edit sequence which will most likely result in a correct segmentation, we added a morphology-specific heuristic to a general string edit algorithm (Vidal et al., 1995). This heuristic always selects an edit sequence containing two subsequences which identify prefix-root and root-suffix boundaries. The heuristic depends on the elementary operations being limited only to INSERT, DELETE and NOCHANGE, i.e. no REPLACEs are allowed. We assume that the target word contains more morphemes than the source word. It therefore follows that there are more INSERTs than DELETEs in an edit sequence. Furthermore, the letters forming the morphemes of the target word appear only as the right-hand components of INSERT operations. Consider the edit sequence to change the string *happy* into the string *unhappier*:

[7]

0:u	INSERT
0:n	INSERT
h:h	NOCHANGE
a:a	NOCHANGE
p:p	NOCHANGE
p:p	NOCHANGE
y:0	DELETE
0:i	INSERT
0:e	INSERT
0:r	INSERT

Note that the prefix *un-* as well as the suffix -*er* consist only of INSERTs. Furthermore, the prefix–root morpheme boundary is associated with an INSERT followed by a NOCHANGE and the root–suffix boundary by a NOCHANGE–DELETE–INSERT sequence. In general, the prefix–root boundary is just the reverse of the root–suffix boundary, i.e. INSERT–DELETE–NOCHANGE, with the DELETE operation being optional. The heuristic resulting from this observation is a bias giving highest precedence to INSERT operations, followed by DELETE and NOCHANGE, in the first half of the edit sequence. In the second half, the precedence is reversed.

3.2 Merging Edit Sequences

A single source-target edit sequence may contain spurious INSERTs which are not considered to form part of a morpheme. For example, the *0:i* insertion in Example 7 should not contribute to the suffix -*er* to form *-ier*, since *-ier* is an allomorph of *-er*. To combat these spurious INSERTs, all the edit sequences for a set of source-target words are merged as follows: A minimal *acyclic finite state automaton* (AFSA) is constructed which accepts all and only the edit sequences as input strings. This AFSA is then viewed as a DAG, with the elementary edit operations as edge labels. For each edge a count is kept of the number of different edit sequences which pass through it. A path segment in the DAG consisting of one or more INSERT operations having a similar count, is then considered to be associated with a morpheme in the target word. The *0:e 0:r* INSERT sequence associated with the *-er* suffix appears more times than the *0:i 0:e 0:r* INSERT sequence associated with the *-ier* suffix, even in a small set of adjectively-related source-target pairs. This means that there is a rise in the edge counts from *0:i* to *0:e* (indicating a root–suffix boundary), while *0:e* and *0:r* have similar frequency counts. For prefixes a fall in the edge frequency count of an INSERT sequence indicates a prefix–root boundary.

To extract the morphemes of each target word, every path through the DAG is followed and only the target-side of the elementary operations serving as edge labels, are written out. The null characters (*0*) on the target-side of DELETEs are ignored while the target-side of INSERTs are only written if their frequency counts indicate that they are *not* sporadic allomorph INSERT operations. For Example 7, the following morphotactic description results:

[8]

Target Word	= Prefix	+ Source	+ Suffix
unhappier	= *un*	+ *happy*	+ *er*

Phase one can segment only one layer of affix additions at a time. However, once the morpheme boundary markers (+) have been inserted, phase two should be able to acquire the correct two-level rules for an arbitrary number of affix additions: *prefix1+prefix2+... +root+suffix1+suffix2+...*.

4 Acquiring Optimal Rules

To acquire the optimal rules, we first determine the full length lexical-surface representation of each word pair. This representation is required for writing two-level rules (Section 2). The morphotactic descriptions from the previous section provide source-target input pairs from which *new* string edit sequences are computed: The right-hand side of the morphotactic description is used as the source and the left-hand side as the target string. For instance, Example 8 is written as:

[9]

Source: *un+happy+er*
Target: *unhappier*

The edit sequence

[10]

u:u n:n +:0 h:h a:a p:p p:p y:i +:0 e:e r:r

maps the source into the target and provides the lexical and surface representation required by the two-level rules:

[11]

Lexical:	*u*	*n*	+	*h*	*a*	*p*	*p*	*y*	+	*e*	*r*
Surface:	*u*	*n*	0	*h*	*a*	*p*	*p*	*i*	0	*e*	*r*

The REPLACE elementary string edit operations (e.g. *y:i*) are now allowed since the morpheme boundary markers (+) are already present in the source string. REPLACEs allow shorter edit sequences to be computed, since one REPLACE does the same work as an adjacent INSERT–DELETE pair. REPLACE, INSERT and DELETE have the same associated cost and NOCHANGE has a cost of zero. The morpheme boundary marker (+) is always mapped to the null character (0), which makes for linguistically more understandable mappings. Under these conditions, the selection of any minimal cost string edit mapping provides an acceptable lexical-surface representation[2].

To formulate a two-level rule for the source-target pair *happy–unhappier*, we need a correspondence pair (CP) and a rule type (op), as well as a left context (LC) and a right context (RC) (see Section 2). Rules need only be coded for *special pairs*, i.e. INSERTs, DELETEs or REPLACEs. The only special pair in the above example is *y:i*, which will be the CP of the rule. Now the question arises as to *how large* the context of this rule must be? It should be large enough to uniquely specify the positions in the lexical-surface input stream where the rule is applied. On the other hand, the context should not be too large, resulting in an overspecified context which prohibits the application of the rule to unseen, but similar, words. Thus to make a rule as general as possible, its context (LC and RC) should be as short as possible[3]. By inspecting the edit sequence in Example 10, we see that *y* changes into *i* when *y* is preceded by a *p:p*, which serves as our first attempt at a (left) context for *y:i*. Two questions must be asked to determine the correct rule type to be used (Antworth, 1990, p.53):

Question 1 Is E the only environment in which L:S is allowed?

Question 2 Must L always be realized as S in E?

The term *environment* denotes the combined left and right contexts of a special pair. E in our example is "*p:p_*", L is *y* and S is *i*. Thus the answer to question one is *true*, since *y:i* only occurs after *p:p* in our example. The answer to question two is also *true*, since *y* is always realized as *i* after a *p:p* in the above edit sequence. Which rule type to use is gleaned from Table 1. Thus, to continue our example, we should use the composite rule type (⇔):

[12]

$$y{:}i \Leftrightarrow p{:}p\ \_$$

[2]Our assumption is that such a minimal cost mapping will lead to an optimal rule set. In most (if not all) of the examples seen, a minimal mapping was also intuitively acceptable.

[3]If abstractions (e.g. *sets* such as *V* denoting vowels) over the regular pairs are introduced, it will not be so simple to determine what is "a more general context". However, current implementations require abstractions to be explicitly instantiated during the compilation process ((Karttunen and Beesley, 1992, pp.19–21) and (Antworth, 1990, pp.49-50)) . Thus, with the current state of the art, abstractions serve only to make the rules more readable.

Q1	Q2	op
false	false	none
true	false	⇒
false	true	⇐
true	true	⇔

Table 1: Truth table to select the correct rule type.

This example shows how to go about coding the set of two-level rules for *a single* source-target pair. However, this soon becomes a tedious and error prone task when the number of source-target pairs increases, due to the complex interplay of rules and their contexts.

4.1 Minimal Discerning Rule Contexts

It is important to acquire the minimal discerning context for each rule. This ensures that the rules are as general as possible (to work on unseen words as well) and prevents rule conflicts. Recall that one need only code rules for the special pairs. Thus it is necessary to determine a rule type with associated minimal discerning context for each occurrence of a special pair in the final edit sequences. This is done by comparing all the possible contiguous[4] contexts of a special pair against all the possible contexts of all the other feasible pairs. To enable the computational comparison of the growing left and right contexts around a feasible pair, we developed a "mixed-context" representation. We call the particular feasible pair for which a mixed-context is to be constructed, a *marker pair* (MP), to distinguish it from the feasible pairs in its context. The mixed-context representation is created by writing the first feasible pair to the left of the marker pair, then the first right-context pair, then the second left-context pair and so forth:

[13]

LC1, RC1, LC2, RC2, LC3, RC3, ..., **MP**

The marker pair at the end serves as a label. Special symbols indicate the start (SOS) and end (EOS) of an edit sequence. If, say, the right-context of a MP is shorter than the left-context, an out-of-bounds symbol (OOB) is used to maintain the mixed-context format. For example the mixed-context of *y:i* in the edit sequence in Example 10, is represented as:

[14]

p:p, +:0, p:p, e:e, a:a, r:r, h:h, EOS, +:0, OOB, n:n, OOB, u:u, SOS, OOB, y:i

The common prefixes of the mixed-contexts are merged by constructing a minimal AFSA which accepts all and only these mixed-context sequences.

[4]A two-level rule requires a contiguous context.

The transitions (or edges, when viewed as a DAG) of the AFSA are labeled with the feasible pairs and special symbols in the mixed-context sequence. There is only one final state for this minimal AFSA. Note that all and only the terminal edges leading to this final state will be labeled with the marker pairs, since they appear at the end of the mixed-context sequences. More than one terminal edge may be labeled with the same marker pair. All the possible (mixed) contexts of a specific marker pair can be recovered by following every path from the root to the terminal edges labeled with that marker pair. If a path is traversed only up to an intermediate edge, a shortened context surrounding the marker pair can be extracted. We will call such an intermediate edge a *delimiter edge*, since it delimits a shortened context. For example, traversing the mixed context path of *y:i* in Example 14 up to *e:e* would result in the (unmixed) shortened context:

[15]

p:p p:p _ +:0 e:e

From the shortened context we can write a two-level rule

[16]

y:i **op** *p:p p:p _ +:0 e:e*

which is more general than a rule using the full context:

[17]

y:i **op** *SOS u:u n:n h:h a:a p:p p:p _ +:0 e:e r:r EOS*

For each marker pair in the DAG which is also a special pair, we want to find those delimiter edges which produce the shortest contexts providing a *true* answer to at least one of the two rule type decision questions given above. The mixed-context prefix-merged AFSA, viewed as a DAG, allow us to rephrase the two questions in order to find answers in a procedural way:

Question 1 Traverse all the paths from the root to the terminal edges labeled with the marker pair *L:S*. Is there an edge e_1 in the DAG which all these paths have in common? If so, then question one is *true* for the environment E constructed from the shortened mixed-contexts associated with the path prefixes delimited by e_1.

Question 2 Consider the terminal edges which have the same L-component as the marker pair *L:S* and which are reachable from a common edge e_2 in the DAG. Do all of these terminal edges also have the same S-component as the marker pair? If so, then question two is *true* for the environment E constructed from the shortened mixed-contexts associated with the path prefixes delimited by e_2.

For each marker pair, we traverse the DAG and mark the delimiter edges *nearest* to the root which allow a *true* answer to either question one, question two or both (i.e. $e_1 = e_2$). This means that each path from the root to a terminal edge can have at most three marked delimiter edges: One delimiting a context for a $\Rightarrow$ rule, one delimiting a context for a $\Leftarrow$ rule and one delimiting a context for a $\Leftrightarrow$ rule. The marker pair used to answer the two questions, serves as the correspondence part (Section 2) of the rule. To continue with Example 14, let us assume that the DAG edge labeled with *e:e* is the closest edge to the root which answers *true* only to question one. Then the $\Rightarrow$ rule is indicated:

[18]

y:i $\Rightarrow$ *p:p p:p _ +:0 e:e*

However, if the edge labeled with *r:r* answers true to both questions, we prefer the composite rule ($\Leftrightarrow$) associated with it although this results in a larger context:

[19]

y:i $\Leftrightarrow$ *a:a p:p p:p _ +:0 e:e r:r*

The reasons for this preference are that the $\Leftrightarrow$ rule

- provides a more precise statement about the applicable environment of the rule and it
- seems to be preferred in systems designed by linguistic experts.

Furthermore, from inspecting examples, a delimiter edge indicating a $\Rightarrow$ rule generally delimits the shortest contexts, followed by the delimiter for $\Leftrightarrow$ and the delimiter for $\Leftarrow$. The shorter the selected context, the more generally applicable is the rule. We therefore select only one rule per path, in the following preference order: (1) $\Leftrightarrow$, (2) $\Rightarrow$ and (3) $\Leftarrow$. Note that any of the six possible precedence orders would provide an accurate analysis and generation of the pairs used for learning. However, our suggested precedence seems to strike the best balance between over- or underrecognition and over- or undergeneration when the rules would be applied to *unseen* pairs.

The mixed-context representation has one obvious drawback: If an optimal rule has only a left or only a right context, it cannot be acquired. To solve this problem, two additional minimal AFSAs are constructed: One containing only the left context information for all the marker pairs and one containing only the right context information. The same process is then followed as with the mixed contexts. The final set of rules is selected from the output of all three the AFSAs: For each special pair

1. we select any of the $\Leftrightarrow$ rules with the shortest contexts of which the special pair is the left-hand side, or

2. if no ⇔ rules were found, we select the shortest ⇐ and ⇒ rules for each occurrence of the special pair. They are then merged into a single ⇔ rule with disjuncted contexts.

The rule set learned is complete since all possible combinations of marker pairs, rule types and contexts are considered by traversing all three DAGs. Furthermore, the rules in the set have the shortest possible contexts, since, for a given DAG, there is only one delimiter edge closest to the root for each path, marker pair and rule type combination.

5 Results and Evaluation

Our process works correctly for examples given in (Antworth, 1990). There were two incorrect segmentations in the twenty one adjective pairs given on page 106. It resulted from an incorrect string edit mapping of *(un)happy* to *(un)happily*. For the suffix, the sequence ... *0:i 0:l y:y* was generated instead of the sequence ... *y:0 0:i 0:l 0:y*. The reason for this is that the root word and the inflected form end in the same letter (*y*) and one NOCHANGE (*y:y*) has a lower cost than a DELETE (*y:0*) plus an INSERT (*0:y*). The acquired segmentation for the 21 pairs, with the suffix segmentation of *(un)happily* manually corrected, is:

[20]

Target	=	Prefix +	Source	+ Suffix
bigger	=		big	+ er
biggest	=		big	+ est
unclear	=	un +	clear	
unclearly	=	un +	clear	+ ly
unhappy	=	un +	happy	
unhappier	=	un +	happy	+ er
unhappiest	=	un +	happy	+ est
unhappily	=	un +	happy	+ ly
unreal	=	un +	real	
cooler	=		cool	+ er
coolest	=		cool	+ est
coolly	=		cool	+ ly
clearer	=		clear	+ er
clearest	=		clear	+ est
clearly	=		clear	+ ly
redder	=		red	+ er
reddest	=		red	+ est
really	=		real	+ ly
happier	=		happy	+ er
happiest	=		happy	+ est
happily	=		happy	+ ly

From these segmentations, the morphotactic component (Section 1) required by the morphological analyzer/generator is generated with uncomplicated text-processing routines. Three correct ⇔ rules, including two gemination rules, resulted for these twenty one pairs[5]:

[21]

0:d	⇔	d:d _ +:0
0:g	⇔	g:g _ +:0
y:i	⇔	_ +:0

To better illustrate the complexity of the rules that can be learned automatically by our process, consider the following set of fourteen Xhosa noun-locative pairs:

[22]

Source Word	→	Target Word
inkosi	→	enkosini
iinkosi	→	ezinkosini
ihashe	→	ehasheni
imbewu	→	embewini
amanzi	→	emanzini
ubuchopho	→	ebucotsheni
ilizwe	→	elizweni
ilanga	→	elangeni
ingubo	→	engubeni
ingubo	→	engutyeni
indlu	→	endlini
indlu	→	endlwini
ikhaya	→	ekhayeni
ikhaya	→	ekhaya

Note that this set contains ambiguity: The locative of *ingubo* is either *engubeni* or *engutyeni*. Our process must learn the necessary two-level rules to map *ingubo* to *engubeni* and *engutyeni*, as well as to map both *engubeni* and *engutyeni* in the other direction, i.e. to *ingubo*. Similarly, *indlu* and *ikhaya* each have two different locative forms. Furthermore, the two source words *inkosi* and *iinkosi* (the plural of *inkosi*) differ only by a prefixed *i*, but they have different locative forms. This small difference between source words provides an indication of the sensitivity required of the acquisition process to provide the necessary discerning information to a two-level morphological processor. At the same time, our process needs to cope with possibly radical modifications between source and target words. Consider the mapping between *ubuchopho* and its locative *ebucotsheni*. Here, the only segments which stay the same from the source to the target word, are the three letters *−buc−*, the letter *−o−* (the deletion of the first *−h−* is correct) and the second *−h−*.

The target words are correctly segmented during phase one as:

[5]The results in this paper were verified on the two-level processor PC-KIMMO (Antworth, 1990). The two-level rule compiler KGEN (developed by Nathan Miles) was used to compile the acquired rules into the state tables required by PC-KIMMO. Both PC-KIMMO and KGEN are available from the Summer Institute of Linguistics.

[23]

Target	=	Prefix +	Source	+ Suffix
enkosini	=	e +	inkosi	+ ni
ezinkosini	=	e +	iinkosi	+ ni
ehasheni	=	e +	ihashe	+ ni
embewini	=	e +	imbewu	+ ni
emanzini	=	e +	amanzi	+ ni
ebucotsheni	=	e +	ubuchopho	+ ni
elizweni	=	e +	ilizwe	+ ni
elangeni	=	e +	ilanga	+ ni
engubeni	=	e +	ingubo	+ ni
engutyeni	=	e +	ingubo	+ ni
endlini	=	e +	indlu	+ ni
endlwini	=	e +	indlu	+ ni
ekhayeni	=	e +	ikhaya	+ ni
ekhaya	=	e +	ikhaya	

Note that the prefix *e+* is computed for all the input target words, while all but *ekhaya* (a correct alternative of *ekhayeni*) have *+ni* as a suffix.

From this segmented data, phase two correctly computes 24 minimal context rules:

[24]

0:e	⇔	o:y +:0 _ n:n
0:i	⇔	u:w +:0 _ n:n
0:s	⇔	p:t _ h:h
+:0	⇐	e:e _
+:0	⇐	o:y _
+:0	⇐	u:w _
+:0	⇐	_ n:n
a:0	⇔	_ m:m
a:e	⇔	_ +:0 n:n
b:t	⇔	_ o:y
h:0	⇔	_ o:o
i:0	⇐	+:0 _ n:n
i:0	⇐	_ h:h
i:0	⇐	_ k:k
i:0	⇐	_ l:l
i:0	⇐	_ m:m
i:0	⇒	+:0 _
i:z	⇔	_ i:i
o:e	⇔	_ +:0 n:n
o:y	⇔	b:t _
p:t	⇔	o:o _
u:0	⇔	+:0 _ b:b
u:i	⇔	_ +:0 n:n
u:w	⇔	l:l _ +:0 0:i

The ⇐ and ⇒ rules of a special pair can be merged into a single ⇔ rule. For example the four rules above for the special pair *+:0* can be merged into

[25]

$$+{:}0 \;\Leftrightarrow\; e{:}e\ \_ \mid o{:}y\ \_ \mid u{:}w\ \_ \mid \_\ n{:}n$$

because both the two questions becomes *true* for the disjuncted environment $e{:}e\ \_ \mid o{:}y\ \_ \mid u{:}w\ \_ \mid \_\ n{:}n$. The vertical bar ("|") is the traditional two-level notation which indicate the disjunction of two (or more) contexts. The five ⇐ rules and the single ⇒ rule of the special pair *i:0* in Example 24 can be merged in a similar way. In this instance, the context of the ⇒ rule (*+:0 _*) needs to be added to some of the contexts of the ⇐ rules of *i:0*. The following ⇔ rule results:

[26]

$$i{:}0 \Leftrightarrow +{:}0\ \_\ n{:}n \mid +{:}0\ \_\ h{:}h \mid +{:}0\ \_\ k{:}k \mid +{:}0\ \_\ l{:}l \mid +{:}0\ \_\ m{:}m$$

In this way the 24 rules are reduced to a set of 16 rules which contain only a single ⇔ rule for each special pair. This merged set of 16 two-level rules analyze and generate the input word pairs 100% correctly.

The next step was to show the feasibility of automatically acquiring a minimal rule set for a wide coverage parser. To get hundreds or even thousands of input pairs, we implemented routines to extract the lemmas ("head words") and their inflected forms from a machine-readable dictionary. In this way we extracted 3935 Afrikaans noun-plural pairs which served as the input to our process. Afrikaans plurals are almost always derived with the addition of a suffix (mostly *−e* or *−s*) to the singular form. Different sound changes may occur during this process. For example[6], gemination, which indicates the shortening of a preceding vowel, occurs frequently (e.g. *kat* → *katte*), as well as consonant-insertion (e.g. *kas* → *kaste*) and elision (*ampseed* → *ampsede*). Several sound changes may occur in the same word. For example, elision, consonant replacement and gemination occurs in *loof* → *lowwe*. Afrikaans (a Germanic language) has borrowed a few words from Latin. Some of these words have two plural forms, which introduces ambiguity in the word mappings: One plural is formed with a Latin suffix (*−a*) (e.g. *emetikum* → *emetika*) and one with an indigenous suffix (*−s*) (*emetikum* → *emetikums*). Allomorphs occur as well, for example *−ens* is an allomorph of the suffix *−s* in *bed* + *s* → *beddens*.

During phase one, all but eleven (0.3%) of the 3935 input word pairs were segmented correctly. To facilitate the evaluation of phase two, we define a *simple rule* as a rule which has an environment consisting of a single context. This is in contrast with an environment consisting of two or more contexts disjuncted together. Phase two acquired 531 *simple rules* for 44 special pairs. Of these 531 simple rules, 500 are ⇐ rules, nineteen are ⇔ rules and twelve are ⇒ rules. The average length of the simple rule contexts is 4.2 feasible pairs. Compare this with the

[6]All the examples comes from the 3935 input word pairs.

average length of the 3935 final input edit sequences which is 12.6 feasible pairs. The 531 simple rules can be reduced to 44 $\Leftrightarrow$ rules (i.e. one rule per special pair) with environments consisting of disjuncted contexts. These 44 $\Leftrightarrow$ rules analyze and generate the 3935 word pairs 100% correctly. The total number of feasible pairs in the 3935 final input edit strings is 49657. In the worst case, all these feasible pairs should be present in the rule contexts to accurately model the sound changes which might occur in the input pairs. However, the actual result is much better: Our process acquires a two-level rule set which accurately models the sound changes with only 4.5% (2227) of the input feasible pairs.

To obtain a prediction of the analysis and generation accuracy over *unseen* words, we divided the 3935 input pairs into five equal sections. Each fifth was held out in turn as test data while a set of two-level rules was learned from the remaining four-fifths. The average recognition accuracy as well as the generation accuracy over the held out test data is 93.9%.

6 Summary

We have described and experimentally evaluated, for the first time, a process which automatically acquires optimal two-level morphological rules from input word pairs. These can be used by a publicly available two-level morphological processor. We have demonstrated that our acquisition process is portable between at least three different languages and that an acquired rule set generalizes well to words not in the training corpus. Finally, we have shown the feasibility of automatically acquiring two-level rule sets for wide-coverage parsers, with word pairs extracted from a machine-readable dictionary.

7 Acknowledgements

Part of this work was completed during the first author's stay as visiting researcher at ISSCO (University of Geneva). We gratefully acknowledge the support of ISSCO, as well as the Swiss Federal Government for providing a bursary which made this visit possible. For helpful comments on an earlier draft of the paper, we wish to thank Susan Armstrong and Sabine Lehmann as well as the anonymous reviewers.

References

Evan L. Antworth. 1990. *PC-KIMMO: A Two-level Processor for Morphological Analysis.* Summer Institute of Linguistics, Dallas, Texas.

Walter Daelemans, Peter Berck and Steven Gillis. 1996. Unsupervised Discovery of Phonological Categories through Supervised Learning of Morphological Rules. In *COLING-96: 16th International Conference on Computational Linguistics*, pages 95–100, Copenhagen, Denmark.

Michael Gasser. 1994. Acquiring Receptive Morphology: A Connectionist Model. In *Proceedings of ACL-94.* Association for Computational Linguistics, Morristown, New Jersey.

Andrew R. Golding and Henry S. Thompson. 1985. A morphology component for language programs. *Linguistics*, 23:263-284.

Joseph E. Grimes. 1983. *Affix positions and cooccurrences: the PARADIGM program.* Summer Institute of Linguistics Publications in Linguistics No. 69. Dallas: Summer Institute of Linguistics and University of Texas at Arlington.

Lauri Karttunen and Kenneth R. Beesley. 1992. *Two-level Rule Compiler.* Technical Report ISTL-92-2. Xerox Palo Alto Research Center.

George Anton Kiraz. 1996. SEMHE: A generalized two-level System. In *Proceedings of ACL-96.* Association for Computational Linguistics, pages 159–166, Santa Cruz, California.

Kimmo Koskenniemi. 1983. *Two-level Morphology: A General Computational Model for Word-Form Recognition and Production.* PhD Dissertation. Department of General Linguistics, University of Helsinki.

Kimmo Koskenniemi. 1990. *A discovery procedure for two-level phonology.* Computational Lexicology and Lexicography: Special Issue dedicated to Bernard Quemada, Vol. I (Ed. L. Cignoni, C. Peters). Linguistica Computazionale, Pisa, Volume VI, 1990, pages 451–465.

Evelin Kuusik. 1996. Learning Morphology: Algorithms for the Identification of Stem Changes. In *COLING-96: 16th International Conference on Computational Linguistics*, pages 1102–1105, Copenhagen, Denmark.

David Sankoff and Joseph B. Kruskal. 1983. *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison.* Addison-Wesley, Massachusetts.

Gary F. Simons. 1988. Studying morphophonemic alternation in annotated text, parts one and two. *Notes on Linguistics*, 41:41–46; 42:27–38.

Richard Sproat. 1992. *Morphology and Computation.* The MIT Press, Cambridge, England.

Enrique Vidal, Andrés Marzal and Pablo Aibar. 1995. Fast Computation of Normalized Edit Distances. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 17:899–902.

Klaus Wothke. 1986. Machine learning of morphological rules by generalization and analogy. In *COLING-86: 11th International Conference on Computational Linguistics*, pages 289–293, Bonn.