Creating and Weighting Hunspell Dictionaries as Finite-State Automata

Tommi A Pirinen and Krister Lindén

University of Helsinki, Department of Modern Languages
Unionkatu 40, FI-00014 University of Helsinki, Finland
{tommi.pirinen,krister.linden}@helsinki.fi
http://www.helsinki.fi/modernlanguages

Abstract. There are numerous formats for writing spell-checkers for open-source systems and there are many lexical descriptions for natural languages written in these formats. In this paper, we demonstrate a method for converting Hunspell and related spell-checking lexicons into finite-state automata. We also present a simple way to apply unigram corpus training in order to improve the spell-checking suggestion mechanism using weighted finite-state technology. What we propose is a generic and efficient language-independent framework of weighted finite-state automata for spell-checking in typical open-source software, e.g. Mozilla Firefox, OpenOffice and the Gnome desktop.

1 Introduction

Currently there is a wide range of different free open-source solutions for spell-checking by computer. The most popular of the spelling dictionaries are the various instances of *spell software, i.e. ispell[1], aspell[2], myspell and hunspell[3] and other *spell derivatives. The hunspell dictionaries provided with the OpenOffice.org suite cover 98 languages.

The program-based spell-checking methods have their limitations because they are based on specific program code that is extensible only by coding new features into the system and getting all users to upgrade. E.g. hunspell has limitations on what affix morphemes you can attach to word roots with the consequence that not all languages with rich inflectional morphologies can be conveniently implemented in hunspell. This has already resulted in multiple new pieces of software for a few languages with implementations to work around the limitations, e.g. emberek (Turkish), hspell (Hebrew), uspell (Yiddish) and voikko (Finnish). What we propose is to use a generic framework of finite-state automata for these tasks. With finite-state automata it is possible to implement the spell-checking functionality as a one-tape weighted automaton containing the language model and a two-tape weighted automaton containing the error model.

In addition, we extend the hunspell spell-checking system by using simple corpus-based unigram probability training [8]. With this probability trained lexicon it is possible to fine-tune and improve the suggestions for spelling errors.

[1] http://www.lasr.cs.ucla.edu/geoff/ispell.html
[2] http://aspell.net
[3] http://hunspell.sf.net

We also provide a path for integrating our finite-state spell-checking and hyphenation into applications through the open-source spell-checking library voikko[4], which has been integrated with typical open-source software, such as Mozilla Firefox, OpenOffice.org and the Gnome desktop via enchant.

2 Definitions

In this article we use weighted two-tape finite-state automata—or weighted finite-state transducers—for all processing. We use the following symbol conventions to denote the parts of a weighted finite-state automaton: a transducer $T = (\Sigma, \Gamma, Q, q_0, Q_f, \delta, \rho)$ with a semi-ring $(S, \oplus, \otimes, \overline{0}, \overline{1})$ for weights. Here Σ is a set with the input tape alphabet, Γ is a set with the output tape alphabet, Q a finite set of states in the transducer, $q_0 \in Q$ is an initial state of the transducer, $Q_f \subset Q$ is a set of finite states, $\delta : Q \times \Sigma \times \Gamma \times S \to Q$ is a transition relation, $\rho : Q_f \to S$ is a final weight function. A successful path is a list of transitions from an initial state to a final state with a weight different from $\overline{0}$ collected from the transition function and the final state function in the semi-ring S by the operation $\otimes$. We typically denote a successful path as a concatenation of input symbols, a colon and a concatenation of output symbols. The weight of the successful path is indicated as a subscript in angle brackets, *input:output*$_{<w>}$. A path transducer is denoted by subscripting a transducer with the path. If the input and output symbols are the same, the colon and the output part can be omitted.

The finite-state formulation we use in this article is based on Xerox formalisms for finite-state methods in natural language processing [2], in practice lexc is a formalism for writing right linear grammars using morpheme sets called lexicons. Each morpheme in a lexc grammar can define their right follower lexicon, creating a finite-state network called a *lexical transducer*. In formulae, we denote a lexc style lexicon named X as Lex_X and use the shorthand notation $Lex_X \cup$ *input:output Y* to denote the addition of a lexc string or morpheme, `input:output Y ;` to the `LEXICON X`. In the same framework, the twolc formalism is used to describe context restrictions for symbols and their realizations in the form of parallel rules as defined in the appendix of [2]. We use $Twol_Z$ to denote the rule set Z and use the shorthand notation $Twol_Z \cap$ *a:b* $\leftrightarrow$ *l e f t_r i g h t* to denote the addition of a rule string `a:b <=> l e f t _ r i g h t ;` to the rule set Z, effectively saying that *a:b* only applies in the specified context.

A spell-checking dictionary is essentially a single-tape finite-state automaton or a language model T_L, where the alphabet $\Sigma_L = \Gamma_L$ are characters of a natural language. The successful paths define the correctly spelled word-forms of the language [8].

For weighted spell-checking, we define the weights in a lexicon as the probability of the word in a text corpus, e.g. Wikipedia. For a weighted model of the automaton, we use the tropical semi-ring assigning each word-form the weight $-\log \frac{f_w}{CS}$, where f_w is the frequency of the word and CS the corpus size as the number of word form tokens. For word-forms not appearing in the text corpus, we assign a small probability using the formula $-\log \frac{1}{CS+1}$.

A spelling correction model or an error model T_E is a two-tape automaton mapping the input text strings of the text to be spell-checked into strings that may be in the

[4] `http://voikko.sf.net`

language model. The input alphabet Σ_E is the alphabet of the text to be spell-checked and the output alphabet is $\Gamma_E = \Sigma_L$. For practical applications, the input alphabet needs to be extended by a special *any* symbol with the semantics of a character not belonging to the alphabet of the language model in order to account for input text containing typos outside the target natural language alphabet. The error model can be composed with the language model, $T_L \circ T_E$, to obtain an error model that only produces strings of the target language. For space efficiency, the composition may be carried out during run-time using the input string to limit the search space. The weights of an error model may be used as an estimate for the likelihood of the combination of errors. The error model is applied as a filter between the path automaton T_s compiled from the erroneous string, $s \notin T_L$, and the language model, T_L, using two compositions, $T_s \circ T_E \circ T_L$. The resulting transducer consists of a potentially infinite set of paths relating an incorrect string with correct strings from L. The paths, $s : s^i_{<w_i>}$, are weighted by the error model and language model using the semi-ring multiplication operation, $\otimes$. If the error model and the language model generate an infinite number of suggestions, the best suggestions may be efficiently enumerated with some variant of the n-best-paths algorithm [7]. For automatic spelling corrections, the best path may be used. If either the error model or the language model is known to generate only a finite set of results, the suggestion generation algorithm may be further optimized.

3 Material

In this article, we present methods for converting the hunspell dictionaries and rule sets for use with open-source finite-state writer's tools. As concrete dictionaries, we use the repositories of free implementations of these dictionaries and rule sets found on the internet, e.g. the hunspell dictionary files found on the OpenOffice.org spell-checking site[5].

In this section, we describe the parts of the file formats we are working with. All of the information of the hunspell format specifics is derived from the `hunspell(4)`[6] man page, as that is the only normative documentation of hunspell we have been able to locate.

The corpora we use for the unigram training of spell-checking dictionaries are Wikipedia database backups[7]. The Wikipedia is available for the majority of languages and it consists of large amounts of language that is typically well-suited for training spell-checking dictionaries.

3.1 Hunspell File Format

A hunspell spell-checking dictionary consists of two files: a dictionary file and an affix file. The dictionary file contains only root forms of words with information about morphological affix classes to combine with the roots. The affix file contains lists of affixes

[5] `http://wiki.services.openoffice.org/wiki/Dictionaries`

[6] `http://manpages.ubuntu.com/manpages/dapper/man4/hunspell.4.html`

[7] `http://download.wikimedia.org`

```
1  # Swedish
   abakus/HDY
3  abalienation/AHDvY
   abalienera/MY
5  # Northern Sámi
   okta/1
7  guokte/1,3
   golbma/1,3
9  # Hungarian
   üzér/1   1
11 üzletág/2        2
   üzletvezetö/3    1
13 üzletszerzö/4    1
```

Fig. 1. Excerpts of Swedish, Northern S|-á-|mi and Hungarian dictionaries

along with their context restrictions and effects, but the affix file also serves as a settings file for the dictionary, containing all meta-data and settings as well.

The dictionary file starts with a number that is intended to be the number of lines of root forms in the dictionary file, but in practice many of the files have numbers different from the actual line count, so it is safer to just treat it as a rough estimate. Following the initial line is a list of strings containing the root forms of the words in the morphology. Each word may be associated with an arbitrary number of classes separated by a slash. The classes are encoded in one of the three formats shown in the examples of Figure 1: a list of binary octets specifying classes from 1–255 (minus octets for CR, LF etc.), as in the Swedish example on lines 2–4, a list of binary words, specifying classes from 1–65,535 (again ignoring octets with CR and LF) or a comma separated list of numbers written in digits specifying classes 1–65,535 as in the North Sámi examples on lines 6–8. We refer to all of these as continuation classes encoded by their numeric decimal values, e.g. 'abakus' on line 2 would have continuation classes 72, 68 and 89 (the decimal values of the ASCII code points for H, D and Y respectively). In the Hungarian example, you can see the affix compression scheme, which refers to the line numbers in the affix file containing the continuation class listings, i.e. the part following the slash character in the previous two examples. The lines of the Hungarian dictionary also contain some extra numeric values separated by a tab which refer to the morphology compression scheme that is also mentioned in the affix definition file; this is used in the hunmorph morphological analyzer functionality which is not implemented nor described in this paper.

The second file in the hunspell dictionaries is the affix file, containing all the settings for the dictionary, and all non-root morphemes. The Figure 2 shows parts of the Hungarian affix file that we use for describing different setting types. The settings are typically given on a single line composed of the setting name in capitals, a space and the setting values, like the NAME setting on line 6. The hunspell files have some values encoded in UTF-8, some in the ISO 8859 encoding, and some using both binary and ASCII data at the same time. Note that in the examples in this article, we have tran-

```
AF 1263
AF VË–jxLnÓéè3ÄäTtYc,4l # 1
AF UmÖyiYcÇ # 2
AF ÖCWRÍ–jþÓíyÉÁÿYc2 # 3

NAME Magyar Ispell helyesírási szótár
LANG hu_HU
SET UTF–8
KEY öüó|qwertzuiopőú| # wrap
     asdfghjkléáűíyxcvbnm
TRY íóútaeslzánorhgkié # wrap
     dmyőpvöbucfjüyxwq–.á

COMPOUNDBEGIN v
COMPOUNDEND x
ONLYINCOMPOUND |
NEEDAFFIX u

REP 125
REP í i
REP i í
REP ó o
REP oliere oliére
REP cc gysz
REP cs ts
REP cs ds
REP ccs ts
# 116 more REP lines

SFX ? Y 3
SFX ? ö ős/1108 ö 20973
SFX ? 0 ös/1108 [^aáeéiíoóöőuüű] 20973
SFX ? 0 s/1108 [áéiíoóúőuúüű–] 20973

PFX r Y 195
PFX r 0 legújra/1262 . 22551
PFX r 0 legújjá/1262 . 22552
# 193 more PFX r lines
```

Fig. 2. Excerpts from Hungarian affix file

scribed everything into UTF-8 format or the nearest relevant encoded character with a displayable code point.

The settings we have used for building the spell-checking automata can be roughly divided into the following four categories: meta-data, error correction models, special continuation classes, and the actual affixes. An excerpt of the parts that we use in the Hungarian affix file is given in Figure 2.

The meta-data section contains, e.g., the name of the dictionary on line 6, the character set encoding on line 8, and the type of parsing used for continuation classes, which is omitted from the Hungarian lexicon indicating 8-bit binary parsing.

The error model settings each contain a small part of the actual error model, such as the characters to be used for edit distance, their weights, confusion sets and phonetic confusion sets. The list of word characters in order of popularity, as seen on line 12 of Figure 2, is used for the edit distance model. The keyboard layout, i.e. neighboring key sets, is specified for the substitution error model on line 10. Each set of the characters, separated by vertical bars, is regarded as a possible slip-of-the-finger typing error. The ordered confusion set of possible spelling error pairs is given on lines 19–27, where each line is a pair of a 'mistyped' and a 'corrected' word separated by whitespace.

The compounding model is defined by special continuation classes, i.e. some of the continuation classes in the dictionary or affix file may not lead to affixes, but are defined in the compounding section of the settings in the affix file. In Figure 2, the compounding rules are specified on lines 14–16. The flags in these settings are the same as in the affix definitions, so the words in class 118 (corresponding to lower case v) would be eligible as compound initial words, the words with class 120 (lower case x) occur at the end of a compound, and words with 117 only occur within a compound. Similarly, special flags are given to word forms needing affixes that are used only for spell checking but not for the suggestion mechanism, etc.

The actual affixes are defined in three different parts of the file: the compression scheme part on the lines 1–4, the suffix definitions on the lines 30–33, and the prefix definitions on the lines 35–37.

The compression scheme is a grouping of frequently co-occurring continuation classes. This is done by having the first AF line list a set of continuation classes which are referred to as the continuation class 1 in the dictionary, the second line is referred to the continuation class 2, and so forth. This means that for example continuation class 1 in the Hungarian dictionary refers to the classes on line 2 starting from 86 (V) and ending with 108 (l).

The prefix and suffix definitions use the same structure. The prefixes define the left-hand side context and deletions of a dictionary entry whereas the suffixes deal with the right-hand side. The first line of an affix set contains the class name, a boolean value defining whether the affix participates in the prefix-suffix combinatorics and the count of the number of morphemes in the continuation class, e.g. the line 35 defines the prefix continuation class attaching to morphemes of class 114 (r) and it combines with other affixes as defined by the Y instead of N in the third field. The following lines describe the prefix morphemes as triplets of removal, addition and context descriptions, e.g., the line 31 defines removal of 'ö', addition of 'ős' with continuation classes from AF line 1108, in case the previous morpheme ends in 'ö'. The context description may also contain bracketed expressions for character classes or a fullstop indicating any character (i.e. a wild-card) as in the POSIX regular expressions, e.g. the context description on line 33 matches any Hungarian vowel except a, e or ö, and the 37 matches any context. The deletion and addition parts may also consist of a sole '0' meaning a zero-length string. As can be seen in the Hungarian example, the lines may also contain an additional number at the end which is used for the morphological analyzer functionalities.

4 Methods

This article presents methods for converting the existing spell-checking dictionaries with error models to finite-state automata. As our toolkit we use the free open-source HFST toolkit[8], which is a general purpose API for finite-state automata, and a set of tools for using legacy data, such as Xerox finite-state morphologies. For this reason this paper presents the algorithms as formulae such that they can be readily implemented using finite-state algebra and the basic HFST tools.

The lexc lexicon model is used by the tools for describing parts of the morphotactics. It is a simple right-linear grammar for specifying finite-state automata described in [2, 6]. The twolc rule formalism is used for defining context-based rules with two-level automata and they are described in [5, 6].

This section presents both a pseudo-code presentation for the conversion algorithms, as well as excerpts of the final converted files from the material given in Figures 1 and 2 of Section 3. The converter code is available in the HFST SVN repository[9] for those who wish to see the specifics of the implementation in lex, yacc, c and python.

4.1 Hunspell dictionary conversion

The hunspell dictionaries are transformed into a finite-state transducer language model by a finite-state formulation consisting of two parts: a lexicon and one or more rule sets. The root and affix dictionaries are turned into finite-state lexicons in the lexc formalism. The lexc formalism models the part of the morphotax concerning the root dictionary and the adjacent suffixes. The rest is encoded by injecting special symbols, called flag diacritics, into the morphemes restricting the morpheme co-occurrences by implicit rules that have been outlined in [1]; the flag diacritics are denoted in lexc by at-sign delimited substrings. The affix definitions in hunspell also define deletions and context restrictions which are turned into explicit two-level rules.

The pseudo-code for the conversion of hunspell files is provided in Algorithm 1 and excerpts from the conversion of the examples in Figures 1 and 2 can be found in Figure 3. The dictionary file of hunspell is almost identical to the lexc root lexicon, and the conversion is straightforward. This is expressed on lines 4–9 as simply going through all entries and adding them to the root lexicon, as in lines 6—10 of the example result. The handling of affixes is similar, with the exception of adding flag diacritics for co-occurrence restrictions along with the morphemes. This is shown on lines 10—28 of the pseudo-code, and applying it will create the lines 17—21 of the Swedish example, which does not contain further restrictions on suffixes.

To finalize the morpheme and compounding restrictions, the final lexicon in the lexc description must be a lexicon checking that all prefixes with forward requirements have their requiring flags turned off.

[8] `http://HFST.sf.net`

[9] `http://hfst.svn.sourceforge.net/viewvc/hfst/trunk/conversion-scripts/`

Algorithm 1 Extracting morphemes from hunspell dictionaries

$finalflags \leftarrow \epsilon$
2: **for all** lines $morpheme/Conts$ in dic **do**
  $flags \leftarrow \epsilon$
4: **for all** $cont$ in $Conts$ **do**
    $flags \leftarrow flags$ + *@C.cont@*
6:    $Lex_{Conts} \leftarrow Lex_{Conts} \cup$ *0:[<cont] cont*
  end for
8: $Lex_{Root} \leftarrow Lex_{Root} \cup$ *flags + morpheme Conts*
end for
10: **for all** suffixes $lex, deletions, morpheme/Conts, context$ in aff **do**
  $flags \leftarrow \epsilon$
12: **for all** $cont$ in $Conts$ **do**
    $flags \leftarrow flags$ + *@C.cont@*
14:    $Lex_{Conts} \leftarrow Lex_{Conts} \cup$ *0 cont*
  end for
16: $Lex_{lex} \leftarrow Lex_{lex} \cup$ *flags* + $[< lex]$ + *morpheme Conts*
  for all del in $deletions$ **do**
18:    $lc \leftarrow context + deletions$ before del
    $rc \leftarrow deletions$ after del + $[< lex] + morpheme$
20:    $Twol_d \leftarrow Twol_d \cap$ *del:0* $\Leftrightarrow$ *lc _ rc*
  end for
22: $Twol_m \leftarrow Twol_m \cap [< lex] : 0 \Leftrightarrow$ *context _ morpheme*
end for
24: **for all** prefixes $lex, deletions, morpheme/conts, context$ in aff **do**
  $flags \leftarrow$ *@P.lex@*
26: $finalflags \leftarrow finalflags$ + *@D.lex@*
  $lex \rightarrow prefixes$ {othewise as with suffixes, swapping left and right}
28: **end for**
$Lex_{end} \leftarrow Lex_{end} \cup$ *finalflags #*

4.2 Hunspell Error Models

The hunspell dictionary configuration file, i.e. the affix file, contains several parts that need to be combined to achieve a similar error correction model as in the hunspell lexicon.

The error model part defined in the KEY section allows for one slip of the finger in any of the keyboard neighboring classes. This is implemented by creating a simple homogeneously weighted crossproduct of each class, as given on lines 1–7 of Algorithm 2. For the first part of the example on line 10 of Figure 2, this results in the lexc lexicon on lines 11–18 in Figure 4.

The error model part defined in the REP section is an arbitrarily long ordered confusion set. This is implemented by simply encoding them as increasingly weighted paths, as shown in lines 9–12 of the pseudo-code in Algorithm 2.

The TRY section such as the one on line 12 of Figure 2 defines characters to be tried as the edit distance grows in descending order. For a more detailed formulation of a weighted edit distance transducer, see e.g. [8]). We created an edit distance model

```
LEXICON Root
    HUNSPELL_pfx ;
    HUNPELL_dic ;

! swedish lexc
LEXICON HUNSPELL_dic
@C.H@@C.D@@C.Y@abakus HDY ;
@C.A@@C.H@@C.D@@C.v@@C.Y@abalienation
    HUNSPELL_AHDvY ;
@C.M@@C.Y@abalienera MY ;

LEXICON HDY
0:[<H]    H ;
0:[<D]    D ;
0:[<Y]    Y ;

LEXICON H
er   HUNSPELL_end ;
ers  HUNSPELL_end ;
er   HUNSPELL_end ;
ers  HUNSPELL_end ;

LEXICON HUNSPELL_end
@D.H@@D.D@@D.Y@@D.A@@D.v@@D.m@ # ;

! swedish twolc file
Rules
"Suffix H allowed contexts"
%[%<H%]: 0 <=> \a_er ;
    \a_ers ;
    a:0_er ;
    a:0_ers ;

"a deletion contexts"
a:0 <=> _ %[%<H%]:0 er ;
    _ %[%<H%]: ers ;
```

Fig. 3. Converted dic and aff lexicons and rules governing the deletions

with the sum of the positions of the characters in the TRY string as the weight, which is defined on lines 14–21 of the pseudo-code in Algorithm 2. The initial part of the converted example is displayed on lines 20–27 of Figure 4.

Finally to attribute different likelihood to different parts of the error models we use different weight magnitudes on different types of errors, and to allow only correctly written substrings, we restrict the result by the root lexicon and morfotax lexicon, as given on lines 1–9 of Figure 4. With the weights on lines 1–5, we ensure that KEY errors are always suggested before REP errors and REP errors before TRY errors. Even

Algorithm 2 Extracting patterns for hunspell error models

for all neighborsets ns in KEY **do**
  for all character c in ns **do**
    for all character d in ns such that $c! = d$ **do**
      $Lex_{KEY} \leftarrow Lex_{KEY} \cup c : d_{<0>}\#$
    end for
  end for
end for
$w \leftarrow 0$
for all pairs $wrong, right$ in REP **do**
  $w \leftarrow w + 1$
  $LEX_{REP} \leftarrow LEX_{REP} \cup wrong : right_{<w>}\#$
end for
$w \leftarrow 0$
for all character c in TRY **do**
  $w \leftarrow w + 1$
  $Lex_{TRY} \leftarrow Lex_{TRY} \cup c : 0_{<w>}\#$
  $Lex_{TRY} \leftarrow Lex_{TRY} \cup 0 : c_{<w>}\#$
  for all character d in TRY such that $c! = d$ **do**
    $Lex_{TRY} \leftarrow Lex_{TRY} \cup c : d_{<w>}\#$ {for swap: replace $\#$ with cd and add $Lex_{cd} \cup d : c_{<0>}\#$}
  end for
end for

though the error model allows only one error of any type[10], simulating the original hunspell, the resulting transducer can be transformed into an error model accepting multiple errors by a simple FST algebraic concatenative n-closure, i.e. repetition.

4.3 Weighting Finite-State Dictionaries

Finite-state automata can be weighted simply by using finite-state composition. For corpus-based weighting, the automata containing a weighted language model simply encodes a probability of a token appearing in a corpus [8]. The weights are formulated as penalty values belonging to the weighted semiring using the formula $-\log \frac{f}{CS}$, where f is the frequency of a token, and CS the size of the corpus in tokens. For tokens not appearing in the corpus, a maximum weight of $-\log \frac{1}{CS+1}$ is used to ensure that they will be suggested last by the error correction mechanism.

Since the error model is weighted as well, the weights need to be scaled so that combining them under the semiring addition operation will produce reasonable results. In our experiment we have opted to scale the weights of the error model so that the weight of making one error is always greater than the back-off weight in the unigram weighting model. Using this scaling ensures that the error model takes precedence over the probability data learned from the dictionary, which may only fine-tune the results in cases where there are multiple choices at the same error distance using the error model.

[10] the manual does not specify how many times and in which order different errors are tried, we assume once for simplicity and as a baseline

```
LEXICON HUNSPELL_error_root
< ? > HUNSPELL_error_root ;
HUNSPELL_KEY "weight: 0" ;
HUNSPELL_REP "weight: 100" ;
HUNSPELL_TRY "weight: 1000" ;

LEXICON HUNSPELL_errret
< ? > HUNSPELL_errret ;
# ;

LEXICON HUNSPELL_KEY
ö:ü HUNSPELL_errret "weight: 0" ;
ö:ó HUNSPELL_errret "weight: 0" ;
ü:ö HUNSPELL_errret "weight: 0" ;
ü:ó HUNSPELL_errret "weight: 0" ;
ó:ö HUNSPELL_errret "weight: 0" ;
ó:ü HUNSPELL_errret "weight: 0" ;
! same for other parts

LEXICON HUNSPELL_TRY
í:0 HUNSPELL_errret "weight: 1" ;
0:í HUNSPELL_errret "weight: 1" ;
í:ó HUNSPELL_errret "weight: 2" ;
ó:í HUNSPELL_errret "weight: 2" ;
ó:0 HUNSPELL_errret "weight: 2" ;
0:ó HUNSPELL_errret "weight: 2" ;
! same for rest of the alphabet

LEXICON HUNSPELL_REP
í:i HUNSPELL_errret "weight: 1" ;
i:í HUNSPELL_errret "weight: 2" ;
ó:o HUNSPELL_errret "weight: 3" ;
oliere:olière HUNSPELL_errret "weight: 4" ;
cc:gysz HUNSPELL_errret "weight: 5" ;
cs:ts HUNSPELL_errret "weight: 6" ;
cs:ds HUNSPELL_errret "weight: 7" ;
ccs:ts HUNSPELL_errret "weight: 8" ;
! same for rest of REP pairs...
```

Fig. 4. Converted error models from aff file

The tokens are extracted from Wikipedia using a dictionary transducer and tokenizing analysis algorithm[4]. This algorithm uses the dictionary automaton to extract tokens that appear in the dictionary from the Wikipedia data. The rest of the tokens are formed from contiguous runs of other dictionary characters which did not result in dictionary word-forms. From this set, the correct tokens are turned into a weighted suffix-tree automaton using the $-\log \frac{f}{CS}$ formula for the weights. This is unioned with

a version of the original dictionary whose final weights have been set to the maximum weight, $-\log \frac{1}{CS+1}$.

5 Tests and Evaluation

We have implemented the spell-checkers and their error models as finite-state transducers using program code and scripts with a Makefile. To test the code, we have converted 42 hunspell dictionaries from various language families. They consist of the dictionaries that were accessible from the aforementioned web sites at the time of writing. The Table 1 gives an overview of the sizes of the compiled automata. The size is given in binary multiples of bytes as reported by `ls -hl`. In the Table 1, we also give the number of roots in the dictionary file and the affixes in affix file. These numbers should also help with identifying the version of the dictionary, since there are multiple different versions available in the downloads.

To test the converted spell-checking dictionaries and error models, we picked 5 dictionaries of varying size and features. For spelling material, we created sets of spelling errors automatically, by introducing spelling errors in the tokens of Wikipedia data. The errors have been made by a python script implementing the edit distance type of errors to the words with a likelihood of 1/33 per character. The words which did not receive any automatic misspellings were not included in the test set, but words where spelling errors introduced another word form of the language were retained. These correct words resulting false positive hits in both tested systems also serve as further check that the systems work equally well. The hunspell results were obtained by `hunspell -1 -d $LL < misspelings`, and the automata were applied using the HFST tool `hfst-ospell error-model dictionary`.

The table 2 gives measures how our FSTs work compared to original Hunspell model, i.e. how accurately the FSTs implement the hunspell functionality. The differences between rankings show how accurately our weighted hunspell error model implements the hunspell's algorithm for generating suggestions, and the differences in false positives come from dictionary implementation. These contain different ordering of equally distant spelling errors and lack of case folding, for example. In the table the colums $1, 2-4$ and $5-$ show numbers of correct results showing as first, other top four, or lower suggestions. The column F is for spelling results, that are found in the dictionary—in this case, false positives. The column M contains misses, where no correct suggestion was given at all—even though all correct strings were originally extracted using the dictionary.

The table 3 summarizes how the probabilities and repetition can be used to change the spelling suggestions made by our finite-state dictionary and error models. Three variants of finite-state automata combinations were tested; one allowing for one hunspell errors without any weighting, one for two errors, and one where two error model was used in conjunction with Wikipedia probability weighted dictionary.

The time requirements of each system was also briefly tested using the standard Unix `time(1)` tool to measure the time of correcting the same misspelled strings previously used for testing the precision and recall. The times were measured on an

Table 1. Compiled Hunspell automata sizes

Language	Dictionary	Roots	Affixes
Portugese (Brazil)	14 MiB	307,199	25,434
Polish	14 MiB	277,964	6,909
Czech	12 MiB	302,542	2,492
Hungarian	9.7 MiB	86,230	22,991
Northern Sámi	8.1 MiB	527,474	370,982
Slovak	7.1 MiB	175,465	2,223
Dutch	6.7 MiB	158,874	90
Gascon	5.1 MiB	2,098,768	110
Afrikaans	5.0 MiB	125,473	48
Icelandic	5.0 MiB	222087	0
Greek	4.3 MiB	574,961	126
Italian	3.8 MiB	95,194	2,687
Gujarati	3.7 MiB	168,956	0
Lithuanian	3.6 MiB	95,944	4,024
English (Great Britain)	3.5 MiB	46,304	1,011
German	3.3 MiB	70,862	348
Croatian	3.3 MiB	215,917	64
Spanish	3.2 MiB	76,441	6,773
Catalan	3.2 MiB	94,868	996
Slovenian	2.9 MiB	246,857	484
Faeroese	2.8 MiB	108,632	0
French	2.8 MiB	91,582	507
Swedish	2.5 MiB	64,475	330
English (U.S.)	2.5 MiB	62,135	41
Estonian	2.4 MiB	282,174	9,242
Portugese (Portugal)	2 MiB	40.811	913
Irish	1.8 MiB	91,106	240
Friulian	1.7 MiB	36,321	664
Nepalese	1.7 MiB	39,925	502
Thai	1.7 MiB	38,870	0
Esperanto	1.5 MiB	19,343	2,338
Hebrew	1.4 MiB	329237	0
Bengali	1.3 MiB	110,751	0
Frisian	1.2 MiB	24,973	73
Interlingua	1.1 MiB	26850	54
Persian	791 KiB	332,555	0
Indonesian	765 KiB	23,419	17
Azerbaijani	489 KiB	19,132	0
Hindi	484 KiB	15,991	0
Amharic	333 KiB	13,741	4
Chichewa	209 KiB	5,779	0
Kashubian	191 KiB	5,111	0

Table 2. Difference between hunspell and FST with 2 errors

Language	Hunspell					FST				
	1	$2-4$	$5-\infty$	Fp	M	1	$2-4$	$5-\infty$	Fp	M
Occitan	164	34	3	4	85	234	35	5	0	16
Kurdish	200	26	4	6	6	214	17	4	0	7
Interlingua	817	95	2	13	73	814	92	17	3	72
Hungarian	338	45	6	16	21	354	38	8	6	22

Table 3. Suggestion algorithm results

Language	FST					FST + 2 errors					FST + 2 errors + unigrams				
	1	$2-4$	$5-\infty$	Fp	M	1	$2-4$	$5-\infty$	Fp	M	1	$2-4$	$5-\infty$	Fp	M
Occitan	185	13	0	0	92	234	35	5	0	16	251	21	2	0	16
Kurdish	171	10	0	0	61	214	17	4	0	7	215	16	4	0	7
Interlingua	631	45	1	3	320	820	80	9	3	72	828	88	9	3	72
Hungarian	266	11	2	6	143	354	38	8	6	22	360	30	10	6	22

application server provided by Centre of Scientific Computing in Finland running 8 AMD 8360 quad-core processors with 512 GiB of RAM memory available [11].

The tests clearly show that increasing the size of error model has greater effect to performance than weighting the error model. For weighted error models the only performance hit is practically the 0.1 s difference in startup time for loading potentially slightly larger dictionary.

6 Discussion

The spelling errors corrected by edit-distance style of error models suggested in order of probability in reference corpora all assume the spelling errors are primarily from mechanic typing mistakes. For other types of errors, only the ordered string replacements and primarily English phonetic are used. To attribute for other types of errors it could be possible to learn longer error models with error corpora[3], in particular it would be interesting to see how this fares with methods of using Wikipedia edit history to find the real world spelling corrections to gather spelling mistake corpora.

The performance of finite-state based spell checking system compared to hunspell approach seems to have typically an order of magnitude faster times as is typical with finite-state systems, providing nearly identical results. The differences in error method w.r.t.repetition depth and other minor details are still to be reverse-engineered to achieve perfectly faithful FST reimplementation of hunspell.

On practical side the current availability of the software supporting finite-state spell checking has already most of the coverage hunspell software does, as it is pluggable to libvoikko spell-checking software, which has ports for the most prominent open source

[11] `http://www.csc.fi/english/pages/hippu\_guide/introduction/overview/index\_html/?searchterm=hippu`

Table 4. Suggestion algorithm speed

Language	Hunspell	FST	FST + 2 errors	FST + 2 errors + unigrams
Occitan	10.7	0.1 s	0.8 s	0.9 s
Kurdish	0.6 s	1.1 s	1.1 s	1.2 s
Interlingua	21.3 s	2.9 s	3.2 s	3.4 s
Hungarian	28.6 s	5.4 s	9.8 s	9.9 s

software and spell checking libraries, such as OpenOffice.org, Mozilla, and libenchant, as well as the Mac OS X's ubiquitous spelling service.

In particular this only captures misspellings in isolation, which prevents us from detecting correctly spelled words in unexpected contexts. We intend to look into extending our model with context-based n-gram models for real-word spelling errors, e.g. [9].

7 Conclusion

We have demonstrated a method and created the software to convert legacy spell-checker data into a more general framework for finite-state automata. We have also provided a path for introducing this in real-life applications.

Acknowledgment

The authors would like to thank Miikka Silfverberg, Sam Hardwick and Brian Croom and others in the HFST research team for contributions to research tools and useful discussions.

References

1. Beesley, K.R.: Constraining separated morphotactic dependencies in finite-state grammars. pp. 118–127. Association for Computational Linguistics, Morristown, NJ, USA (1998)
2. Beesley, K.R., Karttunen, L.: Finite State Morphology. CSLI publications (2003)
3. Brill, E., Moore, R.C.: An improved error model for noisy channel spelling correction. In: ACL '00: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics. pp. 286–293. Association for Computational Linguistics, Morristown, NJ, USA (2000)
4. Garrido-Alenda, A., Forcada, M.L., Carrasco, R.C.: Incremental construction and maintenance of morphological analysers based on augmented letter transducers (2002)
5. Koskenniemi, K.: Two-level Morphology: A General Computational Model for Word-Form Recognition and Production. Ph.D. thesis, University of Helsinki (1983), `http://www.ling.helsinki.fi/~koskenni/doc/Two-LevelMorphology.pdf`
6. Lindén, K., Silfverberg, M., Pirinen, T.: Hfst tools for morphology—an efficient open-source package for construction of morphological analyzers. In: Mahlow, C., Piotrowski, M. (eds.) sfcm 2009. Lecture Notes in Computer Science, vol. 41, pp. 28—47. Springer (2009)
7. Mohri, M., Riley, M.: An efficient algorithm for the n-best-strings problem (2002)

8. Pirinen, T.A., Lindén, K.: Finite-state spell-checking with weighted language and error models. In: Proceedings of the Seventh SaLTMiL workshop on creation and use of basic lexical resources for less-resourced languagages. pp. 13–18. Valletta, Malta (2010), http://siuc01.si.ehu.es/~jipsagak/SALTMIL2010_Proceedings.pdf
9. Wilcox-O'Hearn, L.A., Hirst, G., Budanitsky, A.: Real-word spelling correction with trigrams: A reconsideration of the mays, damerau, and mercer model. In: Gelbukh, A.F. (ed.) CICLing. Lecture Notes in Computer Science, vol. 4919, pp. 605–616. Springer (2008)