Aligning phonemes using finte-state methods

Kimmo Koskenniemi
University of Helsinki
Helsinki, Finland
kimmo.koskenniemi@helsinki.fi

Abstract

The paper presents two finite-state methods which can be used for aligning pairs of cognate words or sets of different allomorphs of stems. Both methods use weighted finite-state machines for choosing the best alternative. Individual letter or phoneme correspondences can be weighted according to various principles, e.g. using distinctive features. The comparison of just two forms at a time is simple, so that method is easier to refine to include context conditions. Both methods are language independent and could be tuned for and applied to several types of languages for producing gold standard data.

The algorithms were implemented using the HFST finite-state library from short Python programs. The paper demonstrates that the solving of some non-trivial problems has become easier and accessible for a wider range of scholars.

1 Background

In this paper, finite-state automata (FSA) and finite-state transducers (FST) are used as the basic tools. In particular, the utility of weighted finite-state transducers (WFST) and automata (WFSA) is demonstrated.

Finite-state toolboxes have been freely available for some time, e.g. OpenFST (Mohri et al., 2002), Xerox XFST (Beesley and Karttunen, 2003), HFST – Helsinki Finite-State Transducer Technology (Lindén et al., 2011), SFST (Schmid, 2005) and Foma (Hulden, 2009). These implementations have been accessible as libraries for C or C++ programmers, some also as command line programs which can be pipelined (HFST), and some as scripting languages (XFST, SFST, Foma, HFST). Combining independent programs using shell commands is easy and suitable for many kinds of tasks, but certain combinations of operations are difficult or impossible to achieve this way. Even the XFST and SFST scripting languages have their restrictions, especially for looping, testing and input/output. It seems that none of the tasks discussed in this paper could be conveniently solved using the XFST or SFST scripting language.

More recently, most finite-state packages have been also available through Python in one way or the other. The HFST Python embedding was used here for a few reasons: it is implemented for Python 3 which uses Unicode as its native code, one could use weighted transducers, and HFST contained the kinds of functions that were needed.

2 Previous work on alignment

Automatic alignment of letters or phonemes is relevant in several areas, e.g. finding the pronunciation of unknown words or names in speech synthesis, see e.g. (Jiampojamarn et al., 2007), phonetically based spelling correction, see e.g. (Toutanova and Moore, 2002), in comparing cognate words in historical linguistics, see e.g. (Covington, 1998; Kondrak, 2000) , reconstructing proto-languages, e.g. (Bouchard-Côté et al., 2009) and in many other areas of linguistics. Character by character alignment can be approached as a machine learning problem, as in (Ciobanu and Dinu, 2014) or as a linguistic task as is done in this paper. The methods presented here make use of general properties of phonology. Still, more specific rules can be included where needed.

Covington (1998) used a special six step measure for phoneme distances and (Nerbonne and Heeringa, 1997) used phonetic features and the plain Levenshtein distance between sequences of features in order to estimate differences between Dutch dialects. (Somers, 1999) used distinctive

Proceedings of the 21st Nordic Conference of Computational Linguistics, pages 56–64,
Gothenburg, Sweden, 23-24 May 2017.

features in the comparison and his algorithm used the (manually marked) stressed vowels as the starting point whereas other approaches progressed from left to right. All these approaches were local in the sense that they measure each pair of phonemes separately and the sum of the measures was the measure of the whole alignment. Their methods appear to depend on this assumption and therefore exclude the inclusion of context dependent phenomena e.g. assimilation, metathesis, constraints of syllable structure etc.

The work presented here separates the definition of the measure from the algorithm for finding the best alignment. Measures are represented as WFSTs and they are built by producing weighted regular expressions using simple Python scripts. The algorithms used here utilize the efficient algorithms already available in the finite-state calculus for finding the best paths from weighted acyclic automata (Mohri, 2009).

3 Weighting the correspondences of phonemes

Simple phoneme by phoneme (or letter by letter) alignment is needed when relating cognate words of related languages or e.g. relating the written words in old texts with their present day forms. The process of alignment consists here of making phonetically similar phonemes correspond to each other and adding zeroes ∅ where necessary (due to epenthesis or elision). E.g. Estonian *leem* and Finnish *liemi* would be aligned by adding one zero at the end of the Estonian form:

```
l e e m ∅
l i e m i
```

In general, consonants may not correspond to vowels or vice versa, except for glides, approximants or semivowels which may correspond to certain vowels and certain consonants. In this paper, many-to-one, and one-to-many correspondences are simply expressed by using combinations of phoneme to phoneme correspondences, phoneme to zero and zero to phoneme correspondences.

Vowels can be described by using distinctive features such as the height of the tongue, frontness/backness and rounding/unrounding, see Figure 1. Using such features, one may compute distances between different vowels. Similarly, consonants can be characterized by their place of articulation, voicing and the manner of articulation, see Figure 2. The measure used here is not intended to be an absolute or universal measure, it is just an ad hoc approximation suitable for Estonian and Finnish.

```
vowels = {
    'i':('Close','Front','Unrounded'),
    'y':('Close','Front','Rounded'),
    'ü':('Close','Front','Rounded'),
    'u':('Close','Back','Rounded'),
    'e':('Mid','Front','Unrounded'),
    'ö':('Mid','Front','Rounded'),
    'õ':('Mid','Back','Unrounded'),
    'o':('Mid','Back','Rounded'),
    'ä':('Open','Front','Unrounded'),
    'a':('Open','Back','Unrounded')}
```

Figure 1: Description of some Finnish and Estonian orthographic vowels using distinctive features. IPA symbol for letters for which they are not the letter itself: ü = y, ö = ø, õ = ɤ, ä = æ, a = ɑ

The work described here permits different kinds of definitions for measuring the distances, including those used in (Covington, 1998; Kondrak, 2000; Nerbonne and Heeringa, 1997; Somers, 1999). Any measure which can reasonably be expressed as a WFST can be used by the algorithm presented in Section 4.

From the phoneme descriptions shown in Figure 1, one can compute simple distances between any two vowels or any two consonants. The formula chosen in this study was heuristic. Tongue height had three steps corresponding to values 1, 2, 3 and the distance was taken to be the difference of those values. The distance between front and back was taken to be 1 as was the distance between rounding and unrounding. The distance between any two vowels was defined to be the sum of these three numbers.[1]

A similar formula was used for consonants where the positions of articulation was numbered from 1 to 5 and their difference was the distance, see Figure 2. Different voicing counted as 1, and so did the manner of articulation. Again, the total distance was the sum of these three numbers.

A double loop through vowels and another through consonants produced a list of feasible combinations of phonemes and the computed measure for their difference. These triplets were then formatted as strings and glued together into a long regular expression, see some samples of it

[1] This is sometimes called Manhattan distance as opposed to the Euclidean distance which would be the square root of the sum of squares.

```
consonants = {
    'm':('Bilab','Voiced','Nasal'),
    'p':('Bilab','Unvoiced','Stop'),
    'b':('Bilab','Voiced','Stop'),
    'v':('Labdent','Voiced','Fricative'),
    'f':('Labdent','Unvoiced','Fricative'),
    'n':('Alveolar','Voiced','Nasal'),
    't':('Alveolar','Unvoiced','Stop'),
    'd':('Alveolar','Voiced','Stop'),
    's':('Alveolar','Unvoiced','Sibilant'),
    'l':('Alveolar','Voiced','Lateral'),
    'r':('Alveolar','Voiced','Tremulant'),
    'j':('Velar','Voiced','Approximant'),
    'k':('Velar','Unvoiced','Stop'),
    'g':('Velar','Voiced','Stop'),
    'h':('Glottal','Unvoiced','Fricative')}
```

Figure 2: Description of some Finnish an Estonian consonants

below:

```
... | u:u::0 | u:y::1 | u:ä::4 | u:ö::2 ...
| k:g::1 | k:h::2 | k:j::2 | k:k::0 ...
```

Note that the weight is after a double colon according to the extended notation for weighted regular expressions used in HFST. Thus, in the above formula, *u* may correspond to *u* at null cost, and *k* may correspond to *g* at a cost of 1.

Any phoneme could be deleted or inserted at a cost. A simple loop produced the additional correspondences and their weights:

```
... | o:∅::3 | p:∅::3 | r:∅::3 | ...
| ∅:o::3 | ∅:p::3 | ∅:r::3 | ...
```

In Finnish and in Estonian, long vowels are represented as two successive vowels. The default correspondences and weights clearly allow shortening of long vowels (or double consonants), but there would be two ways to express it with equal weights: one where the first of the two corresponds to zero, and the other where the second component corresponds to zero. In order to avoid this ambiguity, there was yet another loop which produced the necessary pieces of expressions which had a slightly smaller weight than the deletion alone. Note that e.g. *p:∅ p* will remain acceptable, but the expression below gives a lower weight for the combination where the latter component corresponds to zero.

```
... | o ∅:o::2 | p ∅:p::2 | r ∅:r::2 ...
| o o:∅::2 | p p:∅::2 | r r:∅::2 ...
```

One can use the same technique for handling orthographic conventions. One language might use *kk* and the other *ck* for a geminate *k*, similarly *ks* instead of *x* and *ts* instead of *z*. One can make such correspondences to have a zero distance by listing them with an explicit zero weight, e.g.:

```
k:c::0 k::0 | k:x s:∅::0 | t:z s:∅::0
```

One could use the same mechanism for giving some phoneme relations a lower weight. One could e.g. prefer assimilations over arbitrary combinations by imposing a lower weight for a variant if preceded or followed by phoneme which articulated in the same place. Local metathesis affecting two consecutive phonemes could also be expressed fairly neatly.

When all expressions are glued together, enclosed in brackets and followed by a Kleene star, the regular expression is ready to be compiled into a WFST. After compilation, the WFST which holds the distances as weights, can be stored as a file to be used later by the aligning algorithm.

4 Aligning pairs of words

Now, we are ready to write a small but general Python script which reads in the similarity WFST described in Section 3 and (1) reads in a pair of cognate words from the standard input and (2) converts them into FSTs W1 and W2, (3) adds explicit zero symbols freely to each, (4) compares the zero-filled expressions using the weighed distance transducer ALIGN and produces in this way all possible alignments and their total weights as a weighted transducer RES. Of these many possible alignments accepted by RES, (5) the best one is chosen and (6) printed, see Figure 3.

```
import sys, hfst
algfile = hfst.HfstInputStream("d.fst")
ALIGN = algfile.read()
for line in sys.stdin:                      # (1)
    F1,F2 = line.strip().split(sep=":")     # (1)
    W1 = hfst.fst(F1)                       # (2)
    W1.insert_freely(("∅","∅"))             # (3)
    W2 = hfst.fst(F2)                       # (2)
    W2.insert_freely(("∅","∅"))             # (3)
    W1.compose(ALIGN)                       # (4)
    W1.compose(W2)                          # (4)
    RES = W1.n_best(1).minimize()           # (5)
    paths =                                 # (6)
        res.extract_paths(output='text')    # (6)
    print(paths.strip())                    # (6)
```

Figure 3: Python program for aligning pairs of cognate words

The algorithm in Figure 3 considers all possible ways to add zero symbols to the cognates. It even adds an indefinite number of zeros to each cognate word. For Estonian *leem* and Finnish *liemi* the adding of the zeros would result in strings covered by the following expressions.

```
W1:    Ø* l Ø* e Ø* e Ø* m Ø*
W2:    Ø* l Ø* i Ø* e Ø* m Ø* i Ø*
```

The key idea is in the composition of these two FSAs so that the distance metric transducer *ALIGN* is in the middle:

```
W1 .o. ALIGN .o. W2
```

Transducers W1, W2 and the ALIGN are all epsilon-free, so the composition accepts only same length string pairs. Note that the distance metric ALIGN does not allow a zero to correspond to another zero, so the zero filled strings may only be at most twice so long as the longer cognate was. There would still be quite a few comparisons to do, if one would compare and evaluate them one pair at a time.

The HFST function *n_best(1)* finds the pair of strings which would have the least weight. This is one of the operations that are efficient for WFSTs. The operation produces a FST which accepts exactly the best string pair. Another function is needed for extracting the transition labels which constitute the strings themselves.

The aligner was used among other things, for relating word forms of Modern Finnish and Finnish of a Bible translation of year 1642. With slight tuning of the WFST, the aligner worked as the primary tool for aligning further pairs of old and new words which were used for writing and testing the rules which related the two forms of Finnish. The old orthography was less phonemic than the modern one and there was more orthographic variation in the old texts. After the initial adjusting to the orthography of the old texts, only a little tuning was needed to modify the computation of the similarities until the formula appeared to be stable.

As an informal check, the manually edited and checked set of 203 pairs of old and new word forms was cleaned from the added zeros and realigned using the aligner. Only one difference was observed in the result as compared with the original.[2]

5 Other uses for the distance WFSTs

The aligner was developed for aligning pairs of cognate words given to it. In this task, the aligner can and perhaps must be quite general. When we are studying the relation between two languages, we ought not commit ourselves to certain sound changes (or sound laws) when we start by preparing the source data for analyzing the relation itself.

One might speculate that the aligner could also be used as a predictor of the likely shapes of the missing half of a cognate pair. The WFST alone is, however, not useful for such because it generates too many candidates, not only those which would follow from the sound changes which govern the relation of the languages. The correct candidate would be buried under a heap of incorrect ones.

Instead of proposing the missing halves of cognate pairs from scratch, one can get useful results if one has a word list of the other language. Preparing for the processing, one first converts the word list into a FSA, say *words-et.fst*.

Then, one types in a known word from the first language. The script (1) converts it into a FSA, (2) shuffles it freely with zeros, (3) composes it with the distance WFST, (4) deletes all zeros from the result and then (5) composes it with a FSA containing the word list. From the result of this chain, (6) the best word pairs are retrieved and printed for the user. Using HFST command line tools, the chain of programs looks like:

```
$ hfst-strings2fst |
    hfst-shuffle -2 zeros.fst |
    hfst-compose -2 chardist.fst |
    hfst-compose -2 del-zeros.fst |
    hfst-compose -2 words-et.fst |
    hfst-fst2strings -N 5 -w
```

This is a pipeline of programs which expects the user to feed words of the first language. For each word typed in, the programs do the operations, and from the resulting WFST, the last program prints five best matches, e.g. for the Finnish word *ajo* ('driving', 'trip') it produces:

```
> ajo
ajo:aju 1
ajo:aje 2
ajo:aie 3
ajo:äiu 3
ajo:agu 3
```

The pipeline computes five best guesses as pairs of the Finnish and the Estonian words, and shows their weights. In this case, the first guess Estonian *aju* happens to be the correct one. In other cases, the correct one may not the best scored candidate, as for *vierastus*, the correct candidate *võõrastus* is the third in the list. Some short words may have many similar words in the word list. For them, the desired candidate is often too far down in the list

[2]The manually aligned *lepäØsivät:lewäisiØØt* was problematic because Old Finnish had a different morpheme for the third person past tense *-it* whereas the Modern Finnish has *-ivät*. Thus, no 'correct' alignment actually exists. The pair to be aligned ought to be *lepäisit:lewäisit*.

of best ranking candidates and will not be found this way.

Using the aligner is of course less precise than building a realistic model of sound changes as was done in (Koskenniemi, 2013a) where two-level rules were used for this purpose.

6 Aligning a set of stems or other morphs

Comparing several words is needed when relating more than two historically connected languages but also when relating e.g. different stems of a word in order to build lexical entries which can be used in morphological analysis. The reason for relating stems is usually to construct one common morphophonemic representation for all stems of a word.

In a simplified version of the two-level morphological analysis, one does not postulate underlying forms which are plain sequences of phonemes. Instead, one uses alternations of phonemes (morphophonemes) when different phonemes alternate with each other, cf. (Koskenniemi, 2013b). The problem of aligning three or more different words is similar to the one discussed above but somewhat different methods are needed.

Let us look at the Estonian word *pagu* ('an escape') which inflects in forms like *pagu*, *paos*, *pakku*. Traditional generative phonology would prefer an underlying stem like *paku* and derive the other forms from that using a sequence of rewriting rules. In contrast to this, the linguist following the methods of the simplified two-level morphology, would first locate the boundaries between the stem morphs and the suffix morphs, i.e. *pagu+*, *pao+s* and *pakku+*, then take the stems *pagu*, *pao*, and *pakku*, then insert a few zeros in order to make the stem morphs same length, i.e. *pagØu*, *paØØo*, *pakku*. Thus, the linguist would arrive at the morphophonemic representation *p a* {*gØk*} {*ØØk*} {*uou*} by merging the corresponding letters of each stem morph.

Let us see, how an algorithm could simulate the linguist when it starts from the point where the boundaries have already been located. The task of the algorithm is (1) to produce all alignments, including lots of nonsense alignments, (2) filter out infeasible alignments, and (3) to evaluate the feasible alignments and choose the best among them.

In order to carry out the first task, the algorithm blindly inserts some (or no) zero symbols into each stem in order to make all stem candidates same length. Thus, some stems are expanded to a number of alternatives where the zeros are at all different places. Assuming that five letters suffice for our example word, the first stem needs one zero, the second stem needs two zeros, and the third stem does not need any.[3] The insertion of the zeros to the first stem *pagu* would produce the strings *Øpagu*, *pØagu*, *paØgu*, *pagØu* and *paguØ*. Only one of these five will turn out to be useful but the algorithm does not know yet which. It does not actually produce the set of strings with zeros, but instead, it produces an automaton which accepts all of them.

The feasibility of an alignment is evaluated using the phoneme (or letter) correspondences which are caused by the insertion of zeros. Aligning *Øpagu*, *paØØo* and *pakku* would imply correspondences *Ø-p-p*, *p-a-a*, *a-Ø-k*, *g-Ø-k* and *u-o-u*. Such an alignment is infeasible, as it includes two forbidden correspondences: the second and the third phoneme correspondences *p-a-a* and *a-Ø-k* are both infeasible because they contain both a consonant and a vowel (the dashes of the correspondences will be omitted hereafter). Another alignment, e.g. *pagØu*, *paØØ* and *pakku* would imply the correspondences *ppp*, *aaa*, *gØk*, *ØØk* and *uou* which all seem phonologically feasible containing only identical or closely related sounds and zeros.

Each phoneme correspondence is evaluated separately and assigned a weight according to the similarity of the participating phonemes. The goodness of an alignment is computed as a sum of all weights for the phoneme correspondences in the alignment. In the same manner as when comparing two words, a correspondence consisting of identical phonemes e.g. *ppp* has a zero weight.

Two different distance measures were used. For vowels, the number of distinct vowels participating the correspondence was used as the measure. If the zero was also a member of the correspondence, 0.4 was added. For consonants, the differences of their features was counted. If there were both voiced and unvoiced, then 1 was added, if there were different manners of articulation, then one less than the number of manners was added. The positions of articulation were numbered from

[3] One may start with the shortest possible stems. If no results are produced (because one would have to match some consonant to a vowel), one may add one more zero and try again, and repeat this until the matching succeeds, and then still try with adding one more zero.

0 to 4 and they contributed so that 0.5 times the difference of values for the most back and most front position was added. One or more zeros in the correspondence added 2.6 to the total measure.

Semivowels were included both as consonants and vowels. Their combinations with vowels was restricted to those that are actually feasible, e.g. *j* was allowed to combine with *i* but not with other vowels.

All these measures are ad hoc. Readers are encouraged to experiment and improve the formulas of the measures. In particular, machine learning methods could be utilized for optimizing the parameters.

7 Algorithm for aligning multiple stems

The goal for the aligning of several words or stems were specified in the preceding section. The logic of the algorithm which implements them is shown in Figure 4 as a Python function extracted from the full implementation.

```
def multialign(stems, target_len, weighf):
   R = shuffle_w_zeros(stems[0], target_len)
   for string in stems[1:]:
      S = shuffle_w_zeros(string, target_len)
      R.cross_product(S)
      R.fst_to_fsa()
      T = remove_bad_transitions(R, weightf)
   return = set_weights(T, weighf)
```

Figure 4: Function which produces the set of all feasible alignments and their weights as a weighted FSA

Variables containing binary FSAs or FSTs as their values are in capital letters. The Python function *shuffle_w_zeros()* takes a stem as a string and returns an automaton accepting strings of the required length (by using a few HFST functions) so that exactly the correct amount of zeros are inserted freely.

The first goal of the algorithm (after inserting the necessary zeros) is to produce all combinations of the first and the second stem (with zeros inserted). The function *cross_product()* is a standard HFST function for the cross product of two FSAs. As a result, R is a transducer which maps any of the possible versions of the first stem into any of the versions of the second stem. Our example from Section 6, i.e. *paku pao pakko* would become mappings between two sets of strings:

{*Øpagu, pØagu, paØgu, pagØu, paguØ*}
and
{*ØØpao, ØpØao, ØpaØo, ØpaoØ, pØØao, pØaØo, pØaoØ, paØØo, paØoØ, paoØØ*}

The transducer R maps any of the stings in the first set into any of the second set. The combination of the fourth in the first set and the eighth in the second is a good candidate for the alignment, i.e. *pakØu:paØØo* or equivalently as a sequence of corresponding character pairs (where identical pairs are abbreviated): *p a g:Ø Ø u:o*. At this stage of the algorithm, however, R accepts all 50 combinations.

Two problems arise here: First, we cannot continue immediately, because for the next cross product, we need a FSA instead of a FST. A transducer can be encoded to become an automaton by a standard function *fst_to_fsa()* which substitutes all label pairs *x:y* with label pairs *xy:xy* so that the result becomes a FSA again.

The other problem is, that there may be many unwanted correspondences in the result, letting consonants correspond to vowels or vice versa. Thus, a user-made function *remove_bad_transitions()* checks the encoded FSA for infeasible labels and removes the corresponding transitions.[4]

Now the product of the first two stem expressions is again a FSA, and not too large. So we can proceed by applying the process to the next stem expression and so on. When all stems have been processed, we have a FSA representing all feasible combinations of the zero-padded stems, i.e. all feasible alignments. All alignments are feasible in the sense that they do not contain forbidden combinations of vowels and consonants.

Consider first stems *laps*, *lapse* and *las* and some of their feasible alignments, the morphophonemic representation, the weights for each correspondence (i.e. morphophoneme) and the sum of weights:

```
laps∅ lapse las∅∅
l a pps ss∅ ∅e∅
0 1 2   2.6 1.4    = 7.0

laps∅ lapse la∅s∅
l a pp∅ s ∅e∅
0 1 2.6 0 1.4      = 5.0
```

The former is not entirely impossible. It involves a stem final deletion of *e* which is common

[4] In principle, the cross product multiplies the size of the automaton fast if there would be several stems and several zeros involved.

to both alignments, and in addition the deletion of *s* and changing the *p* into *s* (which would be uncommon). Considering the weights, the second alignment is clearly better because there is no alternation of *s*, only a deletion of *p*.

Other cases may be more difficult to resolve. Consider the stems *litter*, *litri* and *litre* where we would have at least two competing alignments, the latter of which is one inserted zero longer than the former:

```
litter litriØ litreØ
l i t trr eie rØØ
0 1 0  2   2  2.6          = 7.6

litterØ litØØri litØØre
l i t tØØ eØØ  r  Øie
0 1 0 2.6 1.4  0  2.4      = 7.4
```

The linguist would probably choose the latter and reject the former. So does the formula for assigning weights to the morphophonemes. The example also shows that that the algorithm must not stop on the shortest feasible alignment.

The algorithm has a function *weightf()* which computes a weight for each candidate combination of phonemes according to the principles explained in Section 6. It is used first in excluding completely impossible transitions when building the intermediate results. It is used again in another function *set_weights()* which inserts appropriate weights into the transitions of the final result containing still all feasible alignments for the stems.

Once we have a weighted FSA which represents all alignments, it is easy to choose the best alignment with an HFST function *n_best*. Some sets of stems, such as *töö tö* would have two equally good alignments, *töö töØ* and *töö tØö*. Therefore, more than one of the top alignments are chosen first. Out of the equally good top candidates, the program selects the one in which the deletions are more to the right. Behind this choice is the observation that in suffixing languages, the deletions are more likely to occur near the affix morphemes. Using this criterion, *töö töØ* is chosen over the other alternative.

Anyway, the weighting is made in a transparent manner in the program, so it would be easy to experiment, modify and tune the weighting, if it turns out that other types of languages need different principles to resolve such ties.

8 Aligning Estonian noun stems

After the initial implementation of the algorithm it was tested against a language resource *tyvebaas.pmf* containing Estonian words with inflectional information and was freely available from the Institute of the Estonian Language (Eesti Keele Instituut). The file contained among other things, a base form, inflectional class, part of speech code and a list of all actual stem allomorphs of the lexeme. For this experiment the nouns were extracted, and furthermore, only those nouns which had more than one stem given. Out of these entries, only the stems were extracted and all other information was ignored. The original file contained entries like:

='aasima 28_V | at: 'aasi, an: aasi
='aasta 01_S | a0: 'aasta, a0r: 0
=aastak 02_S | a0: aastak, b0: aastaku, b0r:...
~'aastane 10_A | a0: 'aastane, b0: 'aastase,...

The first and the last entries were not nouns and they were discarded. The second entry was a noun but only with one stem. It was discarded as well. The third entry was a noun and had more than one stem, and it was chosen and produced an entry for the purposes of this study. The selected 15 934 sets of noun stem sets entries were like the following examples (after some cleaning):[5]

```
aastak aastaku
aatom aatomi aatome
abajas abaja
```

The whole test material was run through the aligner and the result studied using some sampling and some other checking. Some example alignments made by the algorithm are given in Figure 5.

At least for the author, the alignments in Figure 5 appeared to be acceptable. The whole material was then checked by sampling and by looking at infrequent or untypical letter combinations in a frequency list of all letter combinations. In the whole material, three alignments were found where the author disagrees with the result of the algorithm, see Figure 6. The remaining 15 931 alignments might be acceptable.[6]

[5] The goal was just to test the algorithm, at this time not to produce a lexicon to be accompanied with endings and morphophonological rules. Those other tasks would seem quite feasible to do at a later stage with some cooperation with suitable parties.

[6] One could have used a more phonemic representation for the words by representing long vowels by a single phoneme, e.g. *A* instead of *aa*. Such a representation could have made the task a bit simpler and maybe the mistakes could have been avoided altogether.

```
birmalane birmalase birmalas birmalasi
    b i r m a l a nsss ee∅i
faktuur faktuuri faktuure
    f a k t u u r ∅ie
hämarik hämarikku hämariku hämarikke
                                hämarike
    h ä m a r i k ∅k∅k∅ ∅uuee
koger kogre kokre kogri kokri
    k o ggkgk e∅∅∅∅ r ∅eeii
kuusk kuuske kuuse kuuski kuusi
    k u u s kk∅k∅ ∅eeii
liud liuda liua liudu
    l i u dd∅d ∅aau
mutter mutri mutre
    m u t t∅∅ e∅∅ r ∅ie
pagu pao pakku
    p a g∅k ∅∅k uou
pugu pukku
    p u gk ∅k u
ruga roa ruge rukka
    r uouu g∅gk ∅∅∅k aaea
sugu soo sukku
    s uou g∅k ∅∅k uou
toht tohtu tohu tohte tohe
    t o h tt∅t∅ ∅uuee
vahk vahku vahu vahke vahe
    v a h kk∅k∅ ∅uuee
äie äige
    ä i ∅g e
```

Figure 5: Some example stem sets and their alignments

```
raag raagu rao raage
    raag∅ raagu rao∅∅ raage
    r a aaoa gg∅g ∅u∅e
saad saadu sao saade
    saad∅ saadu sao∅∅ saade
    s a aaoa dd∅d ∅u∅e
sae saaja saaju
    sae∅∅ saaja saaju
    s a eaa ∅jj ∅au
```

Figure 6: The three questionable alignments found in the Estonian word list of nouns

9 Tasks for the future

Distances between phonemes as described in Section 4 seems like a challenge for further research. In essence, the relation looks like a two-level rule relation with weights. No present compiler for two-level rules can directly assign weights to correspondences. It appears to be possible to include some form of weighting to the rule formalism and write a compiler.

A two-level compiler dedicated for alignment tasks could express the language specific patterns of phoneme alternations. A clear case for such rules would be the proper handling of vowel lengthening and gemination. Furthermore, one guiding the insertions and deletions could be made more transparent and robust using rules.

Writing dedicated compilers for two-level rules has now become much easier. It would be a reasonable project to write a compiler which is oriented towards alignment. In this way, one could implement suitable context constraints for the target language(s) and integrate it into the process of alignment.

The algorithm with the weights made for Estonian stems, was tested also for the alignment of Finnish noun stems. The result appeared to be fully clean, but only model words for different inflectional classes were tested. The same weights were also tested for some cognate word sets for Uralic languages found in en.wictionary.org as far as the symbols were available. The alignments for those seemed to be OK. Extensive tests are needed to evaluate the performance on those areas.

All programs made and the testing data used in this project is in the open source and has been made accessible to all[7]. Even without a nicer compiler, the modification of the weighting schemes is easy for anybody who has slight command of Python programming. In particular, researchers interested in statistical or machine learning methods are encouraged to apply their methods for finding optimal weightings for phoneme combinations or using the material as a gold standard for other types of solutions. Even linguists who have better command of Estonian than the author, are encouraged to report mistakes in the aligned material or critique on what the optimal alignment ought to be.

Using the Python programs with the finite state tools requires the availability of Python3, the HFST library and the SWIG interface for using the library from Python.[8]

10 Credits

The FIN-CLARIN language infrastructure project has developed and maintains the HFST software. The author is grateful for their technical support and assistance. Particular thanks are due to Erik Axelsson.

[7] See *https://github.com/koskenni/alignment* for the programs and for the test data

[8] See *https://hfst.github.io/* for the documentation and instructions for downloading, installing and using the Python HFST

References

Kenneth R. Beesley and Lauri Karttunen. 2003. *Finite State Morphology*. Studies in Computational Linguistics, 3. University of Chicago Press. Additional info, see: `www.stanford.edu/~laurik/fsmbook/home.html`.

Alexandre Bouchard-Côté, Thomas L. Griffiths, and Dan Klein. 2009. Improved reconstruction of protolanguage word forms. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 65–73, Boulder, Colorado, June. Association for Computational Linguistics.

Alina Maria Ciobanu and Liviu P. Dinu. 2014. Automatic detection of cognates using orthographic alignment. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 99–105, Baltimore, Maryland, June. Association for Computational Linguistics.

Michael A. Covington. 1998. Alignment of multiple languages for historical comparison. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 1*, pages 275–279, Montreal, Quebec, Canada, August. Association for Computational Linguistics.

Mans Hulden. 2009. Foma: a finite-state compiler and library. In *Proceedings of the Demonstrations Session at EACL 2009*, pages 29–32, Stroudsburg, PA, USA, April. Association for Computational Linguistics.

Sittichai Jiampojamarn, Grzegorz Kondrak, and Tarek Sherif. 2007. Applying many-to-many alignments and hidden markov models to letter-to-phoneme conversion. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 372–379, Stroudsburg, PA, USA, April. Association for Computational Linguistics.

Grzegorz Kondrak. 2000. A new algorithm for the alignment of phonetic sequences. In *1st Meeting of the North American Chapter of the Association for Computational Linguistics, Proceedings*. Association for Computational Linguistics.

Kimmo Koskenniemi. 2013a. Finite-state relations between two historically closely related languages. In *Proceedings of the workshop on computational historical linguistics at NODALIDA 2013; May 22-24; 2013; Oslo; Norway*, number 87 in NEALT Proceedings Series 18, pages 53–53. Linköping University Electronic Press; Linköpings universitet.

Kimmo Koskenniemi. 2013b. An informal discovery procedure for two-level rules. *Journal of Language Modelling*, 1(1):155–188.

Krister Lindén, Erik Axelson, Sam Hardwick, Tommi A. Pirinen, and Miikka Silfverberg. 2011. Hfst – framework for compiling and applying morphologies. In Cerstin Mahlow and Michael Piotrowski, editors, *Systems and Frameworks for Computational Morphology 2011 (SFCM-2011)*, volume 100 of *Communications in Computer and Information Science*, pages 67–85. Springer-Verlag.

Mehryar Mohri, Fernando C. N. Pereira, and Michael Riley. 2002. Weighted finite-state transducers in speech recognition. *Computer Speech and Language*, 16(1):69–88.

Mehryar Mohri. 2009. Weighted automata algorithms. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*. Springer.

John Nerbonne and Wilbert Heeringa. 1997. Measuring dialect distance phonetically. In *Computational Phonology: Third Meeting of the ACL Special Interest Group in Computational Phonology*, pages 11–18. SigPHON, Association for Computational Linguistics.

Helmut Schmid. 2005. A programming language for finite state transducers. In *Proceedings of the 5th International Workshop on Finite State Methods in Natural Language Processing (FSMNLP 2005), Helsinki, Finland*.

Harold L. Somers. 1999. Aligning phonetic segments for children's articulation assessment. *Computational Linguistics*, 25(2):267–275, June.

Kristina Toutanova and Robert Moore. 2002. Pronunciation modeling for improved spelling correction. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 144–151, Philadelphia, Pennsylvania, USA, July. Association for Computational Linguistics.