Effect of Language and Error Models on Efficiency of Finite-State Spell-Checking and Correction*

Tommi A Pirinen
University of Helsinki
Department of Modern Languages
FI-00014 Univ. of Helsinki, PO box 24
tommi.pirinen@helsinki.fi
Sam Hardwick
University of Helsinki
Department of Modern Languages
FI-00014 Univ. of Helsinki, PO box 24
sam.hardwick@helsinki.fi

August 2, 2012(draft)

Abstract

We inspect the viability of finite-state spell-checking and contextless correction of non-word errors in morphologically different languages. Overviewing previous work, we conduct large-scale tests involving three languages — covering a broad spectrum of morphological features; English, Finnish and Greenlandic — and a variety of error models and algorithms, including proposed improvements of our own. Special reference is made to on-line three-way composition of the input, the error model and the language model. Tests are run on real-world text acquired from freely available sources. We show that the finite-state approaches discussed are sufficiently fast for high-quality correction, even for Greenlandic which, due to its morphological complexity, is a difficult task for non-finite-state approaches.

1 Introduction

[1] In most applications of spell-checking, efficiency is a limiting factor for selecting or discarding spell-checking solutions. In the case of finite-state spell-checking it is

[1] This is author's pre-print draft and it may differ from the final publication.

known that finite-state language models can efficiently encode dictionaries of natural languages [1], even for polysynthetic languages. The efficiency of finite-state approaches to spelling correction, however, has often been seen as a problem, and therefore most contemporary spell-checking and correction systems are still based on programmatic solutions (e.g. hunspell[2], and its *spell relatives), or at most specialised algorithms for implementing error-tolerant traversal of the finite-state dictionaries [2, 3]. There have also been few fully finite-state implementations for both the detecting and correcting errors [4, 5]. In this paper we further evaluate the use of finite-state dictionaries with two-tape finite-state automatons for mechanism to correct misspellings, and how to optimise the finite-state error models.

To evaluate the general usability and efficiency of finite-state spell-checking we test the system with three languages of typologically different morphological features[3] and *different reference implementations for contemporary spell-checking applications*: English as a sample of a morphologically more isolating language with a basically word-list approach to spell-checking; as a reference implementation we use hunspell's en-US dictionary[4] and for a finite-state version we use a weighted word-list from [6]. The second language we test is Finnish, whose computational complexity has been just beyond the edge of being too hard to implement nicely in hunspell system for example [7]. For a reference implementation we use Voikko[5], with a LAG-based dictionary using Malaga[6]. Our third test language is Greenlandic, a polysynthetic language which is implemented as finite-state system using Xerox's original finite-state morphology formalism [1]. We use Foma [8] for performing the correction task[7]. As a general purpose finite-state library we use HFST[8], which also provides our spell-checking code. There is no free software for us as a reference implementation apart from our solution, as far as we know.

The baseline feature set and the efficiency of spell-checking we are targeting is defined by the currently de facto standard spelling suite in open source systems,

[2] `http://hunspell.sf.net`

[3] we will not go deep into detail of morphological features of these languages. We thank the anonymous reviewer for pointing us to make this rough comparison using translated text piece. We measure from the translations of the *Universal Declaration of Human Rights* (with pre-amble included) thusly: the number of word-like tokens for English is 1,746, for Finnish 1,275 and for Greenlandic 1,063. The count of 15 most frequent tokens are for English 120—28, for Finnish 85—10 and for Greenlandic 38—7.

The average word length is 5.0 for English, 7.8 for Finnish and 14.9 for Greenlandic. For complexity of computational models refer to the Table 2 in this article.

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

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

[6] `http://home.arcor.de/bjoern-beutel/malaga/`

[7] `http://code.google.com/p/Foma/`

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

hunspell. As neither Finnish nor Greenlandic have been successfully implemented in the hunspell formalism, we mainly use them to evaluate how the complexity of a language model affects the efficiency of finite-state spell-checking. For a full-scale survey on the state-of-the-art non-finite-state spell-checking, refer to [9].

The efficiency results are contrasted with the existing research on finite-state spell-checking in [10] and the theoretical results on finite-state error-models in [11]. Our addition primarily comprises the addition of morphologically complex languages with actual cyclic dictionary automata (i.e. infinite dictionaries formed by compounding and recurring derivation) and more complex structure in general, compared to those of English and Arabic. We also point out that previous approaches have neglected to simultaneously constrain the error model and the dictionary with each other in on-line composition, which affords a significant speed benefit compared to generating the two component compositions.

The rest of the paper is organised as follows. In Section 2 we discuss the spell-checking task, current non-finite-state spell-checkers and previously used finite-state methods for spell-checking and correction and propose some possible speed optimisations for the error models. We also investigate algorithmic limitations of finite-state approaches and ways to remedy them. In Section 3 we present the language models, error models and the testing corpora. In Section 4 we present the comparisons of speed and quality with combinations of different language and error models and corpora for spell-checking. In Section 5 we summarise our findings and results, and outline future goals.

2 Methods

A finite-state spell-checker is typically composed of at least two finite-state automata; one for the dictionary of the language, or the *language model*, which contains valid strings of the language, and one automaton to map misspelt words into correct strings, or the *error model*. Both the language models and the error models are usually weighted finite-state automata, in which case the probabilities are of a word being correctly spelled in the language model and of specific misspellings, respectively. We evaluate here the effect of both the language and error model automatons' structure and complexity on the efficiency of the finite-state spelling task.[9]

[9]The methods introduced in this research as well as all materials are free/libre open source. Please see our svn repository `https://hfst.svn.sf.net/svnroot/trunk/fsmnlp-2012-spellers/` for detailed implementation and scripts to reproduce all the results.

2.1 Language Models

The most basic language model for a spell-checking dictionary is a list of correctly spelled word forms. One of the easiest way to create such spell-checker is collecting the word forms from reasonably large corpus of (mostly) correctly spelt texts; additionally we can count the frequency of words and use that as the likelihood $P = \frac{c(\text{wordform})}{\text{Corpussize}}$. For morphologically more isolating languages such as English, this is often a sufficient approach [6], and we use it to create a dictionary for our English spell-checker as well. As a non-finite-state reference point we use hunspell.

For agglutinative languages like Finnish, for which the word-list approach is likely to miss much greater amount of words, one of the commonest approaches is to use right-linear grammars, possibly combined with finite-state rule languages to implement morphophonological alterations [12]. This approach also applies to the newest available free open source and full fledged finite-state Finnish morphological dictionary we found [13]. This language model features productive derivations, compounding and rudimentary probabilistic models. We take as a reference non-finite state language model for Finnish Voikko's implementation in Malaga, which is currently used as a spell-checking component in open source software. It is implemented in a left-associative grammar formalism, which is a potentially less efficient system with more expressive power. It's similar to finite-state formulations in terms of linguistic coverage—the main rationale of original implementation, however, was that it was first openly available approach at that time.

For polysynthetic languages it will be obvious that coverage of any word-list-based approach will be even lower. Furthermore, most simple extensions to it such as affix stripping (as in hunspell) are not adequate for describing word forms. To our knowledge, the only approaches that have been widely used for spell-checking and morphological analysis of Greenlandic have been based on traditional finite-state solutions, such as the Xerox formalisms. In our case we have obtained one from the Internet, that was based on freely available finite-state morphology implementation[10]. For further details we refer to the authors' website `http://oqaaserpassualeriffik.org/`.

2.2 Error Models

The ubiquitous formula for modeling typing errors since the computer-assisted spelling correction began has been the edit distance metric sometimes attributed to [14] and/or [15]. It maps four typical slips of the fingers on a keyboard to events

[10] `https://victorio.uit.no/langtech/trunk/st/kal`

in the fuzzy matching of the misspelt word forms to correct ones, that is, the deletion of a character (i.e. failing to press a key), addition of a character (i.e. hitting an extra key accidentally), changing a character (i.e. hitting the wrong key) and swapping adjacent characters (i.e. hitting two keys in the wrong order).

When modeling edit distance as finite-state automaton, a relatively simple two-tape automaton is sufficient to implement the algorithm [10]. The automaton will consist of one arc for each type of error, and additionally one state for each swapping of character type of error is needed to keep the character pair to be swapped in memory. This means that the trivial nondetermistic finite-state automaton implementing the algorithm is of space complexity $S(V, E, \Sigma) = O(|\Sigma|^2|V| + |\Sigma|^2|E|)$, where Σ is the alphabet of language, V is the set vertices in automaton and E is the set of edges in automaton. This edit distance formulation is roughly feature equivalent to hunspell's `TRY` mechanism.

To further fine-tune this finite-state formulation of edit distance algorithm, it is possible to attach a probability to each of the error events as a weight in weighted finite-state automaton, which corresponds the likelihood of an error, or a confusion factor. This can be used to implement features like keyboard adjacency or OCR confusion factor to the error correction model. This will not modify the structure of the finite-state error models or the search space—which is why we did not test in this article—, but introduction of non-homogenous weights to resulting finite-state network may have an effect on search time. This addition is equivalent to hunspell's `KEY` mechanism.

For English language spelling correction there is also an additional form of error model to deal with competence related misspellings, as opposed to these other models that mainly deal with mistypings, implemented in form of phonemic folding and unfolding. This type of error is very specific to certain type of English texts and is not dealt with in scope of this experiment. This is the `PHON` part of the hunspell's correction mechanism.

After fine-tuning the error models to reimplement hunspell's feature set, we propose variations of this edit distance scheme to optimise the speed of error correction with little or no negative effect to the quality of the correction suggestions. The time requirement of the finite-state spelling correction algorithm is determined by the size of the search space, i.e. the complexity of resulting network when the error model is applied to the misspelt string and intersected with the dictionary[11].

To optimise the application of edit distance by limiting the search space, many

[11]For non-finite-state solutions, the search space is simply the number of possible strings given the error corrections made in the algorithm. For finite-state systems the amount of generated strings with cyclic language and error models is infinite, so complexity calculation are theoretically slightly more complex, however for basic edit distance implementations used in this article the search space complexities are always the same and the amount of suggestions generated finite

traditional spell checkers will not attempt to correct the very first letter of the word form. We investigated whether this decision is a particularly effective way to limit the search space, but it does not appear to significantly differ from restricting edits at any other position in the input.

Dividing the states of a dictionary automaton into classes corresponding to the minimum number of input symbols consumed by that state, we found that the average ambiguity in a particular class is somewhat higher for the first input symbols, but then stabilises quickly. This was accomplished by performing the following state-categorisation procedure:

1. The start state is assigned to class 0, and all other states are assigned to a candidate pool.
2. All states to which there is an (input) epsilon transition from the start state are assigned to class 0 and removed from the candidate pool.
3. This is repeated for each state in class 0 until no more states are added to class 0. This completes class 0 as the set of states in which the automaton can be before consuming any input.
4. For each state in class 0, states in the candidate pool to which there is a non-epsilon transition are assigned to class 1 and removed from the candidate pool.
5. Class 1 is epsilon-completed as in (2-3).
6. After the completion of class n, class $n + 1$ is constructed. This continues until the candidate pool is empty, which will happen as long as there are no unreachable states.

Having this categorisation, we tallied the total number of arcs from states in each class and divided this total by the number of states in the class. This is intended as an approximate measure of the ambiguity present at a particular point in the input. Some results are summarized in table 1.

Further, the size of a dictionary automaton that is restricted to have a particular symbol in a particular position does not apparently depend on the choice of position. This result was acquired by intersecting eg. the automaton `e.+` with the dictionary to restrict the first position to have the symbol `e`, the automaton `.e.+` to restrict the second position, and so on. The sizes of the transducers acquired by this intersection vary in size of the language, number of states and number of transitions, but without any trend according to the position of the restriction. This is in line with the rather obvious finding that the size of the restricted dictionary in terms of number of strings is similarily position-agnostic.

Presumably, the rationale is a belief that errors predominately occur at other positions in the input. As far as we know, the complete justification for this belief remains to be made with a high-quality, hand-checked error corpus.

Class	Transitions	States	Average
0	156	3	52
1	1,015	109	9.3
2	6,439	1,029	6.3
3	22,436	5,780	3.9
4	38,899	12,785	3.0
5	44,973	15,481	2.9
6	47,808	17,014	2.8
7	47,495	18,866	2.5
8	39,835	17,000	2.3
9	36,786	14,304	2.6
10	45,092	14,633	3.1
11	66,598	22,007	3.0
12	86,206	30,017	2.9

Table 1: State classification by minimum input consumed for the Finnish dictionary

On the error model side this optimisation has been justified by findings where between 1.5 % and 15 % of spelling-errors happen in the first character of the word, depending on the text type [16]; the 1.5 % from small corpus of academic texts [17] and 15 % from dictated corpora [18]. We also performed a rudimentary classification of the errors in the small error corpus of 333 entries from [19], and found errors at first position in 1.2 % of the entries, furthermore we noticed that when evenly splitting the word forms in three parts, 15 % of the errors are in the first third of the word form, while second has 47 % and third 38 %, which would be in favor of selecting the error model to discard initial errors[12].

A second form of optimisation that is used by many traditional spell-checking systems as well is to apply a lower order edit distances separately before trying higher order ones. This is based on the assumption that vast majority of spelling errors will be of lower order. In the original account of edit distance for spell-checking, 80 % of the spelling errors were found to be correctable with distance 1 [20].

The third form of optimisation that we test is an attempt to decrease the number of redundant corrections in error models of higher order edit distances than one. This means that things like adding and deleting the same character in successive moves will not be performed. This makes the error model more complicated but reduces the search space, and does not affect the quality of results.

[12]by crude classification we mean that all errors were forced to one of the three classes at weight of one, e.g. a series of three consecutive instances of the same letters was counted as deletion at the first position.

2.3 Algorithms

The obvious baseline algorithm for the task of finding which strings can be altered by the error model in such a way that the alteration is present in the language model is generating all the possible alterations and checking which ones are present in the language model. This was done in [10] by first calculating the composition of the input string with the error model and then composing the result with the language model.

If we simplify the error model to one in which only substitutions occur, it can already be seen that this method is quite sensitive to input length and alphabet size. The composition explores each combination of edit sites in the input string. If any number of edits up to d can be made at positions in an input string of length n, there are

$$\sum_{i=1}^{d} \binom{n}{i}$$

ways to choose the edit site, and each site is subject to a choice of $|\Sigma| - 1$ edits (the entire alphabet except for the actual input). This expression has no closed form, but as d grows to n, the number of choices has the form 2^n, so the altogether complexity is exponential in input length and linear in alphabet size (quadratic if swaps are considered).

In practice (when d is small relative to n) it is useful to observe that an increase of 1 in distance results in an additional term to the aforementioned sum, the ratio of which to the previously greatest term is

$$\frac{n!/(d! \cdot (n-d!))}{n!/((d-1)! \cdot (n-d+1)!)} = \frac{n-d+1}{d}$$

indicating that when d is small, increases in it produce an exponential increase in complexity. For an English 26-letter lowercase alphabet, swap distance 2 and the 8-letter word "spelling", 700 strings are stored in a transducer. With transpositions, deletions, insertions and edit weights this grows to $100,215$ different combinations of output and weight. We have implemented this algorithm for our results by generating the edited strings by lookup, and performing another lookup with the language model on these strings.

Plainly, it would be desirable to improve on this. The intuition behind our improvement is that when editing an input string, say "spellling", it is a wasted effort to explore the remainder after generating a prefix that is not present in the lexicon. For example, after changing the first character to "z" and not editing the second characted, we have the prefix "zp-", which does not occur in our English lexicon. So the remaining possibilities - performing any remaining edits on the remaining 7-character word - can be ignored.

This is accomplished with a three-way composition, in which the input, the error model and the language model simultaneously constrain each other to produce the legal correction set. This algorithm is presented in some detail in [21].

3 Material

For language models we have acquired suitable free-to-use dictionaries for three languages with a great degree of typological variation between them. The dictionaries are readily obtainable on the Internet.

For error models we have created new implementations of the algorithms to create and modify finite-state error models. For a baseline we have created the basic edit distance error models by hand and then modified them automatically to test different variants.

To test the effect of correctness of the source text to speed of spelling-checker we have retrieved of largest freely available open source text materials from the internet, i.e. Wikipedia. The Wikipedia text as itself is an appropriate real-world material as it contains a wide variety of spelling errors. For material with more errors, we have used a simple script to introduce the (further) errors at uniform probability of 1/33 per character; using this method we can also obtain a corpus of errors with correct corrections along them. Finally we have used a text corpus from language different than the one being spelled to ensure that majority of words are not in vocabulary and not fixable by standard error models.

The corpora statistics are provided here for verification and reproducability: the English Wikipedia dump[13] is 34 GiB, the Finnish Wikipedia[14] is 1 GiB, and the Greenlandic Wikipedia[15] is only 7 MiB. From these we have extracted the contents of the articles and picked 100,000 first word tokens for evaluation purposes.

In Table 2 we summarize the sizes of automata in terms of structural elements. On the first row, we give the effective size of alphabet needed to represent the whole dictionary, this is all the Unicode code points that have been used in the dictionary, including alphabets, punctuation and spaces. Next we give the sizes of automata as nodes and arcs of the finite-state automaton encoding the dictionary. Finally we give the size of the automaton as serialised on the hard disk. While this is not the same amount of memory as its loaded data structures, it gives some indication of memory usage of the automaton in running program. As can be clearly seen from

[13] `http://dumps.wikimedia.org/enwiki/20120211/enwiki-20120211-pages-articles.xml.bz2`

[14] `http://dumps.wikimedia.org/fiwiki/20120213/fiwiki-20120213-pages-articles.xml.bz2`

[15] `http://dumps.wikimedia.org/klwiki/20120214/klwiki-20120214-pages-articles.xml.bz2`

the table, the morphologically less isolating languages have quite directly more in all of structural elements of automaton.

Automaton	En	Fi	Kl
Σ set size	43	117	133
Dictionary FSM nodes	49,778	286,719	628,177
Dictionary FSM arcs	86,523	783,461	11,596,911
Dictionary FSM on disk	2.3 MiB	43 MiB	290 MiB

Table 2: The sizes of dictionaries as automata

In the Table 3 we give the same figures for the sizes of error models we've generated. The sigma set size row here shows the amount of alphabets left, when we have removed the parts of alphabet from the error models that is usually not considered to be part of spell-checking mechanism, such as all punctuation that does not occur word-internally and white-space characters[16]. Note, that sizes of error models can be directly computed from its parameters; i.e., the distance, the Σ set size and the optimisation, this table is provided for reference only.

Automaton	En	Fi	Kl
Σ set size	28	60	64
Edit distance 1 nodes	652	3,308	3,784
Edit distance 1 arcs	2,081	10,209	11,657
Edit distance 2 nodes	1,303	6,615	7,567
Edit distance 2 arcs	4136	20,360	23,252
No firsts ed 1 nodes	652	3,308	3,784
No firsts ed 1 arcs	2,107	10,267	11,719
No firsts ed 2 nodes	1,303	6,615	7,567
No firsts ed 2 arcs	4,162	20,418	23,314
No redundancy and 1.s ed 2 nodes	1,303	6,615	7,567
No redundancy and 1.s ed 2 arcs	4,162	20,418	23,314
Lower order first ed 1 to 2 arcs	6,217	30,569	34,909
Lower order first ed 1 to 2 nodes	1,955	9,923	11,351

Table 3: The sizes of error models as automata

4 Evaluation

The evaluations in this section are performed on quad-core Intel Xeon E5450 running at 3 GHz with 64 GiB of RAM memory. The times are averaged over five test runs of 10,000 words in a stable server environment with no server processes or running graphical interfaces or other uses. The test results are measured using

[16]The method described here does not handle run-on words or extraneous spaces, as they introduce lot of programmatic complexity which we believe is irrelevant to the results of this experiment.

the `getrusage` C function on a system that supports the maximum resident stack size `ru_maxrss` and user time `ru_utime` fields. The times are also verified with GNU `time` command. The results for hunspell, Voikkospell and Foma processes are only measured with time and top. The respective versions of the software are Voikkospell 3.3, hunspell 1.2.14, and Foma 0.9.16alpha. The reference systems are tested with default settings, this means that they will give some first suggestions whereas our system will calculate all strings within given error model.

In the Table 4 we measure the speed of the spell-checking process of the native language Wikipedia with real-world spelling errors and unknown strings. The error model rows are defined as follows: on the *Reference impl.* row, we test the spell-checking speed of the hunspell tool for English, and Voikkospell tool for Finnish. On the *edit distance 2* row we use the basic traditional edit distance 2 without any modifications. On the row *lower order first*, we apply a lower order edit distance model first, then if no results are found, a higher order model is used. On the *No first edits* row we use the error model that may not modify the first character of the word or the first character after compound word boundary. On the *Avoid redundancy* row we use the error model edit distance 2 with the redundant edit combinations removed. On the *Avoid redundancy and first edits* rows we use combined error model of *No first edits* and *Avoid redundancy* functionalities. In the tables and formulae we routinely use the language codes to denote the languages: *en* for English, *fi* for Finnish and *kl* for Greenlandic (Kalaallisut).

Error model	**En**	**Fi**	**Kl**
Reference impl.	9.93	7.96	11.42
Generate all edits 2	3818.20	118775.60	36432.80
Edit distance 1	0.26	6.78	4.79
Edit distance 2	7.55	220.42	568.36
No firsts ed 1	0.44	3.19	3.52
No firsts ed 2	1.38	61.88	386.06
No redundancy ed 2	7.52	4230.94	6420.66
No redundancy and firsts ed 2	1.51	62.05	386.63
Lower order first ed 1 to 2	4.31	157.07	545.91

Table 4: Effect of language and error models to speed (time in seconds per 10,000 word forms)

The results show that not editing the first position does indeed give significant boost to the speed, regardless of language model, which is of course caused by the significant reduction in search space. However, the redundancy avoidance does not seem to make a significant difference. This is most likely because the amount of duplicate paths in the search space is not so proportionally large and their traversal will be relatively fast. The separate application of error models gives the expected timing result between its relevant primary and secondary error models. It should be

noteworthy, that, when thinking of real world applications, the speed of the most of the models described here is greater than 1 word per second (i.e. 10,000 seconds per 10,000 words).

We measured the memory consumption when performing the same tests. Varying the error model had little to no effect. Memory consumption was almost entirely determined by the language model, giving consumptions of 13-7 MiB for English, 0.2 GiB for Finnish and 1.6 GiB for Greenlandic.

The memory measurements show an interesting feature of our method of applying the spell-checking and correction here; the memory usage stays nearly same regardless of the selected error model. We assume that the memory figures shown in the table are dominated by the memory stamp of the dictionary automaton and the error model automaton loaded into memory, and the intermediate structures used by the error-applying traversal of the automaton will not typically contribute much (approx. half mebibyte[17] for each larger edit distance), since they are created and deleted on-the-fly. Notable discrepancy to this pattern is Greenlandic with larger edit distances, it seems to exhaust whole memory which may be caused an inadvertent (epsilon) loop in the morphology. One practical thing to note here is, the memory footstamp does give suggestion of how much available RAM is needed to get speed measurements of the table 4, as our experience shows that if the dictionary automaton is partially in swap memory, the speed will decrease by order of magnitude in current operating systems, e.g. on low-end minilaptops and Greenlandic dictionaries.

To measure the degradation of quality when using different error models we count the proportion of suggestion sets that contain the correct correction among the corrected strings. For this test we use automatically generated corpus of spelling errors to get the large-scale results.

Error model	En	Fi	Kl
Edit distance 1	0.89	0.83	0.81
Edit distance 2	0.99	0.95	0.92
Edit distance 3	1.00	0.96	—
No firsts ed 1	0.74	0.73	0.60
No firsts ed 2	0.81	0.82	0.69
No firsts ed 3	0.82	—	—

Table 5: Effect of language and error models to quality (recall, correct suggestion found per correction set)

This test with automatically introduced errors shows us that with uniformly

[17] this is a standard SI measure unit, if unfamiliar, refer to `http://en.wikipedia.org/wiki/Mebibyte`

distributed errors the penalty of changing error model to ignore the word-initial could be significant. This contrasts to our findings with real world errors, that the distribution of errors tends towards the end of the word, described in 2.2 and [16], but it should be noted that degradation can be as bad as given here.

Finally we measure how the text type used will effect the speed of the spell-checking. As the best-case scenario we use the unmodified texts of Wikipedia, which contain probably the most realistic native language speaker-like typing error distribution. For text with some more errors, where majority of errors should be recoverable we introduce more automatically generated errors in the Wikipedia texts. Finally to see the performance on the worst case scenario where most of the words have unrecoverable spelling errors we use the texts from other languages, in this case English texts for Finnish and Greenlandic spell-checking and Finnish texts for English spell-checking, which should bring us close to the lower bounds on performance. The effects of text type (i.e. frequency of non-words) to speed of spell-checking is given in table 6. All of the tests in this category were performed with error models under row *avoid redundancy and firsts ed 2* in previous tables, which gave us the best speed/quality ratio in the previous tests.

Error model	**En**	**Fi**	**Kl**
Native Lang. Corpus	1.38	61.88	386.06
Added automatic errors	6.91	95.01	551.81
Text in another language	22.40	148.86	783.64

Table 6: Effect of text type on error models to speed (in seconds per 10,000 word-forms)

Here we chiefly note that the amount of non-words in text reflects directly to the speed of spell-checking. This shows that the dominating factor of the speed of spell-checking is indeed in the correcting of wrong results.

5 Conclusions And Future Work

In this article, we built a full-fledged finite-state spell-checking system from existing finite-state language models and generated error models. We showed that using online serial composition of the word form, error model and dictionary instead of previous approaches is usable for morphologically complex languages. Furthermore we showed that the error models can be automatically optimised in several ways to gain some speed at cost of recall.

We showed that the memory consumption of the spell-checking process is mainly unaffected by the selection of error model, apart from the need to store

a greater set of suggestions for models that generate more suggestions. The error models may therefore be quite freely changed in real world applications as needed.

We verified that correcting only the first input letter affords a significant speed improvement, but that this improvement is not dependent on the position of such a restriction. This practice is somewhat supported by our tentative finding that it may cause the least drop in practical recall figures, at least in Finnish. It is promising especially in conjunction with a fallback model that does correct the first letter.

We described a way to avoid having a finite-state error model perform redundant work, such as deleting and inserting the same letter in succession. The practical improvement from doing this is extremely modest, and it increases the size of the error model.

In this research we focused on differences in automatically generated error models and their optimisations in the case of morphologically complex languages. For future research we intend to study more realistic error models induced from actual error corpora (e.g. [22]). Research into different ways to induce weights into the language models, as well as further use of context in finite-state spell-checking (as in [19]), is warranted.

Acknowledgements

We thank the anonymous reviewers for their comments and the HFST research team for fruity discussions on the article's topics. The first author thanks the people of *Oqaaserpassualeriffik* for introducing the problems and possibilities of finite-state applications to the morphologically complex language of Greenlandic.

References

[1] Kenneth R Beesley and Lauri Karttunen. *Finite State Morphology*. CSLI publications, 2003.

[2] Kemal Oflazer. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Comput. Linguist.*, 22(1):73–89, 1996.

[3] Måns Huldén. Fast approximate string matching with finite automata. *Procesamiento del Lenguaje Natural*, 43:57–64, 2009.

[4] Klaus Schulz and Stoyan Mihov. Fast string correction with levenshtein-automata. *International Journal of Document Analysis and Recognition*, 5:67–85, 2002.

[5] Tommi A Pirinen and Krister Lindén. 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*, pages 13–18, Valletta, Malta, 2010.

[6] Peter Norvig. How to write a spelling corrector. referred 2011-01-11, available `http://norvig.com/spell-correct.html`, 2010.

[7] Harri Pitkänen. Hunspell-in kesäkoodi 2006: Final report. Technical report, 2006.

[8] Måns Huldén. Foma: a finite-state compiler and library. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics: Demonstrations Session*, EACL '09, pages 29–32, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[9] Roger Mitton. Ordering the suggestions of a spellchecker without using context*. *Nat. Lang. Eng.*, 15(2):173–192, 2009.

[10] Ahmed Hassan, Sara Noeman, and Hany Hassan. Language independent text correction using finite state automata. In *Proceedings of the Third International Joint Conference on Natural Language Processing*, volume 2, pages 913–918, 2008.

[11] Petar Nikolaev Mitankin. Universal levenshtein automata. building and properties. Master's thesis, University of Sofia, 2005.

[12] Kimmo Koskenniemi. *Two-level Morphology: A General Computational Model for Word-Form Recognition and Production*. PhD thesis, University of Helsinki, 1983.

[13] Tommi A Pirinen. Modularisation of finnish finite-state language descriptiontowards wide collaboration in open source development of morphological analyser. In *Proceedings of Nodalida*, volume 18 of *NEALT proceedings*, 2011.

[14] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics—Doklady 10, 707710. Translated from Doklady Akademii Nauk SSSR*, pages 845–848, 1966.

[15] Fred J Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, (7), 1964.

[16] Meenu Bhagat. Spelling error pattern analysis of punjabi typed text. Master's thesis, Thapar University, 2007.

[17] Emmanuel J Yannakoudakis and D Fawthrop. An intelligent spelling error corrector. *Information Processing and Management*, 19(2):101–108, 1983.

[18] Karen Kukich. Techniques for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, 1992.

[19] Tommi Pirinen, Miikka Silfverberg, and Krister Linden. Improving finite-state spell-checker suggestions with part of speech n-grams. In *Internatational Journal of Computational Linguistics and Applications IJCLA (to appear)*, 2012.

[20] Joseph J. Pollock and Antonio Zamora. Automatic spelling correction in scientific and scholarly text. *Commun. ACM*, 27(4):358–368, April 1984.

[21] Krister Lindén, Erik Axelson, Senka Drobac, Sam Hardwick, Miikka Silfverberg, and Tommi A Pirinen. Using hfst for creating computational linguistic applications. In *Proceedings of Computational Linguistics - Applications, 2012*, page to appear, 2012.

[22] Eric Brill and Robert C. Moore. An improved error model for noisy channel spelling correction. In *ACL '00: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 286–293, Morristown, NJ, USA, 2000. Association for Computational Linguistics.