Efficient generation and ranking of spelling error corrections

Mikael Tillenius

NADA report TRITA-NA-E9621, 1996

Abstract

An efficient method for generating and ranking spelling error corrections is described. This method can be used with dictionaries with only one operation—check if a given word is in the dictionary or not. The method is intended for Swedish, but can easily be modified for other languages. Given a misspelled word, i.e., a word not in the dictionary, the corrections are generated by applying editing operations on the word. An efficient algorithm to generate corrections for compound words is also described. The corrections are the ranked using a combination of edit distances and word frequencies.

Effektiv generering och rangordning av rättstavningsförslag

Sammanfattning

En effektiv metod för att generera och rangordna rättstavningsförslag presenteras. Metoden kan användas för ordlistor med endast en operation – finns ett givet ord i ordlistan eller inte. Metoden är avsedd för svenska men kan lätt modifieras för andra språk. Rättstavningsförslagen för ett felstavat ord, dvs. ett ord som inte finns i ordlistan, genereras genom editering av det felstavade ordet. En effektiv algoritm för att generera förslag för sammansatta ord beskrivs också. Förslagen rangordnas med en kombination av editeringsavstånd och ordfrekvenser.

Acknowledgments

This report is my master's thesis, and was done at the Department of Numerical Analysis and Computing Science (NADA) at Royal Institute of Technology (KTH).

First of all I would like to thank my supervisor, Viggo Kann, for his encouragement, good suggestions and help with practical details.

I would also like to thank some people and organizations who made my work easier by giving their software away for free.

Free Software Foundation for GCC, their excellent compiler, for the editor GNU Emacs, one of the few editors I know that can handle lists of several hundred thousand words, and for gawk, an awk clone, and many other programs I use almost every day.

Donald E. Knuth for TEXwhich was used to typeset this report.

Linus Torvalds and many more who made Linux, a free Unix clone, and therefore made it possible for me to make much of the work at home on my PC.

Many others whose software we use everyday without thinking of it.

Mikael Tillenius

Contents

1 Introduction

This report describes a method for generating and ranking spelling error corrections. It is intended as an addition to the program Stava, which was developed by Joachim Hollman and Viggo Kann at the Department of Numerical Analysis and Computing Science (NADA) at the Royal Institute of Technology (KTH).

Stava is a program to find spelling errors and give error correction suggestion and is primarily intended for Swedish text. Stava uses a method called Bloom filters to store its dictionary. A Bloom filter has only two operations, checking if a word is already in it and adding a new word. This gives some restrictions on how to find error correction suggestions. Stava is described in more detail later in this report.

The goal of my work was to find a fast method to generate all probable spelling corrections and then rank them in order of probability.

Some of the material in this report is specific to spelling error correction in Swedish, but most of it is language independent.

1.1 Some Terminology

I would first like to introduce some terminology which will be used in the rest of this report. A **dictionary** is a set of words (or strings) which are considered to be correct. **Error detection** is the process of finding misspelled words. **Error correction** is the process of generating one or more suggestions of correct words given a misspelled word. These suggestions can be **ranked** in order of probability. When a misspelling of one word results in another correct word (e.g. *form* → *from*), we will call it a **real word** error. An **automatic** spelling corrector would correct any error without any human interaction but an **interactive** spelling corrector will leave the real decision to the user of the spelling corrector. An n**-gram** is a sequence of n characters (or words). Usually n-grams are used are used to represent all possible sequences of characters (or words) that occur in a language. If n is 1, 2 or 3 they are called unigrams, bigrams or trigrams respectively.

1.2 Spelling Error Correction

Detection and correction of spelling errors is an old problem. Much research has been done in this area over the years, and more has to be done. The existing tools are very useful, but they do not entirely replace manual proofreading. I will try to give a quick overview to the subject here. For more details, see [Kukich 1992].

Most of the research has been done for English (and in fact, many spelling correctors for Swedish are only modified versions of the original English version). The results from this research are not always valid for Swedish.

There are two main differences between Swedish and English that I would like to mention. First, in Swedish it is possible to construct new compound words in an almost unlimited way. (E.g. *Förstamajdemonstrationstalarstolsflaggbärarförman* is, strictly speaking, a correct word, even though it might be considered bad style to use such long words.) Secondly there is a stronger relationship between spelling and pronounciation in Swedish than in English. This makes it possible to handle many phonetic errors in the same way as typographic errors.

1.2.1 Spelling Errors

To make a good spelling corrector it is essential that one knows what kinds of spelling errors there are and how often they occur. For typed text the errors can be divided into two categories:

1 **typographic errors** which occur because the typist accidentally presses the wrong key, presses two keys, presses the keys in the wrong order etc. (e.g. *the* → *teh*).
2 **phonetic errors** where the misspelling is pronounced the same as the intended word but the spelling is wrong (e.g. *two* → *to*).

For typographic errors the keyboard is important. It is much more usual to accidentally substitute a key for another if they are placed near each other on the keyboard. The types and frequencies of typographical errors differ much for different writers and also for the same writer in different situations.

For other input devices than keyboards the typographic errors are replaced by other types of errors. Optical character recognition (OCR) for example might substitute an O for a D or might report a character as unrecognizable.

1.2.2 Dictionaries

A dictionary is a list of words that are assumed to be correct. Dictionaries can be represented in many ways, each with their own characteristics like speed and storage requirements.

At first one might think that the more words a dictionary contains the better it is. But when more words are added to the dictionary the risk for real word errors increases. This is mostly a problem for short words; long words generally differ more than short words. A good substitute for a large dictionary might be a dictionary with most common word combined with a set of additional dictionaries for specific topics such as computer science or economy.

A big dictionary also uses more space and may take longer time to search.

1.2.3 Isolated Word Correction

Isolated word correction is the oldest technique. It is much simpler than context dependent correction. To find all misspelled words, one only has to check all words to see if they belong to the dictionary or not. Those words that do not are misspelled. The problem with this is that it is impossible to find real word errors i.e. misspellings that result in another correct word.

Once a misspelled word is found, one has to generate corrections for it. This can be viewed as defining a distance measure between words. The corrections are then the words in the dictionary with the smallest distance to the misspelling. Thus the distance measure should be chosen so that the distance between a misspelling and the intended word is small. There are several ways to define this measure. Traditionally, this measure has been chosen so that the correction suggestions are easy to generate. This has also put some restrictions on how the dictionary is represented. For example, it might be required that the dictionary is sorted alphabetically and partitioned into multiple dictionaries after word length.

It would be nice to define an optimal distance measure, and then compare other distance measures with this optimal distance measure. The distance between a misspelling m and a word w in the dictionary could be defined as

$$D(m, w) = 1 - P(m \text{ is a misspelling of } w).$$

This would maximize the number correct corrections. It would still be impossible to always do the right correction of a misspelled word, and we would still have the problem of real word errors. It would be possible to handle some real word errors by defining the distance measure as

$$D(m, w) = 1 - P(m \text{ was written when } w \text{ was intended})$$

and treat every word where the distance between the word and itself is greater than the distance between the word and any other word in the dictionary. This would make it possible to include many unusual words in the dictionary without increasing the risk for real word errors too much.

It should be noted that this measure is very dependent on the input text. If the input text comes from an OCR device there is a high probability that a D is replaced by an O, but this should be very uncommon for typed text since D and O are placed far from each other on a normal keyboard. There are also big differences between different typists and different situations.

It is easier to correct long words than short words, since there are fewer long words than short words that spell almost the same way as another. With long words I mean simple words, not compound words. Compound words are an even bigger problem since they are built by shorter words and we don't know the word boundaries. Allowing almost unbounded compounding of words (which might be useful for a language like Swedish), increases both the number of real word errors and the number of words spelled similar to another.

Isolated word correction can be useful in an interactive spelling corrector since it catches many errors that might otherwise be uncorrected. In most cases it is also able to suggest the right word, especially if it is a long word. It can only be a complement to manual proofreading since it doesn't catch real word errors. Neither can it be automatic since it sometimes makes wrong corrections.

1.2.4 Context Sensitive Correction

Context sensitive correction is much harder, but also has more capacity than isolated word correction. By using the context in which a word appears it makes better guesses. It can also be used to detect real word errors and to detect grammatical errors.

The amount of context information used may vary very much, from simple word-pair statistics to programs that try to understand the text.

1.3 Stava

Stava uses a Bloom filter to code its dictionary. A Bloom filter is a vector of boolean values and a number of hash functions. To store a word in the dictionary you calculate each hash function for the word and set the vector entries corresponding to the calculated values to true. To find out if a word belongs to the dictionary, you calculate the the hash values for that word and look in the vector. If all entries corresponding to the values are true, the word belongs to the dictionary, otherwise it doesn't.

The good thing about Bloom filters is that they are compact and that the time required to find out if a word belongs to the dictionary or not is independent of the size of the dictionary. The bad things are that it is hard to store any information (such as which part of speech the word belongs to) together with the word and that some words might be reported as belonging to the dictionary even though they never were added to it. The probability for this can be lowered by increasing the size of the vector or increasing the number of hash functions.

Bloom filters also have another advantage—it is impossible to recreate the original dictionary from the Bloom filter. The only way to find out what words are stored in the dictionary, is to generate all possible letter combinations and test if they belong to the dictionary or not. The problem is not to generate all letter combinations because by limiting the word length and using n-grams it is possible to limit the number of combinations.

Instead the problem is that too many incorrect words will be accepted by the dictionary. The reconstructed dictionary will contain too many false words.

Stava really uses three dictionaries, the exception list, the last part list and the first part list. The last part list contains words that may be an independent word or the last part of a compound word. A word in the first part list can be the first or middle part of a compound word. A word in the exception list cannot be part of a compound word. By checking if a word is composed of one or more words from the first part dictionary and a word from the last part dictionary, compounded words can be handled.

Stava also has a heuristics for generating inflections. This consists of rules on the form:

$$-orna \ \leftarrow \ -a, -an, -or$$

This rules says that if *docka* (*doll*), *dockan* (the *doll*) and *dockor* (*dolls*) all belong to the dictionary, then *dockorna* (*the dolls*) also is a valid word. Thus not all inflections have to be stored in the dictionary. These rules have to be selected carefully so that no invalid word can be generated.

For more information about Stava, see [Domeij et al. 1995].

2 Analysis of Methods

To design a spelling corrector one has to consider several factors. First it should do a good job selecting and ranking corrections. Secondly it should be efficient in terms of memory requirements and computational power.

The first requirement is maybe the most challenging. The first observation one should make is that isolated word correction might not be enough. For example, should the correction of *ater* be *after*, *later*, *ate* or *alter*? Many short words are so similar that several of them are likely to be misspelled to the same word. Furthermore, a misspelling of one word may result in another correct word (e.g. *from* $\rightarrow$ *form*).

Using context information it is possible to achieve much better error detection and correction. The question is: How do we use the context information? This is an area where more research needs to be done.

I will make a short survey over correction techniques that have been used by others. The techniques are evaluated with their usability in Stava in mind. For a more general description, see [Kukich 1992].

2.1 Isolated Word Methods

The isolated word methods I will describe are: edit distance, similarity keys, rule-based techniques, n-grams, probabilistic techniques and neural nets. All of these methods can be thought of as calculating a distance between the misspelled word and each word in the dictionary, as described in the previous chapter. The shorter the distance the higher the dictionary word is ranked.

2.1.1 Edit Distance

Edit distance is a simple technique. The distance between two words is the number of editing operations required to transform one of the words into the other. The allowed editing operations are: remove one character, insert one character, replace one character with another or transpose two adjacent characters.

Edit distance is useful for correcting errors resulting from keyboard input, since these are often of the same kind as the allowed edit operations. If we remove the possibility to transpose characters, it is usable for OCR also. It is not quite as good for correcting phonetic spelling errors, especially if the difference between spelling and pronunciation is big as in English or French. It works better for some common Swedish misspellings.

A variation on edit distance is to assign different distances for different editing operations and the letters involved in these operations. This leads to a finer distinction between common and uncommon typographic errors.

2.1.2 Similarity Keys

Similarity keys are based on some method to transform a word into a similarity key. This key should reflect the characteristics of the word. The most important characteristics are placed first in the key. All words in the dictionary are transformed and sorted after the similarity key. A misspelled word is then also transformed and similar keys can be efficiently searched for in the sorted list.

This technique gives a linear ordering of all words. Words that are close to each other in this ordering are considered similar. With a good transformation algorithm this method can handle both keyboard errors and phonetic errors. The problem is to design this algorithm. See [Pollock & Zamora 1984] for an example of a good algorithm for English.

Similarity keys are not very useful in Stava since we cannot use the fast searching if we are using Bloom filters. In fact it will be difficult to do any efficient searching at all.

2.1.3 Rule-based Methods

Rule-based methods are interesting. They work by having a set of rules that capture common spelling and typographic errors and applying these rules to the misspelled word. Intuitively these rules are the "inverses" of common errors. Each correct word generated by this process is taken as a correction suggestion. The rules also have probabilities, making it possible to rank the suggestions by accumulating the probabilities for the applied rules. Edit distance can be viewed as a special case of a rule-based method with limitation on the possible rules.

Rules-based methods could be used in Stava, but for several reasons described in the next section I decided not to.

2.1.4 N-grams

N-grams can be used in two ways, either without a dictionary or together with a dictionary.

Used without a dictionary, n-grams are used to find in which position in the misspelled word the error occurs. If there is a unique way to change the misspelled word so that it contains only valid n-grams, this is taken as the correction. The performance of this method is limited. Its main virtue is that it is simple and does not require any dictionary.

Used together with a dictionary, n-grams are used to define the distance between words, but the words are always checked against the dictionary. This can be done in several ways, for example check how many n-grams the misspelled word and a dictionary word have in common, weighted by the length of the words.

N-grams are used in Stava, not to find or rank error correction suggestions, but to limit the number of false words accepted by the Bloom filter and to increase the speed.

2.1.5 Statistical Methods

Statistical methods are based on some statistical features of the language. Two common methods are transition probabilities and confusion probabilities.

Transition probabilities are similar to n-grams. They give us the probability that a given letter or sequence of letters is followed by another given letter. Transition probabilities are not very useful when we have access to a dictionary.

Confusion probabilities give us the probability that a given letter was has been substituted for another letter.

2.1.6 Neural Networks

Neural networks is also an interesting and promising technique, but it seems like it has to mature a bit more before it can be used in a general purpose spelling corrector like Stava. The current methods are based on back-propagation networks, using one output node for each word in the dictionary and an input node for every possible n-gram in every position of the word, where n usually is one or two. Normally only one of the outputs should be active, indicating which dictionary word the network suggests as a correction.

This method works for small ($<$ 1000 words) dictionaries, but it doesn't scale well. The time requirements are to big on traditional hardware, especially in the learning phase.

2.2 Context-dependent Methods

Context-dependent methods could make it possible to correct some real-word errors and could be used to make a better ranking of the suggestions, possibly good enough to make it automatic. There are two main approaches to use the context information, trying to parse text according to some grammatical rules and word n-grams.

2.2.1 Natural Language Processing

By trying to match the given text against a grammar for the language it is possible to find and correct both spelling errors and grammatical errors. If we find that some word doesn't match the grammar rules we can assume that the word is incorrect. We can then apply some isolated word method to generate corrections and if one of those match the grammar rules we have found a possible correction. This is not as easy as it might sound.

I decided not to try this method since it seemed to be far too much work for a master's degree.

2.2.2 Word N-grams

Using statistics about word n-grams frequencies seems to be a simple method to catch at least some context-information. The biggest problem is the large number of words. Unigrams and bigrams can be handled by a normal personal computer today but trigrams would need some clever compression technique to fit in the memory. Another problem is the huge amount of text that is needed to collect the statistics.

One way to solve this problem would be to use part-of-speech n-grams instead. This decreases the memory requirement, but it is also a much cruder measurement. Since it is hard to save part-of-speech information in an efficient way using Bloom filters I did not consider this method.

2.3 Evaluation of Methods

After considering the methods described above, I decided to try a combination of some isolated word method and word bigrams. It was tempting to use some general rule-based technique, since it is very powerful. The problem is to make it fast enough and to find the rules it should use. Instead of using general rules, I chose to use a variant of minimum edit distance. In normal edit distance measure, each of the operations insert, remove, replace and transpose are counted equal. By modifying the algorithm so that different operations give different penalties, I hoped to increase its usability. The penalties are dependent both of the edit operation and the letters surrounding the place of the operation.

For insertion the penalty is dependent on the letter inserted and the letter following it. For deletion the penalty is dependent on the letter deleted and the letter following it. For replacement the penalty is dependent on the new letter and the letter it replaces. For transposing the penalty is dependent on the two letters that are transposed.

It should also be an extra penalty for changing the first letter in a word since it is uncommon for the first letter to be wrong. These rules can correct all of the normal keyboard typing errors and make it possible to code their probabilities (e.g. an *a* is more often mistyped as a *s* than as a *p* on a normal keyboard). The rules are also powerful enough to correct some phonetic errors. Since, as mentioned before, the correlation between spelling and pronunciation is high in Swedish, these rules work quite well for most common Swedish phonetic errors. They probably work less well for English or French. The phonetic errors they are able to correct include doubling and undoubling of consonants (e.g. *spel* $\leftrightarrow$ *spell*, *tik* $\leftrightarrow$ *tick*).

The penalties can be generated rather easy by collecting statistics of real spelling and typing errors.

By removing the possibility of transposition (for example by setting a very high penalty for transpositions) the rules can also be used for OCR text, but another set of rules (allowing corrections of framing errors such as $ri \rightarrow n$ or $m \rightarrow iii$) would probably work much better.

To increase the precision of the ranking of the generated correction suggestions it is also possible to use word frequency information. It is more probable that a misspelling is a misspelling of a common word than of an uncommon word. Therefore common words should be ranked higher than uncommon words.

To test the correction accuracy of the algorithm I collected a list of misspellings from two different sources. The first source was unedited news articles. By checking the spelling in these articles with Stava I could extract a list of misspelled words. I then removed most names, abbreviations and foreign words from this list. Then a list of corrections of these words could be made by finding the misspellings in the original text and use the context to find the correct word.

The second source was a collection of student essays, which was already marked with corrections. By filtering out the errors and their corresponding corrections and then removing those errors that were grammatical errors, rather than spelling errors, a list of misspellings and corrections was found. This list also contained real word errors.

These two lists were then combined into one list with 729 misspellings and their corresponding corrections.

The methods edit distance and word frequencies were then tested against this list. The results were that edit distance and word frequencies performed rather well by themselves, and even better together (see table 1). Word bigram frequencies did not perform very well (see table 2). It was tested on a set of 50 sentences, both alone and together with one or both of edit distance and word frequencies. The reason for the bad performance was simple, most of the word pairs in the text was not part of my list of 200 000 word pairs. Word bigrams might work better on texts with a smaller vocabulary or it might be used to correct simple grammatical errors. For spelling correction of texts on any subject, a much larger list of bigrams seems to be necessary.

Table 1. Performance with different methods. 729 words. The columns 1, 2 and 3 tell whether the correct word was the first, second or third suggestion. None means no ranking was performed, the suggestions were presented in the order they were generated.

method	1	2	3
none	204 (28%)	71 (10%)	16 (2%)
word freq.	356 (49%)	42 (6%)	26 (4%)
edit dist.	388 (53%)	55 (8%)	16 (2%)
edit dist.+word freq.	440 (60%)	28 (4%)	10 (1%)

Table 2. Performance of bigrams. 50 sentences. The columns 1, 2 and 3 tell whether the correct word was the first, second or third suggestion.

method	1	2	3
none	16 (32%)	4 (8%)	0 (0%)
word bigrams	17 (34%)	4 (8%)	0 (0%)
word freq.	31 (62%)	2 (4%)	0 (0%)
edit dist.	20 (40%)	7 (14%)	5 (10%)

With these results in mind I decided to use a combination of edit distance and word frequencies and not use word bigrams. Using more general rules than editing operation would have corrected a few more phonetic errors, but considering the problems mentioned at the beginning of this sections and the fact that the simple editing operations performed well enough, I decided not to use them.

2.4 Implementation

A spelling corrector must give good correction suggestions but it also has to be fast and easy to use. These aspects must be weighted against each other. I decided to first test how good the selected methods could be and then optimize them for speed if they were good enough.

The distance between a misspelling and a given correction suggestion is the sum of the penalties given to the suggestion as described below. The suggestions are sorted and presented with increasing distance.

2.4.1 Implementation of Edit Distance

First I collected statistics over errors in the test set that could be corrected by a single editing operation. The statistics that was collected was the number of times (where *a* and *b* are any letter in the Swedish alphabet):
(1) *a* was deleted when it appeared directly before *b*
(2) *a* was deleted when it appeared directly after *b*
(3) *a* was directly inserted before *b*
(4) *a* was inserted directly after *b*
(5) *a* and *b* were transposed when *a* appeared directly before *b*
(6) *a* was replaced by *b*.

The number of times each editing operation resulted in a correction was then weighted against the number of times the editing operation was possible.

Both (1) and (2) showed that doubling of consonants was a very common error. Otherwise, (2) gave very little information. In the same way (3) and (4) showed that it was very common not to double a consonant when it should be. Transpositions (5) were very uncommon. No information about the letters involved could be extracted. Finally, (6) showed that letters pronounced in the same or a similar way were often confused. We can see that most errors were phonetic rather than typographic.

Using this statistics I defined the distance for the different editing operations. Since (1) and (2) contained primarily the same information, and likewise with (3) and (4), only (1), (3), (5) and (6) were used. The distances were defined more or less ad hoc. Then they were slightly adjusted by testing how well they performed.

This resulted in the following rules: Deletion, transposition and replacement normally give a penalty of 10, while insertions normally give a penalty of 7. In addition to these general rules, some special cases are handled. Deleting an occurrence of any of the characters *b*, *d*, *f*, *g*, *j*, *l*, *m*, *n*, *p*, *r*, *s*, *t* or *v* before another occurrence of the same character, or a *c* before a *k*, give a penalty of 6. In the same way, inserting one of the characters in the list above before another occurrence of the same character, or inserting a *c* before a *k* gives a penalty of 3. These rules handle common cases of doubling and undoubling of consonants. I also noted that it was slightly more common to replace a vowel whit another vowel rather than by a consonant so replacing a vowel with another vowel gives a penalty of 9 instead of 10. In addition to this, a few pairs of characters were replaced more often than others (e.g. an *ä* was replaced by an *e*, a *g* was replaced by a *j*). These cases give a penalty of 6 or 8.

There is also an extra penalty of 10 for any insertion, deletion or replacement in the first position of a word.

The penalties for each edit operation can be stored in a two-dimensional vector indexed by the two characters involved. This requires very little memory compared to the memory required by the dictionaries.

Since a dictionary based on Bloom filter may accept words that were never inserted into the dictionary, each word generated by the editing operations is also checked against a list of those 4-grams that appear in the words inserted into the dictionary. This list can be coded by a 4-dimensional vector of booleans.

2.4.2 Implementation of Word Frequencies

The word frequencies were then tested. Three questions were interesting, first how should the frequency of a word map to a penalty, how much should word frequencies affect the ranking compared with edit distance, and how should the frequencies be stored. After some testing I decided to divide the words into classes after a logarithmic scale (e.g., all words occurring between 10 and 100 times are in one class, all words occurring between 100 and 1000 times in another and so on). After testing some different numbers of classes I saw that about 10 classes was optimal. Fewer classes resulted in worse performance, more classes didn't increase the performance.

When using both edit distance and word frequencies together, word frequencies should be more important than edit distance. In Stava, an unusual editing operation results in a penalty of 10. A word in the most common class gives no penalty and a word in the most uncommon class gives a penalty of 18. If the word does not appear in the list of word frequencies, a penalty of 25 is given. A list of about 50 000 words and corresponding frequencies was used.

The word frequencies were stored in another dictionary, also coded by a Bloom filter, by concatenating each word with a letter representing the class the word belongs to and then storing it in the dictionary. E.g., the word *och* which belongs to the most common class is concatenated with an *A* resulting in *ochA* and then stored in the dictionary. Then it is possible to find out which class a given word belongs to by probing the dictionary for the word concatenated with the letter representing each class . E.g., to find out which class *och* belongs to, probe for *ochA*, *ochB* and so on until the word is found or all classes have been probed for.

Since the word frequency dictionary is only used to rank the spelling error suggestions generated by the edit distance method, it is possible to use a Bloom filter with comparatively high error probability which means it can be faster and/or require less memory than the other dictionaries.

2.4.3 Speed Optimizations

Generating corrections by applying all possible editing operations to the misspelled word takes too long time. Since the number of editing operations (an thus the time requirements) grows linearly with the length of the word ($59l + 28$ if we have 29 characters in the alphabet and l is the length of the word), long words is a problem. Allowing more than one editing operation per word would take even longer time (roughly proportional to the number of operations possible with only one operation per word to the power of the number of operations per word). This is mostly a problem for compounded words since non-compounded words normally tend to be short.

By using the table of 4-grams present in the words stored in the dictionary, it is often possible to find one or more positions in the misspelled word where 4-grams don't match. Then the editing operations doesn't have to be performed for all possible positions, but only for positions where they might correct the 4-gram. When allowing more than one edit operation per word, the first edit operations must be applied at or before the first 4-gram mismatch, since the second edit operation is always done at a position after the first one.

By setting an upper limit for the length of a non-compound word and another limit for the length of each part of a compound word, it is possible to limit the time required. If these limits are set to the length of the longest word in the dictionary it does not affect the result. In practice, they can be set slightly lower, so that only a few words in the dictionary are longer than this limit without affecting the results noticeably, but decreasing the time required a lot. I use a limit of 22 characters per word in both cases. If a misspelling is longer than this limit or if none of the editing operations results in a valid word, the misspelling is assumed to be a misspelling of a compounded word. In that case the misspelling is divided into parts, and each part is edited by itself and searched for in the dictionary. The division can be made at several places which are all tested. By editing the parts by themselves it is possible to allow two editing operations per compounded word, with only a small addition in time compared with only one editing operation. Since the word parts are limited in length it is even possible have two editing operations in one part, but this increases the time with a factor of about ten without increasing the correction very much.

The default in Stava is to allow one editing operation per non-compounded word and two operations in compounded words (but only one in each part). With an option in is possible to have two editing operations per word or part of word, but this might be to slow for most purposes.

2.5 Performance

The performance of the final version was tested on the same set of 729 misspelled words as before but this time inflections were allowed too, so only 632 of the words were considered misspelled and were corrected. The test was done both allowing and disallowing generation of compounded words and allowing and disallowing two errors per word part. The time was measured on a Sun SparcStation 5 running SunOS 4.1.3. The average length of the words was 8.6 characters. For only 571 of the 632 misspelled words did the correct word exist in the dictionary. Using this we can see that if we allow generation of compounded words, about 77% of the misspellings were correctly corrected when allowing compounded words. In 84% of the cases the right word appeared as on of the first three words. If we do not allow compounded words, about 62% were correctly corrected. Allowing two edit operations did hardly affect the correction rate, but increased the time by a factor of about 40. Two editing operations can still be useful in some cases, for example to find the correct spelling of a single word. See table 3.

Table 3. Performance based on 632 misspelling words. The columns 1, 2 and 3 tell whether the correct word was the first, second or third suggestion. Not found means that Stava didn't find any suggestion for that word.

method	time (s)	not found	1	2	3
no compund words	7.2	188 (30%)	352 (56%)	20 (3%)	7 (1%)
2 edit oper.	285	146 (23%)	362 (57%)	20 (3%)	7 (1%)
compund words	20.7	16 (3%)	440 (67%)	28 (4%)	10 (2%)
compund + 2 edit	521	13 (2%)	444 (70%)	24 (4%)	10 (2%)

3 Conclusions

The spelling error correction method described in this report generates spelling correction suggestions by transforming the misspelled word with the editing operations insertion, deletion, transposition and replacement. These operations match typographic errors as well as some phonetic errors common in Swedish. Each possible transformation is then searched for in the dictionary. If a transformation is found in the dictionary it is accepted as a correction suggestion. Any number of editing operations may be performed on a misspelling but it is not practical to use more than one per word or maybe two for longer words (if you have a fast computer).

With an option, compound words can also be generated. This is done by dividing the misspelled word into all possible parts. These parts are then transformed and checked against the dictionary. In fact three different dictionaries are used, one for words that can only appear by themselves, one for words that may appear by themselves or as the last part of a compound word and a list of words that only may appear as part of compound word and not as the last part.

I tested three different methods to rank spelling corrections, modified edit distance, word frequencies and word bigram frequencies. The edit distance method was modified so that editing operations had different weight depending both on the kind of operation and on the letters near the edit operation. For insertion the weight depends on the letter following the insertion and on the letter inserted. For deletion the weight depends one the letter after the one deleted and on the letter deleted. Transpositions depends on the two transposed letters. The weight for replacements depends on the letter replaced and the letter that replaces it.

Word bigrams did hardly affect the ranking at all. The reason for the bad performance was that most of the word pairs in my test set did not appear in the list of 200 000 bigrams. Bigrams might be more useful if the list of bigrams is longer or for other types of text. Modified edit distance and word frequencies gave a good ranking, both by them self and used together. When used together, the correct word was ranked highest for 77% of the misspellings in my test set (counting only words where the correct word appeared in the dictionary, about 7% of the words did not appear in the dictionary). In 5% of the cases, the correct word was ranked as the second highest and 2% as the third highest word.

I believe this is near the optimal performance for a general purpose spelling corrector based only on isolated words. Further improvements would probably need to use context information.

Bibliography

[Domeij et al. 1995] Domeij, Rickard, Hollman, Joachim and Kann, Viggo. Detection of spelling errors in Swedish not using a word list en clair, *Journal of Quantitative Linguistics*, Vol 1(3), 1995

[Kukich 1992] Kukich, Karen. Techniques for Automatically Correcting Words in Text, *ACM Computing Surveys*, Vol 24(4) Dec. 1992, pp. 377-439

[Pollock & Zamora 1984] Pollock, Joseph J., and Zamora, Antonio. Automatic Spelling Correction in Scientific and Scholary Text, *Communications of the ACM*, Vol 27(4) April 1984, pp. 358-368

Appendix A Source Code

A.1 rattstava.h

```
/* Rättstavningsprogram. Version 2.1  1995-11-29
   Copyright (C) 1990-1996
   Joachim Hollman och Viggo Kann
   joachim@nada.kth.se viggo@nada.kth.se
*/
/* rattstava.h - gränssnitt till rattstava.c */
extern void GenereraAlternativaOrd(char *ord);
extern void OppnaRattstavaFil(void);
extern int Finns2Petig(char *ord, int fyrKoll);
/* LagraFyrgrafer ser till att ett ords alla fyrgrafer är tillåtna */
INLINE extern void LagraFyrgrafer(char *ord);
/* FyrKollaHela kollar om ett ords alla fyrgrafer är tillåtna */
INLINE extern int FyrKollaHela(char *ord);

extern void SkrivForslag(char *ordin);
```

A.2 rattstava.c

```
/* Rättstavningsprogram. Version 2.1  1995-11-29
   Copyright (C) 1990-1996
   Joachim Hollman och Viggo Kann
   joachim@nada.kth.se viggo@nada.kth.se
*/

/* rattstava.c - modul för rättstavningsfunktioner */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SVENSKA
#ifdef SVENSKA
#include "stava.h"
#endif
#ifdef ENGELSKA
#define ISO8BITAR
#include "spell.h"
#endif
#include "rattstava.h"
#include "suffix.h"

typedef unsigned char uchar;

extern int xAndelser, xSammansatta, xContext, xMaxOneError;
extern INLINE void SkrivOrd(unsigned char *s);

#define MAXSUGGESTIONS  20
#define CHECK_EL    1
#define CHECK_FL    2
#define CHECK_IL    4
```

```
#define DELPVAL 10
#define SWAPVAL 10
#define REPPVAL 10
#define INSPVAL 7
#define FIRSTP  10

#define DELP(a, b)      delap[(uchar)(a)][(uchar)(b)]
#define INSP(a, b)      insap[(uchar)(a)][(uchar)(b)]
#define SWAP(a, b)      swapp[(uchar)(a)][(uchar)(b)]
#define REPP(a, b)      replp[(uchar)(a)][(uchar)(b)]

static signed char delap[128][128];
static signed char insap[128][128];
static signed char swapp[128][128];
static signed char replp[128][128];

static unsigned char fyrtabell[FGRAMSIZE];
static FILE *fyrf;
static char **fyrOrd;
static int fyrAntalOrd = 0, fyrMaxAntalOrd = 0;
static char **delOrd[2*MAXORDDELAR];
static int delAntalOrd[2*MAXORDDELAR], delMaxAntalOrd[2*MAXORDDELAR];
static int addedWord;

static INLINE void *xmalloc(size_t s)
{ void  *p = malloc(s);
  if (!p) {
    fprintf(stderr, "Out of memory\n");
    exit(1);
  }
  return p;
}

static INLINE void *xrealloc(void *p, size_t s)
{
  p = realloc(p, s);
  if (!p) {
    fprintf(stderr, "Out of memory\n");
    exit(1);
  }
  return p;
}

/* Calculate points for word frequencies */
INLINE static int WordFreq(char *word)
{ int       len, i;
  char  buf[LANGD+1];
  static int p[] = { 0, 2, 4, 6, 8, 10, 12, 14, };

  strcpy(buf, word);
  len = strlen(buf);

  buf[len+1] = '\0';
  for (i=0; i<8; i++) {
    buf[len] = 'A' + i;
    if (InXL(buf))
      return p[i];
  }
  return 25;
}
```

```
/* FyrKollaHela kollar om ett ords alla fyrgrafer är tillåtna */
int FyrKollaHela(char *ord)
{ static char buf[LANGD+4];
  char *s;
  long l;
  int plats;
  sprintf(buf, "-%s-", ord);
  for (s = buf; *s; s++)
    if (*s >= 'a' && *s <= '}') *s -= 'a'; else
    if (*s >= 'A' && *s <= ']') *s -= 'A'; else
      *s = 29;
  for (plats = 3; buf[plats]; plats++) {
    l = ((buf[plats - 3] * 30 + buf[plats - 2]) * 30 + buf[plats - 1]) * 30 +
        buf[plats];
    if (!(fyrtabell[l >> 3] & (1 << ((int) l & 7)))) return(plats);
  }
  return(0);
}

/* LagraFyrgrafer ser till att ett ords alla fyrgrafer är tillåtna */
void LagraFyrgrafer(char *ord)
{ static char buf[LANGD+4];
  char *s;
  long l;
  int plats;
  sprintf(buf, "-%s-", ord);
  for (s = buf; *s; s++)
    if (*s >= 'a' && *s <= '}') *s -= 'a'; else
    if (*s >= 'A' && *s <= ']') *s -= 'A'; else
      *s = 29;
  for (plats = 3; buf[plats]; plats++) {
    l = ((buf[plats - 3] * 30 + buf[plats - 2]) * 30 + buf[plats - 1]) * 30 +
        buf[plats];
    fyrtabell[l >> 3] |= (1 << ((int) l & 7));
  }
}

/* FyrKoll kollar om ett ords fyrgrafer är tillåtna, plats anger index i
ord för en ändring. Om plats är negativt kollas hela ordet.
extra är antalet extra positioner som är ändrade */
static INLINE int FyrKoll(char *ord, int plats, int extra)
{ char buf[LANGD+4];
  char *s;
  long l;

  s = buf;
  *s++ = 29;
  for ( ; *ord; s++, ord++)
    if (*ord >= 'a' && *ord <= '}') *s = *ord - 'a'; else
    if (*ord >= 'A' && *ord <= ']') *s = *ord - 'A'; else
      *s = 29;
  *s++ = 29;
  *s = '/';
  if (plats >= 2) {
    plats++;
    buf[plats + 4 + extra] = '/';
  } else {
    if (plats >= 0) buf[plats + 5 + extra] = '/';
    plats = 3;
  }
  while (buf[plats] != '/') {
    l = ((buf[plats - 3] * 30 + buf[plats - 2]) * 30 + buf[plats - 1]) * 30 +
        buf[plats];
#ifdef DEBUG
printf("%c%c%c%c - %d\n", buf[plats - 3]+'A', buf[plats - 2]+'A',
       buf[plats - 1]+'A', buf[plats]+'A',
       (fyrtabell[l >> 3] & (1 << ((int) l & 7))) != 0);
#endif
    if (!(fyrtabell[l >> 3] & (1 << ((int) l & 7)))) return(0);
    plats++;
  }
  return(1);
}

static void FyrAdderaOrd(char *ord, int point)
{ int     i;

  if (strlen(ord) <= 1) return; /* Strunta i enbokstavsord */
  if (fyrMaxAntalOrd == 0) {
    fyrMaxAntalOrd = 20;
    fyrOrd = (char **) malloc(sizeof(char *) * fyrMaxAntalOrd);
    i = 0;
  }

  for (i = 0; i < fyrAntalOrd; i++) {
    if (!strcmp(ord, fyrOrd[i]+1))
      if (fyrOrd[i][0] > point)
        goto overwrite;
      else
        return;
      if (!strcmp(ord + 1, fyrOrd[i] + 2)) {
        if (abs(*ord - *(fyrOrd[i])+1) == 'a' - 'A')
          if (fyrOrd[i][0] > point)
            goto overwrite;
          else
            return;
      }
  }
  if (fyrAntalOrd >= fyrMaxAntalOrd) {
    fyrMaxAntalOrd += 20;
    fyrOrd = (char **) realloc(fyrOrd, sizeof(char *) * fyrMaxAntalOrd);
  }
  fyrAntalOrd++;
  addedWord = 1;

overwrite:
  fyrOrd[i] = (char *) malloc(strlen(ord)+2);
  fyrOrd[i][0] = point;
  strcpy(fyrOrd[i]+1, ord);
}

static void FyrSudda(void)
{ int i;
  for (i = 0; i < fyrAntalOrd; i++) free(fyrOrd[i]);
  fyrAntalOrd = 0;
}

static void Concat(char *to, char *word, int point, int part, int lastpart)
{ int       len, i;

  if (part == lastpart) {
    strcat(to, word);
    FyrAdderaOrd(to, point+WordFreq(to));
  } else {
    for (i=0; i<delAntalOrd[part]; i++) {
      len =strlen(to);
      strcat(to, delOrd[part][i]+1);
      Concat(to, word, point+delOrd[part][i][0], part+1, lastpart);
      to[len] = 0;
    }
  }
}

/* Sätt ihop de olika orddelarna i alla kombinationer och stopa in i
   listan över ordförslag */
static void ConcatParts(char *word, int point, int part)
{ char  buf[LANGD];

  addedWord = 1;
  buf[0] = 0;
  Concat(buf, word, point, 0, part);
}

/* Lägg till en ny orddel */
static void AddPart(char *ord, int point, int part)
{ int   i;

  if (strlen(ord) <= 1) return; /* Strunta i enbokstavsord */
  if (delMaxAntalOrd[part] == 0) {
    delMaxAntalOrd[part] = 20;
    delOrd[part] = (char **) xmalloc(sizeof(char *) * delMaxAntalOrd[part]);
    i = 0;
  }

  for (i = 0; i < delAntalOrd[part]; i++) {
    if (!strcmp(ord, delOrd[part][i]+1))
      if (delOrd[part][i][0] > point)
        goto overwrite;
      else
        return;
    if (!strcmp(ord + 1, delOrd[part][i] + 2)) {
      if (abs(*ord - *(delOrd[part][i])+1) == 'a' - 'A')
        if (delOrd[part][i][0] > point)
          goto overwrite;
        else
          return;
    }
  }
  if (delAntalOrd[part] >= delMaxAntalOrd[part]) {
    delMaxAntalOrd[part] += 20;
    delOrd[part] = (char **) xrealloc(delOrd[part], sizeof(char *) *
                                  delMaxAntalOrd[part]);
  }
  delAntalOrd[part]++;
  addedWord = 1;

  overwrite:
  delOrd[part][i] = (char *) malloc(strlen(ord)+2);
  delOrd[part][i][0] = point;
  strcpy(delOrd[part][i]+1, ord);
}

/* rensa listan med orddelar */
static void PartClear(int part)
{ int i;
  for (i = 0; i < delAntalOrd[part]; i++) free(delOrd[part][i]);
  delAntalOrd[part] = 0;
}

/* Kolla om word finns i ordlistan. check styr vilka av ordlistorna EL, IL
   och FL som ska användas */
INLINE void Check(char *word, int point, int part, int check)
{
  /*fprintf(stderr, "(check %s %d %d %d)", word, point, part, check);*/
  if (check & CHECK_IL) {
    if (InIL(word)) {
FyrAdderaOrd(word, point+WordFreq(word));
      return;
    }
  }
  if ((check & CHECK_EL)) {
    if (InEL(word) || (xAndelser && CheckSuffix(word))) {
      ConcatParts(word, point, part);
      return;
    }
  }
  if (check & CHECK_FL) {
    if (InFL(word, strlen(word))) {
      AddPart(word, point, part);
      return;
    }
  }
  return;
}

void Generera1(char *word, int from, int errors, int point, int part, int check)
{ int      i, j, len, left, right, first, last, onlyswap;
  long  l;
  char  tmp1;
  char  fyrbuf[LANGD+2], word2[LANGD];
  char  *s;

  len = strlen(word);
  if (len < 2)
    return;

  sprintf(fyrbuf, "-%s-", word);
  for (s = fyrbuf; *s; s++) {
    if (*s >='a' && *s <= 'z'+3)
      *s-= 'a';
    else if (*s >='A' && *s <= 'Z'+3)
      *s -= 'A';
    else
      *s = 29;
  }

  last = -1;
  first = -1;
  for (i=0; i<len-1; i++) {
    l = ((fyrbuf[i] * 30 + fyrbuf[i+1]) * 30 + fyrbuf[i+2]) * 30 +
      fyrbuf[i+3];
    if (!(fyrtabell[l>>3] & (1 << (l&7)))) {
      last = i;
      if (first == -1)
        first = i;
    }
  }
  onlyswap = 0;
```

```
if (errors == 1) {
  if (last == -1) {   /* ingen felaktig fyrgraf */
    left = from;
    right = len;
  } else if (last-first > 4) { /* minst 2 fel */
    return;
  } else if (last-first == 4) {
    onlyswap = 0;
    left = last-1;
    right = first+3;
  } else {    /* möjliga positioner för felet */
    left = last-1;
    right = first+3;
  }
} else { /* se till att alltid fixa första felaktiga fyrgrafen */
  left = from;
  if (first == -1 || first+3 > len)
    right = len;
  else
    right = first+3;
}
if (left < from)
  left = from;

/*fprintf(stderr, "[%s %d %d %d %d]", word, first, last, left, right);*/

if (last == -1)
  Check(word, point, part, check);

if (errors == 0)
  return;

/* swap two chars */
for (i=left-1; i<right; i++) {
  if (i<0)
    continue;
  tmp1 = word[i];
  word[i] = word[i+1];
  word[i+1] = tmp1;
  if (errors == 1) {
    if (FyrKoll(word, i, 1))
      Check(word, point+SWAP(word[i+1], word[i]), part, check);
  } else {
    Generera1(word, i+1, errors-1,
              point+SWAP(word[i+2], word[i]), part, check);
  }
  tmp1 = word[i];
  word[i] = word[i+1];
  word[i+1] = tmp1;
}

if  (onlyswap)
  return;

/* remove char */
tmp1 = word[left];
strcpy(word2, word);
for (i=left; i<right; i++) {
  strcpy(word2+i, word+i+1);
  if (errors == 1) {
    if (FyrKoll(word2, i, 0))
      Check(word2, point+DELP(word[i], word2[i])+((i==0)?FIRSTP:0),
              part, check);
    } else {
      Generera1(word2, i+1, errors-1,
                point+DELP(word[i], word2[i])+((i==0)?FIRSTP:0), part,
                check);
    }
    word2[i] = word[i];
  }

  /* replace char */
  for (i=left; i<right; i++) {
    tmp1 = word[i];
    for (j='a'; j<='z'+3; j++) {
      if (j != tmp1) {
        word[i] = j;
        if (errors == 1) {
          if (FyrKoll(word, i, 0))
            Check(word, point+REPP(tmp1, j)+((i==0)?FIRSTP:0),
                  part, check);
        } else {
          Generera1(word, i+1, errors-1,
                    point+REPP(tmp1, j)+((i==0)?FIRSTP:0), part,
                    check);
        }
      }
    }
    word[i] = tmp1;
  }

  /* insert char */
  strcpy(word2, word);
  strcpy(word2+left+1, word+left);

  for (i=left; i<right+1; i++) {
    for (j='a'; j<='z'+3; j++) {
      word2[i] = j;
      if (errors == 1) {
        if (FyrKoll(word2, i, 0))
          Check(word2, INSP(word2[i], word2[i+1])+((i==0)?FIRSTP:0),
                part, check);
      } else {
        Generera1(word2, i+1, errors-1,
                  INSP(word2[i], word2[i+1])+((i==0)?FIRSTP:0), part,
                  check);
      }
    }
    word2[i] = word2[i+1];
  }
  word2[len] = '\0';
}

/* Generera rättstavningsförslag för word. Högst errors fel får förekomma */
static int Generera(char *word, int errors, int point, int part, int check)
{ int      len, i;

  len = strlen(word);

  if (len > DELORDMAX)
    return -1;

  if (errors > 2)
    errors = 2;
  if (errors > 1 && (len <= 6 || xMaxOneError))
    errors = 1;

  for (i=0; i<=errors; i++) {
    addedWord = 0;
    /*fprintf(stderr, "{Gen %s %d %d}", word, i, errors);*/
    if (i == 0)
      Check(word, 0, part, check);
    else
      Generera1(word, 0, i, point, part, check);

    if (addedWord) {
      /*fprintf(stderr, "Found\n");*/
      return i;
    }
  }
  return -1;
}

/* Kolla om word är sammansatt som FL* EL. Om xTillatSIFogar
   så tillåt 's' i all fogar utom mellan 1:a och 2:a delen.
   Om xTillatSIAllaFogar så tillåt 's' i alla fogar. */
INLINE static void CompoundEdit(char *word, int offset, int part, int errors)
{ int       end, tmp, len, before, res;

  len = strlen(word);

  AddPart("s", 0, part);

  if (offset == 0) {
    if (Generera(word, errors, 0, part, CHECK_EL | CHECK_IL) != -1)
      return;
  } else {
    if (Generera(word, errors, 0, part, CHECK_EL) != -1)
      return;
  }

  if (!xGenereraSammansatta || part+1 >= MAXORDDELAR)
    return;

  for (end=DELORDMIN; end<=len-DELORDMIN; end++) {
    tmp = word[end];
    word[end] = 0;
    before = fyrAntalOrd;
    PartClear(part);
    res = Generera(word, errors, 0, part, CHECK_FL);

    if (res != -1) {
      errors -= res;
      word[end] = tmp;
      CompoundEdit(word+end, offset+end, part+1, errors);

      if (((xTillatSIFogar && offset) || xTillatSIAllaFogar) &&
          word[end] == 's' &&
          bindebokstav[(unsigned char)word[end-1]] == 's') {
        PartClear(part+1);
        AddPart("s", 0, part+1);
        CompoundEdit(word+end+1, offset+end+1, part+2, errors);
      }

#if 0
```

```
      /* Hantera tex toppolitiker som topp|politiker */
      if (word[end-1] == word[end-2]) {
        CompoundEdit(word+end-1, offset+end-1, part+1, errors);
      }
#endif
      errors += res;
    }
    word[end] = tmp;
  }
}

void GenereraAlternativaOrd(char *wordin)
{ int       len;

  len = strlen(wordin);
  if (len < 2)
    return;

  if (len < 7)
    CompoundEdit(wordin, 0, 0, 1);
  else
    CompoundEdit(wordin, 0, 0, 2);
}

void OppnaRattstavaFil(void)
{ int       i, j;
  char  slask;
  char  dub[] = "bdfgjlmnprstv";
  char  vov[] = "aeiouy{|}";

  if (!(fyrf = fopen(XFYRGRAFER, "r"))) {
#ifdef ISO8BITAR
    fprintf(stderr,"Kan inte öppna filen %s\n", XFYRGRAFER);
#else
    fprintf(stderr,"Kan inte |ppna filen %s\n", XFYRGRAFER);
#endif
    exit(1);
  }
  if (fread(fyrtabell, sizeof(unsigned char), FGRAMSIZE, fyrf) !=
      FGRAMSIZE || fread(&slask, sizeof(char), 1, fyrf) == 1) {
#ifdef ISO8BITAR
    fprintf(stderr, "%s har fel format för att vara en fyrgraffil\n",
            XFYRGRAFER);
#else
    fprintf(stderr, "%s har fel format f|r att vara en fyrgraffil\n",
            XFYRGRAFER);
#endif
    fclose(fyrf);
    exit(1);
  }
  fclose(fyrf);

  for (i=0; i<128; i++) {
    for (j=0; j<128; j++) {
      delap[i][j] = DELPVAL;
      insap[i][j] = INSPVAL;
      swapp[i][j] = SWAPVAL;
      replp[i][j] = REPPVAL;
    }
  }

  for (i=0; i<sizeof(dub); i++)
```

```
    insap[(int)dub[i]][(int)dub[i]] -= 4;
  insap[(int)'c'][(int)'k']--;

  for (i=0; i<sizeof(dub); i++)
    delap[(int)dub[i]][(int)dub[i]] -= 4;
  delap[(int)'c'][(int)'k']--;

  for (i=0; i<sizeof(vov); i++)
    for (j=0; j<sizeof(vov); j++)
      replp[(int)vov[i]][(int)vov[i]]--;

  replp[(int)'e'][(int)'a'] = REPPVAL-4;
  replp[(int)'a'][(int)'e'] = REPPVAL-4;
  replp[(int)'e'][(int)'i'] = REPPVAL-4;
  replp[(int)'i'][(int)'e'] = REPPVAL-4;
  replp[(int)'n'][(int)'m'] = REPPVAL-4;
  replp[(int)'m'][(int)'n'] = REPPVAL-4;
  replp[(int)'{'][(int)'e'] = REPPVAL-4;
  replp[(int)'e'][(int)'{'] = REPPVAL-4;
  replp[(int)'}'][(int)'o'] = REPPVAL-4;
  replp[(int)'o'][(int)'}'] = REPPVAL-4;
  replp[(int)'s'][(int)'c'] = REPPVAL-4;
  replp[(int)'g'][(int)'j'] = REPPVAL-4;
  replp[(int)'j'][(int)'g'] = REPPVAL-4;
  replp[(int)'v'][(int)'w'] = REPPVAL-2;
  replp[(int)'w'][(int)'v'] = REPPVAL-2;
  replp[(int)'c'][(int)'k'] = REPPVAL-2;
}

void SkrivForslag(char *ordin)
{ char ord2[LANGD + 3], Ord[LANGD + 3];
  int i, swapped;
  char *tmp;

  FyrSudda();
  GenereraAlternativaOrd(ordin);

  if (fyrAntalOrd == 0) {
    VersalerGemena(ordin, ord2, Ord);
    if (*ord2) GenereraAlternativaOrd(ord2);
  }

  if (fyrAntalOrd > 0) {
    /* bublesort the list */
    do {
      swapped = 0;
      for (i=0; i<fyrAntalOrd-1; i++) {
        if (fyrOrd[i][0] > fyrOrd[i+1][0]) {
          tmp = fyrOrd[i];
          fyrOrd[i] = fyrOrd[i+1];
          fyrOrd[i+1] = tmp;
          swapped = 1;
        }
      }
    } while (swapped);

    /* display the list */
    for (i = 0; i < fyrAntalOrd; i++) {
      /*printf("%d ", fyrOrd[i][0]);*/
      SkrivOrd(fyrOrd[i]+1);
      putchar(' ');
      if (i > MAXSUGGESTIONS)
```

```
        break;
    }
    putchar('\n');
  } else printf("? hittar inga liknande ord\n");
}
```