Int. J. Appl. Math. Comput. Sci., 2005, Vol. 15, No. 2, 275–285

CORRECTING SPELLING ERRORS BY MODELLING THEIR CAUSES

SEBASTIAN DEOROWICZ, MARCIN G. CIURA

Silesian University of Technology, Institute of Computer Science
ul. Akademicka 16, 44–100 Gliwice, Poland
e-mail: {Sebastian.Deorowicz, Marcin.Ciura}@polsl.pl

This paper accounts for a new technique of correcting isolated words in typed texts. A language-dependent set of string substitutions reflects the surface form of errors that result from vocabulary incompetence, misspellings, or mistypings. Candidate corrections are formed by applying the substitutions to text words absent from the computer lexicon. A minimal acyclic deterministic finite automaton storing the lexicon allows quick rejection of nonsense corrections, while costs associated with the substitutions serve to rank the remaining ones. A comparison of the correction lists generated by several spellcheckers for two corpora of English spelling errors shows that our technique suggests the right words more accurately than the others.

Keywords: spelling correction, finite state automata, spelling errors

1. Introduction

Introducing texts into computer memory always entails a possibility of making errors. The state of the art in automated finding and correcting these errors is reviewed in (Kukich, 1992; Mitton, 1996).

Spelling correction consists of *detecting* and *correcting* errors. These subproblems can be addressed in two ways. In *isolated-word error detection and correction*, each word is checked separately, abstracting from its context. To detect an error, it suffices to search for a word in the set of all correct words, i.e., a *lexicon*. This approach fails when a spelling error produces another correct word, as when, e.g., *then[1] is written instead of than.[2] Besides, candidates of correct forms, i.e., *suggestions* may be contextually inappropriate. Notwithstanding this, many contemporary spellcheckers use this approach, since the alternative, *context-dependent error detection and correction*, requires grammatical analysis and is thus more complex and language dependent. Even in context-dependent methods, though, the list of suggestions is obtained from an isolated-word method before making a choice depending on the context. In this paper, we focus on isolated-word methods only.

Error correction means providing correct words that could have been misspelled as a given non-word. In *interactive spelling correction*, the accurate word is chosen by the user and to make this task easier the corrections should be ranked in order of a decreasing probability. In *automatic spelling correction*, the correct word is chosen without user interaction, therefore only one correct word should be generated.

In this paper, we use a minimal acyclic deterministic finite automaton (ADFA) as an effective representation of the lexicon, and employ a method for its making and searching introduced in our paper (Ciura and Deorowicz, 2001). We also present methods of correcting non-words, traversing the word space, and ranking suggestions. We show how the proposed rules can be used to cover many types of common spelling errors. Our approach is evaluated on two corpora of typical spelling errors against other popular methods.

2. Spelling Errors

2.1. Detecting Non-Word Spelling Errors

Detecting whether or not a word is correct seems simple—why not to look up the word in a set of all words? Unfortunately, there are some problems with this simple strategy. Firstly, a lexicon containing all correct words could be extremely large, which entails space and time inefficiency. Secondly, in some languages it is practically impossible to list all correct words, because they are highly productive. Thirdly, making a spelling error can sometimes result in a real word, which belongs to the lexicon—such an error is called a *real-word error*. It is impossible to decide that this word is wrong without some contextual information.

[1] Through the paper, the incorrect examples are denoted by an asterisk in front of the word to distinguish them from the correct ones.

[2] Our approach can be applied to many languages but for clear presentation, we provide examples in English only.

Fourthly, the bigger the lexicon, the more esoteric words it contains, making real-word errors more likely.

The appropriate lexicon size is dependent on the language. In only slightly inflective languages as English, lexicons of size 50 000–200 000 words were recommended (Damerau, 1990; Damerau and Mays, 1989; Peterson, 1986). For highly inflective languages, the lexicon has to be much larger, and typically contains millions of words.

The classic data structure offering a fast search is a hash table (Knuth, 1973). Its disadvantage is the need to properly choose the hash function and the size of the hash table to mitigate the problem of collisions. Minimal perfect hashing (Czech *et al.*, 1997) eliminates collisions but requires storing the hash table of a size equal to the number of words in the lexicon and the whole lexicon (possibly compressed by some method).

Another popular data structure used for lexicon storage is a trie (Knuth, 1973). It is a character-oriented tree, in which every path from a root to a leaf corresponds to a key, and branching is based on successive characters. A trie offers fast lookup and some compression of the lexicon. Its size, however, is typically comparable to the lexicon size due to the need of storing pointers to the nodes. There are many works on reducing trie sizes; some of the alternative versions of tries are the C-trie (Maly, 1976), PATRICIA (Morrison, 1968), and Bonsai (Darragh *et al.*, 1993).

An acyclic deterministic finite automaton (ADFA) can be considered a generalisation of the trie. If all equivalent subtries of a full trie are merged, we obtain a minimal ADFA. This data structure is discussed later in detail as we use it in our solution.

Yet another approach is to store directly root forms of each lexicon word and rules for stripping affixes existing in an actual language. If a text word is absent from the lexicon, the existing affixes are stripped one by one and the obtained forms are checked. For example, the lexicon contains the word check, and judging the correctness of the text word uncheckable might go like this: uncheckable is absent from the lexicon; stripping un- results in checkable which is also absent from the lexicon; stripping -able results in check which is present in the lexicon, so the text word uncheckable is judged as correct. Such a method requires low space but, unfortunately, may lead to false acceptance of some words, as not all affixes can be appended to all root forms. For example, a text word *unmark is wrong, but stripping un- results in the word mark so the text word *unmark is judged as correct. This method will not work either for some languages like Finnish and Turkish, since they need much more sophisticated processing.

A similar approach is to store only a root form for each word together with rules of its inflection. Therefore, we know how to produce all correct forms of the word. This solution is used in such spellcheckers as Ispell (Kuenning, 2003) and Aspell (Atkinson, 2003). Some examples from Ispell are: boolean/S, which means that the plural form, booleans, is also correct; frizzle/DGS, allowing: frizzle, frizzled, frizzles, frizzling.

2.2. Types of Spelling Errors

Typing texts consists of three main stages (cf. Fig. 1). An error may occur at each of them.

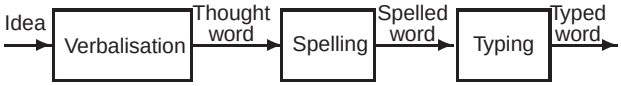


Fig. 1. From an idea to a typed word.

In the first stage, verbalisation, an idea crystallises into a thought word. Usually it is simple, but sometimes may not be such, e.g., one may want to write a negative form of a word and is unsure which of the negative prefixes should be used in that particular case. As there are a few negative prefixes, e.g., im- (imperfect), in- (incorrect), un- (unnatural), one may create a negative form choosing the improper one like perfect → *inperfect. An error can also appear when one tries to create a word from a different part of speech, e.g., typical suffixes for adjectives are -ical (cynical), -ly (mannerly), -ally (magically), and when one does not know the correct adjective derived from the adverb tragic, may write *tragicly instead of tragically. Spelling errors of this kind, called *vocabulary incompetence*, are typical for children and non-native language users.

In the second stage, the thought word is converted into its spelt form. In this stage, an error may appear when one is unsure of its spelling or pronunciation. When the pronunciation is known, one may use a different grapheme (letters representing a phoneme) for the phoneme existing in the word. This is reflected in such errors like *occurrance instead of occurrence, *fourty instead of forty, *grammer instead of grammar. When the pronunciation is known only approximately, one may try to write a different phoneme, e.g., *egsistence instead of existence. *Misspellings* are frequent in languages in which phonemes are rendered by several graphemes. These error types are usual for children and non-native language users.

The third stage is typing. In this stage, one knows the word, knows its proper spelling, but makes a spelling error, a *mistyping*, while pressing the keys. In our research, we follow the works (Damerau, 1964; Pollock and Zamora, 1984) and define four main kinds of mistypings:

insertion of a letter, e.g., *speklling instead of spelling; deletion of a letter, e.g., *spellng instead of spelling; substitution of one letter by another, e.g., *spellong instead of spelling; transposition of two adjacent letters, e.g., *seplling instead of spelling. All these errors appear when one types a word, so the keyboard layout influences the introduced mistypings. It is much more likely to press the wrong key which is close to the one in question than other, e.g., many people use QWERTY keyboards, so pressing the letter e instead of o is unusual, but for those using Dvorak keyboards this is typical. It is possible to define other mistyping categories, e.g., doubling of a letter, but such types can be usually modelled using the mentioned ones, so we refrain from introducing other types to make the model concise.

The above-described error classes are typical when a person types a text. There are, however, various ways to introduce a text into computer, like scanning and processing by an optical character recognition (OCR) program. In such a situation, a common error is to misinterpret letters that look similar, e.g., e and c, ı /ai/ and l /el/. Sometimes an OCR application can also misinterpret two adjacent letters as one and vice versa, e.g., ni and m.

2.3. Techniques for Isolated-Word Spelling Error Correction

Isolated-word spelling error correction means providing one or more suggestions for a non-word. If more than one suggestion is provided, they should be ranked from the most to the least probable candidate, e.g., the suggestion spelling is a more probable candidate for a non-word *speling than the suggestion spilling, but we cannot be sure.

The discussion from the previous section gives us a key to construct suggestions. We simply try to revert the effect of human-made errors. It is impossible to be sure which kind of error (or errors) the person made, so usually a list of some number of possibilities is produced. Each suggestion has a score, which tells us how much it is similar to the non-word. The list is then sorted according to the scores. In automatic spelling correction, all words but the top-ranked one are discarded, but in interactive spelling correction, several best candidates may be presented to the user.

The simplest method is based on the assumption that the person usually makes few errors, if any. Therefore, for each lexicon word, we determine the minimal number of the basic editing operations (i.e., character *insertions*, *deletions*, and *substitutions*) necessary to convert a lexicon-word into the non-word. The lower the number, the higher the probability that the user has made such errors. This approach is called the *minimum edit distance* technique and was introduced in (Damerau, 1964) (in fact, Damerau uses also *transpositions* in his work) and redefined later in (Wagner, 1974). A similar technique was used at about the same time assuming other three basic editing operations: *insertions*, *deletions*, and *transpositions* (Levenshtein, 1966). This technique covers only one of the three main categories of spelling errors, i.e., mistypings, however, it is possible to extend its usage to the others. A straightforward implementation needs to compare all lexicon words with the non-word, which is costly. A faster solution (Baeza-Yates and Navarro, 1998) allows, however, reducing the number of compared strings by about an order of magnitude if the number of errors in the non-word is low.

In the *similarity key technique*, a key is assigned to each lexicon word and only lexicon keys are compared with the key computed for the non-word. The words for which the keys are most similar are selected as suggestions. Such an approach is speed effective as only the words with similar keys have to be processed. The very first such a technique was based on the SOUNDEX system (Odell and Russell, 1918), which was developed to correct names written phonetically. The SOUNDEX code is rather short and consists of a letter and three digits, so it is imprecise and cannot distinguish between many words. Therefore, a similar, but more precise, technique, SPEEDCOP, was developed (Pollock and Zamora, 1984) for the spelling checking problem. The system offers two keys, skeleton and omission, and is often more suitable. Both techniques can be used, however, only for misspelling and mistyping correction.

Rule-based techniques use the knowledge of the most common error types to transform the non-word into lexicon words. First such a knowledge-based system (Yannakoudakis and Fawthrop, 1983a; 1983b) was based on rules determined from experiments with over 1000 spelling errors. The rules model various spelling errors so the application of them to the non-word results in a number of correct words. There are two main problems with these techniques. Firstly, the rules have to be obtained from experiments with real spelling errors and must be built into an algorithm or obtained by the algorithm from another source. Secondly, it is difficult to organise the lexicon in such a way that the words obtained from the rules are rapidly verified if they are real words, so producing the list of suggestions may be slow. An advantage of these methods is their ability to cover all three spelling error categories.

Probabilistic techniques rely on the probability of introducing some types of errors. The analysis of the probabilities of substituting one letter by other, letter insertion, letter deletion, or making a transposition of letters is the basis of the approach developed at Bell Labs (Church and Gale, 1991). The probabilities were obtained analysing a corpus containing millions of words. The suggestions are

sorted according to the probabilities of making changes in the word necessary to obtain the non-word. The probabilities of making errors can be used also for spell checking in several ways, i.e., for improving the similarity key or rule-based techniques.

The *phonetic similarity* technique is suitable especially to deal with misspellings. The method is based on the assumption that many types of errors are caused by the person who knows the pronunciation of a word, but does not know its correct spelling. Such people often render phonemes using improper graphemes. There are several possibilities to address the problem of phonetic similarity. The SOUNDEX code is one of them. Some other solutions are PHONIX (Gadd, 1990), Metaphone (Philips, 1990), and Double Metaphone (Philips, 2000), used in the Aspell spellchecker (Atkinson, 2003). The most complex techniques employ grapheme to phoneme conversions (Berkel and Smedt, 1988).

Other popular solutions in spelling correction which offer good results are based on neural networks (Hodge and Austin, 2003; Kukich, 1988) and the noisy channel (Brill and Moore, 2000; Toutanova and Moore, 2002).

2.4. Why Perfect Spelling Correction Is Impossible

The spelling correction problem is difficult to be solved completely, because people make various errors. Without contextual information we cannot be sure which word was intended to be typed. Often, several candidates are possible. Let us consider the non-word *stat. In the document concerning astronomical problems, it is often a mistyping of star, but in other documents it is rather a mistyping of stay or state. Abstracting from the document contents, we are sometimes unable to choose the accurate suggestion. There is also the possibility that the typist does not know a word and guesses its spelling thus producing a sequence of letters which is far from the correct word, e.g., *sicolagest instead of psychologist[3]. It is hard to propose an accurate suggestion in such a case.

Even if we know the context of the text non-word, there are situations in which we are unable to decide how to correct it. The simplest example are enumerations—several words can be enumerated and we usually have little contextual information.

3. Modelling Spelling Errors with Substitution Rules

3.1. Substitution Rules

Our approach to spelling correction is based on rules modelling typical spelling errors. Each rule is a triple $\langle input\_string, output\_string, cost\rangle$. The rules are applied to the non-word in such a way that the $input\_string$, located anywhere within the non-word, is substituted for the $output\_string$. The cost of making a substitution is related with the new word. If the obtained word is valid, it is added to the suggestion list. Different rules can be applied to the non-word, and all possibilities are examined. When two or more rules are applied to the non-word, the costs sum up.

Since the number of rules can be large, we specify them in an equivalent but more compact form by gathering the rules with the same $input\_string$: $input\_string > c_1o_1, c_2o_2, \ldots, c_no_n$, where c_k means $cost$ related to the kth $output\_string$, o_k. There are some special symbols used in the rule notation. The first one, . (dot), appearing as the $output\_string$, means an empty string, and appearing as the $input\_string$, means any position in the word except for initial and final. The symbols $ and ^ can appear only in the $input\_string$, and mean respectively the end of the word and the beginning of the word. When one of these two symbols appears as the $input\_string$, the rule defines the insertion of the $output\_string$ at the end or beginning of the word. The sample rules mean: $\langle$ai, ey, 5$\rangle$—substitute ai by ey while the cost of making the substitution is 5; $\langle$^in, un, 5$\rangle$—substitute in appearing at the beginning of the word by un; $\langle$ize$, ate, 5$\rangle$—substitute ize appearing at the end of the word by ate; $\langle$e, ., 3$\rangle$—remove e; $\langle$., e, 3$\rangle$—insert e in any position in the word except for initial and final; $\langle$$, e, 2$\rangle$—insert e in the final position of the word.

We define also the special symbol $*$, which means any letter. It is introduced to define rules handling traditional mistypings: $\langle$*, *, 7$\rangle$—substitute any character by any other, $\langle$*, ., 5$\rangle$—remove any letter, $\langle$., *, 4$\rangle$—insert any letter in any position in the word except for the initial and the final one. The last special symbol, _, means a space character. It allows defining the cost of the insertion of a space into the examined non-word, e.g., $\langle$., _, 9$\rangle$. We also allow using the digraph $*\langle symbol\rangle$ to distinguish different "any letter" in the rule. This allows defining the rules covering the transpositions of adjacent letters, e.g., $\langle$*a*b, *b*a, 2$\rangle$ defines changing the ordering of the two consecutive letters and $\langle$*a*b*c, *c*a*b, 4$\rangle$ defines a more complex transposition between three letters.

Note also that since we allow substituting substrings instead of only single characters, our approach is different from the calculation of the edit distance. It can be considered as an extension of the known algorithms (Oflazer, 1996; Savary, 2001), which can handle only mistypings. The idea of substituting strings instead of single characters is not novel (Brill and Moore, 2000). The authors of the mentioned work show a way of automatic training of their spelling errors model with a set of spelling errors. The substitution rules are also recovered from the train-

[3] This example comes from one of our test data sets.

ing set. Our approach is different since we propose to use some insight into the origin of spelling errors. Therefore, we have some knowledge of the causes of the errors and may propose a set of rules which is probably to be smaller than the one obtained automatically. Such a strategy is less flexible, but likely may lead to faster spelling correction and similar quality of suggestions.

3.2. Mistypings

Mistypings are the simplest errors. In our approach, there are rules specifying the cost of a letter insertion, a letter deletion, a substitution of any letter with any other, and a transposition of two adjacent letters. We extend the basic rule of transposition from two to three letters, since we observed this helps in correction. In our implementation, the rules cannot overlap, so the change of the ordering of three adjacent letters cannot be simulated as a sequence of two overlapping 2-letter transpositions.

The probability of letter substitutions depends on the keyboard layout (Grudin, 1983), so the costs of making them should reflect the distance between the letters on the keyboard. Figure 2(a) presents rules for letter substitutions and letters transpositions. (We assume here the QWERTY layout.)

(a)	(b)	(c)
a>6q,6w,5s,6z	ay>5a,5ai,5ei,5ey,5ea	^un>5in,5im,5il,5ir
	a>5ay,5ai,5ei,5ey,5ea	^in>5un,5im,5il,5ir
b>5v,6g,6h,5n	ai>5ay,5a,5ei,5ey,5ea	^im>5un,5in,5il,5ir
	ei>5ay,5a,5ai,5ey,5ea	^il>5un,5in,5im,5ir
c>5x,5d,5f,5v	ey>5ay,5a,5ai,5ei,5ea	^ir>5un,5in,5im,5il
	ea>5ay,5a,5ai,5ei,5ey	
k>5j,6i,6o,5l,6m		ize$>5ate,5ify,5en
	t>4tt,4th	ise$>5ate,5ify,5en
*a*b>2*b*a	tt>4t,4th	ate$>5ize,5ise,5ify,5en
	th>4t,4tt	ify$>5ize,5ise,5ate,5en
*a*b*c>4*b*c*a		en$>5ize,5ise,5ate,5ify
*a*b*c>4*c*a*b	.>3e	
*a*b*c>4*c*b*a	e>3.	

Fig. 2. Sample substitution rules covering various types of spelling errors. Three types of errors are shown: (a) mistypings, (b) misspellings, (c) vocabulary incompetence.

3.3. Misspellings

Misspellings are a bit more complex to handle. This type of errors appears when a person knows the pronunciation of a word, but does not know its spelling or even knows the pronunciation only approximately. In such a case, one usually tries to write the word rendering the phonemes in some way, often in the way the phonemes can be legally rendered. Let us think of the word misspelling /mɪsspelɪŋ/. According to the typical representations of phonemes, it may be typed as *misspelyng.

For each phoneme, we distilled its grapheme representations to produce the rules covering misspellings. Each rule defines the cost of substituting different representations of a phoneme with its other representations. The more so, in English, some letters can be inaudible and are often omitted by the typist, e.g., the letter h is inaudible in the word hour. The word-ending e is also often missed as it is mute. A typical misspelling is also adding some letters, e.g., the word-ending e, when the person is unsure whether the letter exists in the word or not. To deal with such situations, we created rules inserting or removing the letters a, e, h, o. Yet another misspelling is rendering the pair of phonemes /ks/ or /gz/ with the letter x. Sample rules covering misspellings are presented in Fig. 2(b).

3.4. Vocabulary Incompetence Errors

Covering vocabulary incompetence errors is most complex when the typist does not know the word and guesses it. The typical errors committed in such a situation are the usage of wrong prefixes or suffixes of words. We model such errors providing rules for groups of often misused affixes. Some of the rules are for: negating prefixes {il-, in-, im-, ir-, un-}, {mal-, mis-}, {a-, an-}, {de-, dis-}; other prefixes: {macro-, mega-}, {multi-, poly-}, {col-, com-, con-}, {em-, en-, in-}; verb suffixes: {-ate, -en, -ify, -ise, -ize}; noun suffixes: {-er, -ian, -ist, -or}, {-dom, -ness, -ship}, {-ness, -ty}, {-alist, -ist}, {-ance, -ion, -ism, -ity, -ment, -ship}, {-ation, -tion}; nationality suffixes: {-an, -e, -ean, -er, -ese, -i, -ian}, adjective and adverb suffixes: {-al, -ar, -ic, -ical, -ed, -ive}, {-al, -ly, -ally}. Sample rules addressing vocabulary incompetence errors are shown in Fig. 2(c).

3.5. Other Spelling Error Types

The above-described errors are typical for a typed text. If the way of introducing the text into a computer is different, other spelling errors may appear. The rules covering errors introduced during the optical character recognition stage reflect such situations like misinterpreting the pair of letters ni as a single letter m, misinterpreting the letter e as c, and similar. In our implementation, we assume that the text is typed, so such OCR-specific rules are absent from the set of rules examined in the experiments described in Section 6.

3.6. Gathering It All Together

There are a number of substitution rules that may lead to a lot of suggestions. Since higher cost means smaller relevance between the suggestion and the original non-word we define a maximal cost of suggestion acceptance. In our method, all possible words which can be obtained using substitution rules from the non-word, which do not exceed the maximum cost are examined. We, however, refrain from overlapping the rules. If there are several ways of obtaining a suggestion from the non-word (using various rules), we choose the one with the lowest cost.

All suggestions are sorted according to decreasing costs. The ones with equal costs are sorted according to the length of the common prefix of the non-word and the suggestion. Such a criterion extends the observation (Mitton, 1987; Pollock and Zamora, 1983) that the first letter of the non-word and of the correct word are rarely different. If some words have the same costs and the same lengths of the common prefixes, they are sorted according to the length of the common suffixes with the non-word. The last criterion is lexicographical ordering.

Substitution rules modelling spelling errors have been, in general, hand-crafted. We, however, used the research described in (Church and Gale, 1991; Grudin, 1983) to develop rules for mistypings. The discussion on grapheme representations of phonemes described in (Jassem, 1983; Mateescu, 2003) has been used to prepare rules handling misspellings. Rules handling vocabulary incompetence errors result from the analysis of the description presented in (Merriam-Webster, 2002). Some additional rules (a few) are also based on our experience. The costs for the rules were selected according to the experiments with the aspell corpus.[4] Therefore, the results for this data set cannot be considered objective. We, however, tried to refrain from tuning to this data set.

Our model is, of course, a bit simplistic, since we ignore some more complicated cases of spelling errors, e.g., people's attempt to spell a word by analogy with another word, which is often when the correct spelling is unknown. Such spelling errors are, however, hard to cover with such a simple strategy of modelling errors by substitution rules.

4. Minimal Acyclic Deterministic Finite Automaton as Lexicon Representation

One of the most important decisions in the development of a spelling checking solution is the choice of lexicon representation. There is no single answer which way is the best since the problem has to be considered together with the method of correcting spelling errors.

[4] This data set will be introduced in Section 6.

We propose to use a minimal ADFA (Fig. 3), similarly to (Mihov and Schulz, 2004; Oflazer, 1996; Savary, 2001; Schulz and Mihov, 2002), which allows storing the lexicon in a compact form. There are highly efficient static algorithms for making, searching, and listing the minimal ADFA (Ciura and Deorowicz, 2001; Daciuk *et al.*, 2000). (In the research, we use ADFAs with terminal transitions, the so-called Mealy automata.) The making algorithm is fast, but requires an alphabetically sorted list of words, and there is no simple method to delete or insert a word into an existing minimal ADFA. Fortunately, the lexicon for a spellchecker is fixed and there is no need to extend it, so these limitations are unimportant. If necessary, there are also ways of maintaining the minimal ADFA dynamically (Carrasco and Forcada, 2002; Daciuk *et al.*, 2000), but these methods are slower, and we do not consider them further.

The usage of such a lexicon representation allows fast checking for the existence of a word in the lexicon during the spelling detection stage, and also helps to easily employ substitution rules during the spelling correction stage.

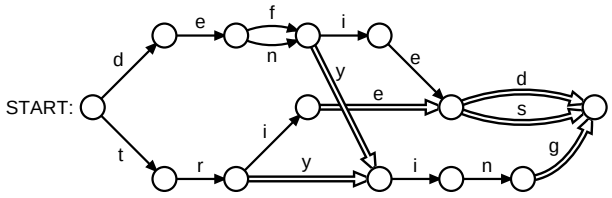


Fig. 3. Minimal ADFA storing a lexicon: {defied, defies, defy, defying, denied, denies, deny, denying, trie, tried, tries, try, trying}. Thick arrows indicate terminal transitions.

5. Implementation Details

Vocabulary incompetence errors, misspellings, and almost all mistypings are handled by substitution rules. A word's splitting is, however, implemented separately. The algorithm of our spelling correction method is presented in Fig. 4.

The parameters of the procedure $find$ are: $word$—the examined non-word, p—the current position in $word$, $trans$—a transition pointing to the next state, $cost$—a total cost of substitution rules employed so far. In the initial execution of this procedure, $p = 0$, $cost = 0$, and $trans$ is the transition pointing to the starting state. The variable $word$ contains the examined non-word at the beginning, but during the process it is a variant of the original non-word obtained from it by a sequence of substitutions or transpositions.

```
    procedure find(word, p, trans, cost);
    begin
01      if trans = NIL or cost > max_cost then return;
                                                  end if;
02      state ← trans.dest; q ← length(word);
03      if p = q and trans.terminal then
            add_suggestion(word, cost); end if;
04      if p > q then return; end if;
        {increment the word position (p)}
05      find(word, p + 1, move(trans, word_p), cost);
        {try all applicable substitutions}
06      for all r ∈ substitution_rules(word, p) do
07          find(word_{0..p-1}||r.out_str||
                word_{p+length(r.in_str)..q-1},
                p + length(r.out_str),
                move(trans, r.out_str), cost + r.cost);
08      end for;
        {try to split the word}
09      if trans.terminal then
10          find(word_{0..p-1}||space||word_{p..q-1}, p + 1,
                trans_to_start, cost + C_S);
11      end if;
    end;
```

Fig. 4. Pseudocode of the spelling correction method implemented in our approach. (The operator $||$ means string concatenation and *space* means a space character.)

The lines 01–04 validate the current transition, $trans$, and examine if we are at the end of the current word or beyond the end. The procedure $add\_suggestion$ adds a $word$ to the list of suggestions. It is executed if the current transition is terminal. In the line 05, we simply move one letter forward in the word $word$. The function $move$ traverses the ADFA from the state pointed by the $trans$ according to the string given as its second parameter. It returns the transition for the last letter of the traversed sequence. It is necessary to obtain the transition, not the pointed state, since we have to verify in the procedure whether the last traversed transition was terminal or not. When there is no possibility to traverse the ADFA, i.e., there is no transition related with the letter, the function $move$ returns NIL. The lines 06–08 implement the application of substitution rules, which are applied to the actual position of the $word$. The substitution rules are stored using a hash table of a size a bit larger than the number of rules. Therefore, they can be easily accessed. The function $substitution\_rules$ returns a set of triples $\langle in\_str, out\_str, cost\rangle$ describing the rules that can be applied in the current position, p, in the word. (If the subindex of the word $word$ exceeds the length of the $word$, an empty character is returned.)

A special case of mistypings—a deletion of the space between consecutive words—reflects in a sequence of letters being joined words. We handle such a situation in the lines 09–11, where the current word is split (the variable C_S means the cost of splitting a word and is also defined by a rule). We insert a space character (denoted in the figure as *space*) into the *word* and start searching from the starting state of the ADFA (the global variable *trans_to_start* means the transition pointing the starting state), since we are looking for another correct word (or try to change the remaining part of the sequence of letters to be a correct word).

6. Experimental Results

To evaluate our spelling checking method, we chose two data sets containing real-world spelling errors. The first one, aspell, is a collection of hard-to-correct errors used for testing GNU Aspell (Atkinson, 2002). The second one, wikipedia, is a data set of typical spelling errors made by the editors of the Wikipedia Project (Wikipedia, 2003).[5] The Wikipedia is edited by many people from the whole world, so the spelling errors are of any kind and reflect spelling problems of different people. The structure of these collections is as follows: The aspell data set contains pairs: wrong word–correct word. The wikipedia data set is a bit more complex. With a single wrong word one or more correct forms is related, as the same wrong word can appear as a spelling error from several correct words. The fraction of wrong words with more than one correct word related is, however, small, about 8%.

We compare our proposition with three popular spellcheckers: Ispell, GNU Aspell, and Microsoft Word[6] built-in spellcheckers.[7] (All the examined spellcheckers make no use of contextual information, similarly to our approach.) To provide an honest comparison of spelling correction methods, not the lexicon contents, the spelling errors were filtered to remove all wrong words for which the correct form (or at least one correct form for wikipedia) did not appear in all the three basic lexicons: Ispell American Extra Large, Aspell American, Word 97. After such a filtering, aspell and wikipedia data sets contain respectively 525 and 2240 wrong words. Our method can be used with any list of words but for the experiments we decided to use the same word list as GNU Aspell, containing $153\,662$ entries, to avoid tuning the lexicon contents to the collections.

Unfortunately, we were unable to compare our approach to the ones presented recently in (Brill and Moore, 2000; Toutanova and Moore, 2002). The authors report impressive results, but they examine the proposed solutions on a set of errors, which is not publicly available and we cannot obtain it for tests. Also, the implementation of

[5] These sets of spelling errors contain no information on the context.

[6] We used the GetSpellingSuggestions method from Word's Visual Basic for Applications to obtain the list of suggestions.

[7] In all spellcheckers, the American English dictionaries were used.

the algorithms, the data sets used for their training, and the lexicons used are unavailable.

It would be ideal to choose the same lexicon for all the spellcheckers. Unfortunately, we were unable to do so, since word lists are usually integrated into spellcheckers. The only thing we could do was choosing the size of the lexicons for the Ispell program to be similar to the others. The sizes of the examined spellchecker lexicons were: Ispell—134 689, Aspell—153 662. The developer of the MS-Word built-in spellcheckers does not publish any information on the size of the lexicons. With respect to our preliminary experiments we can only estimate these sizes to be about 150 000 for Word 97 and over 200 000 for Word 2000 and Word 2003.

Because of language evolution, some wrong words from our data sets exist in spellcheckers' lexicons as correct words. Therefore, we cannot obtain a list of suggestions for them. Fortunately, the fraction of such words is small (the columns denoted as *Present words* in the tables give the number of wrong words from our data sets existing in the spellcheckers' lexicons). To calculate the accuracy of the spellchecker, we proceeded as follows. For each pair $\langle wrong\_word, list\_of\_correct\_forms \rangle$ (for aspell the list contains only one word, but for wikipedia in about 8% cases it is longer), we check the *wrong_word*. If it exists in the spellchecker's lexicon, we increment the t_p counter (number of *Present words*). If it is absent from the lexicon, we obtain the *list_of_suggestions*. Then we increment all the counters t_n ($1 \leq n \leq 10$) for which any of the forms from the *list_of_correct_forms* appears within the top n positions of the *list_of_suggestions*. The values Top-n (for $1 \leq n \leq 10$) denoting the accuracy of suggestions is calculated as

$$\text{Top-}n = \frac{t_n}{t_t - t_p} \times 100\%,$$

where t_t denotes the number of pairs in the data set.

The experiments were made for the Ispell program, the four modes of the GNU Aspell program, the three versions of the Microsoft Word program, and our proposal. Table 1 shows the results of all the examined methods for the aspell data set; the best methods from each family are also compared graphically in Fig. 5(a). As can be observed, the percentage of accurate words returned as the first suggestion by the spellcheckers is rather small, and does not exceed 67%. Only three programs break the 60% threshold, i.e., Word 2000, Word 2003, and our proposition, which provides the best result. If we compare the ratios Top-n, we can observe that our solution is best for all $n \leq 5$, and only for $n = 10$ GNU Aspell is better. In the real world, the ideal situation is, however, to propose the correct form as the first suggestion, or in a very early position as the user can be confused obtaining a list of many suggestions. The more important is, therefore, to propose the highest possible fraction of accurate suggestions at the top of the list. Since the substitution rules, in our research, were adjusted according to the experiments with the aspell data set, the results cannot be considered objective.

Table 1. Experimental results on the aspell data set (525 errors). (The column *Present words* contains the number of wrong words appearing in the spellcheckers' lexicons.)

Method	Present words	Top-1 [%]	Top-2 [%]	Top-3 [%]	Top-5 [%]	Top-10 [%]
Ispell	30	36.0	43.6	47.7	50.3	51.7
Aspell (fast)	14	56.4	65.8	72.8	78.5	84.3
Aspell (ultra)	14	54.6	64.2	70.8	75.9	79.1
Aspell (normal)	14	56.9	66.9	74.4	81.0	87.9
Aspell (bad spellers)	14	55.6	65.8	72.2	78.9	85.3
Word 97	18	59.0	65.7	69.0	71.0	72.6
Word 2000	20	62.6	71.1	74.1	77.0	78.0
Word 2003	20	62.8	71.3	74.1	77.2	78.2
our	14	66.3	75.5	79.6	83.6	85.5

Table 2. Experimental results on the wikipedia data set (2240 errors). (The column *Present words* contains the number of wrong words appearing in the spellcheckers' lexicons.)

Method	Present words	Top-1 [%]	Top-2 [%]	Top-3 [%]	Top-5 [%]	Top-10 [%]
Ispell	68	76.0	81.2	82.8	83.2	83.4
Aspell (fast)	44	83.5	90.5	94.1	95.6	96.5
Aspell (ultra)	44	82.7	90.1	94.2	95.6	96.5
Aspell (normal)	44	84.7	91.8	95.6	97.4	98.5
Aspell (bad spellers)	44	81.4	89.9	93.5	95.4	97.1
Word 97	31	89.0	93.1	94.3	94.7	95.0
Word 2000	42	92.5	95.4	96.1	96.4	96.5
Word 2003	41	92.6	95.5	96.1	96.5	96.6
our	44	94.1	97.4	98.3	98.9	99.0

The second experiment, performed on the wikipedia data set, is impartial, since the errors from this corpus were absent from the training data. Table 2 contains the results, while Fig. 5(b) provides the graphical comparison of the best methods. Since the spelling errors from this corpus are typical, the ratio of accurate suggestions is much higher. Three methods produce over 90% of accurate suggestions in the first position of the list. The ranking of the spellcheckers is similar—again our proposal offers the highest fraction of accurate suggestions in the first position. Even if we compare the methods using the Top-n ratio, for all $n \leq 10$ our algorithm outperforms the others.

Table 3. Average times necessary to produce a list of suggestions of a maximal length 10 for the compared spellcheckers.

Method	wikipedia set [ms/word]	aspell set [ms/word]
Ispell	0.52	0.51
Aspell (fast)	3.40	4.37
Aspell (ultra)	1.81	2.38
Aspell (normal)	51.04	59.05
Aspell (bad spellers)	28.86	32.98
Word 97	15.40	15.14
Word 2000	18.46	16.11
Word 2003	18.14	20.58
our	1.32	1.93

We measured also the speed of spelling correction of the examined methods on a computer equipped with an Athlon XP 2500+ processor clocked at 1833 MHz. The results (Table 3) show that the fastest algorithm is Ispell. Approximately, it is three times faster than our approach. Ispell, however, covers only mistypings and offers poor results when we measure the accuracy of suggestions. The speeds of algorithms from the Aspell family are highly diversified, and the version offering the most accurate suggestions, Aspell normal, is about 30–40 times slower than our proposal. The spellcheckers implemented in Word are also much slower, about 10 times, than our solution.

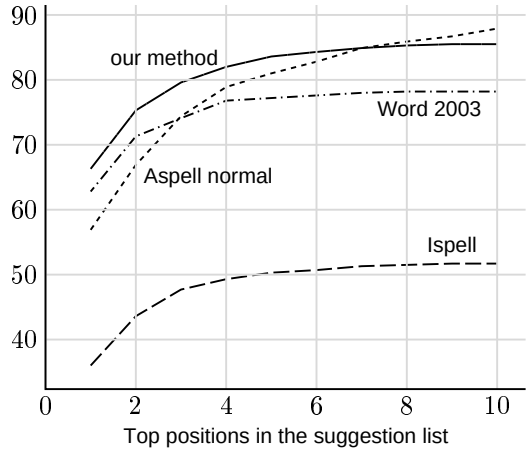
We cannot compare directly our solution to the one described in (Brill and Moore, 2000), since we used different data sets and ran our implementation on approximately a 3–4 times faster processor. Nevertheless, we can roughly say that our solution is several times faster, since the method based on the noisy channel produces a list of suggestions in about 50 milliseconds on Pentium III clocked at 500 MHz, and we do that in less than 2 milliseconds on Athlon XP 2500+ clocked at 1833 MHz.

To make future comparisons possible, we provide the set of rules used in our research at `http://www-zo.iinf.polsl.gliwice.pl/~sdeor/pub/dc05abs.htm`. At the same URL the contents of the aspell and wikipedia collections are available.

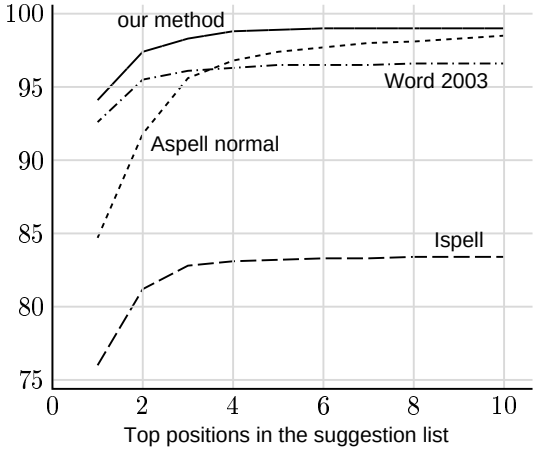
7. Conclusions

We proposed a method of spelling correction based on modelling causes of errors. Using substitution rules we model classic types of spelling errors: vocabulary incompetence, misspellings, mistypings. Other error types, as OCR-related, can also be handled by our proposal. The lexicon is represented as a minimal acyclic deterministic finite automaton, therefore it is compact and spell-checking is extremely fast—a word can be verified in less than a microsecond on a modern PC. Spelling correction is slower, because many words are examined as potential candidates for correct forms of a non-word. The time depends on the type of errors, the number of suggestions, and the lexicon size. We examined two different sets of spelling errors: difficult to correct, aspell, and typical, wikipedia. The time necessary to produce the suggestion list was about 1.5 millisecond per single case. The lexicon size was 153 662 and the maximal number of suggestions was set to 10. Our algorithm outperforms the other examined methods of spell-checking in the ratio of accurate suggestions and speed.

Our method can be applied to many languages (it suffices to collect a set of correct words and define substitu-



(a) aspell data set



(b) wikipedia data set

Fig. 5. Comparison of the accuracy of suggestions for aspell and wikipedia data sets. (At the vertical axes, the percentage of accurate suggestion is given.)

tion rules covering errors typical for the language), except for some truly agglutinative languages, like Turkish, Finnish, etc., since their lexicons cannot be represented as AFDAs. The proposed method of producing the list of suggestions can be, however, used also for cyclic deterministic finite automata that can be employed to represent lexicons for such languages. For a clear presentation, we discussed, however, only the correction of English spelling errors.

The costs of rules in our implementation are integer numbers from the range $[1, 9]$. In the proposed set of rules, we adjusted the costs according to the experiments on one of the test data sets of real errors. The results of correction could be slightly improved if the costs were selected more precisely. We, however, tried to refrain from tuning the set of rules to the chosen sets of spelling errors.

The main disadvantage of our proposal is its restriction to an isolated-word spelling detection and correction problem. An intention of this paper is, however, to show how good suggestions can be proposed when we resign from contextual information. The method can be also employed as a part of a more complex contextual spellchecker as a generator of suggestions which are later ranked according to the context.

An important problem in real applications is that we use a static ADFA, which forbids users to extend the lexicon with new words or prepare their own private lexicons. This can be, however, handled thanks to the second ADFA storing only the user's private lexicon, which is typically much smaller than the main one. The user's ADFA can be implemented dynamically. Also, a static automaton (rebuilt when necessary) is an interesting proposal, since for lexicons smaller than 10,000 words the ADFA is rebuilt with in a few milliseconds on contemporary computers. The list of suggestions obtained from the second ADFA, searched in the same way like the main ADFA, and the main list can be merged in order to offer a single list to the user.

Acknowledgments

We would like to thank Szymon Grabowski for reading a preliminary version of this paper and suggesting improvements. We are also grateful to the referees for their comments on the paper, which helped to make the presentation clearer.

References

Atkinson K. (2002): *Spell checker test kernel results*. — Available at `http://aspell.net/test/`.

Atkinson K. (2003): *GNU Aspell*. — Available at `http://aspell.sourceforge.net/`.

Baeza-Yates R. and Navarro G. (1998): *Fast approximate string matching in a dictionary*. — Proc. 5th Int. Symp. *String Processing and Information Retrieval, SPIRE'98*, Santa Crus de la Sierra, Bolivia, pp. 14–22.

Berkel B. van and Smedt K.D. (1988): *Triphone analysis: A combined method for the correction of orthographical and typographical errors*. — Proc. 2nd Conf. *Applied Natural Language Processing*, Austin, pp. 77–83.

Brill E. and Moore R.C. (2000): *An improved error model for noisy channel spelling correction*. — Proc. 38th Annual Meeting of the Association for Computational Linguistics, Hong Kong, pp. 286–293.

Carrasco R. and Forcada M. (2002): *Incremental construction and maintenance of minimal finite-state automata*. — Comput. Linguistics, Vol. 28, No. 2, pp. 207–216.

Church K.W. and Gale W.A. (1991): *Probability scoring for spelling correction*. — Statist. Comput., Vol. 1, No. 1, pp. 93–103.

Ciura M.G. and Deorowicz S. (2001): *How to squeeze a lexicon*. — Soft. Pract. Exper., Vol. 31, No. 11, pp. 1077–1090.

Czech Z.J., Havas G. and Majewski B.H. (1997): *Perfect hashing*. — Theoret. Comput. Sci., Vol. 182, No. 1–2, pp. 1–143.

Daciuk J., Mihov S., Watson B.W. and Watson R.E. (2000): *Incremental construction of minimal acyclic finite-state automata*. — Comput. Linguistics, Vol. 26, No. 1, pp. 3–16.

Damerau F.J. (1964): *A technique for computer detection and correction of spelling errors*. — Comm. ACM, Vol. 7, No. 3, pp. 171–176.

Damerau F.J. (1990): *Evaluating computer-generated domain-vocabularies*. — Inf. Process. Manag., Vol. 26, No. 6, pp. 791–801.

Damerau F.J. and Mays E. (1989): *An examination of undetected typing errors*. — Inf. Process. Manag., Vol. 25, No. 6, pp. 659–664.

Darragh J.J., Cleary J.G. and Witten I.H. (1993): *Bonsai: A compact representation of trees*. — Softw. Pract. Exper., Vol. 23, No. 3, pp. 277–291.

Gadd T.N. (1990): *PHONIX: The algorithm*. — Program: Automat. Library Inf. Syst., Vol. 24, No. 4, pp. 363–366.

Grudin J. (1983): *Error patterns in skilled and novice transcription typing*, In: Cognitive Aspects of Skilled Typewriting, (W.E. Copper, Ed.). — New York: Springer-Verlag, pp. 121–143.

Hodge V.J. and Austin J. (2003): *A comparison of standard spell checking algorithms and a novel binary neural approach*. — IEEE Trans. Knowl. Data Eng., Vol. 15, No. 5, pp. 1073–1081.

Jassem W. (1983): *The Phonology of Modern English*. — Warsaw: Polish Scientific Publishers.

Knuth D.E. (1973): *The Art of Computer Programming* (Vol. 3. Sorting and Searching Algorithms). — Reading, MA: Addison–Wesley.

Kuenning G.H. (2003): *International ispell*. — Available at `http://www.cs.hmc.edu/~geoff/ispell.html`.

Kukich K. (1988): *Variations on a back-propagation name recognition net*. — Proc. *Advanced Technology Conference*, U.S. Postal Service, Wash. D.C., USA, pp. 722–735.

Kukich K. (1992): *Techniques for automatically correcting words in text*. — ACM Comput. Surveys, Vol. 24, No. 4, pp. 377–439.

Levenshtein V.I. (1966): *Binary codes capable of correcting deletions, insertions and reversals*. — Soviet Physics Doklady, Vol. 10, pp. 707–710.

Maly K. (1976): *Compressed tries*. — Comm. ACM, Vol. 19, No. 7, pp. 409–415.

Mateescu D. (2003): *English phonetics and phonological theory*. — University of Bucharest, Romania. Available at `http://www.unibuc.ro/eBooks/filologie/mateescu`

Merriam-Webster (2002): *A dictionary of prefixes, suffixes, and combining forms from Webster's third new international dictionary, unabridged*. — Merriam–Webster. Available at `http://www.spellingbee.com/pre_suf_comb.pdf`

Mihov S. and Schulz K.U. (2004): *Fast approximate search in large dictionaries*. — Comput. Linguistics, Vol. 30, No. 4, pp. 451–477.

Mitton R. (1987): *Spelling checkers, spelling correctors, and the misspellings of poor spellers*. — Inf. Process. Manag., Vol. 23, No. 5, pp. 495–505.

Mitton R. (1996): *Spellchecking by computer*. — J. Simplif. Spell. Soc., Vol. 20. No. 1, pp. 4–11.

Morrison D.R. (1968): *PATRICIA—Practical Algorithm to Retrieve Information Coded in Alphanumeric*. — J. ACM, Vol. 15, No. 4, pp. 514–534.

Odell M.K. and Russell R.C. (1918): *U.S. Patent Numbers, 1,261,167 (1918) and 1,435,663 (1922)*. — U.S. Patent Office, Washington, D.C.

Oflazer K. (1996): *Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction*. — Comput. Linguistics, Vol. 22, No. 1, pp. 73–89.

Peterson J.L. (1986): *A note on undetected typing errors*. — Comm. ACM, Vol. 29, No. 7, pp. 633–637.

Philips L. (1990): *Hanging on the metaphone*. — Comput. Lang. Mag., Vol. 7, No. 12, pp. 38–44.

Philips L. (2000): *The double metaphone search algorithm*. — C/C++ Users Journal, Vol. 18, No. 6.

Pollock J.J. and Zamora A. (1983): *Collection and characterization of spelling errors in scientific and scholary text*. — J. Amer. Soc. Inf. Sci., Vol. 34, No. 1, pp. 51–58.

Pollock J.J. and Zamora A. (1984): *Automatic spelling correction in scientific and scholarly text*. — Comm. ACM, Vol. 27, No. 4, pp. 358–368.

Savary A. (2001): *Typographical nearest-neighbor search in a finite-state lexicon and its application to spelling correction*. — Lect. Notes Comput. Sci., Vol. 2494, pp. 251–260.

Schulz K.U. and Mihov S. (2002): *Fast string correction with Levenshtein-automata*. — Int. J. Docum. Anal. Recognit., Vol. 5, No. 1, pp. 67–85.

Toutanova K. and Moore R.C. (2002): *Pronunciation modelling for improved spelling correction*. — Proc. 40th Annual Meeting of the Association for Computational Linguistics, Hong Kong, pp. 144–151.

Wagner R.A. (1974): *Order-n correction for regular languages*. — Comm. ACM, Vol. 17, No. 5, pp. 265–268.

Wikipedia (2003): *Wikipedia: List of common misspellings*. — Available at `http://en2.wikipedia.org/wiki/Wikipedia:List_of_common_misspellings`.

Yannakoudakis E.J. and Fawthrop D. (1983a): *An intelligent spelling corrector*. — Inf. Process. Manag., Vol. 19, No. 12, pp. 101–108.

Yannakoudakis E.J. and Fawthrop D. (1983b): *The rules of spelling errors*. — Inf. Process. Manag., Vol. 19, No. 2, pp. 87–99.

Received: 7 January 2005
Revised: 25 February 2005