# Spelling Correction Using Context*

**Mohammad Ali Elmi** and **Martha Evens**
Department of Computer Science, Illinois Institute of Technology
10 West 31 Street, Chicago, Illinois 60616 (csevens@minna.iit.edu)

*Proc. 36 th ACL, 17 th COLING*, Aug. 10-14 1998, Montreal, Quebec, Canada

## Abstract

This paper describes a spelling correction system that functions as part of an intelligent tutor that carries on a natural language dialogue with its users. The process that searches the lexicon is adaptive as is the system filter, to speed up the process. The basis of our approach is the interaction between the parser and the spelling corrector. Alternative correction targets are fed back to the parser, which does a series of syntactic and semantic checks, based on the dialogue context, the sentence context, and the phrase context.

## 1. Introduction

This paper describes how context-dependent spelling correction is performed in a natural language dialogue system under control of the parser. Our spelling correction system is a functioning part of an intelligent tutoring system called Circsim-Tutor [Elmi, 94] designed to help medical students learn the language and the techniques for causal reasoning necessary to solve problems in cardiovascular physiology. The users type in answers to questions and requests for information.

In this kind of man-machine dialogue, spelling correction is essential. The input is full of errors. Most medical students have little experience with keyboards and they constantly invent novel abbreviations. After typing a few characters of a long word, users often decide to quit. Apparently, the user types a few characters and decides that (s)he has given the reader enough of a hint, so we get 'spec' for 'specification.' The approach to spelling correction is necessarily different from that used in word processing or other authoring systems, which submit candidate corrections and ask the user to make a selection. Our system must make automatic corrections and make them rapidly since the system has only a few seconds to parse the student input, update the student model, plan the appropriate response, turn it into sentences, and display those sentences on the screen.

Our medical sublanguage contains many long phrases that are used in the correction process. Our filtering system is adaptive; it begins with a wide acceptance interval and tightens the filter as better candidates appear. Error weights are position-sensitive. The parser accepts several replacement candidates for a misspelled string from the spelling corrector and selects the best by applying syntactic and semantic rules. The selection process is dynamic and context-dependent. We believe that our approach has significant potential applications to other types of man-machine dialogues, especially speech-understanding systems. There are about 4,500 words in our lexicon.

## 2. Spelling Correction Method

The first step in spelling correction is the detection of an error. There are two possibilities:

1. The misspelled word is an *isolated* word, e.g. 'teh' for 'the.' The Unix spell program is based on this type of detection.
2. The misspelled word is a valid word, e.g. 'of' in place of 'if.' The likelihood of errors that occur when words garble into other words increases as the lexicon gets larger [Peterson 86]. Golding and Schabes [96] present a system based on *trigrams* that addresses the problem of correcting spelling errors that result in a valid word.

We have limited the detection of spelling errors to isolated words. Once the word *S* is chosen for spelling correction, we perform a series of steps to find a replacement candidate for it. First, a set of words from the lexicon is chosen to be compared with *S*. Second, a configurable number of words that are close to *S* are considered as candidates for replacement. Finally, the context of the sentence is used for selecting the best candidate; syntactic and semantic information, as well as phrase lookup, can help narrow the number of candidates.

The system allows the user to set the limit on the number of errors. When the limit is set to *k*, the program finds all words in the lexicon that have up to *k* mismatches with the misspelled word.

## 3. Algorithm for Comparing Two Words

This process, given the erroneous string *S* and the word from the lexicon *W*, makes the minimum number of deletions, insertions, and replacements in *S* to transform it to *W*. This number is referred to

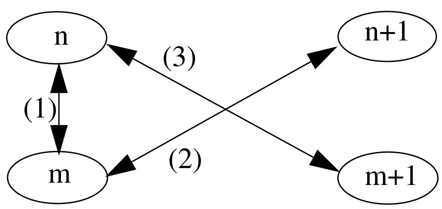
*This work was supported by the Cognitive Science Program, Office of Naval Research under Grant No. N00014-94-1-0338, to Illinois Institute of Technology. The content does not reflect the position or policy of the government and no official endorsement should be inferred.

as the edit distance. The system ignores character case mismatch. The error categories are:

| Error Type | Example | |
|---|---|---|
| reversed order | h**ae**rt | h**ea**rt |
| missing character | hert | he**a**rt |
| added character | h**u**eart | heart |
| char. substitution | h**u**art | h**e**art |

We extended the edit distance by assigning weights to each correction which takes into account the position of the character in error. The error weight of 90 is equivalent to an error distance of one. If the error appears at the initial position, the error weight is increased by 10%. In character substitution if the erroneous character is a neighboring key of the character on the keyboard, or if the character has a similar sound to that of the substituted character, the error weight is reduced by 10%.

**3.1 Three Way Match Method.** Our string comparison is based on the system developed by Lee and Evens [92]. When the character at location *n* of *S* does not match the character at location *m* of *W*, we have an error and two other comparisons are made. The three way comparison, and the order of the comparison is shown below:




| Comparison name | Comparison number 1 | 2 | 3 |
|---|---|---|---|
| no error | T | — | — |
| reversed order | F | T | T |
| missing character | F | F | T |
| added character | F | T | F |
| char. substitution | F | F | F |

For example, to convert the misspelled string *hoose* to *choose*, the method declares missing character 'c' in the first position since the character *h* in *hoose* matches the second character in *choose*.

The three way match (*3wm*) is a fast and simple algorithm with a very small overhead. However, it has potential problems [Elmi, 94]. A few examples are provided to illustrate the problem, and then our extension to the algorithm is described. Let *char(n)* indicate the character at location *n* of the erroneous word, and *char(m)* indicate the character at location *m* of the word from the lexicon.

**3.1.1 Added Character Error.** If the character *o* of *choose* is replaced with an *a*, we get: *chaose*. The *3wm* transforms *chaose* to *choose* in two steps: drops *a* and inserts an *o*.

**Solution:** When the *3wm* detects an added character error, and char(n+1)=char(m+1) and char(n+2)≠ char(m+1), we change the error to character substitution type. The algorithm replaces 'a' with an 'o' in *chaose* to correct it to *choose*.

**3.1.2 Missing Character Error.** If *o* in *choose* is replaced with an *s*, we get the string: *chosse*. The *3wm* method converts *chosse* to *choose* in two steps: insert 'o' and drop the second *s*.

**Solution:** When the *3wm* detects a missing character and char(n+1)=char(m+1), we check for the following conditions: char(n+1)≠char(m+2), or char(n+2)=char(m+2). In either case we change the error to "character substitution". The algorithm replaces 's' with 'o' in *chosse* to correct it to *choose*. Without the complementary conditions, the algorithm does not work properly for converting *coose* to *choose*, instead of inserting an *h*, it replaces *o* with an *h*, and inserts an *o* before *s*.

**3.1.3 Reverse Order Error.** If *a* in *canary* is dropped, we get: *cnary*. The *3wm* converts *cnary* to *canary* with two transformations: 1) reverse order 'na': *canry* and 2) insert an 'a': *canary*.

Similarly, if the character *a* is added to *unary*, we get the string: *uanary*. The *3wm* converts *uanary* to *unary* with two corrections: 1) reverse order 'an': *unaary* and 2) drop the second 'a': *unary*.

**Solution:** When the *3wm* detects a *reverse order* and char(n+2) ≠ char(m+2), we change the error to:

- **Missing character error**: if char(n+1) = char(m+2). Insert char(m) at location n of the misspelled word. The modified algorithm inserts 'a' in *cnary* to correct it to *canary*.
- **Added character error**: if char(n+2) = char(m+1). Drop char(n). The algorithm drops 'a' in *uanary* to correct it to *unary*.

**3.1.4 Two Mismatching Characters.** The final caveat in the three way match algorithm is that the algorithm cannot handle two or more consecutive errors. If the two characters at locations n and n+1 of *S* are extra characters, or the two characters at locations m and m+1 of *W* are missing in *S*, we get to an obvious index synchronization, and we have a disaster. For example, the algorithm compares *enabcyclopedic* to *encyclopedic* and reports nine substitutions and two extra characters.

Handling errors of this sort is problematic for

*Proc. 36th ACL, 17th COLING*, Aug. 10-14 1998, Montreal, Quebec, Canada

Proc. 36th ACL, 17th COLING, Aug. 10-14 1998, Montreal, Quebec, Canada

many spelling corrector systems. For instance, both FrameMaker (Release 5) and Microsoft Word (Version 7.0a) detect *enabcyclopedic* as an error, but both fail to correct it to anything. Also, when we delete the two characters 'yc' in *encyclopedic*, Microsoft Word detects *enclopedic* as an error but does not give any suggestions. FrameMaker returns: *inculpated*, *uncoupled*, and *encapsulated*.

**Solution:** When comparing *S* with *W* we partition them as $S=x\boldsymbol{u}z$ and $W=xvz$. Where $x$ is the initial segment, $z$ is the tail segment, $u$ and $v$ are the error segments. First, the initial segment is selected. This segment can be empty if the initial characters of *S* and *W* do not match. In an unlikely case that $S=W$, this segment will contain the whole word. Second, the tail segment is selected, and can be empty if the last characters of *S* and *W* are different. Finally, the error segments are the remaining characters of the two words:

| initial segment | error segment in *S* / error segment in *W* | tail segment |
|---|---|---|

Using the modified algorithm, to compare the string *enabcyclopedic*, to the word *encyclopedic*, the matching initial segment is *en* and the matching tail segment is c*yclopedic*. The error segment for the misspelled word is *ab* and it is empty for *encyclopedic*. Therefore, the system concludes that there are two extra characters *ab* in *enabcyclopedic*.

## 4. Selection of Words from the Lexicon

To get the best result, the sure way is to compare the erroneous word *S* with all words in the lexicon. As the size of the lexicon grows, this method becomes impractical since many words in a large lexicon are irrelevant to *S*. We have dealt with this problem in three ways.

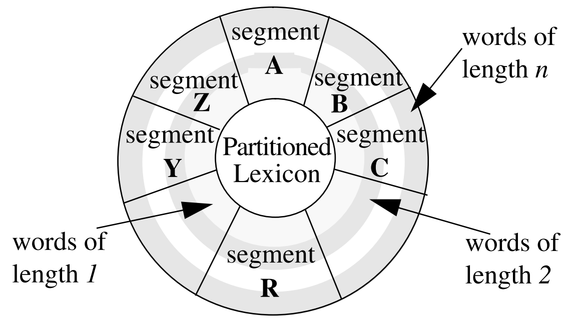
**4.1 Adaptive Disagreement Threshold.** In order to reduce the time spent on comparing *S* with irrelevant words from the lexicon, we put a limit on the number of mismatches depending on the size of *S*.

The disagreement threshold is used to terminate the comparison of an irrelevant word with *S*, in effect acting as a filter. If the number is too high (a loose filter), we get many irrelevant words. If the number is too low (a tight filter), a lot of good candidates are discarded. For this reason, we use an adaptive method that dynamically lowers the tolerance for errors as better replacement candidates are found.

The initial disagreement limit is set depending on the size of *S*: 100 for one character strings, 51* length of *S* for two or more character strings. As the two words are compared, the program keeps track of the error weight. As soon as the error weight exceeds this limit, the comparison is terminated and the word from the lexicon is rejected as a replacement word. Any word with error weight less than the disagreement limit is a candidate and is loaded in the replacement list. After the replacement list is fully loaded, the disagreement limit is lowered to the maximum value of disagreement amongst the candidates found so far.

**4.2 Use of the Initial Character.** Many studies show that few errors occur in the first letter of a word. We have exploited this characteristic by starting the search in the lexicon with words having the same initial letter as the misspelled word.

The lexicon is divided into 52 segments (26 lower case, 26 upper case) each containing all the words beginning with a particular character. Within each segment the words are sorted in ascending order of their character length. This effectively partitions the lexicon into subsegments (314 in our lexicon) that each contains words with the same first letter and the same character size:




The order of the search in the lexicon is dependent on the first letter of the misspelled word, *chr*. The segments are dynamically linked as follows:

1. The segment with the initial character *chr*.
2. The segment with the initial character *as* reverse case of *chr*.
3. The segments with a neighboring character of *chr* as the initial character in a standard keyboard.
4. The segments with an initial character that has a sound similar to *chr*.
5. The segment with the initial character as the second character of the misspelled word.
6. The rest of the segments.

**4.3 Use of the Word Length.** When comparing the misspelled string *S* with length *len* to the word *W* of the lexicon with length *len+j*, in the best case scenario, we have at least *j* missing characters in *S* for positive value of *j*, and *j* extra characters in *S* for negative value of $j$. With the initial error weight of 51*$len$, the program starts with the maximum error limit of $limit=len/2$. We only allow comparison of words from the lexicon with the character length between $len\text{-}limit$ and $len+limit$.

Combining the search order with respect to the initial character and the word length limit, the correction is done in multiple passes. In each alphabetical segment of the lexicon, $S$ is compared with the words in the subsegments containing the words with length $len \pm i$, where $0 \leq i \leq limit$. For each value of $i$ there is at least $i$ extra characters in $S$ compared to a word of length $len\text{-}i$. Similarly, there is at least $i$ missing characters in $S$ compared to a word of length $len+i$. Therefore, for each i in the subsegments containing the words with length $len \pm i$, we find all the words with error distance of $i$ or higher. At any point when the replacement list is loaded with words with the maximum error distance of $i$ the program terminates.

## 5. Abbreviation Handling

Abbreviations are considered only in the segments with the same initial character as the first letter of the misspelled word and its reverse character case.

In addition to the regular comparison of the misspelled string $S$ with the words with the character length between $len\text{-}limit$ and $len+limit$, for each word $W$ of the lexicon with the length $len+m$ where $m>limit$, we compare its first $len$ characters to $S$. If there is any mismatch, $W$ is rejected. Otherwise, $S$ is considered an abbreviation of $W$.

## 6. Word Boundary Errors

Word boundaries are defined by space characters between two words. The addition or absence of the space character is the only error that we allow in the word boundary errors. The word boundary errors are considered prior to regular spelling corrections in the following steps:

1. $S$ is split into two words with character lengths $n$, and $m$, where $n+m=len$ and $1 \leq n < len$. If both of these two words are valid words, the process terminates and returns the two split words. For example, 'upto' will be split into '<u>u</u> pto' for n=1, '<u>up</u> <u>to</u>' for n=2. At this point since both words 'up' and 'to' are valid words, the process terminates.
2. Concatenate $S$ with the next input word $S_2$. If the result is a valid word, return the result as the replacement for $S$ and $S_2$. For example, the string '*specifi*' in '<u>*specifi*</u> <u>*cation*</u>' is detected as an error and is combined with '*cation*' to produce the word '*specification*.' Otherwise,
3. Concatenate $S$ with the previous input word $S_1$. If the result is a valid word, return the result as the replacement for $S$ and $S_1$. For example, in the input '<u>*specific*</u> <u>*ation*</u>' the word '*specific*' is a valid word and we realize we have a misspelled word when we get to '*ation*.' In this case, '*ation*' is combined with the previous word '*specific*' and the valid word '*specification*' is returned.

## 7. Using the Context

It is difficult to arrive at a perfect match for a misspelled word most of the time. Kukich [92] points out that most researchers report accuracy levels above 90% when the first three candidates are considered instead of the first guess. Obviously, the syntax of the language is useful for choosing the best candidate among a few possible matching words when there are different parts of speech among the candidates. Further help can be obtained by applying semantic rules, like the tense of the verb with respect to the rest of the sentence, or information about case arguments.

This approach is built on the idea that the parser is capable of handling a word with multiple parts of speech and multiple senses within a part of speech [Elmi and Evens 93]. The steps for spelling correction and the choice of the best candidates are organized as follows:

1. Detection: The lexical analyzer detects that the next input word $w$ is misspelled.
2. Correction: The spelling corrector creates a list of replacement words: $((w_1\ e_1) \ldots (w_n\ e_n))$, where $w_i$ is a replacement word, and $e_i$ is the associated error weight. The list is sorted in ascending order of $e_i$. The error weights are dropped, and the replacement list $(w_i\ w_j \ldots)$ is returned.
3. Reduction: The phrase recognizer checks whether any word in the replacement list can be combined with the previous/next input word(s) to form a phrase. If a phrase can be constructed, the word that is used in the phrase is considered the only replacement candidate and the rest of the words in the replacement list are ignored.
4. Part of speech assignment: If $w_i$ has $n$ parts of speech: $p_1, p_2, \ldots, p_n$ the lexical analyzer replaces $w_i$ in the list with: $(p_1\ w_i)\ (p_2\ w_i) \ldots (p_n\ w_i)$. Then, factors out the common part of speech, $p$, in: $(p\ w_i)$ $(p\ w_j)$ as: $(p\ w_i\ w_j)$. The replacement list: $((p_1\ w_i\ w_j \ldots)\ (p_2\ w_k\ w_m \ldots) \ldots)$ is passed to the parser.
5. Syntax analysis: The parser examines each sublist $(p\ w_i\ w_j \ldots)$ of replacement list for the part of speech $p$ and discards the sublists that violate the syntactic rules. In each parse tree a word can

Proc. 36th ACL, 17th COLING, Aug. 10-14 1998, Montreal, Quebec, Canada

*Proc. 36th ACL, 17th COLING*, Aug. 10-14 1998, Montreal, Quebec, Canada

have a single part of speech, so no two sublists of the replacement list are in the same parse tree.

6. Semantic analysis: If $w_i$ has $n$ senses $(s_1, s_2, ..., s_n)$ with the part of speech $p$, and $w_j$ has $m$ senses $(t_1, t_2, ..., t_m)$ with the part of speech $p$, the sublist $(p\ w_i\ w_j\ ...)$ is replaced with $(p\ s_1, s_2, ..., s_n, t_1, t_2, ..., t_m\ ...)$. The semantic analyzer works with one parse tree at a time and examines all senses of the words and rejects any entry that violates the sematic rules.

## 8. Empirical Results from Circsim-Tutor

We used the text of eight sessions by human tutors and performed the spelling correction. The text contains 14,703 words. The program detected 684 misspelled words and corrected all of them but two word boundary errors. There were 336 word boundary errors, 263 were split words that were joined (e.g., 'nerv' and 'ous' for *nervous*) and 73 were joined words that were split (e.g., *ofone* for 'of' and 'one'). Also, 60 misspelled words were part of a phrase. Using phrases, the system corrected 'end dia volum' to: 'end diastolic volume.'

The two word boundary failures resulted from the restriction of not having any error except the addition or the absence of a space character. The system attempts to correct them individually:

... quite a sop[h isticated one ...

.... is a deter miniic statement ...

## 9. Performance with a Large Lexicon

To discover whether this approach would scale up successfully we added 102,759 words from the *Collins English Dictionary* to our lexicon. The new lexicon contains 875 subsegments following the technique described in section 4.2.

Consider the misspelled string *ater* [Kukich, 92]. The program started the search in the subsegments with character length of 3, 4, and 5 and returned: *Ayer Aten Auer after alter aster ate aver tater water*. Note that character case is ignored.

Overall, the program compared 3,039 words from the lexicon to 'ater', eliminating the comparison of 99,720 (102759-3039) irrelevant words. Only the segments with the initial characters 'aAqwszQWSZt' were searched. Note that characters 'qwsz' are adjacent keys to 'a.' With the early termination of irrelevant words, 1,810 of these words were rejected with the comparison of the second character. Also, 992 of the words were rejected with the comparison of the third character. This took 90 milliseconds in a PC using the Allegro Common Lisp.

We looked for all words in the lexicon that have error distance of one from *ater*. The program used 12,780 words of length 3, 4, and 5 character to find the following 16 replacement words: *Ayer Aten Auer after alter aster ate aver cater eater eter later mater pater tater water.* Out of these 12,780 words, 11,132 words were rejected with the comparison of the second character and 1,534 with the comparison of the third character.

Finally, lets look at an example with the error in the first position. The program corrected the misspelled string: 'rogram' into: *grogram program engram roam isogram ogham pogrom.* It used 32,128 words from the lexicon. Out of these 32,128 words, 3,555 words were rejected with the comparison of the second character, 21,281 words were rejected with the comparison of the third character, 5,778 words were rejected at the fourth character, and 1,284 at the fifth character.

## 10. Summary

Our spelling correction algorithm extends the three way match algorithm and deals with word boundary problems and abbreviations. It can handle a very large lexicon and uses context by combining parsing and spelling correction.

The first goal of our future research is to detect errors that occur when words garble into other words in the lexicon, as *form* into *from*. We think that our approach of combining the parser and the spelling correction system should help us here.

## 11. References


Elmi, M. 1994. *A Natural Language Parser with Interleaved Spelling Correction, Supporting Lexical Functional Grammar and Ill-formed Input.* Ph.D. Dissertation, Computer Science Dept., Illinois Institute of Technology, Chicago, IL.

Elmi, M., Evens, M. 1993. An Efficient Natural Language Parsing Method. *Proc. 5th Midwest Artificial Intelligence and Cognitive Science Conference*, April, Chesterton, IN, 6-10.

Golding, A., Schabes, Y., 1996. Combining Trigram-based and Feature-based Methods for Context-Sensitive Spelling Correction. *Proc. 34th ACL*, 24-27 June, 71-78.

Kukich, K. 1992. Techniques for Automatically Correcting Words in Text. *ACM Computing Surveys*, Vol. 24, No. 4, 377-439.

Lee, Y., Evens, M. 1992. Ill-Formed Natural Input Handling System for an Intelligent Tutoring System. *The Second Pacific Rim Int. Conf. on AI.* Seoul, Sept 15-18, 354-360.

Peterson, J. 1986. A Note on Undetected Typing Errors. *Commun. ACM,* Vol. 29, No. 7, 633-637.