PRACTICES

Computer Programs for Detecting and Correcting Spelling Errors

James L. Peterson
The University of Texas at Austin

1. Introduction

Computers are being used extensively for document preparation. The typical system is a time-shared one with file systems, text editors, and text formatting programs. The author can enter a document directly on-line with the text editor, storing it in the computer's file system. The document has formatting commands indicating appropriate margins, spacing, paging, indenting, and so on. The text formatter interprets this to produce a file suitable for printing. The text file can be modified using a text editor which is generally an interactive program, while the formatter is more often a batch program.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Key words and phrases: spelling, spelling programs, spelling correction, spelling dictionary CR Categories: 3.42, 4.0, 4.40, 4.49
Author's address: J.L. Peterson, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712.

© 1980 ACM 0001-0782/80/1200-0676 75g.

SUMMARY: With the increase in word and text processing computer systems, programs which check and correct spelling will become more and more common. Peterson investigates the basic structure of several such existing programs and their approaches to solving the problems which arise when this type of program is created. The basic framework and background necessary to write a spelling checker or corrector are provided.

This method of document preparation has created a market for word processing systems. This market will grow as more people use computers to improve their document preparation

Once this mode of document preparation is adopted, several new operations on text files become possible. Our interest here is in the possibility of analyzing documents for spelling errors. Several systems can analyze a text file for potential spelling errors, pointing out probable correct spellings. We expect spelling programs to be a standard part of all future text processing systems. Several companies, including IBM, have

recently begun to promote systems including spelling programs.

There are two types of spelling programs: spelling checkers and spelling correctors. The problem for a spelling checker is quite simple: Given an input file of text, identify those words which are incorrect. A spelling corrector both detects misspelled words and tries to find the most likely correct word. This problem contains elements of pattern recognition and coding theory. A solution is possible only because of redundancy in the English language.

Each word can be considered a point in a multidimensional space of letter sequences. Not all letter se-

Communications of the ACM

quences are words, and the spelling checker must classify each candidate word (called a token) as a correctly spelled word or not. We want to minimize errors, both type I (indicating a correctly spelled word is incorrect) and type 2 (indicating an incorrectly spelled word is correct). Correcting a misspelling involves a search of the word space to select the nearest neighbor(s) of the incorrectly spelled word. (This view of words in a multidimensional space can lead to peculiar representations of words such as in [13] in which each word is represented as a vector in a two-dimensional space.)

Spelling errors can be introduced in many ways. The following three are probably most important.

Author ignorance. Such errors can lead to consistent misspellings and are probably related to the difference between how a word sounds and is actually spelled.

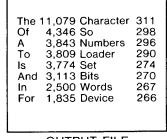
Typographical errors on typing. These are less consistent but perhaps more predictable, since they are related to the position of keys on the keyboard and probably result from errors in finger movements. Studies have shown that large databases may have significant keyboarding errors [4].

Transmission and storage errors. These are related to the specific encoding and transmission mechanisms. Early work in spelling correction was often focused on the problem of optical character recognition (OCR) input [2, 5, 16, 34, 42] and the recognition of Morse code [23].

Different sources of error suggest that some algorithms will be better for some errors.

The original motivation for research on spellers was to correct errors in data entry; much early work was directed at finding and correcting errors resulting from specific input devices in specific contexts. Davidson [8] was concerned with finding the (potentially misspelled) names of passengers for a specific airline flight. Either the stored or inquiry name (or both) might be misspelled. Carlson [5] was concerned with

Computers, like automobiles, television, and telephones, are becoming more and more an integral part of our society. As such, more and more people come into regular contact with computers. Most of these people know how to use their computers only for specific purposes.



INPUT FILE

OUTPUT FILE

Fig. 1. Constructing a Word List. The words have been ordered by number of occurrences.

names and places in a genealogical database. Freeman [10] was working only with variable names and key words in the CORC programming language, while McElwain and Evans [23] were concerned with improving the output of a system so that it would recognize Morse code. Each of these projects considered the spelling problem as only one aspect of a larger problem, and not as a separate software tool.

Many academic studies on the general problem of string matching and correction algorithms have been conducted [1, 7, 22, 32, 33, 43, 44], but not with the aim of producing a working spelling program for general text.

Recently, several spelling checkers have been written for the sole purpose of checking text. Research extends back to 1957, but the first spelling checker written as an applications program (rather than research) appears to have been SPELL for the DEC-10, written by Ralph Gorin at Stanford in 1971. This program and its revisions are widely distributed today.

The UNIX operating system provides two spelling checkers: TYPO and SPELL. These both represent different approaches. Another spelling checker has been written for IBM/360 and IBM/370 systems and is in use at the IBM Thomas J. Watson Research Center in Yorktown Heights.

This paper reports on an investi-

gation of these programs: how they work and their advantages and disadvantages. As a result, we determined how a spelling checker can be written using existing software technology. On the basis of this study, we then proceeded to write our own spelling checker/corrector, using a top-down programming methodology [31].

2. Token Lists

The simplest assistance in checking spelling is an algorithm which lists all distinct tokens in the input (see Figure 1). This requires a person to identify any misspellings; a program of this sort is a simple exercise.

Even at this stage significant decisions must be made: e.g., What is a potential word (token)? It is a sequence of letters, separated by delimiters. Delimiters include blanks and special characters, such as commas, periods, colons, and so on. In most cases, the classification of a character as a letter or a delimiter is clear, but careful thought should be given to the interpretation of numbers, hyphens, and apostrophes.

Tokens which include numbers are often ignored by spelling checkers. This includes both tokens which are totally numbers (1978, 10, and so on) and tokens which are part number and part letter (3M, IBM360, R2D2, and so on). Whether or not these sorts of tokens should be con-

sidered by a speller is best left as an option to be selected by the user.

Hyphens are generally considered delimiters; each part of the hyphenated token is considered a separate token. This follows from the normal use of a hyphen to construct compound words (such as German-American, great-grandmother, four-footed, and so on). It is best not to allow hyphenation at the end of a line in the input to a spelling checker, since it is not possible, in general, to correctly undo such hyphenation.

Apostrophes are generally considered to be letters for the purpose of spelling correction. This is due to their use as an indicator of omitted letters (as in don't and I'd) and of possessives (King's and John's). An apostrophe at the beginning or end of a token might be considered a delimiter, since they are sometimes used as quote markers (as in, "the token 'ten'..."), but plural possessives also have apostrophes at the ends of words (kids').

Another concern is the case of letters in a token. Most frequently, all letters will be lowercase, although the first letter at the start of a sentence is capitalized. Since the is normally considered the same word as The, most spellers map all letters into one case (upper or lower) for analysis. A flag may be associated with the token to allow proper capitalization for spelling correctors, since we want the capitalization of corrected tokens to match the capitalization of the input token. Thus, amung should be corrected to among, while Amung is corrected to Among.

The case problem is actually more complex because of the existence of words, particularly in computer literature, which are strictly uppercase (such as FORTRAN and IBM) and the widespread use of acronyms (GCD and CPU) which are also uppercase. Should fortran or ibm be considered misspellings or separate words? What should be the cases of the letters used to correct an input

token such as aMunG? These problems are minor but require thought in designing spelling correctors. Spelling checkers need only report that aMunG is not correctly spelled in any case; this allows all tokens to be mapped into a uniform case for checking.

A separate problem is a possible decision that tokens which consist solely of uppercase characters should (optionally) not even be considered by a spelling corrector since they will most likely be proper names, variable names, or acronyms, none of which would typically be understood by a spelling program anyway.

Once these decisions are made, creating a list of all distinct tokens is relatively straightforward. The basic algorithm is

Initialize. Get the name of the input file and the output file, open them, and set all variables to their initial state.

Loop. While there are still tokens in the input file, get the next token and enter it into an internal table of tokens.

To enter the token, first search to see if it is already in the table. If not, add it to the table.

Print. When all tokens have been read, print the table and stop.

The key to this program is obviously its internal table of tokens. We need to be able to search the table quickly and find a token (if it exists), or determine that it is not in the table. We also need to be able to add new tokens to the table, although we need not delete them. The speed of the program is especially dependent on the search time. A hash table approach would seem the most reasonable for the table.

The table would be most useful if it were *sorted*. Sorting can be done either alphabetically or by frequency. Attaching a frequency count to each table entry provides the number of uses of each token. This can speed the process by searching higher frequency items first (a selfmodifying table structure); it may also help in determining misspellings. Typographical errors are generally not repeated, and so tokens typed incorrectly will tend to have a very low frequency. Any token with low frequency is thus suspect. Consistent misspellings (due to the author not knowing the correct spelling) are not as easily found using this technique.

This type of spelling program was used to check spelling for the text of Computer Organization and Assembly Language Programming (Academic Press, 1978). Although it was quite tedious to check the resulting list of words (5,649 unique words out of 156,134 tokens), the result was a book with no (known) spelling errors.

3. TYPO

An extension of this token list concept is the basis of the TYPO program on UNIX [24, 26]. This program resulted from research on the frequency of two-letter pairs (digrams) and three-letter triples (trigrams) in English text. If there are 28 letters (alphabetic, blank, and apostrophe), then there are 28 letters, 28² (= 784) digrams and 28^3 (= 21,952)trigrams. However, the frequency of these digrams and trigrams varies greatly, with many being extremely rare. In a large sample of text, only 550 digrams (70 percent) and 5,000 trigrams (25 percent) actually occurred. If a token contains several very rare digrams or trigrams, it is potentially misspelled. The use of digrams and trigrams to both detect probable spelling errors and indicate probable corrections has been one of the most popular techniques in the literature [5, 6, 16, 23, 24, 26, 32, 33, 42].

TYPO computes the actual frequency of digrams and trigrams in the input text and a list of the distinct tokens in the text. Then for each distinct token, an index of peculiarity is computed. The index for a token is the root-mean-square of the indices for each trigram of the token. The index for a trigram xyz given digram and trigram frequencies f(xy), f(yz), and f(xyz) is $[\log(f(xy))]$ -1) + log (f(yz)-1)]/2 - log(f(xyz)-1). (The log of zero is defined as -10 for this computation.) This index is a statistical measure of the probability that the trigram xyz was produced by the same source which yielded the rest of the text.

The index of peculiarity measures how unlikely the token is in the context of the rest of the text. The output of TYPO is a list of tokens, sorted by index of peculiarity. Experience indicates that misspelled tokens tend to have high indices of peculiarity, and hence appear toward the front of the list. Errors tend to be discovered since (a) misspelled words are found quickly at the beginning of the list (motivating the author to continue looking), and (b) the list is relatively short. In a document of 10,000 tokens, only approximately 1,500 distinct tokens occur. This number is further reduced in TYPO by comparing each token with a list of over 2,500 common words. If the token occurs in this list, it is known to be correct and is not output, thereby eliminating about half the distinct input tokens and producing a much shorter list.

4. Dictionary Look-Up

The use of an external list (a dictionary) of correctly spelled words is the next level of sophistication in spelling checkers. The spelling checker algorithm using a dictionary is

Initialize.

Build List. Construct a list of all distinct tokens in the input file.

Search. Look up each token of the list in the dictionary.

If the token is in the dictionary, it is correctly spelled.

If the token is not in the dictionary, it is not known to be correctly spelled and is put on the output list.

Print. Print the list of tokens which were not found in the dictionary.

If the list of input tokens and the dictionary are sorted, then only one pass through the list and dictionary is needed to check all input tokens, thereby providing a fast checker.

The major component of this algorithm is the dictionary. Several dictionaries exist in machine readable form. It may be most practical to use the output of the spelling checker to create the dictionary: i.e., a list of tokens which are not in the dictionary. Starting with a small dictionary, many correctly spelled words will be output by the checker. By deleting spelling errors and proper names

from this list, we have a list of new words which can be added to the dictionary. A new dictionary can easily be created by merging the two.

The size of a reasonable dictionary can be estimated by glancing at books which list words mainly for use by secretaries; e.g., [21]. These usually have 20,000–40,000 words.

One must be careful not to produce too large a dictionary. A large dictionary may include rare, archaic, or obsolete words. In technical writing, these words are most likely to be the result of a misspelling. Another problem with most large dictionaries is the tendency to include incorrectly spelled words. It is neither interesting nor pleasant to verify the correct spelling of 100,000 words.

A dictionary of 10,000 words would be quite reasonable for a small community of users. Since the average English word is about eight characters long, this requires about 100K bytes of storage. Controls on who may modify the dictionary will be needed.

The need for controls creates the post of system dictionary administrator. This person would be responsible for maintaining the shared system dictionary: the addition of new words and the selective deletion of those infrequently used.

One approach to limiting the size of the system dictionary is to create multiple dictionaries, one for each major topic area. This leads naturally to the concept of a common base dictionary with multiple local dictionaries of special words specific to a given topic. When a particular file is to be checked, a temporary master dictionary is created by augmenting the common base with selected local dictionaries. These local dictionaries can be created and maintained by the dictionary administrator or by individual users reflecting their own vocabularies.

The above algorithm for dictionary look-up is a batch algorithm. It is essentially the algorithm used for the UNIX and IBM spelling programs. There are some problems with this approach. First, a substantial real-time wait may be required while the program is running; this

can be quite annoying to a user at an interactive console. Second, the output list of misspelled and unknown tokens lacks context. It may be difficult using some text editors to find these tokens in the text due to differences in the case (for editors which pay attention to upper/lower case) and search difficulties (the file may be quite large, and/or the misspelled token may be a commonly occurring substring in correctly spelled words).

5. Interactive Spelling Checkers

Problems such as those just discussed can be easily corrected with an *interactive* spelling checker. An interactive checker uses the following basic algorithm:

Initialize. Possibly asking the user for mode settings and the names of local dictionaries.

Check. For each input token, search the dictionary.

If the token is not in the dictionary, ask the user what to do.

This is the approach of the DEC-10 SPELL program and the approach which we used with our own spelling corrector.

5.1 Modes of Operation

Several modes of operation may be available in an interactive checker. The interaction with the user may be verbose (novice users) or terse (experts familiar with the program). Local dictionaries of specialized terms may be temporarily added to the large shared system dictionary. A training mode (in which all tokens are defined to be correctly spelled) may be useful for constructing local dictionaries.

The options available to the user are determined by user needs. The following list indicates some options:

Replace. The unknown token is taken to be misspelled and will be replaced. The token is deleted and a correctly spelled word is requested from the user.

Replace and remember. The unknown token is replaced by a user specified word, and all future uses of this token are also replaced by the new word.

Accept. The token is to be considered correct (in this context) and left alone.

December 1980 Volume 23 Number 12

the ACM

Accept and remember. The unknown token is correct and will be left alone. In addition, all future uses of this token are correct in this document; the user should not be asked about them again.

Edit. Enter an editing submode, allowing the file to be modified in its local context.

The use of a CRT as the terminal allows the spelling checker to display the context of an unknown token: At least the line containing the token should be displayed. On terminals with sufficient bandwidth and features, a larger context of several lines or an entire page can be displayed with the token emphasized by brightness, blinking, size, or font. (See Figure 2.)

5.2 The Dictionary

The performance of an interactive spelling checker is important. The user is waiting for the checker and the checker must therefore be sufficiently fast to avoid frustration. Also, unlike batch checkers which need to look up each distinct token once (sorting them to optimize the search), an interactive checker must look up each occurrence in the order in which they are used. Thus, the interactive checker must do more work.

(It would be possible to batch small portions, as much as a page, of the input file. A list of all distinct tokens on a page would be created and the dictionary searched for each token. Each token which was not found would be presented to the user. This is repeated for each page in the input file. The main problems are response time and keeping track of the context of each usage of a token for display.)

The structure of the dictionary is thus of great importance. It must allow very fast searches. The "correct" structure must be determined for the configuration of the computer system. Such factors as memory size, file access methods, and the existence of virtual memory can be significant in determining appropriate structures. If the memory is large enough, the entire dictionary can be kept in memory, making operations easier and faster. If the memory is virtual as opposed to physical, however, the dictionary structure should minimize page faults while searching. If the memory is too small, a two-layer structure is needed, keeping the most frequently referenced words in memory, while accessing the remainder with disk reads as necessary.

Algorithms and data structures for text processing may be tailored to the properties of the English language. Several studies have created large computer files of text which can be analyzed to determine the statistical properties of the language. The most commonly cited collection is the Brown Corpus [20] created at Brown University. The Brown Corpus contains 1,014,232 total tokens

with 50,406 distinct words. The longest word is 44 characters, while 99 percent are 17 characters or less in length; the average word length is 8.1 characters. It is well-known, however, that short words are used more frequently than long words. The ten most common words are the, of, and, to, a, in, that, is, was, and he. Because of this usage of shorter words, the average token length, when weighted by frequency of use, is only 4.7 characters, with 99 percent having 12 characters or less.

Knuth [19] is a good source for different data structures and search algorithms. However, there is no "best" algorithm; each machine and each system make different demands on the dictionary data structure. We will now outline a few approaches.

The DEC-10 SPELL program uses a hash chain table of 6,760 entries. The hash function for a token is the first two letters (L1 and L2) and the length (WL) of the token (2,

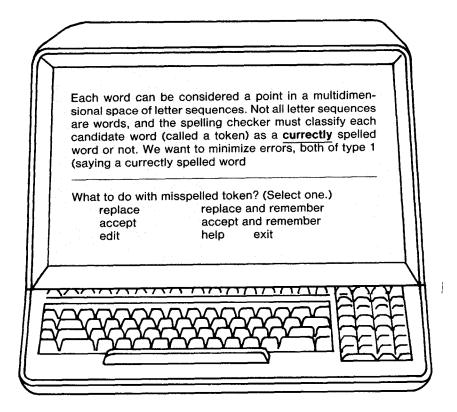


Fig. 2. An Interactive Spelling Program as It Might Appear on a CRT Terminal. Here, a page of text with a highlighted misspelled word, along with a list of corrective actions, are displayed.

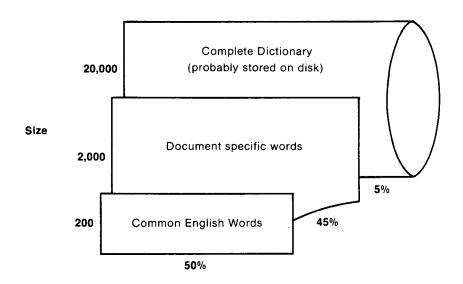
 $3, \ldots, 10, 11$ and over) as (L1 * 26 + L2) * 10 + min(WL - 2, 9). Each hash table entry is a pointer to a chain of words, all of which have the same first two letters and the same length. (This program assumes all tokens of length 1 are correctly spelled.)

Another suggested structure for the dictionary is based on tries [19, 27, 30]. A large tree structure represents the dictionary. The root of the tree branches to 26 different nodes, one for each of the possible first letters of words in the dictionary. Each of these nodes would branch according to the possible second letters, given the known first letter associated with the node. These new nodes branch on possible third letters, and so on. A special bit would indicate the possible ends of words. This structure is particularly appropriate for small computer systems in which a token must be compared, with a dictionary entry, one character at a time. Searching for a token of WL characters requires following the tree WL levels down and checking the end-of-word bit.

Alternative structures are generally based on the frequency of use of words. It is known that usage is extremely variable, and a small number of words are most often used. Hence, we want to structure our search strategy so that the most common words are searched first. One suggestion [35] is a two-level search strategy.

In the two-level search strategy, the token would first be compared with entries in the small in-core table of most frequently used words. If the token is not in this table, a search of the remaining words would be made. This larger table might be stored on secondary disk or drum storage, or in a separate part of virtual memory, requiring longer access and search times. If this table is on secondary storage, access to a block can be accomplished by an in-core index.

From the Brown Corpus, we can determine the following statistics for the percentage of most common English words. The size of the table determines the percentage of tokens found. As the table size is doubled, a larger percentage is found.



frequency of use of tokens

Fig. 3. The Possible Size and Use of Three Separate Dictionaries.

Table Size	% of Tokens Found	Gain
16	28.8	
32	36.0	7.2
64	43.2	7.2
128	49.6	6.4
256	55.6	6.0

Thus, over half of the tokens in normal English occur in a vocabulary of only 134 words. A table of this size can easily be stored in memory. A good search strategy can be designed for such a small table; for example, Knuth [19] gives a hashing algorithm which results in only one comparison to identify any of the 31 most frequently used words in English (35.7 percent of all tokens).

Another improvement in search time can be made by noticing that the total number of distinct tokens in a document tends to be small to moderate (1,500 for a 10,000-word document) and often uses words of particular interest to the subject area. This means that for each specific document, there exists a (small) table of words which occur frequently in that document (but not in normal English). (See Figure 3.) Thus, it is wise to build another table of words which have been used in this specific document. This three-table structure would be searched by

First, the small table of most common English words.

Next, the table of words which have already been used in this document.

Finally, the large list of the remaining words in the main dictionary. If a word is found at this level, add it to the table of words for this document.

Distinct data structures may be appropriate for these three tables since they exhibit the following different properties:

Most common English words—static, small (100–200);

Document specific words—dynamic, small to moderate (1,000–2,000);

Secondary storage dictionary—static, very large (10,000–100,000).

(You may recognize that the latter two data structures are similar to a paging or segmentation scheme. Notice that even if the underlying computer system provides a virtual memory by such a memory management scheme, this data structure will provide improved locality and hence a lower fault rate.)

6. Spelling Correction

An interactive spelling checker can also be helpful in correction. In the simplest case, the checker re-

Communications of the ACM

members all tokens which the user has indicated should be replaced and the words with which the tokens are to be replaced. After the first such replacement, future occurrences of the misspelled token can be automatically corrected, although this feature might be undesirable in some instances.

A similar approach is including common misspellings in the dictionary with the correct spelling. Thus, greatfully would be tagged as a misspelling of gratefully and prehaps as a misspelling of perhaps. This approach has not been included in any current spellers, probably because of the lack of an obvious source of known misspellings and the low frequency of even common misspellings.

Most misspellings can be generated from correct spellings by a few simple rules. Damerau [7] indicates that 80 percent of all spelling errors are the result of:

- (1) transposition of two letters;
- (2) one letter extra;
- (3) one letter missing;
- (4) one letter wrong.

These four rules are the basis for the correction portion of the DEC-10 spelling corrector. (See Figure 4.) The basic algorithm for correction is

For each token which is not found in the dictionary, construct a list of all words which could produce this token by one of the above rules. These are candidate spellings.

If the list has exactly one candidate, guess that word and ask the user if that word is the correct spelling.

If the list contains several words, state this.

The user may then request the list and select one as the correct word, or indicate one of the normal options of replace, replace and remember, accept, accept and remember, or edit.

The candidate list is formed by multiple searches of the dictionary. Transpositions can be detected by transposing each pair of adjacent characters in the token, one at a time, and searching for the resulting token. If the resulting token is found, it is a

candidate and is added to the list. For a token of WL characters, this requires WL-1 searches of the dictionary. Checking for one extra letter requires deleting each character one at a time, and searching for the resulting tokens. This requires an additional WL searches; most tokens are short (five characters on the average) so this need not be expensive. However, no statistics are available on the average length of misspelled words.

The remaining two types of errors (one missing letter and one wrong letter) are difficult to detect. Two strategies are possible. The comparison operation can be defined to include a special match-any character. This means that a search for a missing letter could be achieved by inserting or appending the special match-any character and searching. The characteristics of this type of search are more complex than a simple bit-wise compare and will take more time. Considering the critical nature of performance, two separate searches may be needed, with and without the match-any character feature. A search with a match-any character feature cannot stop when the first word match is found, but must continue, since many words may match the token. This approach requires WL + 1 searches for a missing letter error and WL searches for a wrong letter error.

A second approach is to substitute each potential character in each character position and search normally. If there are N potential characters (N approximately 27 depending on the classification of the apostrophe and similar characters), then a missing letter error requires $N^*(WL + 1)$ searches and a wrong letter requires N^*WL searches.

A specific dictionary structure may allow improvements. The DEC-10 SPELL program separates the dictionary into 6,760 chains by the first two letters and length. For a token of length WL, candidate spellings for one-letter-wrong and transposition errors are of the same length, candidates for one-letterextra errors are of length WL - 1, and one-letter-missing candidates are of length WL + 1. This considerably shortens the number of chains which must be searched. The search strategy is best described by quoting from the comments of the SPELL program itself.

There are four kinds of errors that the program attempts to correct:

one wrong letter; one missing letter; one extra letter; two transposed letters.

For a wrong letter in the third or subsequent character, all words which are candidates must exist on the same chain that the suspect token hashes to. Hence, each entry on that chain is inspected to determine if the suspect differs from the entry by exactly one character. This is accomplished by an exclusive-or (XOR) between the suspect and the dictionary.

Then a JFFO instruction selects the first nonzero byte in the XOR. This byte is zeroed and if the result is all zero, then the dictionary word differs from the suspect in only one letter. All such words are listed at CANDBF, where they can be inspected later.

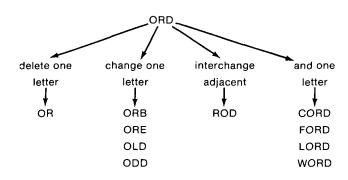


Fig. 4. An Illustration of an Incorrectly Spelled Token and the Correctly Spelled Tokens Which Result When Certain Simple Rules Are Applied.

For a wrong letter in the first or second character, the program tries varying the second letter through all 26 possible values, searching for an exact match. Then all 26 possible values of the first letter are tried, after setting the second letter to its original value. This means that 52 more chains are searched for possible matches.

To correct transposed letters, all combinations of transposed letters are tried. There are only WL-1 such combinations, so it is fairly cheap to do this.

To correct one extra letter, WL copies of the token are made, each with some letter removed. Each of these is looked up in the dictionary. This takes WL searches.

To correct one missing letter, WL+1 copies of the token are made, each time inserting a null character in a new position in the suspect. The null character is never part of any word, so the suspect token augmented by an embedded null can be thought of as a word with one wrong letter (the null). Then the algorithm for matching one wrong letter is used. If the first character is omitted, all 26 possible first characters are tried. Also, 26 more tokens are formed by varying the second character in case that had been omitted.

Counting, we find that a total of 3*WL + 103 chains must be searched, with WL such chain searches requiring a special algorithm to look for (exactly) one wrong character. The dictionary (of say 100,000 words) is split into 6,760 chains, so a chain search ordinarily looks at only 5 to 50 words. For a token of 8 characters this might mean 3,000 comparisons.

Spelling correction is certainly not cheap. It is, however, neither prohibitively expensive nor normally needed; it is needed only for those tokens not found in the dictionary.

Other correction algorithms have been suggested. One theoretically pleasing suggestion [1] is to determine the lengths of common substrings in a token and a dictionary word. This can be used to determine an *index of matching*. The index of matching can be considered the probability the token resulted from a misspelling of the dictionary word. The word with the highest probability can be selected as the most likely correct spelling. This seems to be a very expensive algorithm.

Similar algorithms have been investigated by Wagner and Fischer [43] and Lowrance and Wagner [22].

Additional information can often be used to increase correction accuracy or speed. One important source of additional information is the source of possible errors. For example, errors introduced by OCR input devices are only the substitution of one letter for another; characters are never inserted or deleted by the OCR reader. In addition, it is possible to create a confusion matrix giving the probability $p_{i,j}$ of reading a character i when the correct character is j. This confusion matrix can be used to indicate which characters should be considered when an input letter is to be corrected. In a more general system, digram and trigram frequencies could be used to limit the number of alternative spellings which must be considered.

The layout of the standard typewriter keyboard may also be of use in correcting errors introduced in keyboarding text.

Another technique attacks the problem of words which the original author spelled incorrectly. The misspellings in this case are often based on word sound. It has been suggested that tokens should be converted to standard phonetic spelling, which would be used to find similarly sounding words in the dictionary. This would help correct such apparent double errors as using f for ph or k for qu.

On the other hand, correction based upon the four rules used by the DEC-10 SPELL program is quite successful and relatively cheap, leaving the more difficult corrections to the user.

7. Affix Analysis

For performance reasons, it is desirable to keep the dictionary in memory. Thus, serious thought has been given to ways of reducing the size of the dictionary. In the Brown Corpus of 50,406 words, half were used only once or twice. A dictionary of 8,000 words would represent 90 percent of the words; 16,000 words would yield 95 percent.

A major reduction may be possible, however, by considering common affixes. (An affix is either a suffix or a prefix.) Most plural nouns and verbs result from adding an -s to

the end of the singular forms. Other common suffixes include -ly, -ing, -est, -er, -ed, -ion, and so on. By removing these suffixes and storing only the root word, the dictionary can be reduced significantly. Similar approaches are possible for prefixes.

Two approaches are possible. In the simplest case, each token is examined for a list of common affixes. These are removed if found. Hence, any token ending in -s has the -s removed. (The order of search can assure that larger suffixes like -ness are removed first.) Then the dictionary is searched. If found, the token is judged correct. A major problem with this approach is that it does not catch misspelled tokens which are the result of correct affixes incorrectly applied to correct words, such as implyed or fixs. This can allow misspelled tokens to escape detec-

A solution to this problem is to flag each word in the dictionary with its legal affixes. Then the dictionary is first searched for the token. If the token is not found, it is examined for possible affixes. These are removed and the dictionary searched for the root. If the root is found, its flags are examined to see that the particular affix is legal for this root.

The rules associated with affix removal can be complex. As an example, the following flags and their interpretation are used in the DEC-10 SPELL program. These rules show how to add a suffix to a root word. Let a and b be "variables" that can stand for any letter. Uppercase letters are constants. "..." stands for any string of zero or more letters.

```
"G" flag:

... E \rightarrow ... ING

... a \rightarrow ... aING, if a is not E.

"J" flag:

... E \rightarrow ... INGS

... a \rightarrow ... aINGS, if a is not E.

"O" flag:

... E \rightarrow ... ED

... bY \rightarrow ... bIED, if b is not A, E, I,

O, or U

... ba \rightarrow ... bED, otherwise.

"S" flag:

... bY \rightarrow ... bIES, if b is not A, E, I,

O, or U

... a \rightarrow ... aES, if a is S, X, Z, or H

... a \rightarrow ... aS, otherwise.
```

Suffix removal requires the correct reversal of the above rules, showing how to produce a root and suffix from an input token. Similar rules are defined for prefix removal.

The interaction of affix removal and spelling correction is not well understood, nor is iterated affix removal (removing several affixes). See Figure 5 for an example.

Of the three spelling checkers, each uses a different approach to affix analysis. The IBM speller uses no affix analysis, preferring a larger but simpler dictionary. The UNIX speller removes a few common suffixes and prefixes with no check for the legality of these affixes on the resulting root. Some of the DEC-10 SPELL programs maintain the flags listed above.

8. Design and Implementation

Based upon the research on spelling programs recounted above, we implemented a program to check and correct spelling errors [31]. Our intent was twofold: (1) to produce a machine-independent spelling program design, and (2) to test modern programming principles in the construction of a nontrivial program.

A spelling program is an excellent example of a programming project, especially for students. The concept of a program which detects (and possibly corrects) spelling errors in a text file excites students with its apparent sophistication. In its simplest implementation, however, it is little more than a table look-up program. At the same time, advanced students can extend the project to almost arbitrary complexity.

Our spelling program is an interactive spelling corrector. A three-level dictionary of common English words (256 of them), a hash table of document words, and a disk-based dictionary are used. Local dictionaries are supported. The program was developed completely top-down in a Pascal-like pseudolanguage.

The design consists of two parts. Starting with the main program, a text portion discusses the possible alternative designs and the decisions which lead us to our choice. This choice is then written in a code portion. The code portion uses procedure calls to represent those parts of the design yet to be completed. This approach is then repeated with the undefined procedures, alternating between a text discussion of the design and an eventual code representation of our design choice.

The result of this approach to design is a document which details not only what the program is, but also why the program has developed as it has. The design document serves to both describe and explain.

To implement the speller, a text editor was used to delete the text portion of the design document, leaving only the code portion. The text editor was then used to effect a massive inversion of the code. Note that a top-down design first produces the main program and then its subpro-

cedures. This is the inverse order required by, for example, Pascal. Thus, it is necessary to completely reorder the procedures as produced by the top-down design.

In addition, the use of abstract data types resulted in associating data type and variable declarations with the few procedures and functions which actually use the data items. The declarations in a Pascal program must be at the beginning of the program, preceding all procedure declarations. This required additional movement of code. Finally, some limitations of the Pascal language (e.g., inability to use a set value in a case statement) were overcome by local recoding.

The result of this relatively mechanical editing of the design document was an executable Pascal program. There were no major errors; the errors were either syntactic (e.g., missing semicolons or index variable declarations) or due to a failure to properly initialize some local variables.

There are still some problems with our spelling program, particularly in terms of performance. The program is somewhat sluggish. We are investigating the cause of this difficulty, but feel it is probably the result of one or more of the following three factors:

(1) The dictionary is disk-based. With the dictionary stored on disk, a search for a new word requires considerable time, both for the search and for the disk I/O. This is particularly a problem when the program is attempting to generate correct spellings of a misspelled word. As many as 200 potential spellings,

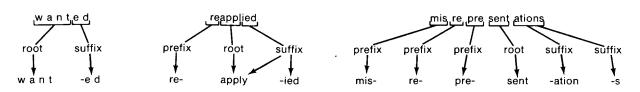


Fig. 5. Prefix and Suffix Analysis in a Spelling Corrector Showing Three Different Levels of Difficulty.

almost all of them incorrect, must be searched for, each generally requiring a disk access.

- (2) The use of separate procedures for subgoals in the top-down design has resulted in, as it now stands, 105 procedures. No single procedure is more than a page in length; many are only a couple of lines long. All but a handful of these procedures are called from exactly one place in the program. This means that substantial procedure call overhead could be eliminated by textually substituting the body of the procedure for the procedure call, renaming local variables if necessary. This might (substantially) improve execution time, but would destroy program readability. Hence, we are investigating the possibility of mechanically performing these substitutions as a prepass before compila-
- (3) Finally, the use of a high-level language, such as Pascal, for this program may result in a substantial increase in execution time over a lower level language. We expect to translate the program design into lower level languages, such as C and various assembly languages, in order to improve both memory storage requirements (for both code and data) and execution time. This problem may be aggravated in the case of Pascal since it lacks a built-in string data type.

9. Future Checkers

The existing spelling checkers clearly demonstrate the feasibility of spelling checkers and correctors for English, and by implication, for similar languages. Suffix and prefix analysis for Romance languages might be easier and more necessary. The applicability of this work to German, Russian, or Oriental languages is unknown.

The existence of a spelling checker and corrector suggests that the next step is to build into the checking programs more knowledge. The following checkers and correctors are possible areas for further research and refinement.

Consistency. In some cases, several correct spellings exist for a given word, such as grey and gray. In addition to assuring that all tokens in a file are correctly spelled, we should assure that a consistent spelling is used. Also, we should check that the appropriate case is used for proper names such as IBM, Algol, Jim, and so on.

Syntax. Given that we can identify words in a sentence, the next step is to classify each word according to its word class (noun, verb, adjective, article, and so on). Given this identification, the structure of each sentence can be checked to assure that each sentence is properly constructed and syntactically correct.

Grammar. Once syntax can be checked, we must check grammar. This includes not only spelling and syntax, but also proper usage of the words and punctuation as well.

Semantics. Finally, a program which examines a document for correct meaning would be useful. This program would check that the ideas in the document are properly developed and presented, that conclusions are correctly based on the facts presented (or known), that the document is complete and consistent, and that the document contributes new information or insights into significant problems.

Each checker would have an associated corrector which would attempt to correct any problems. These programs would greatly aid the preparation of documents, although they would be more difficult to construct.

We have begun efforts to construct a syntax checker. The on-line dictionary is being expanded to include parts of speech for each word, and various table-driven parsing methods are being researched. A question of considerable importance is how much syntax checking can be done without semantic information, particularly for such problems as number agreement and missing articles.

Acknowledgments

Discussions with W.B. Ackerman, R. Amsler, M.H. Conner, and L.R. Power have aided the author in writing this paper.

References

- 1. Alberga, C.N. String similarity and misspellings. Comm. ACM 10, 5 (May 1967), 302–313. This master's thesis reviews previous work, referring to the work of two researchers at IBM Watson who suggest finding the longest common substrings and assigning probabilities based on the portion of the correct string matched. The paper presents rather extensive but unreadable analysis of different algorithms, with no real results. (Reviewed in Compting. Reviews 8, 5 (Sept.-Oct. 1967), 440–441)
- 2. Bledsoe W.W., and Browning, I. Pattern recognition and reading by machine. Proc. Eastern Joint Comptr. Conf., Boston, Mass., Dec. 1959, pp. 225–232. Here, the authors have used a small dictionary with the probability of each word for OCR.
- 3. Blair, C.R. A program for correcting spelling errors. *Inform. and Control 3*, 1 (March 1960), 60-67. Blair weights the letters to create a four- or five-letter abbreviation for each word. If abbreviations match, the words are assumed to be the same. The possibility (impossibility) of building in rules like "i before e except after c and when like a as in neighbor and weigh" is mentioned.
- 4. Bourne, C.P. Frequency and impact of spelling errors in bibliographic data bases. *Inform. Processing and Mgmt. 13*, 1 (1977), 1–12. Bourne examines the frequency of spelling errors in a sample drawn from 11 machine-readable bibliographic databases. He finds that spelling errors are severe enough to influence the search strategy to find information in the database. Errors are not only in the input queries, but also in the database itself.
- 5. Carlson, G. Techniques for replacing characters that are garbled on input. Proc. 1966 Spring Joint Comptr. Conf., AFIPS Press, Arlington, Va., 1966, pp. 189–192. The author uses trigrams to correct the OCR input of genealogical records.
- **6.** Cornew, R.W. A statistical method of spelling correction. *Inform. and Control 12*, 2 (Feb. 1968), 79–93. Cornew employs digrams first, then a dictionary search to correct one character substitutions. The dictionary already exists for a speech output problem.
- 7. Damerau, F.J. A technique for computer detection and correction of spelling errors. Comm. ACM 7, 3 (March 1964), 171–176. The four errors (wrong, missing, extra, and transposed) are mentioned here as accounting for 80 percent of errors. Damerau uses a bit vector for preliminary compare (bit [i] is on if letter i is in word). (Reviewed in Compting. Reviews 5, 4 (July-Aug. 1964), 219.)
- **8.** Davidson, L. Retrieval of misspelled names in an airlines passenger record system. *Comm. ACM* 5, 3 (March 1962), 169–171. The author abbreviates a name to match a stored one. Either name (token or dictionary) may be wrong.

- 9. Fisher, E.G. The use of context in character recognition. Tech. Rep. 76-12, Dept. of Comptr. and Inform. Sci., Univ. of Massachusetts, Amherst, July 1976. Fisher considers the problems of automatically reading addresses from envelopes in the Post Office and also Morse code recognition.
- 10. Freeman, D.N. Error correction in CORC: The Cornell computing language. Ph.D. Th., Dept. of Comptr. Sci., Cornell Univ., Ithaca, N.Y., Sept. 1963.
- 11. Galli, E.J. and Yamada, H.M. An automatic dictionary and verification of machine-readable text. *IBM Syst. J.* 6, 3 (1967), 192–207. This article is a good discussion of the general problem of token identification and verification.
- 12. Galli, E.J., and Yamada, H.M. Experimental studies in computer-assisted correction of unorthographic text. *IEEE Trans. Engineering Writing and Speech EWS-11*, 2 (Aug. 1968), 75–84. Galli and Yamada provide a good review and explanation of techniques and problems.
- 13. Giangardella, J.J., Hudson, J., and Roper, R.S. Spelling correction by vector representation using a digital computer. *IEEE Trans. Engineering Writing and Speech EWS-10*, 2 (Dec. 1967), 57–62. The authors define hash functions to give a vector representation of a word as norm, angle, and distance. This speeds search time (over linear search) and aids in localizing the search for correct spellings since interchanged characters have the same norm and the extra or deleted letter is within fixed range.
- 14. Glantz, H.T. On the recognition of information with a digital computer. J. ACM 4, 2 (April 1957), 178–188. Glantz seems to want either exact match or greatest number of equal characters in equal positions; his approach is good for OCR.
- 15. Hanson, A.R., Riseman, E.M., and Fisher, E.G. Context in word recognition. *Pattern Recognition* 8, 1 (Jan. 1976), 35-45. The authors suggest positional binary trigrams to correct words with wrong letters. See also [33].
- 16. Harmon, L.D. Automatic reading of cursive script. Proc. Symp. on Optical Character Recognition, Spartan Books, Washington, D.C., Jan. 1962, pp. 151–152. Harmon uses digrams and a "confusion matrix" to give the probability of letter substitutions.
- 17. Harmon, L.D. Automatic recognition of print and script. *Proc. IEEE 60*, 10 (Oct. 1972) 1165–1176. Here, Harmon surveys the techniques for computer input of print, including a section on error detection and correction. He indicates that digrams can catch 70 percent of incorrect letter errors.
- 18. James, E.B., and Partridge, D.P. Tolerance to inaccuracy in computer programs. *Comptr. J. 19*, 3 (Aug. 1976), 207–212. The authors describe an approach to correcting spelling errors in Fortran key words.
- 19. Knuth, D.E. The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, Reading, Mass., 1973. This is a classic reference on sorting and searching methods.

- 20. Kucera H., and Francis, W.N. Computational Analysis of Present-Day American English. Brown Univ. Press, Providence, R.I., 1967. The authors give frequency and statistical information for the Brown Corpus of over a million tokens.
- 21. Leslie, L.A. 20,000 Words. McGraw-Hill, N.Y., 1977. This work is representative of several books which list words.
- 22. Lowrance, R., and Wagner, R.A. An extension of the string-to-string correction problem. J. ACM 22, 2 (April 1975), 175–183. This article extends the work of Wagner and Fischer [43] to include adjacent transpositions as an edit operation.
- 23. McElwain, C.K., and Evans, M.E. The degarbler—a program for correcting machineread Morse code. *Inform. and Control* 5, 4 (Dec. 1962), 368–384. The authors' program uses digrams, trigrams, and a dictionary to correct up to 70 percent of errors in machine recognized Morse code. It also uses 5 special rules for the types of errors which can occur (dot interpreted as dash, etc.)
- 24. McMahon, L.E., Cherry, L.L., and Morris, R. Statistical text processing. *Bell Syst. Tech. J.* 57, 6, part 2 (July-August 1978), 2137–2154. McMahon et al. provide a good description of how computer systems can be used to process text, including spelling correction and an attempt at a syntax checker.
- 25. Morgan, H.L. Spelling correction in systems programs. Comm. ACM 13, 2 (Feb. 1970), 90–94. Morgan discusses the use of spelling correction for compilers and operating system JCL. He uses a dictionary with the four classes of errors and also notes that it is possible to use syntax and semantics to narrow search space. He reports on the CORC compiler [10] which associated a probability with each possible misspelling.
- 26. Morris, R., and Cherry, L.L. Computer detection of typograhical errors. *IEEE Trans. Professional Comm. PC-18*, 1 (March 1975), 54-64. Morris and Cherry describe the TYPO program for the UNIX system.
- 27. Muth, F., and Tharp, A.L. Correcting human error in alphanumeric terminal input. *Inform. Processing and Mgmt.* 13, 6 (1977), 329–337. This paper suggests using a tree structure (like a trie) with special search procedures to find corrections. Damerau's review points out that their search strategies need improvement and that their tree is much too big to be practical. Each node of the tree has one character (data) and three pointers (father, brother, son). (Reviewed in *Comptng. Reviews 19*, 6 (June 1978), 231.)
- 28. O'Brien, J.A. Computer program for automatic spelling correction. Tech. Rep. RADC-TR-66-696, Rome Air Development Ctr., New York, March 1967. This report describes an early prototype of a spelling correction program designed to correct input from an OCR reader. It was implemented on a CDC 160A with special hardware.
- 29. Okuda, T., Tanaka, E., and Kasai, T. A method for the correction of garbled words based on the Levenshtein metric. *IEEE Trans. Comptrs. C-25*, 2 (Feb. 1976), 172–177. The authors suggest a method for correcting an incorrectly spelled token.

- **30.** Partridge, D.P., and James, E.B. Natural information processing. *Internat. J. Man-Machine Studies* 6, 2 (March 1974), 205–235. Here, the authors use a tree structure representation of words to allow checks for incorrect input words. This is done in the context of correcting key words in a Fortran program, but more is there. Frequencies are kept with tree branches to allow the tree to modify itself to optimize search.
- 31. Peterson, J.L. Design of a spelling program: An experiment in program design. Lecture Notes in Comptr. Sci. 96, Springer-Verlag, New York, Oct. 1980. In this work the complete top-down design of a program to detect and correct spelling errors is given. The design is machine-independent, but directly results in a Pascal program which implements the design.
- **32.** Riseman, E.M., and Ehrich, R.W. Contextual word recognition using binary digrams. *IEEE Trans. Comptrs. C-20*, 4 (April 1971), 397–403. The authors indicate that the important property of digrams is only their zero or nonzero nature.
- 33. Riseman, E.M., and Hanson, A.R. A contextual postprocessing system for error correction using binary *n*-grams. *IEEE Trans. Comptrs. C-23*, 5 (May 1974), 480-493. Riseman and Hanson suggest using digrams (2-grams), trigrams (3-grams), or in general *n*-grams, but only storing whether the probability is zero or nonzero (1 bit). They also favor using positional *n*-grams which means a separate *n*-gram table for each pair of positions is kept (for each *i* and *j* we have the digram table for characters in position *i* and position *j* in a word).
- 34. Rosenbaum, W.S., and Hilliard, J.J. Multifont OCR postprocessing system. *IBM J. Res. and Develop. 19*, 4 (July 1975), 398-421. This paper focuses very specifically on OCR problems. Searching with a *match-any* character is discussed.
- 35. Sheil, B.A. Median split trees: A fast look-up technique for frequently occurring keys. Comm. ACM 21, 11 (Nov. 1978), 947-958. Sheil defines a tree data structure for searching and applies this, as an example, to a dictionary of common English words.
- 36. Szanser, A.J. Error-correcting methods in natural language processing. *Inform. Processing 68*, North-Holland, Amsterdam, Aug. 1968, pp. 1412–1416. This is a confused paper dealing with correction for machine translation and automatic interpretation of shorthand transcript tapes; it suggests using "elastic" matching.
- 37. Szanser, A.J. Automatic error-correction in natural languages. *Inform. Storage and Retrieval* 5, 4 (Feb. 1970), 169–174.
- **38.** Tanaka, E., and Kasai, T. Correcting method of garbled languages using ordered key letters. *Electronics and Comm. in Japan* 55, 6 (1972), 127–133.
- 39. Tenczar, P.J., and Golden, W.W. Spelling, word and concept recognition. Rep. CERL-X-35, Univ. of Illinois, Urbana, Ill., Oct. 1962.

- **40.** Thomas, R.B., and Kassler, M. Character recognition in context. *Inform. and Control 10*, 1 (Jan. 1967), 43–64. Thomas and Kassler consider tetragrams (sequences of 4 letters); of 27⁴ possible tetragrams, only 12 percent (61,273) are legal.
- **41.** Thorelli, L. Automatic correction of errors in text. *BIT 2*, 1 (1962), 45-62. This paper is a kind of survey/tutorial. It mentions diagrams and dictionary look-up and suggests maximizing probabilities.
- 42. Vossler, C.M. and Branston, N.M. The use of context for correcting garbled English text. Proc. 19th ACM Nat. Convention, Aug. 1964, pp. D2.4-1-D2.4-13. The authors use confusion matrix and word probabilities to select the most probable input word. They also use digrams and strive to improve OCR input.
- 43. Wagner, R.A., and Fischer, M.J. The string-to-string correction problem. J. ACM
- 21, 1 (Jan. 1974), 168-173. Here, an algorithm is presented for determining the similarity of two strings as the minimum number of edit operations to transform one into the other. The allowed edit operations are add, delete, or substitute one character.
- 44. Wong, C.K., and Chandra, A.K. Bounds for the string editing problem. J. ACM 23, 1 (Jan. 1976), 13–16. Wong and Chandra show that the complexity bounds of [43] are not only sufficient but also necessary.

Coming in COMPUTING PRACTICES:

"One More Time-How to Update a Master File"
"Experience with ILIAD: A High-Level Process Control Language"
"The Effect of Programming Team Structures on Programming Tasks"