Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

96

James L. Peterson

Computer Programs for Spelling Correction:

An Experiment in Program Design

Springer-Verlag
Berlin Heidelberg New York

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

96

James L. Peterson

Computer Programs for Spelling Correction:

An Experiment in Program Design

Springer-Verlag
Berlin Heidelberg New York 1980

Author

James L. Peterson
The Department of Computer Sciences
The University of Texas
Austin, Texas 78712/USA

AMS Subject Classifications (1979): 68A30
CR Subject Classifications (1974): 5.2, 3.42

ISBN 3-540-10259-0 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-10259-0 Springer-Verlag New York Heidelberg Berlin

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

PREFACE

"Anyone who can spell in English can't be very bright."

- George Bernard Shaw

The automatic detection and correction of spelling errors by computers has been a subject of interest for a long time. (Our literature search revealed work as early as 1957.) There have been several papers investigating various algorithms and showing their application to various tasks, generally data entry. Now, however, with the increased interest in computer based text processing (word processing) and the storage of large amounts of textual information in computers (data bases), we suggest that spelling correction will become commonplace. This volume brings together the diverse and scattered work on this topic and shows how it can be applied to create a real general purpose spelling corrector.

The theory behind spelling error detection and correction is only one of the considerations of this volume, however. More generally, we consider the design of the program to implement that theory. This work is an *experiment in program design*. Our objective is to demonstrate the application of modern program design principles on a large real (non-toy) program. The approach taken is:

1. A complete literature search provides the background information upon which our program will be based, introducing the problems and solutions which have been considered before.

2. Using our new familiarity with the subject, a complete, top-down program design will be created.

3. Finally, the design will be implemented by converting it into working code. The complete working program is given in Appendix III.

The purpose of this project is as much to investigate and demonstrate modern programming principles as to produce a spelling corrector. We feel that this project can be used as an example of the design of a large program. As such, it is suitable for study.

Consider using this volume in the classroom. The topic appeals to students; it is new, modern and has the appearance of intelligence. The simplest spelling error detector is an exercise in data structures, but it can be expanded and developed to illustrate many topics (disk I/0, interactive programming, data structures, performance, design, ...). Our experience has shown that it can be a very satisfactory programming project. Combining such a project with the reading of this volume provides both the experience and the example needed to motivate modern programming principles.

The program of Appendix III is available in machine readable form from the author (for a minimal cost to cover the computer time for preparing a copy of the program and shipping charges) for educational purposes.

James L. Peterson
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Acknowledgements - Discussions with William B. Ackerman, Robert Amsler, Michael Conner and Leigh Power helped to understand the problems of spelling programs. Jeanne Peterson assisted with proofreading and defining the style of this volume. Art Rinn, Emmett Griner, Mark McCulloch, Barney McCartney and the staff of Texas Student Publications aided in the preparation of the typeset copy. I greatly appreciate their help and time.

TABLE OF CONTENTS

Part I Background

Part II Design of a Spelling Program

PART I

Background

1.0 Introduction

Computers can assist in the production of documents in many ways. Formatting and editing can be consistently applied in a document, producing a high quality document. Appropriate software can be devised to improve the appearance of output on any output device, including typewriters and line printers. Use of sophisticated output devices, such as computer driven phototypesetters, or xerographic or electrostatic printers, can produce documents of outstanding quality.

Computers are being extensively used for document preparation. The systems used are typically a time-shared computer system (such as a DEC-10 or PDP-11 UNIX system) with their file systems, text editors and text formatting programs. The author can enter the document directly on-line with the text editor, storing it in the computer's file system. The document is stored with formatting commands indicating appropriate margins, spacing, paging, indenting, and so on. The text formatter interprets this file to produce an output file suitable for printing. The text file can be quickly modified using a text editor and a new version of the document produced as necessary. The text editor is generally an interactive program, while the formatter is more often a batch program.

This method of document preparation was originally used only by computer science researchers, but the advantages of computer assisted document handling have created a market for *word processing* systems. This market will grow as more and more groups apply computers to improve their document preparation.

Once this mode of document preparation is adopted, however, several other operations on the text files other than editing and formatting become possible. Of particular interest here is the possibility of analyzing documents for *spelling errors*. Several systems have programs which can analyze a text file for potential spelling errors, often pointing out probable correct spellings. This is a fairly recent form of text processing, but its potential is clear to those who have used spelling programs. We expect spelling programs to be a standard part of all future text processing systems.

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 incorrectly spelled. A spelling corrector both detects misspelled words and tries to determine the most likely correctly spelled word which was meant. This problem has elements of pattern recognition and coding theory. A solution is possible only because of the rather great redundancy in the English language.

Each word can be thought of as a point in a multi-dimensional space of letter sequences. Not all letter sequences are words, and the spelling checker must classify each candidate word from the input (called a *token*) as a correctly spelled word or not. We want to minimize errors in classification, both errors of Type I (saying a correctly spelled word is incorrect) and errors of Type II (saying an incorrectly spelled word is correct). In spelling correction, we want to also identify the correct spelling. This involves a search of the word space to select the nearest neighbor(s) of the incorrectly spelled word as the candidates for correct spelling. (This view of words in a multi-dimensional space can lead to peculiar representations of words such as in [Giangardella, *et al* 1967] in which each word is represented as a vector in a 2-dimensional space.)

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

Author ignorance of correct spelling. These errors would lead to consistent misspellings, probably related to the differences between how a word sounds and how it is spelled.

Typographical errors when text is keyboarded. These errors would be less consistent but perhaps more predictable, since they would be related to the position of keys on the keyboard, and probably would result for specific errors in finger movements. Studies have shown that large data bases may have significant errors introduced in keyboarding [Bourne 1977].

Transmission and storage errors within the computer system. These errors would be related to the specific encoding and transmission mechanisms used. Some early work in spelling correction was aimed at the specific problem of optical character recognition (OCR) input [Bledsoe and Browning 1959; Harmon 1962; Vossler and Branston 1964; Carlson 1966] and the recognition of Morse code [McElwain and Evans 1962].

Some checking or correcting algorithms may be better for some types of errors.

The original motivation for research on spellers was to correct errors in data entry. Hence much of the early work was directed at finding and correcting errors resulting from specific input devices in specific contexts. For example, Davidson [1962] was concerned with finding the (potentially misspelled) names of passengers for a specific airline flight. Either (or both) name (in the passenger list or the query) might be misspelled. Similarly, Carlson [1966] was concerned with names and places

in a genealogical data base. Freeman [1963] was working only with variable names and keywords in the CORC programming language while McElwain and Evans [1962] were concerned with improving the output of a system to 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 on its own.

Similarly, many academic studies have been on the general problem of string matching and correction algorithms [Damerau 1964; Alberga 1967; Riseman and Ehrich 1971; Wagner and Fisher 1974; Riseman and Hanson 1974; Lowrance and Wagner 1975], but not with the aim of producing a working spelling program for general text.

Recently, however, several spelling checkers have been written for the sole purpose of checking arbitrary text files. Research on spelling correction extends back to 1957, but the first spelling checker written as an application program (rather than a research experiment) 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 on both DEC-10 and DEC-20 computer systems.

The UNIX operating system provides two spelling checkers: TYPO and SPELL. These are different approaches to the same problem, as will be discussed later.

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 at Yorktown Heights.

These programs have been investigated; this volume reports on the operation of these programs: how they work and the resulting advantages and disadvantages. As a result, we illustrate how a spelling checker can be written for other systems, using existing software technology.

2.0 Token Lists

The simplest form of computer assistance in detecting spelling errors is a program which simply lists all distinct tokens in the input document. This requires a person to scan the list, identifying any misspellings. A program of this sort is a simple exercise and should be easy to write.

Even at this point, however, significant decisions must be made about what is a potential word (token). A word 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 these sorts of tokens should be considered by a speller might be 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). The use of hyphens for hyphenation at the end of a line is best not allowed in the input to a spelling checker, since it should be well known that it is not possible in general to undo such hyphenation correctly.

Apostrophes are generally considered to be letters for the purpose of spelling correction. This is because of their use as an indicator of omitted letters (as in "don't", "I'd", and so on), and possessives ("King's", "John's", and so on). 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 for the *case* of letters in a token. Most frequently all letters will be in lower case, although the first letter is capitalized at the start of a sentence. 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 than this because of the existence of words, particularly in computer literature, which are strictly upper case (such as "FORTRAN", "IBM", and so on) and the widespread use of acronyms ("GCD", "CPU", and so on) which are typically upper case. Should "fortran" or "ibm" be considered misspellings or separate words from their upper case equivalents? (This is especially true in languages such as German where use of incorrect upper or lower case is considered a spelling error.) What should be the cases of the letters used to correct an input token such as "aMunG"? These problems are minor but need thought for spelling correctors. Spelling checkers need only report that "aMunG" is not a correctly spelled token 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 upper case characters should (optionally) not even be considered by a spelling corrector since they most likely will be proper names, variable names, or acronyms, none of which would typically be understood by a spelling program anyway.

Once these decisions are made, it is relatively straightforward to write a program to create a list of all distinct tokens. The basic algorithm is:

Initialize. Get the name of the input file and the output file for the list, 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, then add it to the table.

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

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

The table would be most useful if it were *sorted*. Sorting can be either alphabetically or by frequency. Attaching a frequency count to each table entry provides a count of the number of uses of each token. This can speed the search of the table by searching higher frequency items first (a self-modifying table structure) and also may give help in determining misspellings. Typographical errors are generally not repeated, and so tokens typed incorrectly will tend to have 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 by this technique.

3.0 TYPO

An extension of this idea is the basis of the TYPO program on UNIX [Morris and Cherry 1975; McMahon, *et al* 1978]. 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 28^2 (= 784) digrams and 28^3 (= 21,952) trigrams. However, the frequency of these digrams and trigrams varies greatly, with many being extremely rare. Typically, in a large sample of text, only 550 digrams (70%) and 5000 trigrams (25%) actually occur. 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 [Harmon 1962; McElwain and Evans 1962; Vossler and Branston 1964; Carlson 1966; Cornew 1968; Riseman and Ehrich 1971; Riseman and Hanson 1974; Morris and Cherry 1975; McMahon, *et al* 1978].

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 that produced 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 towards the front of the list. Errors tend to be found 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 ten thousand tokens only approximately 1500 distinct tokens occur. This number is further reduced in TYPO by comparing each token with a list of over 2500 common words. If the token occurs in this list, it is known to be correct and is not output by TYPO. This simple process will typically eliminate half of the distinct input tokens, producing a much shorter list to be inspected by the author as potential misspellings.

4.0 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 for the user's attention.

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, providing a very fast checker.

The major component of this algorithm is the dictionary. Several dictionaries exist in machine readable form, or one could sit and type one in. More practical

would be to use the output of the spelling checker to create the dictionary. The output is a list of tokens which are not in the dictionary. Starting with a small dictionary, many correctly spelled but unknown 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. Since both the old dictionary and the new words are sorted, a new dictionary can be easily created by merging the two.

The size of a reasonable dictionary can be estimated by the existence of books which simply list words, mainly for use in spelling by secretaries, such as [Leslie 1977]. A survey of local bookstores reveals such books with 20K, 20K, 35K and 40K words.

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

A dictionary of 10,000 words, particularly if produced by a small community of users, should be quite reasonable. Since the average word in English is about eight characters, this requires about 100K bytes of storage in the computer system. Hence, the dictionary will probably be shared among the users of the system. Controls will be needed on who may modify the dictionary.

The need for controls on the modification of the shared dictionary creates the need for a system *dictionary administrator*. The dictionary administrator would be responsible for maintaining the shared system dictionary. This involves the addition of new words (or words whose frequency of use suddenly increases) and the selective deletion of infrequently used words.

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 to reflect their own vocabularies.

The algorithm mentioned above 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 to find some tokens in the text using some text editors due to differences in 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).

5.0 Interactive Spelling Checkers

These problems can be easily corrected with an *interactive* spelling checker. An interactive checker uses the following basic algorithm.

Initialize. This may include asking the user for mode settings and the names of local dictionaries to be used.

Check. For each token of the input file, search the dictionary for it.

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

This is the approach of the DEC-10 SPELL program.

5.1 Modes of Operation

Several *modes* of operation may be useful in an interactive checker. The interaction with the user may be *verbose* (for novice users) or *terse* (for experts who are familiar with the program). Local dictionaries of specialized terms may be temporarily added to the large shared system dictionary. A training mode (where all tokens not in the system dictionary are defined to be correctly spelled) may be useful for constructing such local dictionaries.

The options available to the user when an unknown token is found are determined by user needs. The following list indicates some possible options.

Replace. The unknown token is taken to be misspelled and should be replaced. The token is deleted from the input stream and a correctly spelled word is then requested from the user to be inserted in its place.

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

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

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

Edit. Enter an editing submode allowing the file to be arbitrarily modified in the local context of the token.

Use of a CRT as the terminal for interaction with the speller allows the spelling checker to display the context of an unknown token. At least the line in which the

token is found should be displayed. On terminals with sufficient bandwidth and features, a larger context of several lines or an entire page could be displayed with the token emphasized by brightness, blinking, size or font.

5.2 *The Dictionary*

The performance of an interactive spelling checker is of great concern. The user is waiting for the checker and so the checker must be sufficiently fast to avoid frustrating the user. Also, unlike the batch checkers which need to look up each distinct token only once (and can sort these tokens to optimize the order of search), an interactive checker must look up each occurrence of a token in the order in which they are used. Thus, the interactive checker intrinsically must do more work.

(It would be possible to batch small portions, up to say 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 with response time and in keeping track of the context of each usage of a token for display.)

The structure of the dictionary is therefore of great importance. The dictionary structure must allow very fast searches. The "correct" structure must be determined for the configuration of the local computer system. Such factors as memory size, file access methods, and the existence of virtual memory can be significant factors determining appropriate data structures. If memory is large enough, the entire dictionary can be kept in memory making things much easier and faster. If this memory is virtual, as opposed to physical, however, the dictionary structure should be selected to minimize page faults while searching. If 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.

Many algorithms and data structures for text processing are affected by the properties of the English language. Several studies have created large computer files of text which can then be analyzed to determine the statistical properties of the language. The most commonly cited such collection is the Brown Corpus [Kucera and Francis 1967] 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% 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 higher usage of shorter words, the average token length when weighted by frequency of use is only 4.7 characters, with 99% of the words being of 12 characters or less in length.

Knuth [1973] is a good source for different data structures and search strategies. Several different algorithms and their properties are considered. However, there is no "best" algorithm; each machine and each system makes different demands on the dictionary data structure. A few approaches are outlined below.

The DEC-10 SPELL program uses a hash chain table of 6760 entries. The hash function for a token is the first two letters (L1 and L2) and the length (*WL*) of the token (2, 3, ..., 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* [Knuth 1973; Partridge and James 1974; Muth and Tharp 1977]. 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. To search 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 used most often. Hence, we want to structure our search strategy so that the most common words are searched first. One suggestion [Sheil 1978] is for a *two-level search strategy*.

In the two-level search strategy, a given token would first be looked up in the small in-core table of most frequently used words. If the token is not in this table, a search would be made of a larger table of the remaining words. 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 were on secondary storage, direct access to a particular block, which could be identified by an in-core index structure, would be most effective.

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

Table Size	Percent Tokens Found	Gain
16	28.8	
32	36.0	7.2
64	43.2	7.2
128	49.6	6.4
256	55.8	6.2

Over half of the tokens in normal English result from a vocabulary of only 168 words. A table of this size can easily be stored in memory. A good search strategy can be designed to speed search of this small table. For example, Knuth [1973] gives a hashing algorithm which results in only one comparison to identify any of the 31 most frequently used words in English (35.7% of all tokens).