Forgive Them, For They Know Not What They Do (Luke, 23:34)

by Winfried Gerum

Winfried Gerum

Some old hands still say garbage in, garbage out. But times change, and it has become fashionable to use the word "fuzzy" instead. Users are still notoriously fuzzy in their input, i.e., requirements, orders, or even plain data entry. Instead of replying with some obscure error message, we now try to make sense of input that is wrong if taken literally.

What's in a name? Names are often in the fuzzy area. If you have a database with names, you have to take care of a frequent problem of finding an entry if the exact spelling is unknown. And names sometimes have really strange spellings. The time-honored SOUNDEX algorithm for handling names is well known in the M world. For those who missed it so far, here is a short introduction before going on to another related concept about names.

The solution is mapping a name to a class of similar sounds. The workings are as follows:

Map lowercase to uppercase characters. Let the first character represent itself. For additional characters proceed by mapping AEIOU and HWY to class 0, the characters BFPV to class 1, the characters CGJKQSXZ to class 2, the characters DT to class 3, the character L to 4, M and N to 5, and R to 6, and discard all nonalphabetic characters. Then treat adjacent occurrences of the characters of the same class as a single occurrence. Then purge all references to class 0 (mostly vowels). Finally, use only the first four characters of the result.

```
          ;-Compute SOUNDEX value of X
SOUNDEX(X) NEW A,C,D,I
        +1 SET D=$TR(X,"abcdefghijklmnopqrstuvwxyz","ABCDEF
           GHIJKLMNOPQRSTUVWXYZ")
        +2 SET A=$E(D),$E(D)="",C=""
        +3 SET D=$TR(D,"AEHIOUWYBFPVCGJKQSXZDTLMNR"_D,"0000
           00001111222222223334556")
        +4 FOR I=1:1:$L(D) SET:C'=$E(D,I) C=$E(D,I),A=A_C
        +5 QUIT $E($TR(A,0),1,4)
```

This is the basic algorithm. Some refinements are possible, such as mapping PH to F, and so on. As long as there are not too many foreign names in a database, there is no need for many refinements.

This algorithm looks very crude, since a wide range of characters are mapped to the same value. If you take combinations of up to six characters, there are 321,272,406 combinations mapped into just 6,734 different SOUNDEX values. That means there are an average of 47,709 "words" mapped into each SOUNDEX value.

The distribution, however, is not even: A666 is a rare SOUNDEX value. Only sixty-four six-character words map into it. In contrast, A22 represents 205,312 "words." In reality, that huge realm of "possible" names is populated by only a small number of real names. Therefore, the performance of the SOUNDEX algorithm in real applications is quite satisfying.

To see just how good it is, I checked against some databases with real names (from German environments).

Database 1 is a small collection of surnames, Database 2 a large collection of surnames, and Database 3 a list of first names. Table 1 gives the number of entries (different names) in each database, the number of different SOUNDEX values derived from the names, the number of names that produce a unique SOUNDEX value, the average number of names mapping to one SOUNDEX value, and the maximum number of names mapping to one single SOUNDEX value.

The interpretation is clear: the smaller a database, the more useful the

	Entries	SOUNDEX Values	Unique SOUNDEX	Average Entry/Sdx	Max # Entry/Sdx
Database 1	1,502	841	528	1.8	11
Database 2	26,709	3,718	957	7.2	123
Database 3	2,119	919	451	2.3	17

Table 1. Results of SOUNDEX matches from German-language databases.

SOUNDEX method. It is wonderful for a physician in solo practice, but it has limitations if used unrefined in a clinic.

Even in small databases, the SOUNDEX method has its limitations. A small change in a word may produce a very different SOUNDEX value. Let's use $$SOUNDEX ("Marlboro") = "M614" $$SOUNDEX ("Mallboro") = "M416". That example demonstrates that the SOUNDEX is not suitable for use in something like a spelling checker. A simple spelling checker that just looks at whether a given word is in a list (a global) is extremely straightforward to have in M. But today more sophistication is required. A word processor such as WordPerfect does not just say that a word might be wrong, it also gives a list of possibly correct alternatives. The amazing thing is that in most cases one of the first alternatives in a given list is the right one. How is this done?

Quite simply, you need a definition of a distance between two words, so that a "small" change gives a small value for "distance." There is something called the "Levenshtein Distance" between two words. It is "the minimum number of character insertions, character deletions, or character replacements to change one word into another." That sounds reasonable in the definition but awfully complicated to implement. As happens so often, just a few lines of M code will do the job:

```
            ;Function $$LSD
            ;Computes Levenshtein Distance between two words
LSD(W1,W2)  NEW A,I1,I2,L1,L2,R,T,X
            SET L1=$L(W1),L2=$L(W2)
            QUIT:'L1 L2 QUIT:'L2 L1
            SET T=0 FOR I1=0:1:L1 SET A(0,I1)=I1
            FOR I2=1:1:L2 DO
            .SET T='T,A(T,0)=I2
            .FOR I1=1:1:L1 DO
            ..SET R=A('T,I1-1)+$$DCHAR($E(W1,I1),$E(W2,I2))
            ..SET X=A('T,I1)+1  SET:X<R R=X
            ..SET X=A(T,I1-1)+1 SET:X<R R=X
            ..SET A(T,I1)=R
            QUIT R

            ;Function $$DCHAR
            ;"Unequalness" between two characters
            ;returns 0 if two characters are equal
            ;returns 1 if they are "completely" different
DCHAR(C1,C2) QUIT C1'=C2
```

How can this be used in a spelling checker? Naturally, you need a list of words appropriately stored in a global. When words are checked against the list, there is no problem. Otherwise the checker scans the list, computes the Levenshtein Distance between each word and the word in question, then presents words with a small distance (or change) for selection.

Checking a word against every entry in a database seems very wasteful, but there is no simple remedy. The Levenshtein Distance does not impose an order, and no matter how you arrange the words, this unwieldy sense of convolution persists. To reduce the search, you can sort the names by their length. The difference in the number of characters imposes an upper limit on the Levenshtein Distance between two words.

Just store your words in a two-level global with the length of the word as the first subscript and the word as the second subscript. When searching for lookalikes of, say, a four-character word, scan the sublists with three-, four-, and five-character words. That still leaves a lot of entries to check against. If errors in the first character were impossible, you could drastically reduce the search. Compare words of suitable length beginning with the same character. Unfortunately, mistakes have "minds" of their own. Still, the idea looks too good to be discarded. If your database contains regular words (straight) and $REVERSEd, then you could do a fast check against all entries that share either the first or the last character with the word in question. Then your global has two entries for each word:

```
SET ^DICT($L(WORD),"STRAIGHT",
    WORD)=""
SET ^DICT($L(WORD),"REVERSED",
    $RE(WORD))=""
```

```
Spelling Checker using Levenshtein Distance
Function $$SPELLCHK
Checks WORD and returns
   1_";"_WORD       if word is invalid,
or 1_";"_ALTERNATE if alternate has been selected,
or 0               if word cannot be (re)interpreted.
```

```
SPELLCHK(WORD) NEW $R
            QUIT:$D(^DICT(WORD)) "1;"_WORD ;word in list
            NEW D,I,L,MAXD,R,W
            SET MAXD=2 ;-Maximum distance of interest
            SET W=""
            ;search for similar words
            FOR SET W=$O(^DICT(W)) QUIT:W=""  DO
            .SET D=$$LSD(WORD,W)
            .QUIT:D>MAXD  ;no interest in large distance
            .SET L(D,W)=W ;note word
            ;something similar ...
            QUIT:'$D(L) 0 ;...no
            ;display  alternates
            WRITE !,"Word '",WORD,"' not in dictionary,"
            WRITE " please select alternate"
            SET I=0
            WRITE !,"(",I,")",?5,"edit word"
            FOR D=0:1:MAXD F  S W=$O(L(D,W)) Q:W=""  D
            .SET I=I+1 W !,"(",I,")",?5,W
            ;select alternate
            READ !,"Your Choice>",R
            ;here we should test for valid input
            ;no alternate selected
            QUIT:'R 0
            ;return selected alternate
            SET W="L"
            FOR I=1:1:R SET W=$Q(@W) QUIT:W=""
            QUIT:W="" 0
            QUIT "1;"_@W
```

Another possible modification of the basic Levenshtein Distance is to assign small difference values to characters that are neighbors on the keyboard. That metric could help a poor typist.

None of the algorithms presented in this article fits all needs to remedy poor input. The SOUNDEX method is too crude to be used as a spelling checker, except in cases when there are fewer than 2,000 words. It is perfect in small databases dealing with names. You can fine tune a spelling checker with the Levenshtein Distance to a high level of sophistication, but it is very demanding on your processor.

While the core algorithms can be written in any programming language, the surrounding database machinations are best done with—what else?—M.

Then some words will be checked twice, but the global dramatically confines the search, and in that way a spelling checker becomes practical.

Introducing the function DCHAR seems to be overkill for a simple comparison between two characters. Two characters are either equal or not, aren't they? Well, matters are not that simple: You probably feel that uppercase *A* is very different from *X*, but only slightly different from lowercase *a*. Characters sharing the same SOUNDEX class might be considered less different than two characters belonging to different SOUNDEX classes.

Using that modified idea of a difference between two characters, a spelling checker can become more forgiving about phonetic misspellings in a similar fashion to the SOUNDEX algorithm. The spelling checker in WordPerfect seems to use something like this Levenshtein distance: It is very good on phonetic mistakes, but it does not find some nonphonetic errors. WordPerfect seems to compare phonemes instead of characters.

```
                ;Function $$DCHAR
                ;"Unequalness" between two characters
                ;returns 0 if two characters are equal
                ;returns .1 if two chars differ in case only
                ;returns .5 if two chars share a SOUNDEX class
                ;returns 1 if they are "completely" different
DCHAR(C1,C2)    Q:C1=C2 0
                S X1=$$UC(C1),X2=$$UC(C2)
                Q:X1=X2 .1
                Q:$$SDXC(X1)=$$SDXC(X2) .5
                Q 1
                ;$$UC makes characters uppercase
UC(X)           Q $TR(X,"abcdefghijklmnopqrstuvwxyz","ABCDEFG
                HIJKLMNOPQRSTUVWXYZ)
                ;$$SDXC SOUNDEX class of a character
SDXC(X)         Q +$TR($E(X),"BFPVCGJKQSXZDTLMNR"_D,"11112222
                 2222334556")
```

Winfried Gerum is with Winner Software GmbH in Röttenbach, Germany. Send your ideas for topics to him by phone at 011-49-9195-940022 or by fax at 011-49-9195-940030.