Writing a spellchecker

Spellcheckers seem simple enough, but how do they know which corrections to suggest?

In this issue...

▶ WHAT'S COVERED

We all use spellcheckers. Without them, our documents would be full of avoidable errors. By using some crafty algorithms and a simple state machine, we have all we need to create a system that can identify rogue words, quickly compare them to a word list and produce a list of sensible alternative spellings.

None of us are too old to remember that strange time when writers had to use word processors without the benefits of an automated spellchecker. It was a time when the jokes came fast and furious, even to the point of *The Guardian* newspaper becoming known as *The Grauniad* because of the legendary typos sent to print by its tired subbing staff.

Pretty soon though, word processors started to come with spell-checking utilities that would check the text for spelling errors. At first they were activated manually – you had to select the option to check the spelling in your document – but soon Microsoft shipped a version of Word (Word 95, in, er, 1995) that was able to check your spelling as you typed and would underline misspelled words with a red squiggly line.

Alongside these user interface improvements came algorithms to enable the word-processing program to suggest corrections for the misspellings encountered in the document. Later still came grammar checkers, which are still in their infancy and can even now get things very wrong (for instance, as I type this paragraph, Word 2007's grammar checker is insisting that the phrase 'these user' in the first sentence should be changed to 'this user').

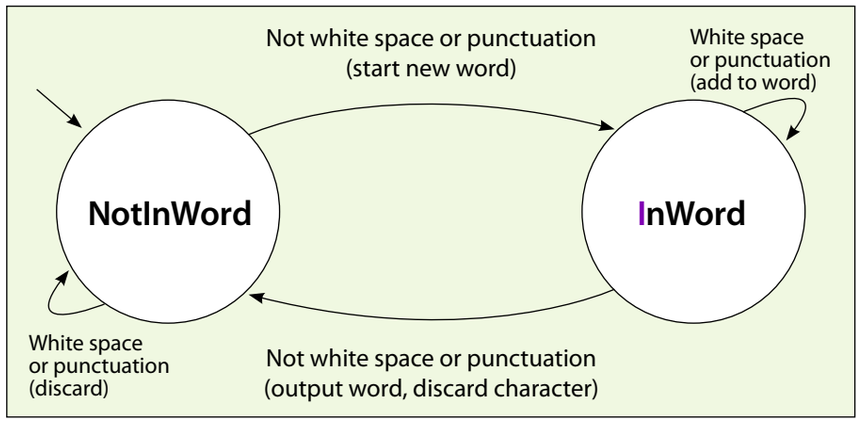
Nowadays, the spellchecker is so ingrained in word-processing applications that we no longer think about checking our spellings manually. Indeed, the jokes have now moved on to mock the sometimes-wacky suggestions that correction algorithms produce. And because we often assume that if there are no red squiggly lines in the document then the text is good, a new issue has arisen: the possibility of correctly spelled yet inappropriate words in the text going unnoticed (my personal issues are 'from' and 'form', and 'class' and 'calls').

So the spellchecker is now invaluable for people from all works of life. But how do they work, and how might we go about writing one?

Creating a state machine

There are two algorithms at play in the most basic kind of spellchecker. The first algorithm takes a body of text and splits it up, identifying the separate words. These words are then checked to see if they're spelt correctly using the second algorithm.

There are several ways to break up a chunk of text into its component words. Sometimes the run-time of the language you use may have ▶



▲ **Figure 1: A simple state machine for extracting words from text.**

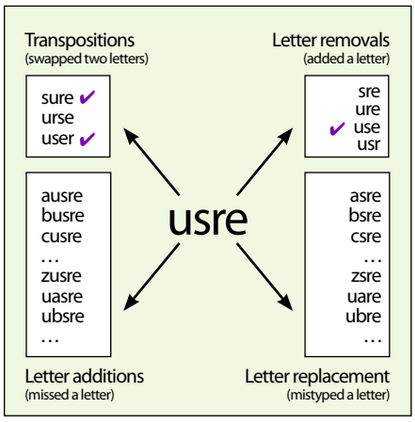
The Levenshtein method

The Russian scientist and mathematician Vladimir Levenshtein devised the Levenshtein distance algorithm in 1965 as part of his investigations into information theory and error-correcting codes. The Levenshtein distance is the minimum number of edits needed to get from one word to another, where an edit is defined as the insertion of a letter, the deletion of a letter or the substitution of one letter for another.

For small strings, calculating the Levenshtein distance is a quick operation. However, calculating the edit distance between a misspelled word and all the words in a large word list could take a significant amount of time, especially as after the calculation the results have to be sorted into distance order.

In reality, we'd want to be much more choosy about what our spellchecker suggests as a spelling correction. Even though the Levenshtein distance from 'begind' to 'end' is only three (delete three letters, essentially), there's no way that any self-respecting writer would accept 'end' as a spelling correction for what was clearly supposed to be 'begin'. Clearly a more efficient algorithm was needed.

The Levenshtein distance is still an important concept, however. In his paper on spelling corrections in 1964, F J Damerau noticed that four errors caused 80 to 90 per cent of misspellings: leaving out a letter, adding a letter, using a wrong letter or transposing two adjacent letters. These errors all come to a Levenshtein distance of one. He created a variant algorithm based on this research called the Damerau-Levenshtein distance. ■



▲ **Figure 2: The process behind generating alternative spellings for a mistyped word.**

► a function that takes a string and returns an array of the words within it. Another option is to use the regular expressions package that comes with your run-time (especially if you're using one of the interpreted languages such as JavaScript, Perl, PHP or Python) to split up the words. These are relatively easy solutions, but we're going to take a look at the issue from the first principles of spellchecking, and instead use a state machine.

To create a state machine, we need first to determine the states. After that, we can define the transitions between the states, and what happens within those transitions. For our 'separation of text into words' state machine, we will be using two states that we can call 'InWord' and 'NotInWord'. Let's start off by looking at the NotInWord state.

The easiest transition to think about is the transition from the InWord state to the NotInWord state. I would categorise this as reading a white-space character (like a space, new line or tab) or as reading a punctuation character (like a comma, semicolon, colon, full stop or dash). This makes the transition from NotInWord to InWord much simpler: it's any character that isn't white space or punctuation. While we're in the InWord state we collect characters, appending them to a string (which we can call Word, imaginatively enough). In the transition to NotInWord, we release this Word value. Figure 1 shows this very simple state machine.

It's worth noting that the reason we define the transitions in this way is that it's more culture-neutral than doing things the other way around, where we define the characters that can appear in a word. The problem with this is that there are many nonstandard characters, including accented characters such as the 'é' in 'café' or the 'ï' in 'naïve'. Because the number of characters that can be defined as white space or punctuation is much smaller than the number of possible characters constituting a word, it's easier to define the transitions using the former type.

Compound words

Compared to English, German is a more complex language to spellcheck because you can create compound words (words comprising two or more other words joined together). These appear in English as well (an apposite example here is 'spellchecker'), but, in general, compound words in English are kept separate using a space or a hyphen. In German, compound words may be created at will (although there are grammar rules that govern this) and as a result they might not appear in any word list. Other languages that allow this are Swedish and Finnish. ■

Dealing with possessives

Natural text will still produce problems for the simple spellchecking algorithm outlined in this article. For example, the algorithm will reject the phrase 'the officer's car' since it will assume that the apostrophe is punctuation and then flag the possessive 's' as a spelling error. A workaround for this is to assume that all single letter words are spelled correctly. This is what Microsoft Word does by default. ■

Finding misspellings

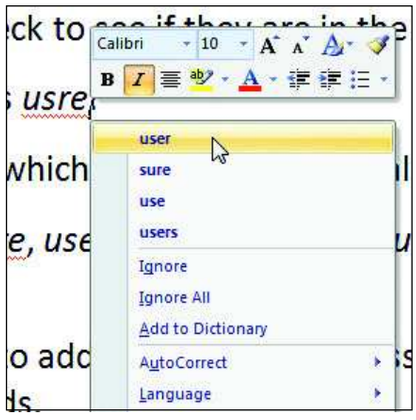
Now that we can receive a series of words one at a time from the text, we can think about how to spellcheck them. The first thing is to assume that any words that have digits in them are spelt correctly. So words like '2009', '1st', and 'Q1' are always assumed to be correct spellings. In normal text, I would say that this is generally the best assumption to make. The alternative – flagging all words with digits in them – is likely to produce a lot of false negatives and therefore annoy the writer. A possible middle ground is to accept all words that consist of digits only and reject all others.

So once we have a candidate, how do we check its spelling? By far the simplest algorithm that we can use is a hash table or dictionary. The first step is to locate a complete word list. There are several to be had on the Internet; just Google 'word list' to find one. Write a routine to read this list (it'll usually be in a one-word-per-line format) and add each word to a hash table.

Now the process becomes much faster: for each word identified in the text, check to see if it's present in the hash table. The reason for using a hash table here is that this checking process is linear in time (it doesn't depend on the number of entries) compared to a simple array (time is proportional to the number of entries) or to a sorted array or binary search tree (time is proportional to the logarithm of the number of entries).

Suggesting corrections

There are two main reasons for a misspelled word. The first is physical: the writer knows exactly how to spell the word, but in the



▲ **Figure 3: The list of suggestions provided by Word 2007 to correct 'usre' matches ours.**

> The process of checking spellings is not an exact science, but there's no denying it's made our lives a little easier

Letter	Code
b, f, p, v	1
c, g, j, k, q, s, x, z	2
d, t	3
l	4
m, n	5
r	6

▲ **Table 1 : Converting letters to Soundex codes.**

process of getting his thoughts down as text, he mistypes it. He may hit a key that's next to the key he wanted ('thr' instead of 'the'), transpose a couple of letters ('teh'), miss a key completely ('te') or inadvertently manage to type an extra adjacent letter ('thge').

The second reason for a misspelling is that the writer has guessed how to spell a word. Here the misspelled word is likely to be phonetically similar to the actual word. An example is 'definately' instead of 'definitely'.

Because there are two distinct reasons for misspellings, we have to use two different algorithms for identifying possible corrections. The algorithm used to correct words that have been mistyped is based on the Levenshtein method, which calculates the distance between the misspelled word and all the words in the word list (see the box on page 122). However, the original Levenshtein method is too lengthy for modern spellchecking. A far better algorithm for suggesting corrections is the Damerau algorithm, which generates variants of the mistyped word according to the errors listed below and checks them against the word list.

Let's see this method in action (shown in Figure 2). Suppose our misspelled word is 'usre':

- Transpose two of the letters. This produces 'sure', 'urse' and 'user', of which only 'urse' isn't a valid word.
- Omit a letter in case one of them has been inadvertently added: this produces 'sre', 'ure', 'use' and 'usr', of which only 'use' is a valid word.
- Add a missed letter back: there are five possible positions to add a letter and 26 possible letters to use in each, giving a total of 130 words. None of them are valid words.
- Replace a mistyped letter: there are four letters in the word and 25 possible replacements for each (100 possible words).

Again, in this example none are valid words. So, for the cost of generating 237 words and checking each in the word hash table, we have three suggestions for correcting the original word: 'sure', 'user' and 'use'. As Figure 3 shows, these are the first three suggestions that Word 2007 produces for 'usre', suggesting that our spellchecker is working on the right lines. This is vastly faster than calculating the Levenshtein distance for every word in the list we're using, which contains 170,000 words.

Now that we've analysed how to suggest spelling corrections for a mistyped word, let's have a look at how to generate suggestions for a genuinely misspelled word (where the writer tries a phonetic approximation). Our previous example ('definately') would already have been corrected by the Damerau algorithm, so let's contrive a more difficult example: 'enonesiate' for 'enunciate'. In this case, it's time to try the Soundex method (see the box below for details).

To use the Soundex method to suggest corrections, you would need to create another hash table that's populated with the Soundex codes for all of the words in the word list. Obviously there will be duplicates generated by the Soundex algorithm for all these words, so you would have to save a list of 'soundalikes' for each Soundex key in the hash table rather than the single original word.

Once you have created this hash table, suggesting spelling corrections is merely a case of calculating the Soundex value for the misspelled word, keying it into the Soundex hash table and retrieving the list of soundalikes given for the misspelling. In our example, 'enonesiate' would get converted to a value of E552. Since E552 is also the Soundex for 'enunciate', it will be provided as a suggestion for the correction.

As you can see, the process of checking spellings and suggesting corrections is not an exact science, but there's no denying that it has made our lives a little easier and our publications a little less unpredictable. ■

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pcplus.co.uk

The Soundex method

In 1918, Robert Russell and Margaret Odell devised and patented an algorithm for approximating and encoding the phonetic value of a word. They called it Soundex, and it's implemented like this:

- Keep the first letter of the word and discard all instances of 'a', 'e', 'h', 'i', 'o', 'u', 'w' and 'y' in the remainder of the word.
- Apart from the first letter, convert the letters to a digit using the table shown in Table 1.
- If two or more letters with the same code were adjacent in the original word (or would have been except for intervening 'h's or 'w's), drop the second and later codes (so the 'sack' part of 'knapsack' would be encoded as 22: the first digit replacing the 's', and the second for the 'ck').
- Convert to the form X999, where X is the initial letter and the nines indicate the codes. Pad with zeros or truncate if needs be.

The Soundex algorithm is a good approximation for encoding a word phonetically, but it has a big problem when used for spelling corrections: it requires that the initial letter of the phonetic misspelled guess be correct. (It would not notice that 'cereal' and 'serial' were the same phonetically, for example.) Other problems with the algorithm are that it generates fixed-size codes and that it does not distinguish between different-sounding phonemes.

In 1990, Lawrence Phillips devised another phonetic algorithm that corrects many of the issues with Soundex. It's called Metaphone, and it uses a much larger set of rules to create a phonetic encoding of a word. While more accurate, it is also much harder to implement.

Eight years later, Phillips produced an improved version which he named the Double Metaphone algorithm. This version produces two codes for every word. ■

Using n-grams

Another way of testing whether a word is spelled correctly is to preprocess the word list to produce a table of n-grams. An n-gram is a combination of *n* letters, usually four. We create a large bitmap (26^4 bits, so about 56KB) and set a given bit if a particular four-gram appears in any word in our word list. (The first bit, then, would correspond to 'aaaa' and the last to 'zzzz'.) To see if a word is spelled correctly, every four-gram in the candidate word formed by four consecutive letters has to have its bit set in the bitmap. ■