Project Overview:
The basic process for spell checking a document is very simple. Beforehand, a dictionary of the acceptable words must be created. The dictionary is typically stored in one or more text files or in a data structure such as a tree or hash table. The spell checking software takes the source document and steps through each word, searching through the dictionary to determine if there is a match and thus, the validity of the word. The first spell checking algorithms would stop here, simply pointing out spelling errors for the user to correct. While this approach was fine for correcting typographical errors, if the user knew what word they wanted to type but did not know the correct spelling, this approach was not particularly helpful. People began to improve spell checking software to make the best suggestions as to how to spell the desired word to allow the user to fix their error. This was a vast improvement and, for most cases, was sufficient; modern spell checkers attempt to address the cases of using correctly-spelled words in the wrong context ("there/their", "affect/effect") or homonyms ("tale/tail", "heir/air"). Since there are no spelling mistakes in these cases, the identification and correcting of these errors is much more difficult. This paper will provide an overview of some common context-insensitive spell checking algorithms and outline the approach used by the author to create a simple spell checking program using C.

Constructing the Dictionary
The underlying backbone of any spell checking algorithm is the dictionary. The dictionary must be comprehensive, yet still allow for quick searching and retrieval. A common word list is the Spell Checking Oriented Word List (SCOWL); the list offers various sizes of list that reflect the complexity and frequency of use of the words found in the list. The largest list has over 200,000 English words and proper nouns so it is apparent that some thought needs to be given to forming the dictionary for effective searching. The overhead from constructing the dictionary is not insignificant when pulling from a large word list and needs to be addressed.

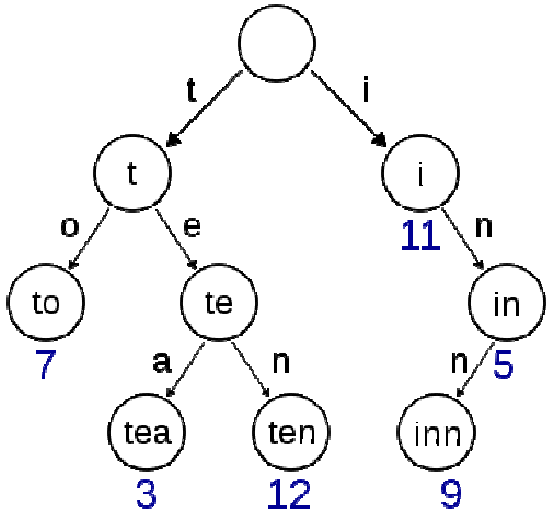
Approach 1: Alphabetical text files (linear search)
The easiest approach to constructing the dictionary would be to just store all the words in alphabetical order in one or more text files. However, this approach certainly leaves much to be desired in terms of efficiency. If we assume that the word list is split into 26 files (one for each starting letter in the words) and we also assume that the target source word has the correct first letter, without using any tricks or shortcuts, there are several thousand comparisons that need to be made to determine whether or not the source word is valid. The time complexity is *O(dictionary file size)*, that is, regardless of how long or short the source word is, the speed is based on the number of words in the dictionary file. This way is very similar to stored the data in a sorted list structure; searching though a list requires *O(n)* time in the worst-case.

You can play some games that reduce the number of comparisons that need to be made. For example, you could separate the files based on "triphones", which are three letter strings. With this approach, you would have files that contained works that start with "aaa", "aab", "aac", …etc which would reduce the size of n. Another space saving method would be to strip all the words of common suffixes or prefixes. This would

reduce similar words such as “play”, “plays”, “replay”, “played”,…etc but could potentially miss invalid words (“unplay” for example). Regardless of how much we are able to shrink the dictionary, we are still limited to *O(n)* searching and we now need more overhead to process the list. Since we are dealing with quite a large number of words, *O(n)* performance is really not practical for use in a spell-checking application.

Approach 2: Trie – prefix tree

Based on searching performance, the next best data structure for searching is a tree. A special type of tree called a trie, also known as a prefix tree, is an ordered data structure that stores a set of data that uses strings for keys. The nodes are arranged such that all its descendants have a common prefix of the string represented by that node. The advantage that a trie has over a regular binary search tree is that looking up a string of length *m* can be done in *O(m)* compared to *O(log n)*, where n is the number of elements for a BST. This is because a trie lookup is based on the depth of the tree, though in the worst case, *log(n)* will approach *m*. Tries also have simple operations for inserting or deleting entries (since it is just a special type of tree), which could be useful for more advancing spell checking features such as adding a new word to the dictionary on the fly (a common feature on most modern spell checkers).



Example of a trie data structure

Approach 3: Hash table

A hash table could provide another viable solution for indexing and searching through a dictionary. As with all hash tables, assuming a good hashing function could be determined, performance for searching can reach constant, *O(1)* time complexity. Though a perfect hashing collision scheme would be ideal, the use of separate chaining would probably prove to be the most realistic and easy to implement. Since the performance degradation for separate chaining is more gradual than other schemes (namely open addressing), it would be suitable for use in a large data set such as a spell check dictionary. Assuming the load factor is kept under control, the possible performance decline should not be an issue. Hashing also has the capability for easy persistent storage in a database, which could be useful for a desktop spell checking application.

Correction Suggestions

Once it has been determined that a word is not in the dictionary and thus a misspelling, the battle is only half over. A good spell checking application will be able to intelligently make a suggestion of how the word should be spelled so that the user will be able to quickly fix their mistakes. The spell checker should be able to identify, for example, that "pncake" should really be "pancake" and "teh" should be "the". The follow are two common approaches used by spell checkers:

Approach 1: Levenshtein edit distance

A common metric used in computer science that was developed in 1965 by Vladimir Levenshtein, the Levenshtein edit distance is the minimum number of transformation operations needed to transform one string into another string. The transformation operations are as follow:

1. Insertion of one character anywhere in the string ("aple" to "apple")
2. Deletion of one character anywhere in the string ("applze" to "apple")
3. Substitution of one character anywhere in the string ("apzle" to "apple")
4. Swap any two characters in the string ("aplpe" to apple")

So in the previous "pancake" example, the Levenshtein edit distance would be just 1, since "pncake" can be transformed to the correct word "pancake" in just one operation (inserting "a"). For any given string that is not found in the dictionary, the algorithm will perform transformations and check if any of the new words is valid. Then, the new words can be ranked based on their edit distance and suggested to the user. For simplicity, assume "pncake" was transformed into various "new" strings such as "cake", "poundcake", "pancake", etc; since "pancake" took the fewest edits (1) compared to "cake" (2) and "poundcake" (3), it should be ranked highest in the list of suggested corrections.

This approach is quite useful when the spelling error is due to a typo, instead of a fundamental mistake in the words spelling. Researchers have found that between 80 to 95% of the errors in large texts were due to typing errors and could be correcting with just one transformation operation.

The time complexity of performing the Levenshtein transformation depends heavily on how many edits are to be made. For simple, single-edit transformations, the complexity is not too bad. For example, operation 1 (insertion), there are 26 possible characters that could be added at each of the n locations in the string; the transformation would take $O(26*n) => O(n)$ time and generate $26*n$ possible new words ($O(n)$ memory complexity). However, if we wish to consider words that can be generated in two-edit transformations, then each of the $26*n$ new words each needs to be transformed again. The time and memory complexities would both grow to $O(n^2)$. This pattern continues such that the complexity is bound by $O(n^k)$, where k is the maximum edit distance that we wish to consider, which can lead to very bad performance if we allow k to grow too large.

Approach 2: Phonetic algorithms

While the Levenshtein distance approach seems to be great for correcting typographical errors, it often has trouble when the user genuinely is spelling a word incorrectly. While the first approach discussed may be able to correct "poorfackt" into "perfect", the number of edits will be larger (remember that most typographical errors only require one edit) and other, incorrect suggestions may be found with less edits. But if you read both strings aloud, you could tell that the first string was intended to be the second. There are two popular algorithms that deal with these phonetic similarities: Soundex and Metaphone.

Approach 2.1: Soundex

Soundex, originally developed for indexing American surnames based on sound, is the most widely used "phonetic algorithm". The algorithm converts the given string into a Soundex code, which is the first letter of the string followed by three digits that represent the rest of the string. The algorithm to compute the Soundex code is as follows:

1. Replace consonants with digits as follows (but do not change the first letter):
   - b, f, p, v => 1
   - c, g, j, k, q, s, x, z => 2
   - d, t => 3
   - l => 4
   - m, n => 5
   - r => 6
2. Collapse adjacent identical digits into a single digit of that value.
3. Remove all non-digits after the first letter.
4. Return the starting letter and the first three remaining digits. If needed, append zeroes to make it a letter and three digits.

The Soundex codes for "poorfackt" and "perfect" are both P-612 so the algorithm would return an exact match. Since performing the Soundex encoding for a word is a one-time operation, the time complexity is *O(n)*, where *n* is the number of characters in the string. Additionally, the memory complexity is *O(1)*, since the Soundex value is four characters regardless of string size.

Approach 2.2: Metaphone

Metaphone, developed in order to address some of the short-comings of Soundex, is more apt at comparing any word instead of just names because it uses a larger set of pronunciation rules. A major difference is that the Metaphone algorithm outputs a variable length coded value, compared to Soundex's first letter and three digit code. The transformation process is much more advanced the Soundex algorithm and is shown below:

Exceptions:

Initial kn-, gn-, pn, ac- or wr-	=> drop first letter
Initial x-	=> change to "s"
Initial wh-	=> change to "w"

Transformations:

Vowels are kept only when they are the first letter.

B =>	B unless at the end of a word after "m" as in "dumb"
C =>	X (sh) if -cia- or -ch-
	S if -ci-, -ce- or -cy-
	K otherwise, including -sch-
D =>	J if in -dge-, -dgy- or -dgi-
	T otherwise
F =>	F
G =>	silent if in -gh- and not at end or before a vowel
	in -gn- or -gned- (also see dge etc. above)
	J if before i or e or y if not double gg
	K otherwise
H =>	silent if after vowel and no vowel follows
	H otherwise
J =>	J
K =>	silent if after "c"
	K otherwise
L =>	L
M =>	M
N =>	N
P =>	F if before "h"
	P otherwise
Q =>	K
R =>	R
S =>	X (sh) if before "h" or in -sio- or -sia-
	S otherwise
T =>	X (sh) if -tia- or -tio-
	0 (th) if before "h"
	silent if in -tch-
	T otherwise
V =>	F
W =>	silent if not followed by a vowel
	W if followed by a vowel
X =>	KS
Y =>	silent if not followed by a vowel
	Y if followed by a vowel
Z =>	S

The Metaphone code for "poorfackt" and "perfect" are both PRFKT so this algorithm would also return an exact match. Similar to Soundex, the time complexity is *O(n)*, however, the memory complexity can vary depending on the string length. In the worst case, the size of the Metaphone value will approach the string size, so the memory complexity is bound by *O(n)*. However, the extra memory cost is a small trade-off for the improved accuracy of the Metaphone value compared to Soundex.

Code Implementation

After analyzing some of the more popular approaches to creating a spell checking program, the author created a small, "proof-of-concept" spell checking program written in C. The program is context-insensitive and features phonetic-based suggestions. The author chose to implement "Approach 1" (file based dictionary with linear search) for dictionary construction and "Approach 2.1" (Soundex phonetics) for correction suggestions. The dictionary used contains 58,112 English words (a "simple" dictionary by comparison). Along with using the Soundex algorithm to find phonetic matches, a "sameness" parameter was devised to filter out results that had a similar phonetic, but few similar characters. If a word matches the Soundex value and is above a "sameness threshold", it is determined to be a valid suggestion and displayed to the user; this seemed to improve the suggestions for words of medium to long length.

The finished code is under 200 lines and provides fairly good suggestions relative to its complexity. The biggest improvement would be to reduce the number of suggestions while increasing the accuracy; for some words, the suggestions are concise and highly accurate, but for others there is a large set of suggestions (though nearly all test cases returned suggestion sets that include the "intended" word). Also, the program does not order the suggestions in any way (they are displayed in alphabetically order), so this could be added to improve functionality by placing "better" suggestions higher up in the list.

References

Baeza-Yates, R. A. and Navarro, G. 1996. "A Faster Algorithm for Approximate String Matching." In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching* (June 10 - 12, 1996). D. S. Hirschberg and E. W. Myers, Eds. Lecture Notes In Computer Science, vol. 1075. Springer-Verlag, London, 1-23.

Gonzalo Navarro. "A guided tour to approximate string matching". *ACM Computing Surveys,* 33(1):31–88, 2001.

Eric Sven Ristad, Peter N. Yianilos, "Learning String-Edit Distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, pp. 522-532, May, 1998.

http://www.dcs.bbk.ac.uk/~roger/spellchecking.html
http://www.norvig.com/spell-correct.html
http://en.wikipedia.org/wiki/Soundex
http://en.wikipedia.org/wiki/Fuzzy_string_searching
http://en.wikipedia.org/wiki/Spellchecker
http://en.wikipedia.org/wiki/Levenshtein_distance
http://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings
http://www.ibm.com/developerworks/java/library/j-jazzy/
http://goanna.cs.rmit.edu.au/~jz/fulltext/sigir96.pdf