Spelling correction algorithm: using hash table, separate chaining with linked list, and move-to-front technique

Algorithm Report

By Leonardo Linardi (linkedin.com/in/leonardolinardi)

Section 1 - Algorithm Overview

Let us define,

- n as the total number of words in the dictionary
- m as the total number of words in the document

III. Spell Checking

At this function, an approach using a hash table is implemented to the dictionary words with the details as follows,

- Hash function : XOR Hash (taken from Workshop 9)
- Hash table size : n
- Collision management : Separate chaining with linked list (with move-to-front technique)
- Load factor : 1
- Probability of collision : 0.26424 (1 - P(no collision), approximated using Poisson distribution)

and every document word is hashed with the same function, to check whether the word exists in the hash table (such that, it's correctly spelled)

Upon initializing the hash table, that is, hashing the dictionary words and storing them inside the hash table, requires $O(n)$ time at best and average case, when all or most of the words are hashed into different buckets, and $O(n^2)$ time at worst case, when all the words are hashed into 1 single bucket.

Searching the hash table, that is, checking whether a document word is inside the hash table, only requires $O(1)$ time at best and average case, when all or most of the dictionary words are hashed into different buckets, and $O(n)$ time at worst case, when all the words are hashed into 1 single bucket.

Searching the hash table is relatively fast, however, this approach requires an extra of O(n) space to store the hash table.

Overall, this approach would take O(m) at best and average case, and O(nm) at worst case.

IV. Spelling Correction

At this function, the same hash table is implemented to the dictionary words with the details and time complexity mentioned above. However, different approaches are made upon checking whether the word exists in the dictionary (that is, if it's a correctly spelled word) and finding the corrected word (which is, a word with Levenshtein edit distance of 1, 2 or 3 from the dictionary words). The approaches for each case are,

1. Correctly spelled word : Hash table
2. 1 edit distance away : Hash table
3. 2 edit distance away : Hash table
4. 3 edit distance away : Direct lookup to the dictionary

Upon finding a corrected word with 3 edit distance from the dictionary words, a direct lookup (directly comparing the wrong word to the dictionary with edit distance 3) is used instead. This is preferred as there would be an enormous number of combinations of 3 edit distance words from the wrong word to be compared with the hash table, which would be inefficient compared to a simple direct lookup (even though the dictionary is also huge).

A direct lookup would cost O(n) time at any case and comparing edit distance between two words (say w_1 and w_2) would take $O(w_1 . w_2)$, using a 2D matrix of size $w_1 \times w_2$. Overall, a direct lookup would take $O(n . w_1 . w_2)$.

Section 2 - Alternative Approaches

III. Spell Checking

Some of the alternative approaches are,

1. A simple direct lookup and comparing all the document words to all the dictionary words
2. Array to store collisions on separate chaining
3. Not applying move-to-front (MTF)

Firstly, performing a direct lookup would be ineffective as it would cost O(nm) at any case, which could be worst if the dictionary and document size gets larger and larger.

Secondly, an array for separate chaining would be faster when searching an item (compared to a linked list) due to the effect of CPU cache, but the MTF technique would be harder to implement, as every insertion or search would require that item searched being moved to the front of the array, and the rest of the items shifted 1 slot away (which takes O(q) time, with q as the number of words in that bucket or the size of the array). Deletion of an item would behave similarly.

Lastly, implementing the move-to-front technique would increase the speed of a search in the hash table, especially when there are lots of collisions (that is, lots of words stored inside the same bucket). As MTF could adapt to skew access patterns, with the most frequent words being searched located in front of the list, reducing lookup time.

Overall, the method implemented, explained on Section 1, is one of the best method.

IV. Spelling Correction

Some of the alternative approaches are,

1. The same alternatives on the hash function itself, as mentioned above.
2. Using a hash table for searching 3 edit distance away words but is inefficient, as explained on Section 1.
3. Create a threshold on searching 2 edit distance away words

Explaining the last point, one of the alternative approach is to find the average length (`avglen`) of words in the dictionary and set it as a threshold upon searching 2 edit distance away words. If a word from the document, is below (shorter in length) the threshold, then a hash table is performed. But if the word is above the threshold, a simple direct lookup is performed. Upon implementing this approach, the following result is achieved,

Without threshold	**With threshold, of:**		
	`avglen`	**1.5 x** `avglen`	**2 x** `avglen`
37.2 s	70 s	37.5 s	37.45 s

Table 2.1 Comparisons of program time with or without threshold

The program was tested using `words-5M.txt` as the dictionary and `jabberwocky.txt` as the document. Time is measured in seconds (s) using the function `time` provided by the command shell (bash).

We could see that providing a threshold wouldn't make much difference in terms of speed. So, the approach is implemented without threshold and in overall, is one of the best approach to find spelling corrections.