1000x Faster Spelling Correction algorithm

Posted on **June 7, 2012**

Share on

Update: We released a C# implementation as Open Source.
Update2: We are 100,000 times faster for edit distance=3.
Update3: Spelling correction is now also part of FAROO search.
Update4: The source code is now also on GitHub.
Update5: Improved implementation now 1,000,000 times faster for edit distance=3.

Recently I answered a question on Quora about spelling correction for search engines. When I described our algorithm I was pointed to Peter Norvig's page where he outlined his approach.

Both algorithms are based on Edit distance (Damerau-Levenshtein distance).
Both try to find the dictionary entries with smallest edit distance from the query term.
If the edit distance is 0 the term is spelled correctly, if the edit distance is <=2 the dictionary term is used as spelling suggestion. But our way to search the dictionary is different, resulting in a significant performance gain

and language independence. Three ways to search for minimum edit distance in a dictionary: **1. Naive approach**
The obvious way of doing this is to compute the edit distance from the query term to each dictionary term, before selecting the string(s) of minimum edit distance as spelling suggestion. This exhaustive search is inordinately expensive.
Source: Christopher D. Manning, Prabhakar Raghavan & Hinrich Schütze: Introduction to Information Retrieval.

The performance can be significantly improved by terminating the edit distance calculation as soon as a treshold of 2 or 3 has been reached.

2. Peter Norvig
Generate all possible terms with an edit distance <=2 *(deletes + transposes + replaces + inserts)* from the query term and search them in the dictionary.
For a word of length n, an alphabet size a, an edit distance d=1, there will be n deletions, n-1 transpositions, a*n alterations, and a*(n+1) insertions, for a total of 2n+2an+a-1 terms at search time.
Source: Peter Norvig: How to Write a Spelling Corrector.

This is much better than the naive approach, but still expensive at search time (114,324 terms for n=9, a=36, d=2) and language dependent (because the alphabet is used to generate the terms, which is different in many languages and huge in Chinese: **a=70,000** Unicode Han characters)

3. Symmetric Delete Spelling Correction (FAROO)
Generate terms with an edit distance <=2 *(deletes only)* from each dictionary term and add them together with the original term to the dictionary. This has to be done only once during a pre-calculation step.
Generate terms with an edit distance <=2 *(deletes only)* from the input term and search them in the dictionary.
For a word of length n, an alphabet size of a, an edit distance of 1, there will be just n deletions, for a total of n terms at search time.

This is **three orders of magnitude less expensive** (36 terms for n=9 and d=2) and **language independent** (the alphabet is not required to generate deletes).
The cost of this approach is the pre-calculation time and storage space of x deletes for every original dictionary entry, which is acceptable in most cases.

The number x of deletes for a single dictionary entry depends on the

maximum edit distance: x=n for edit distance=1, x=n*(n-1)/2 for edit distance=2, x=n!/d!/(n-d)! for edit distance=d (combinatorics: k out of n combinations without repetitions, and k=n-d),
E.g. for a maximum edit distance of 2 and an average word length of 5 and 100,000 dictionary entries we need to additionally store 1,500,000 deletes.

Remark 1: During the precalculation, different words in the dictionary might lead to same delete term: delete(sun,1)==delete(sin,1)==sn.
While we generate only one new dictionary entry (sn), inside we need to store both original terms as spelling correction suggestion (sun,sin)

Remark 2: There are four different comparison pair types:

1. dictionary entry==input entry,
2. delete(dictionary entry,p1)==input entry
3. dictionary entry==delete(input entry,p2)
4. delete(dictionary entry,p1)==delete(input entry,p2)

The last comparison type is required for replaces and transposes only. But we need to check whether the suggested dictionary term is really a replace or an adjacent transpose of the input term to prevent false positives of higher edit distance (bank==bnak and bank==bink, but bank!=kanb and bank!=xban and bank!=baxn).

Remark 3: Instead of a dedicated spelling dictionary we are using the search engine index itself. This has several benefits:

1. It is dynamically updated. Every newly indexed word, whose frequency is over a certain threshold, is automatically used for spelling correction as well.
2. As we need to search the index anyway the spelling correction comes at almost no extra cost.
3. When indexing misspelled terms (i.e. not marked as a correct in the index) we do a spelling correction on the fly and index the page for the correct term as well.

Remark 4: We have implemented query suggestions/completion in a similar fashion. This is a good way to prevent spelling errors in the first place. Every newly indexed word, whose frequency is over a certain threshold, is stored as a suggestion to all of its prefixes (they are created in the index if they do not yet exist). As we anyway provide an instant search feature the lookup for suggestions comes also at almost no extra

cost. Multiple terms are sorted by the number of results stored in the index.

Reasoning

In our algorithm we are exploiting the fact that the edit distance between two terms is symmetrical:

1. We can generate all terms with an edit distance <2 from the query term (trying to reverse the query term error) and checking them against all dictionary terms,
2. We can generate all terms with an edit distance <2 from each dictionary term (trying to create the query term error) and check the query term against them.
3. We can combine both and meet in the middle, by transforming the correct dictionary terms to erroneous strings, and transforming the erroneous input term to the correct strings.
   Because adding a char on the dictionary is equivalent to removing a char from the input string and vice versa, we can on both sides restrict our transformation to deletes only.

We are using variant 3, because the delete-only-transformation is language independent and three orders of magnitude less expensive.

Where does the speed come from?

- **Pre-calculation**, i.e. the generation of possible spelling error variants (deletes only) and storing them at index time is the first precondition.
- A fast index access at search time by **using a hash table** with an average search time complexity of O(1) is the second precondition.
- But **only our Symmetric Delete Spelling Correction** on top of this allows to bring this O(1) speed to spell checking, because it allows a tremendous reduction of the number of spelling error candidates to be pre-calculated (generated and indexed).
- **Applying pre-calculation to Norvig's approach would not be feasible** because pre-calculating all possible delete + transpose + replace + insert candidates of all terms would result in a huge time and space consumption.

Computational Complexity

Our algorithm is constant time (O(1) time), i.e. independent of the dictionary size (but depending on the average term length and maximum

edit distance), because our index is based on a Hash Table which has an average search time complexity of O(1).

Comparison to other approaches

BK-Trees have a search time of O(log dictionary_size), whereas our algorithm is constant time (O(1) time), i.e. independent of the dictionary size.

Tries have a **comparable search performance** to our approach. But a Trie is a prefix tree, which requires a common prefix. This makes it suitable for autocomplete or search suggestions, but **not applicable for spell checking**. If your typing error is e.g. in the first letter, than you have no common prefix, hence the Trie will not work for spelling correction.

Application

Possible application fields of our algorithm are those of fast approximate dictionary string matching: spell checkers for word processors and search engines, correction systems for optical character recognition, natural language translation based on translation memory, record linkage, de-duplication, matching DNA sequences, fuzzy string searching and fraud detection.

———

BTW, by using a similar principle our web search is three orders of magnitude more efficient as well. While Google touches 1000 servers for every query, we need to query just one (server/peer).

That's not because of DHT! Vice versa, because even for a complex query in a web scale index only one of the servers needs to be queried, it enables the use of DHT for web search.

Our algorithm improves the efficiency of central servers in a data center to the same extent.

This entry was posted in **FAROO**, **Open Source**, **Spelling correction** by**Wolf**. Bookmark the **permalink**.