# CS575: Final Project Report

**Project Title: Spell Checker using Levenshtein Distance and BKTrees**

**Team Member(s): Devarapalli Vinay**

**Mense Himani**

**Nikam Hitesh**

## I. PROBLEM

Search engines generally find the data based on keywords entered; therefore, there are times when the user can't find the data they need because there is an error while entering a keyword. Thus, we need the ability to detect the entered word is correct. That's where our spell check using levenshtein distance comes into picture. A spell checker is a technique which identifies the incorrect or misspelled words and replaces them with the nearest best possible combination of correct words. The purpose of this algorithm is to retrieve the nearest match from your dictionary if a particular word is not present. The system suggests such words with the smallest edit distance in accordance with the existing data. Approximate string matching (ASM) is a well-known computational problem with important applications in spell checking, plagiarism detection and many more. Levenshtein distance is one of the approximate string-matching algorithms used in string search based on the estimation approach. This algorithm is a weighted approach to a cost of 1 to every edit operation (Insertion, Replacement and deletion). This distance is known as Levenshtein distance (aka edit distance), a special case of edit distance where unit costs apply. By using this algorithm, the search performed turns out to be efficient with less lookup time.

## II. ALGORITHMS

(1) SymSpell: It finds all strings within a maximum edit distance from a huge list of strings in a very short time. SymSpell derives its speed from the symmetric delete spelling correction algorithm and keeps its memory requirement in check by prefix indexing. The symmetric delete spelling correction algorithm reduces the complexity of edit generation and dictionary lookup for a given Levenshtein distance. Opposite to other algorithms, only delete operations are required here. The other operations are transformed into deletes of the dictionary term. This works as below:

1. Generate terms with an edit distance (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.
2. Generate terms with an edit distance (deletes only) from the input term and search them in the dictionary.

Pre-calculate main dictionary and store it in hash table

1;

Create hash table with generated words

Code:

Search keyword in Hash table

If found {
Search term is correct, no spelling correction is required
}
else {
Perform IRD operations
If number records matched >1
Retrieve the original word and
Output the word with highest frequency
}

(2) Peter Norvig's approach: The function looks at every possible edit to the input- deletion, insertion, replace, transposition of any 2 adjacent characters and calculates the edit distance accordingly. For a word of length n, an alphabet size of a, an edit distance of 1, there will be just n deletions, n-1 transpose, a*n replace and a*(n+1) insertions for a total of n terms at search time.

(3) Faroo's approach: It is similar to SymSpell in terms of performing only deletes out of all other IRD operations. It works by storing tokens with all possible characters deletions up to distance 2. As your dictionary grows, the lookup and retrieval time get affected. It becomes difficult to retrieve all tokens thus we store only top n most frequent ones to save space and reduce correction effort. For better time complexity, we limit the maximum edit distance. This turns out to be better than Norvig's approach.

## III. Software Design and Implementation

Our spell checker was implemented using java 8 on eclipse IDE. The newly introduced concepts used in this project are Levenshtein distance and BKTrees.

### A. *Software Design*

User Interface: We input a word and select an algorithm through which we want to perform spell check. Once the algorithm is selected, we perform spell check and if word exists in dictionary it is retrieved along with the retrieval time as a pop-up.

### B. *Implementation and Tools Used*

Levenshtein distance: Levenshtein distance between any two strings is defined as the minimum number of edits that we do to transform one string into another. This measure is used to determine the similarity of two strings. For instance, the levenshtein edit distance between "dog" and "cat" is 3 (replacing d by c, o by a, g by t). It calculates the minimum number of elementary operations executed. There are 3 kinds of major operations that can be performed, namely character insertion to add a certain character into a string, then character replacement to replace a specific character in a string and character deletion operation to remove a particular character in a string.

BKTrees: It is a data structure used to perform spell check based on edit distance concept. It is also used for approximate string matching. Let's say we have a dictionary of words and then we have some other words to be checked in the dictionary for spelling errors. We need to have collection of all words in the dictionary which are very close to the given word. For instance, if we are checking a word "ruk" we will have {"truck", "buck", "duck"}. Therefore, spelling mistakes can be corrected by deleting a character from the word or adding a new character in the word or by replacing the character in the word by some appropriate one. Therefore, we use the edit distance as a measure for correctness and matching of the misspelled word from the words in our dictionary.

The structure of BK tree is similar to other trees, consisting of nodes and edges. The nodes in the BK tree represent the individual words in our dictionary and there will be exactly the same number of nodes as the number of words in our dictionary. The edge will contain some integer weight that will tell us about the edit distance from one node to another. Let's say we have an edge from node u to node v having some edge-weight w, then w is the edit distance required to turn the string u to v. Every node in the BK tree will have exactly one child with same edit distance. In case, if we encounter some collision for edit-distance

while inserting, we will then propagate the insertion process down the children until we find an appropriate parent for the string node. The root node can be any word from our dictionary. That was about how we build a BK tree.

Now the question arises, how do we find the closest correct word for our misspelled word? For this, we firstly set a tolerance value. This tolerance value is simply the maximum edit distance from our misspelled word to the correct words in our dictionary. So, to find the eligible correct words within the tolerance limit, Naïve approach would be to iterate over all the words in the dictionary and collect the words which are within the tolerance limit. But this approach has O (n*m*n) time complexity (n is the number of words in the dictionary, m is the average size of correct word and n is the length of misspelled word) which times out for larger size of dictionary.

At this stage, BK trees come into action. Since we traverse from the root node to specific nodes that lie within the tolerance limit, we don't iterate over all children but only those that have edit distance in range [dist-TOL, dist+TOL], dist being the edit distance of current node from the misspelled word. This reduces our complexity by a large extent. It is quite evident that the time complexity majorly depends on the tolerance limit. We will consider tolerance limit to be 2. Now, roughly let's estimate the depth of BK tree to be log n, where n is the size of dictionary. At every level we are visiting 2 nodes in the tree and performing edit distance calculation. Thus, our time complexity will be O (L1*L2*log n), here L1 is the average length of word in our dictionary and L2 is the length of misspelled.

### C. *Performance Evaluation*

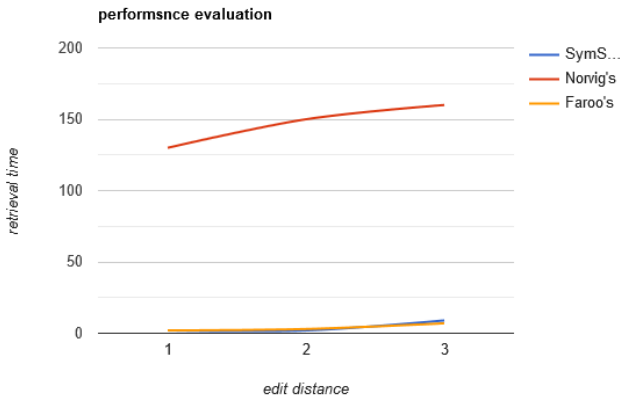
All algorithms strive for the same goals to achieve short lookup times: reducing the number of lookups and comparisons, possibly reducing further the number of full edit distance calculations and finally reducing the computational complexity of the edit distance calculation itself, while not compromising its accuracy.

SymSpell: The time complexity of SymSpell algorithm is constant time i.e. O (1) due to the factor that the dictionary is stored on a hash table which has an average search time complexity of O (1). Hence it is also independent of the dictionary size (but depending on the average term length).

SymSpell algorithm is used if speed is important. It is 5-6 orders of magnitude faster than Norvig's algorithm. SymSpell lookup time grows only moderate with dictionary size and maximum edit distance. The retrieval time in case of our algorithm implementation is 2ms for a search through 35,000 words. It outperforms all other algorithms but at a cost of higher memory consumption.

Norvig's Approach: This approach turns out to be expensive as a lot of computation takes place compared to SymSpell. Also, this computation grows exponentially with respect to length of input string.

Faroo's Approach: It has implementation same as SymSpell in terms of performing only deletes. The only difference between SymSpell and Faroo's is the retrieval time. Faroo's should turn out to be better than SymSpell with faster retrieval. But in our case, we have the same retrieval time of 2ms.




## ATTACHMENTS

- https://github.com/hnikam1/SpellingChecker

## REFERENCES

[1] Scientific Journal of Informatics, Vol. 5, No. 1, May 2018

[2]https://signal-to-noise.xyz/post/bk-tree/

[3]https://codereview.stackexchange.com/questions/48908/java-implementation-of-spell-checking-algorithm
[4] www.ijcst.com/vol8/2/15-liny-varghese.pdf
[5] https://medium.com/@wolfgarbe/1000x-faster-spelling-correction-algorithm-2012-8701fcd87a5f