

# Damerau-Levenshtein Algorithm and Bayes Theorem for Spell Checker Optimization

Iskandar Setiadi 13511073  
Program Studi Teknik Informatika  
Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia  
iskandarsetiadi@students.itb.ac.id

**Abstract**—Google, as the world largest search engine, has been well-known worldwide for its features. One of the main concern in giving a nice user experience is spell checker. There are two main concerns in spell checker application, that's it, speed and accuracy. Google's spell checker could perform quickly with a rather good accuracy in its correctness. Yet, an algorithm itself couldn't give a hundred-percent accuracy as users' expectation. However, there's always a trade-off between better accuracy and faster speed in spell checker. We need to create an algorithm with an acceptable rate of running time while maintaining an acceptable rate of accuracy. We will implement several analyses with Bayes theorem and probabilities in order to improve the accuracy of spell checker. Also, we will integrate this implementation with Damerau-Levenshtein algorithm.

**Index Terms**—Bayes theorem, Damerau-levenshtein algorithm, language processing, spell checker.

## I. INTRODUCTION

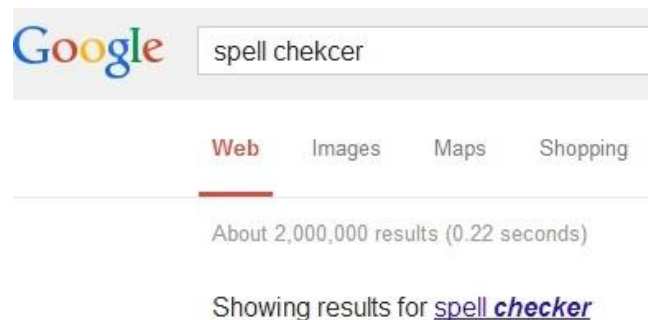
In this information era, spell checker application has a lot of functionalities such as an integrated application in word processor, email client, electronic dictionary, and search engine. A spell checker is defined as an application program which flags words in a document that may not be spelled correctly.

Spell checker is started to be developed in 1957. It's used to find records in database in spite of incorrect entries. Les Earnest, who headed the research at Stanford University, saw it necessary to include the first spell checker that accessed a list of 10.000 acceptable words. In 1971, Ralph Gorin created the first true spelling checker program written as an application program for general English text.

The first spelling checker for personal computers appeared for CP/M and TRS-80 computers in 1980, followed by IBM PC in 1981. On the personal computers, these spelling checkers were standalone programs, many of which could be run in TSR mode from within word-processing packages on personal computers. The first usage of spelling checkers as an integrated application was started by the mid-1980s. Developers of popular word-processing packages like WordStar and WordPerfect had incorporated spell checkers in their packages. However, this required increasing

sophistication in the morphology routines of the software in rather difficult languages.

Recently, spell checkers has moved beyond word processors. A web browser may have a spell checker which helps user in editing Wikitext, writing on many webmail sites, blogs, and social networking websites.



Picture 1.1 Google spell checker in search engine

As shown in picture 1.1, Google itself has implemented a spell checker feature in its search engine. Almost all applications which are related to word-processing have implemented this spell checker feature.

A basic spell checker carries out the following processes:

1. It scans all the text and parses the words contained in it
2. It compares each word with a known list of correctly spelled words. This list might contain another information to increase the accuracy of spell checker
3. The next step is a language-dependent algorithm for handling morphology (language specific). The different forms of the same word, such as plurals, verbal forms, contractions, and possessives need to be considered.

Normal algorithm such as brute force (comparing each letter one by one manually) will have relative slow speed in its running time. As we want to improve the accuracy of spelling checker, brute force algorithm is no longer feasible for being implemented. We will analyze several methods to increase the processing speed and accuracy of a spell checker.

## II. SOME THEORIES

### 2.1 Dynamic Programming

Dynamic programming is widely used in solving computational problems by breaking the solution down to several steps. The main characteristic of dynamic programming is solving greater problem from its sub-problems. Optimality principle states that if the total solution is optimal, then all of its sub-solutions up to the total solution are also optimal.

There are two general methods in dynamic programming, that's it, bottom-up and top-down DP. Dynamic programming usually stores its memorization in tables.

### 2.2 Levenshtein Distance

In computer science and information theory, Levenshtein distance algorithm is an algorithm which is used for string processing. This algorithm measures the difference between two string sequences. Levenshtein distance counts the minimum number of single-character edits (insertion, deletion, substitution) required to change one word into the other. Basically, this algorithm is widely used for sequence alignment between two different string patterns.

Let's take an example between two words "kitten" and "sitting". After running Levenshtein algorithm between both words, we'll find that the Levenshtein distance between both of them is 3.

- ➔ kitten → sitten (substitution of "s" for "k")
- ➔ sitten → sittin (substitution of "i" for "e")
- ➔ sittin → sitting (insertion of "g" at the end)

There is no way to change "kitten" to "sitting" in less than 3 single-character edits. The Levenshtein distance between two strings is always at least the difference of the sizes of the two strings and at most the length of the longer string.

Algorithm for Levenshtein distance is shown below:

```
function levenshteinDistance(input s : array[1..m] of char,
    input t : array[1..n] of char) → integer
{function to compute Levenshtein distance between two
strings using Levenshtein algorithm}
DECLARATION
i, j : integer
d : array [0..m][0..n] of integer
ALGORITHM
for i ← 1 to m do { source prefixes initialization }
    d[i][0] ← i
endfor
for j ← 1 to n do { target prefixes initialization }
    d[0][j] ← j
endfor
{ using Levenshtein Algorithm to check }
for i ← 1 to n do
    for j ← 1 to m do
        if (s[i] == t[j]) then
```

```
        d[i][j] ← d[i-1][j-1] {same character}
    else
        d[i][j] ← minimum
        (
            d[i-1][j] + 1, { deletion }
            d[i][j-1] + 1, { insertion }
            d[i-1][j-1] + 1 { substitution }
        )
    endif
endfor
end for
→ d[m][n] { return results }
```

Using the algorithm above, comparison between "kitten" and "sitting" can be tabularized into table below:

-	k	i	t	t	e	n	
s	0	1	2	3	4	5	6
i	1	1	2	3	4	5	6
t	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
i	4	4	3	2	1	2	3
n	5	5	4	3	2	2	3
g	6	6	5	4	3	3	2
	7	7	6	5	4	4	3

Table 2.1 DP Table for "kitten" and "sitting"

We can also use less space,  $O(\min(n,m))$  instead of  $O(m.n)$ , since it only requires one previous row to process the current row at any one time.

### 2.3 Damerau-Levenshtein Distance

Frederick J. Damerau has improved Levenshtein algorithm with an additional operation to check the distance between strings, that's it, a transposition of two adjacent characters. Damerau stated that this algorithm has corresponded to more than 80% of human misspellings. By taking four string operations (insertion, deletion, substitution, and transposition), this algorithm has also been used in biology to measure the variation between DNA.

Algorithm for Damerau-Levenshtein distance, which is almost the same with Levenshtein distance, is shown below:

```
function damerauLevenshteinDistance(input s :
array[1..m] of char, input t : array[1..n] of char) → integer
{function to compute Damerau-Levenshtein distance
between two strings using Damerau-Levenshtein
algorithm}
DECLARATION
i, j : integer
cost : integer
d : array [0..m][0..n] of integer
ALGORITHM
for i ← 1 to m do { source prefixes initialization }
    d[i][0] ← i
```

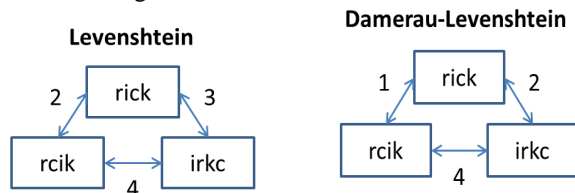
```

endfor
for j ← 1 to n do { target prefixes initialization }
  d[0][j] ← j
endifor
{ using Damerau-Levenshtein Algorithm to check }
for i ← 1 to n do
  for j ← 1 to m do
    if (s[i] == t[j]) then
      cost ← 0
    else
      cost ← 1
    endif
    d[i][j] ← minimum
    (
      d[i-1][j] + 1, { deletion }
      d[i][j-1] + 1, { insertion }
      d[i-1][j-1] + cost { substitution }
    )
    if (i > 1 and j > 1 and s[i] == t[j-1] and s[j-1] == t[i])
then
      d[i][j] ← minimum
      (
        d[i][j],
        d[i-2][j-2] + cost { transposition }
      )
    endif
  endfor
endfor
end for
→ d[m][n] { return results }

```

We're using different color to emphasize the difference between Damerau-Levenshtein and Levenshtein algorithm. This algorithm differs only in an additional condition for transposition case. Overall, this algorithm has the same time complexity with Levenshtein algorithm ( $O(m.n)$ ).

The following image summarizes the edit-distance difference between Levenshtein and Damerau-Levenshtein algorithm:



Picture 2.1 The edit distance between two algorithms<sup>[4]</sup>

## 2.4 Bayes Theorem

Bayes theorem is a probability theory which focuses on the manipulation of conditional probabilities. Bayes theorem is a result that derives from the more basic axioms of probability. Bayes theorem has applications in a wide range of calculations involving probabilities. Bayes theorem can be formularized by the following expression:

$$P(A | B) = \frac{P(B | A)P(A)}{P(B)}$$

Let's take a simple example on how this theory works. Suppose that a lecturer told you that he wanted to meet a certain student in his class. There exists two majors, IF and STI. Not knowing the distribution of student in both major, the probability that he wanted to meet either IF or STI student is 50%. Now suppose he also told you that the student is a male. Suppose that 90% of students in IF are male and 80% students in STI are male. Our goal is to calculate the probability that the lecturer wanted to meet a male student from IF major. Let's denote male student as M and female student as F. Let's also denote IF major as I and STI major as S. Using the formula of Bayes theorem, we have:

$$P(I | M) = \frac{P(M | I)P(I)}{P(M)} = \frac{P(M | I)P(I)}{P(M | I)P(I) + P(M | S)P(S)}$$

where we have used the law of total probability. This yields:

$$P(I | M) = \frac{0.9 * 0.5}{0.9 * 0.5 + 0.8 * 0.5} \approx 0.53$$

The probability that the lecturer wanted to meet an IF major student, given that the person is male, is about 53%.

## 2.5 Terms

The following are several terms which will be generally used in this paper.

### 2.5.1 n-Gram

n-Gram is a contiguous sequence of  $n$  items from given sequence of text, which is widely used in computational linguistics and probability. n-Grams typically are collected from a text or speech corpus. A n-gram of size 1 is usually called as a "unigram", size 2 is a "bigram", and size 3 is a "trigram".

In spell checker, correcting mistakes in spelling sometimes need more than *unigram* for verifying its correctness. For example, it's impossible for having two adjacent verbs in one complete sentence, and we can infer that one of them must be mistyped. The following case needs to take *bigram* into consideration.

More concisely, an n-gram model predicts  $x_i$  based on  $x_{i-(n-1)}, \dots, x_{i-1}$ . In Bayes theorem, we can denote this model as  $P(x_i | x_{i-(n-1)}, \dots, x_{i-1})$ . For our simplicity, we'll only take *unigram* to our current account.

### 2.5.2 Edit Distance

Edit distance is a term in computer science and defined as a way of quantifying how dissimilar two strings are to one another by counting the minimum number of operations required to transform one string into the other. Both Levenshtein and Damerau-Levenshtein have their own definitions regarding operations to determine the edit-distance between two strings. For our simplicity, we'll only take two strings which have a value of 1 in their edit distance to our current account.

### III. IMPLEMENTATION AND EXPERIMENTS

In this experiment, we will use two test-case, both are sized of 15 and 60. The first test-case is created from random example while the second test-case is created from a list of top 60 common mistakes in spelling, provided by Wikipedia. The dictionary contains 6.000 common words which are widely used by people.

#### 3.1 Naïve Implementation with Damerau-Levenshtein

The easiest way to implement spell checker is traversing each word through all words in the dictionary. The implementation of this algorithm is almost the same with pseudo-code in 2.3 above.

The result from running the first test-case:

Normal = 333 ms
Average Processing per Query = 22.2 ms
Accuracy = 73 %
Number of Found = 14 / 15

The result from running the second test-case:

Normal = 1642 ms
Average Processing per Query = 27.3667 ms
Accuracy = 88 %
Number of Found = 45 / 60

The results show that each word needs about  $\pm 25$  ms to be processed using 6.000 words in dictionary. The accuracy of spell checker using this method of implementation is  $\pm 81\%$ .

#### 3.2 Pruned Naïve Implementation

The further analysis shows that we don't need to check words with length below than (checked\_length - 1) and greater than (checked\_length + 1) in edit distance of 1. We will only check all words in range of [checked\_length - 1, checked\_length + 1].

The result from this pruning method for the first test-case:

Pruned = 138 ms
Average Processing per Query = 9.2 ms
Accuracy = 73 %
Number of Found = 14 / 15

While the result from running the second test-case:

Pruned = 737 ms
Average Processing per Query = 12.2833 ms
Accuracy = 88 %
Number of Found = 45 / 60

The results above show that there's an improvement from  $\pm 25$  ms to  $\pm 10.5$  ms using pruned naïve implementation. The accuracy is obviously still the same with previous method, around  $\pm 81\%$  accuracy.

#### 3.3 Bayes Theorem Implementation

In this method, we will have several analyses before we implement the algorithm. Suppose that we are trying to find the correction of  $c$  given the original word  $w$ :

$$\max_c P(c | w)$$

We want to choose the correction  $c$  which is having greatest value of  $P(c|w)$ . By substituting Bayes theorem, this is equivalent to:

$$\max_c \frac{P(w | c)P(c)}{P(w)}$$

Since  $P(w)$  is the same for all kinds of correction, we can eliminate  $P(w)$ , simplify the equation to:

$$\max_c P(w | c)P(c)$$

Let's say,  $P(c)$  is a probability that the proposed correction  $c$  stands on its own. In this experiment,  $P(c)$  will be determined by word ranks in the dictionary. For example, the word "nice" has greater probability than "niece" based on words' usage statistics.

$P(w|c)$  is a probability that  $w$  would be typed when the user meant  $c$ . Simply said, this is the probability of how likely the user would type  $w$  by mistake when  $c$  was intended.

We will choose the word with maximum probability from all possible words in dictionary. Of course, word that is having edit distance greater than 1 has probability of 0. In 3.1 and 3.2, we are only using  $P(c)$  to check.

There are many factors of  $P(w|c)$  that we need to take into account, but since some factors are not completely independent (increasing probability of  $x$  may decrease the probability of  $y$ ), we'll make simple analysis through it.

Let's have a little brainstorming. Which is having greater probability, mistyped a word which has less character, more character, or two characters swapped? For example, "burnd" is closer to "burned" than "burn" in  $P(w|c)$ , even though both of them are having edit distance of 1. By using several statistics, we'll get the priority as below:

1. The probability of  $w$  having less character than  $c$  is greater than the probability of  $w$  having same number of characters with  $c$ .
2. The probability of  $w$  having same number of characters with  $c$  is greater than the probability of  $w$  having more character than  $c$ .

Since we don't have enough data to separate the probability in quantitative measure, let's assume that each categories have 1/3 marginal from dictionary total size. As there're 6.000 words in our dictionary, we'll have 2.000 ranks marginal.

Probability of $w$ having less character
--

=

Probability of $w$ having same number of characters with $c + 2.000$
--

=

Probability of $w$ having more character than $c + 4.000$
---

The word “burned” will have 4.000 ranks greater than “burn” when we are comparing it with “burnd”. In this method, we’ll assume that swapping and mistyping operations in probability of  $w$  having same number of characters with  $c$  are the same.

The result from running the first test-case:

Bayes Spell Checker = 121 ms  
 Average Processing per Query = 9.512 ms  
 Accuracy = 86 %  
 Number of Found = 14 / 15

The result from running the second test-case:

Bayes Spell Checker = 791 ms  
 Average Processing per Query = 13.1833 ms  
 Accuracy = 91 %  
 Number of Found = 45 / 60

The results show that each words needs about  $\pm 11.3$  ms using Bayes theorem implementation. The accuracy of spell checker is around  $\pm 88.5\%$ . This method has the same complexity with pruned naïve implementation, with running time difference  $< 1$  ms for a query (from  $\pm 10.5$  ms to  $\pm 11.3$  ms). Also, the implementation of Bayes theorem has improved its accuracy by  $\pm 7.5\%$  (from  $\pm 81\%$  to  $88.5\%$ ), which is feasible enough to be used as a spell checker.

Method	Average Time	Accuracy
Naïve	$\pm 25$ ms	$\pm 81\%$
Pruned	$\pm 10.5$ ms	$\pm 81\%$
Bayes T.	$\pm 11.3$ ms	$\pm 88.5\%$

Table 3.1 Summary of Methods Performance

#### IV. FURTHER ANALYSIS WITH BAYES THEOREM

In this part, we will have further analyses with Bayes theorem in order to improve the accuracy of our spell checker. We’ll assume that our algorithm has relative good running time and does not need special improvement in its time complexity.

Beforehand, we need to **improve our dictionary size** since 6.000 common words only give around  $\pm 84\%$  matching from common mistakes. For example, the following is taken from the second test-case:

fiery not found. Expected : fiery. Do you mean:  
 (\*) fire(482), fired(3293)

In the example above, the expected word is not existed in our current dictionary. The expected number of words in dictionary should be around 10.000 common words.

Let’s back to our previous assumption. We’re assuming that swapping and mistyping operations are the same. In

reality, we should take several other factors into account, for example, **keyboard distance**. The following is taken from the second test-case:

clear not found. Expected : clerk. Do you mean:  
 (\*) clear(355), clerk(3095), clark(4387)

Both “clear” and “clerk” are having the same number of words. In our previous assumption, both words will not be affected by  $P(w|c)$ . But our intuitive should be able to prioritize “clerk” over “clear” because of keyboard distance. Word “r” and “k” has a distance of 5 in QWERTY keyboard, while “e” and “a” has only a distance of 2 in QWERTY keyboard. Hence, “clear” should have lower probability than “clerk” in  $P(w|c)$ . Of course, we should gather more data in order to convert keyboard distance into quantitative numeric.

The other factor that we need to take into account is **error character location**. It’s more unlikely for user to have a misspelling in first character of the word than two same adjacent consonants.

Yet, language processing could be tricky. It’s possible for a word with edit distance of 0 to be misspelled. The obscurity of its language model  $P(c)$  must be taken into account if we want to improve our algorithm with not only edit distance of 1, but also other value of edit distance.

$P(c)$ , the language model, becomes more important as we take **bigram or trigram** into our account. The probability  $P(c)$  of two adjacent verbs should be near 0. Sometimes, people who have **mother language other than English** may make typical / same mistakes. The main reason behind it is language transformation.

#### V. CONCLUSION

We can implement Damerau-Levenshtein algorithm with Bayes theorem to improve the accuracy of spell checker application.

There are still many rooms of improvement that could be applied based on this analysis. There are many aspects that we need to take account into in order to improve the effectiveness of spell checker, such as:

- Number of common words and obscure words in dictionary
- Type of keyboard and its distance between two specific characters
- Common knowledge of people (related to mother language and geographical area)
- Edit-distance (greater than 1 or 0, even though edit-distance of 1 has covered at least 80% of correctness probability)
- N-grams (greater than 1, Google has provided n-gram data based on billion search queries from its search engine which can be downloaded at <http://storage.googleapis.com/books/ngrams/books/datasetsv2.html>)

- Language grammar structures (such as subject – predicate – object in English)

The trade-off between higher accuracy (even by a slight) and time is important, since users often need real-time result from a spell-checking application. Number of data is the most important thing in improving the accuracy of spell checker, which Google has the most benefits from its search engine.

## VI. ACKNOWLEDGMENT

Iskandar Setiadi, as the author of this paper, wants to express his deepest gratitude to Dr. Ir. Rinaldi Munir, M.T. and Masayu Leylia Khodra, ST., MT. as the lecturers of IF 2211 – “*Strategi Algoritma*”. Special thanks to my family and all of my friends in Informatics 2011.

## REFERENCES

- [1] *Dictionary of 6000 Most Frequently Used Words in English*. November 29, 2013 (5.00 PM) < <http://www.insightin.com/esl/> >
- [2] Munir, Rinaldi, 2009. *Diktat Kuliah IF2211 Strategi Algoritma*. Program Studi Teknik Informatika STEI ITB
- [3] Peter Norvig, *How to Write a Spelling Corrector*. November 30, 2013 (4.45 AM) < <http://norvig.com/spell-correct.html> >
- [4] Richard Minerich. *Levenshtein Distance and the Triangle Inequality*. December 18, 2013 (2.10 AM) < <http://richardminerich.com/tag/damerau-levenshtein-distance> >
- [5] Steve Hanov. *Fast and Easy Levenshtein Distance using a Trie*. November 30, 2013 (4.30 AM) < <http://stevehanov.ca/blog/index.php?id=114> >
- [6] Walpole, Ronald, Raymond H. Myers, Sharon L. Myers, 2007. *Probability & Statistics for Engineers & Scientists*. Pearson Prentice Hall

## ADDITIONAL NOTES

All of the test-case which used in this experiment can be downloaded at:

[https://dl.dropboxusercontent.com/u/58181220/spellchecker\\_testcase.rar](https://dl.dropboxusercontent.com/u/58181220/spellchecker_testcase.rar)


Source code (written in C++) and dictionary <sup>[1]</sup> used in this experiment can be downloaded at:

[https://dl.dropboxusercontent.com/u/58181220/spellchecker\\_source.rar](https://dl.dropboxusercontent.com/u/58181220/spellchecker_source.rar)

## PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Desember 2013



Iskandar Setiadi 13511073