World Academics Journal of Management **Research Paper**
Vol.6, Issue.2, pp.01-04, December (2018) **E-ISSN:** 2321-905X

An Levenshtein Transpose Distance Algorithm for approximating String Matching

Meena Malviya[1], Rajendra Gupta[2*]

[1,2]Rabindranath Tagore University, Raisen, India

Available online at: www.isroset.org

Received: 29/Jul/2018, Accepted: 26/Aug/2018, Online: 31/Dec/2018

Abstract- The string-matching is a very essential issue in the wider area of text processing. String-matching algorithms are basic components used in implementations of practical existing software under most operating systems. Furthermore, they emphasize programming methods that serve as paradigms in other fields of computer science. There are different solutions have been proposed that allow solving the string matching issues. The main characteristic of string matching algorithm is the fact that it attempts to establish the correspondence of the substring with the pattern in the reverse direction. This paper focuses on an approach by implementing Levenshtein distance algorithm using transposition of characters. After the testing of data, it is found that the implemented algorithm gives significant results while string matching.

***Keywords*:** Levenshtein Distance Algorithm, Approximating String Matching, Stop words

I. INTRODUCTION

With the term, most usual understanding of "approximate" or just "like" is that of similarity between two strings. By some inspection identification process, the two strings can be determined to be similar or not. The important property of similarity which makes it very dissimilar from equivalence is that similarity is not necessarily transitive; that is,

if *r* is similar to *s*
and *s* is similar to *t*,
then it does not necessarily follow
that *r* is similar to *t*.

An Approximate String Matching (ASM) has been mostly applied in many fields, including network intrusion detection systems, web searching, voice recognition and also computational biology. Fundamentally, Approximate String Matching is the problem of finding all positions of a string where a given pattern occurs, allowing a limited number of *errors* in the matches. The closeness of a match is measured by the minimum number of edit operations used to convert a factor of the input string into an exact match of the pattern. The usual edit operations are insertion, deletion, replacement, and transposition. There is a popular method for ASM that allows three edit operations of insertion, deletion, and substitution to transform a factor of the input string into the pattern, which is called ASM with differences. This method is also called as ASM with edit distances or ASM with Levenshtein distance.

II. RELATED WORK

The exact pattern atching mean to find all occurrences of a pattern positioned in a text. As suggested by Woo & Cheol et al., the exact pattern matching can be useful in finding sequences in DNA (Woo & Cheol et al., 2016). Moreover, it can be useful to perform exact pattern matching prior to approximate pattern matching to save computational time by removing queries that exactly match before employing algorithms with higher computational cost. This concept is given by Smith and Waterman and Needleman and Wunsch in 2015. The two algorithms can be clearly distinguished from the mass of proposed methods for exact pattern matching.

An another work done by Musser and Nishanov claim that the skip loop of the fast Boyer and Moore algorithm performs inadequately with small alphabets and long patterns. This approach is to solve the problem is straight forward and uses hashing. This is clearly transforming the alphabet to a different space, using hashing also called q-grams (Kytojoki, et al., 2013). The two problems arise in strings from small alphabets. The time spent in the skip loop is reduced while the number of times that a match needs to be evaluated in detail is increased. This is even more pronounced when a large number of matches are expected in the text or if the suffix of the pattern is abundant in the text.

The author Raita (2016) created a variant of the Boyer and Moore algorithm which introduced sentinels in order to speed up searches by first comparing the parts of the pattern with the weakest dependencies. The author reported an

improvement of approximately 35% over the Boyer and Moore algorithm which has been shown by Smith (2010) to be solely due to sentinel use, as opposed to character dependencies within the pattern, as Raita concluded (Smith, 2015).

Moreover, two changes to the Boyer-Moore algorithm are proposed which even allow for shifts when an initial match needs to be evaluated. The look-up table for the skip-loop seems to be un-necessary and it is therefore fully removed in most of the algorithms and slightly reducing the algorithm's overhead. The semantics of the b1-shift table of the original BM algorithm is slightly changed so that it can be utilized for the skip loop as well as for all other evaluations. It also allows for shifts in case of partial matches which is a great benefit in case of small alphabets. It transpires that the shift can be calculated by using the maximum of the suffix shift and the shift at the position of the mismatch. In order to save computation, either the suffix shift or the mismatch shift is used in the algorithms developed in this study.

III. APPROXIMATE STRING MATCHING ALGORITHMS

3.1 Brute-force search

Brute Force string matching algorithm refers to a programming style that does not include any shortcuts to improve performance, but instead relies on sheer computing power to try all possibilities until the solution to a problem is found. The brute force solution is simply to calculate the total distance for every possible route and then select the shortest one. This is not particularly efficient because it is possible to eliminate many possible routes through clever algorithms.

In an example in which there is a 5 digit password, in the worst case scenario would take 10^5 tries to crack. The time complexity is determined in brute force is O(n*m). Therefore if we search for a string of 'n' characters in a string of 'm' characters using brute force, it will give n* m tries.

The Brute-force algorithm make parallel pattern at the beginning of text and in further step, the text moving from left to right, compare each character of pattern to the corresponding letter in text until all letters are found to match or an mismatch is detected in the next step while pattern is not found.
The functioning of Brute-force algorithm is as given below :

```
Brute_Force_String_Match (T[0...n-1], P[0...m-1])
        for i ← 0 to n-m do
                j ← 0
                while j < m and P[j] = T[i+j] do
                        j++
                if j = m then return i
        return -1
```

3.2 Rabin-Karp algorithm

The Rabin-Karp is a type of pattern searching algorithm which find the pattern in a more efficient manner. The algorithm also checks the pattern by moving window one-by-one, but without checking all letters in all cases. The algorithm calculates a numerical (also called hash) value for the pattern *p*, and for each *m*-character substring of text *t*. Then, it compares the numerical values instead of comparing the actual symbols. If any similarity is found, it compares the pattern with the sub-string by Naive approach else it shift to next sub-string of *t* to compare with the value of *p*.

The value can be calculated in the form of mathematical (hash) values by using Horner's rule. According to this algorithm, lets assume,

$h_0 = k,$
$h_1 = d(h_0 - d^{m-1} .p[1] + p [m + 1]$

Where *p* and substring t_i can be too large to work with conveniently.

The very simple solution is given here in which we can compute *p* and the t_i modulo a suitable modulus *q*. So for the each value of i,

$$h_{i+1} = (d\ h_i - t[i + 1].dm{-}1 + t[m + i + 1]) \bmod q$$

The modulus q is typically selected as a prime such that *d.q* fills within one computer word.

3.3 Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt algorithm is a linear time algorithm, more accurately O(N + M). The main characteristic of Knuth-Morris-Pratt algorithm (KMP) is each time when a match between the pattern and a shift in the text fails, the algorithm then use the information given by a specific table, obtained by a preprocessing of the pattern to avoid re-examine the characters that have been previously checked. Thus, limiting the number of comparison required. So this algorithm can be composed by two parts, a searching part which consists to find the valid shifts in the text, where the time complexity is O(N), obtained by comparison of the pattern and the shifts of the text, and a preprocessing part which consists to pre-processes the pattern. The algorithm work in the following manner :

```
function kmp_search:
integer;
var i, j: integer;
        begin i:= 1; j:= 1;
initnext;
repeat if (j = 0) or (a[i] = p[j])
then
        begin i:= i+1; j:= j+1
end
```



```
else
        begin j:= next[j] end;
        until (j > M) or (i > N);
if j > M
        then kmp_search:= i - M
else
        kmp_search:= i;
end;
```

IV. PROPOSED LEVENSHTEIN TRANSPOSE ALGORITHM :

In the proposed work, the Levenshtein distance algorithm is implemented with Levenshtein transpose distance. It can be treated as an extension to Levenshtein Distance, which allows one extra operation that is *Transposition* of two adjacent characters. It can be illustrated with the following example :

Suppose we have two words; TSART to START. In this case the value is 1 (since shifting of S and T positions cost only one operation). This can be implemented using Levenshtein distance which is known as Levenshtein Transpose Distance.

Here Levenshtein distance = 2 (Replace S by T and T by S)
This brings to Damerau-Levenshtein, which does not have the limitations of restricted edit distance. The main difference between Damarau-Levenshtein and the implemented Levenshtein edit distance algorithm is that when Damerau-Levenshtein computes a transposition, it generally look much further backwards to find a match than the reduced edit distance algorithm.
The cost of a transposition is calculated by the following concept;

(cost before transposition) + (distance between rows) + (distance between columns) + 1
The cost for all the four operations is calculated and the minimum cost operation is selected. For each element in the matrix, we just look at the three neighbor elements, in that add 1 to their values, and use the minimum of the three for the given element.

- the element to the left = previous cost for a delete operation + 1 ... this is the current deletion cost
- the element above = previous cost for an insert operation + 1 ... this is the current insertion cost
- the element to the upper left = previous cost for a substitute operation + 1 if the letters differ, or + 0 if the letters are the same ... this is the current substitution cost

V. RESULT AND DISCUSSION

The algorithm Levenshtein edit distance makes a supposition that causes some difficulty in some cases, although it assumes no characters added or deleted between the transposed characters.

The Levenshtein Transpose Distance algorithm calculates the minimum edit operations that are needed to modify one document to obtain second document. A matrix is initialized measuring in the (m, n)-cell in which Levenshtein distance between the m-character prefix of one with the n-prefix of the other word. For testing the algorithm, two documents 'A' and 'B' has been taken in the study and compared by including stop words and removal of stop words. All the experiments are done in MATLAB analytical tool.

Table 1 : Text length of document ‘A’ and ‘B’ with and without using stop words

Text length of First Document	Text length of Second Document	First document ‘A’ after removing stop words	Second document ‘B’ after removing stop words
49	64	29	40
101	94	65	58
199	190	122	114
285	385	220	221

Table 2 : Time taken to calculate Levenshtein’s distance after removing stop words

Text length of first document ‘A’	Text length of second document ‘B’	Time taken to calculate Levensthein distance with stop words (in milliseconds)
49	64	13
101	94	15
199	190	23
285	385	60

In the above table, 49 text length of document ‘A’ and ‘B’ means the document size. The document size means it contains the defined number of words. The experiment is done by taking different document size from 50–1000.

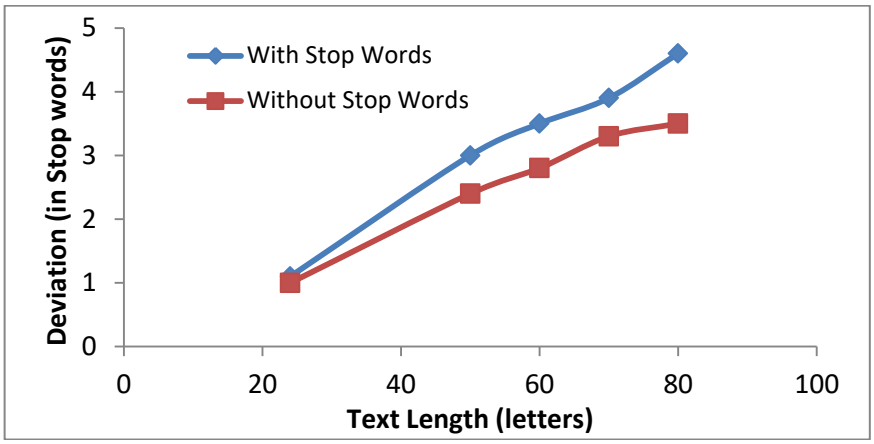

Fig. 1. Text length of document A with and without using stop words

The length of document 'A' and the length of same document after using and removing the stop words are shown in Figure 1. In which, blue colour represents the complete text length of document and red colour represents the length of document after removing the stop words.

VI. CONCLUSION

The main difference between Damarau-Levenshtein distance and the implemented Levenshtein edit distance algorithm is that when Damerau-Levenshtein computes a transposition it generally look much ahead backwards to find a match than the reduced edit distance algorithm does. If there is a transposition, the system in a way reverts to that earlier state, than calculates the cost of getting back. It clearly shows that there is a transposition between here and there.

The value of the cell is the lowest of the costs of addition, deletion, substitution, or transposition, than we do not actually have to check whether the differences in the middle of a transposition are additions or deletions. In this case, we just count both directions, and if neither is zero, the cost will simply be too high for that cost to be chosen.

The documents with different text length of 100, 150, 200, 350, 500 is taken to calculate the Levenshtein edit distance and the time need to compare both documents by using Levenshtein edit distance algorithm. It is noticed that each document consists of 25–40 per cent stop words, which are not useful for any calculation. Therefore, it is observed that if 25 per cent stop words are removed from any text document, 40 per cent time can be reduced to calculate the Levenshtein's edit distance. Finally, it is observed that the proposed systems produce around 85 to 92 per cent successful result.

REFERENCES

[1] C. Charas, T. Lecroq and J. D. Pehoushek. 2005. A very fast string matching algorithm for small alphabets and long patterns, Lecture notes in Computer Science, Volume 1448, pp. 55-64, 2018

[2] J. J. McConnell, "Analysis of Algorithms", Canisius College, pp 125-128, 2018

[3] M. Crochemore and D. Perrin, 1991, Two Way String Matching, Journal of the Association for Computing Machinery, Vol. 38, No 3, pp.65I-675, 2018

[4] T. Raita, 1992, Tuning the Boyer–Moore–Horspool String, Searching Algorithm, Software Practice and Experience, Vol 22(10),pp 879–884, 2018

[5] R. Baeza-Yates and G. H. Gonnet, 1992, A New Approach To Text Searching, Communication of the ACM,Vol.35, No.10, 2018

[6] James Lee Holloway, "Algorithms for string matching with applications in Molecular Biology", Oregon State University Corvallis, OR, USA, 2018

[7] Luqman Gbadamosi, "Voice Recognition System Using Template Matching", International Journal of Research in Computer Science, Volume 3 Issue 5, pp. 13-17, 2017

[8] Saurabh Tiwari and Deepak Motwani, "Feasible Study on Pattern Matching Algorithms based on Intrusion Detection Systems", International Journal of Computer Applications 96(20):13-17, 2017

[9] Kavita Ahuja, Preeti Tuli, "Object Recognition by Template Matching Using Correlations and Phase Angle Method", International Journal of Advanced Research in Computer and Communication Engineering, Vol. 2, Issue 3, 2017

[10] Prabhudeva S, "Plagiarism Detection by using Karp-Rabin and String Matching Algorithm Together", IJCSNS International Journal of Computer Science Engineering and Network Security, VOL.8 No.10, 2017

[11] Koloud Al-Khamaiseh, "A Survey of String Matching Algorithms", International Journal of Engineering Research and Applications, Vol. 4, Issue 7, Version 2, pp.144-156, 2017

[12] Knuth, D. E., Morris, J. H. Jr., and Pratt, V. R., "Fast pattern matching in strings," SIAM Journal of Computer, Vol. 6, No. 2, pp. 323-350, 2017.

[13] Boyer, R. S., and Moore, J. S., "A fast string searching algorithm" Communication in ACM, Vol. 20, No. 10, pp. 762-772, 2016

[14] Rivest, R. L., "On the worst case behavior of string-searching algorithms", SIAM Journal of Computer, Vol. 6, No. 4, pp. 669-674, 2016

[15] Galil, Z., "An improving the worst-case running time of the Boyer-Moore string matching algorithms" Communication ACM, Vol. 22, No. 9, pp. 505-508, 2016

[16] Rytter, W., "A correct preprocessing algorithm for Boyer-Moore string-searching" SIAM Journal of Computer, Vol. 9, No. 3, pp. 509-512, 2016

[17] Guibas, L. J. and Odlyzko, A. R., "A new proof of the linearity of the Boyer-Moore string searching algorithm" SIAM, Journal of Computer, Vol. 9, No. 4, pp. 672-682, 2016

[18] Horspool, R. N., "fast searching in strings" Software Practice Experience, Vol. 10, pp. 501-506, 2016

[19] Smit, G. "Comparison of three string matching Algorithms" Software Practice Experience, Vol. 12, pp. 57-66, 2015

[20] Ushijima, K., Kurosaka, T., and Yoshida, K., "SNOBOL with Japanese text processing facility" Proc. ICTP 83, pp. 235-240, 2015

