Using Levenshtein Distance Algorithm to Increase Database Search Efficiency and Accuracy

Baleid Mohammed Aldoukali[1], Ebrahem Ali Elburase[2]
[1,2] *University of Tripoli. - Libya*
[1]b.aldoukali@uot.edy.lu, [2]*eb.elburase@uot.edu.ly*

Abstract

Names are frequently used as a search criterion in databases to retrieve information, so names play an important role in information systems, but some names in the case of application may have some defects, including misspellings, in addition to those cultural differences complicate the retrieval of information Based on names. In a string fuzzy match, the goal is to find short text matches from many long texts, in which case fewer matches in variance are expected. For example, a short text can come from a dictionary, here usually one of the strings is short and the other is arbitrarily long. Levinstein's space has a wide range of applications, such as spell checking, optical character correction systems, and memory-based natural language translation utilities. In this paper, the Levenstein algorithm was used and included in the SQL Server database engine. The purpose of this is to combine the queries with Levenstein's algorithm to obtain the best search results and to compare them with the results of traditional search and filtering in queries. The results were better in terms of accessing the required records, but that was at the expense of the time it takes to get the results.

Keywords— text similarity, Levenshtein Algorithm, database efficiency

Introduction

Most database applications use queries to extract data and display search results in a tabular form. The query language SQL has been based on the relational algebra [1]. The searches are based on a set of parameters and commands, such as comparing two words and making sure that they are equal, or searching for a part of the word within the text, whether it begins or ends with it, and so on. But searches in databases have some shortcomings when the search words are entered incorrectly, such as writing the word missing from some letters or the word is written incorrectly which is an old problem [2], especially when searching for the names of people. Hence the need to increase the efficiency and accuracy of search operation to get better results. In order to achieve attention from the database community. It has a widespread real application such as web search, spell checking, translation to sign language [3] and DNA sequence discovery in bio-informatics [4]. In this paper, the Levenshtein algorithm was used and included in SQL Server database after converting it to a code written in C# and compiling it into a DLL file. The purpose is to get the search results using this algorithm and compare it with the traditional search results in database.

Edit Distance Algorithms

The edit distance between strings $a_1..a_m$ and $b_1..b_n$ is the minimum cost s of a sequence of editing steps that convert one string into another. [5] There are two variants in string similarity search. The first identifies the strings from a string set whose edit distances to the query are not larger than a given threshold (Threshold-based Similarity Search). The second finds top-k strings with the smallest edit distances to the query (Top-k Similarity Search) [6]. In the Threshold-based Similarity Search, Given a string set S, a query q, and a threshold τ, threshold-based similarity search finds all strings s ∈ S such that ED(s, q) ≤ τ. For example, consider the strings s= "أحمد, حماد محمود" and the query q= "أحمتد", τ = 1. The threshold-based similarity search returns {"أحمد"} since the edit distance between "أحمد" and q= "أحمتد" is 1 and the edit distances between other strings and q are larger than 1

Levenshtein Distance Algorithm

This algorithm is named after Vladimir Levenshtein, who developed it in 1965. The algorithm calculates the distance between two texts, where this distance is measured by the number of changes required to be made to the first text, so that it becomes equal to the second text [7]. This change occurs by substituting a letter by a letter, or deleting a letter, or adding a

letter. If the space between the two texts is zero, this means that they are equal, and if they are 1, this means that one of them differs from the other by a letter (insert, delete or change). These three operations can be represented in the following steps:

1) *Deleting a letter from any position say i, to give* $a_1 \dots a_{i-1}a_{i+1} \dots a_m$.
2) *Inserting a letter* $b \in \Sigma$ *at position i to give* $a_1 \dots a_i b a_{i+1} \dots a_m$.
3) *Replacing a letter at position i to a new letter* $b \in \Sigma$ *to give* $a_1 \dots a_{i-1} b a_{i+1} \dots a_m$. [5]

The Levenshtein distance between two strings a, b (of length $|a|$ and $|b|$ respectively) is given by $lev(|a|,|b|)$ [4] , where:

$$lev(i,j) = \begin{cases} \max(i,j) & \textbf{if } \min(i,j) = 0, \\ min \begin{cases} lev(i,j-1)+1 \\ lev(i-1,)+1 \quad otherwise \\ lev(i-1,j-1)+k_{(a_i \neq b_j)} \end{cases} \end{cases}$$

In this equation, k is the indicator function equal to 0 if $a_i = b_j$ and 1 otherwise. By |a| we denote the length of the string a. The equation above can be tabulated as follows: For all $0 \le i \le m$ and $0 \le j \le n$, denote by $lev_{i,j}$ the edit distance $ED(a_1 \dots a_i, b_1 \dots b_j)$ from string $a_1 \dots a_i$ to string $b_1 \dots b_j$. The matrix $lev((m+1) \times (n+1))$ can be obtained from the recurrence:

$$lev_{00} = 0$$
$$lev_{ij} = \min \begin{cases} lev_{i,j-1}+1, \\ \quad i > 0 \text{ or } j > 0 \\ lev_{i-1,j}+1, \\ lev_{i-1,j-1} + IF\ a_1 = b_1\ THEN\ 0\ ELSE\ I \end{cases}$$

If we used the previous example and calculate according to the previous recurrence, we will obtain the following table:

	""	أ	ح	م	د
""	0	1	2	3	4
أ	1	0	1	2	3
ح	2	1	0	1	2
م	3	2	1	0	1
ت	4	3	2	1	1
د	5	4	3	2	1

Operations key

change	insert
delete	You r here

The above table is a 2d matrix of string a = "أحمد" and string b = "أحمتـد" which can be evaluated starting from point lev_{00} and going over it row by row or column by column calculating the three operations until you reach the end of the matrix, the last cell $lev(n,m)$ holds the lowest cost required to change string b to string a which can be compared with the threshold τ specified earlier. Applications that use the edit distance algorithm convert the previously mentioned operations, in addition to the equation into a code to run on the computer, and the following code illustrates this:

Edit distance algorithm

Input: $a = a_1 \dots a_n$ *and* $b = b_1 \dots b_m$
1: *for i* ← *0 to n do*
2: $lev_{i,0} \leftarrow i;$
3: *for j* ← *0 to m do*
4: $lev_{0,j} \leftarrow j;$
5: *for I* ← *1 to n do*
6: *for j* ← *1 to m do*
$k = (a_i = b_j)?\ 0:\ 1;$
$lev_{i,j} \leftarrow \min(lev_{i,j-1}+1,$
$lev_{i-1,j}+1,$
$lev_{i-1,j-1}+k);$
7: *end for*
8: *return* $lev_{n,m;}$

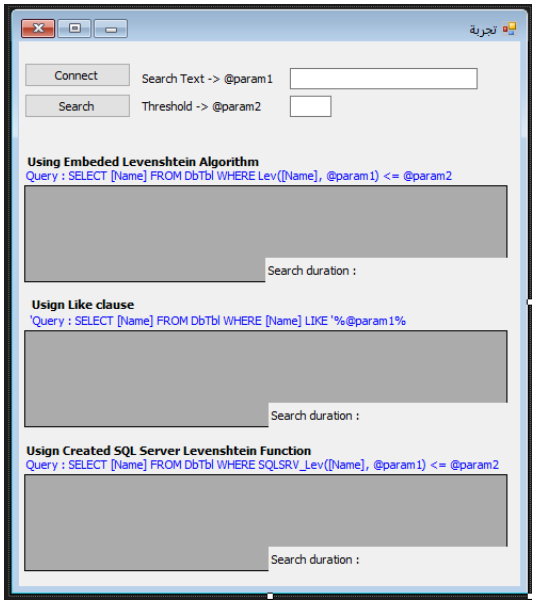
We converted the illustration above into C# function.

Implementation

In this paper, we proposed a list of deferent full Arabic names (first, middle, last, surname) with deferent variations (5000 and 15000 names) pumped into SQL server table. Two types of queries were applied to the table, one is a normal query that uses the (Like) clause to approximate the search results to the desired name, And the second is the Levenstein logarithm as a function in the database and used in a query to find the best approximation based on the specified threshold. The function for calculating Levenstein's distance is configured in two ways, first by C# and converting it to a DLL file included in SQL server, and second by creating a function directly inside SQL server, the purpose is to try the two functions separately and compare the results that obtained from them.

After preparing and including the functions, we designed a test tool through Visual Studio 2019 in C# language, its purpose is to connect to SQL server express 2014 database server and send 3 queries.

Figure 1. Test tool



The first query uses a LIKE clause to get the search result and the query was as follows:

$Select\ [name]\ from\ DbTbl\ Where\ [name]\ LIKE$

$'\%@param1\%'$

The second query uses the Levenstein algorithm, which was written in C-Sharp and embedded in the database server. The query was as follows:

$Select\ [name]\ from\ DbTbl\ Where\ CSharp\_Lev([name],$

$@param1) > @param2$

Finally, the third query uses the Levenstein algorithm, but this time by writing a function inside the database server instead of including it from outside, and the query was as follows:

$Select\ [name]\ from\ DbTbl\ Where\ SQLSrv\_Lev([name],$

$@param1) > @param2$

A laptop with the following specifications was used to connect to the database server

Table 1. Laptop specification

Brand	HP
Processor	Inter Core I7
Memory	8 GB
Operating System	Windows 10 Pro 64Bit

Results

The results were focused primarily on the accuracy of the search by comparing the results of the resulting search from the three queries, in addition, the time taken to extract the results is an important factor. As such, the names to be searched in the first parameter of the three queries are sent, written in full without errors or missing letters. The results were as follows:

- **In case of records returned**

1. First query (CSharp_Lev): The query returned two records, since the name is repeated twice in the database table but the last character is different.
2. Second query (Like clause): This query returns one record because Like Claus cannot replace or guess letters or round the results if there is an error in writing the name incorrectly
3. The third query (SqlSrv_Lev): The result was similar to the first query, but at the expense of the time it takes to return the results.
4. We made some changes to the name to be searched for by writing the name missing some letters, to determine whether the results will change or not. We found that the second query (LIKE clause) returned no records. As for the first and third query returned the same previous records.

Table 2. Search results

Name	CSharp_Lev	LIKE clause	SqlSrv_Lev
محمد سالم علي رمضان	**2 records**	**1 record**	**2 records**
محمد ستلم رمضان	**2 records**	**No records**	**2 records**

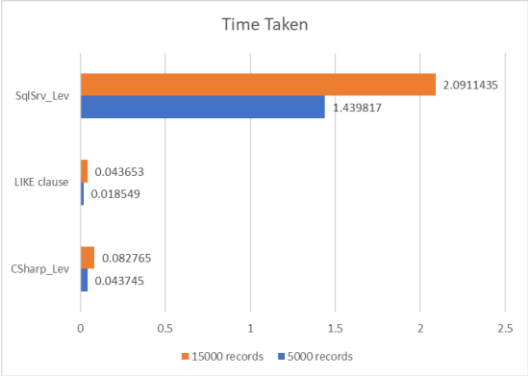
- **In case of time taken**

  The time taken was calculated on 5000 records and 15,000 records for the three queries and the results were as follows:

  1. First query (CSharp_Lev): the time was (0.043445) milliseconds for the 5000 records and (0.082765) milliseconds for the 15000 records.
  2. Second query (LIKE clause): the time taken was (0.018544) milliseconds for the 5000 records and (0.043653) milliseconds for the 15000 records.
  3. third query (SqlSrv_Lev): the time taken was (1.439887) seconds for the 5000 records and (2.911435) seconds for the 15000 records.

With a simple comparison, it shows us that the query (Like clause) is faster than the rest of the two queries, this was in terms of time, but as soon as you forget or change a letter or two, the results change. The query (Like clause) did not return any results, the winner was the query (CSharp_Lev) which showed two records and was faster than the third query (SqlServ_Lev) that showed the same results.

Figure 2. Results comparison



Conclusion

The Levenstein algorithm discussed in this paper is a method for determining the extent to which two texts match each other. This algorithm has been used in many applications, the most famous of which are genetic sequence analysis, translation and spell checker. In this paper, the algorithm is used to search database records and compare the results with traditional query statements such as LIKE Clause. The results concluded that it is possible to improve searching in databases by using this algorithm to get more accurate results.

References

[1] G. Sreenivasulu and M. Basha, "Searching As-You-Type in Databases Using SQ," *IJRECS*, vol. 1, no. 2, pp. 621-638, 2014.

[2] G. Navarro, "A Guided Tour to Approximate String Matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31-88, 2001.

[3] A. Almohimeed, M. Wald and R. I. Damper, "Arabic Text to Arabic Sign Language Translation for the Deaf and Hearing-Impaired Community," in *2nd Workshop on Speach and Language Proceissing and Assistive Technologies*, Edinburgh, UK, 2011.

[4] M. Bartoszuk and M. Gagolewski, "A Fuzzy R Code Similarity Detection Algorithm," in *Information Processing and Management of Uncertainty in Knowledge-Based Systems*, Montpellier, 2014.

[5] E. Ukkonen, "Algorithms of Aproximate String Matching," *Imformation And Control*, vol. 64, pp. 101-117, 1985.

[6] M. Yu, J. Wang, G. Li, Y. Zhang, D. Deng and J. Feng, "A unified framework for string similarity search with edit-distance," *The VLDB Journal*, p. 26, 2016.

[7] D. K. Po, "Similarity Based Information Retrieval Using Levenshtein," *International Journal of Advances in Scientific Research and Engineering*, vol. 6, no. 4, pp. 6-17, 2020.

[8] P. Thompson and D. Christopher, "Name Searching and Information Retrieval," in *Second Conference on Empirical Methods in Natural Language Processing*, 1997.

[9] S. Marcos-Pablos and F. J. García-Peñalvo, "Information retrieval methodology for aiding scientific database search," Soft Computing, vol. 24, no. 8, p. 5551–5560, 2020.