ISSN: 2321-7782 (Online)
Volume 4, Issue 1, January 2016
International Journal of Advance Research in Computer Science and Management Studies
Research Article / Survey Paper / Case Study
Available online at: www.ijarcsms.com

Automatic Data Cleansing of Incorrect Citynames Using LCS Algorithm

Subhani Shaik[1]
Assoc. Prof & Head of the Department
Department of CSE, St.Mary's Group of Institutions Guntur
Chebrolu(V&M), Guntur(Dt), Andhra Pradesh, India

Naga Malleswara Rao Nalamothu[2]
Professor,
Department of CSE, RVR & JC College of Engineering
Chowdavaram, Guntur(Dt),Andhra Pradesh, India

Abstract: ***In today's competitive environment, there is a need for more precise information for a better decision making. Yet the inconsistency in the data submitted makes it difficult to aggregate data and analyze results which may delays or data compromises in the reporting of results. The purpose of this paper is to give detailed study on the algorithms for data cleansing. The data cleansing algorithms can increase the quality of data while at the same time reduce the overall efforts for data collection. If the data quality is poor, wrong conclusions may be drawn from the data and the consequences may be tragic. Hence, poor data quality may lead to completely unexpected results. In this paper, Longest Common Subsequence algorithm is implemented for automatic city name correction to cleanse a large spatial database without requiring human intervention or a prior knowledge of the context. The longest common subsequence (LCS) problem is to find the longest subsequence common to all sequences in a set of sequences (often just two). The Longest Common Subsequence algorithm achieves a precision of 91% which is significantly better than the traditional Levenshtein distance.***

Key words: Data Cleansing, LCS, Levenshtein Distance.

I. INTRODUCTION

The longest common subsequence (LCS) problem is one of the classical and well-studied problems in computer science which has extensive applications in diverse areas ranging from spelling error corrections to molecular biology a spelling error correction program tries to find the dictionary entry which resembles most a given word. Data quality is a very important issue in data mining as well as to any information system. We have been looking at what is called "bottom-up Dynamic Programming". Here is another way of thinking about Dynamic Programming that also leads to basically the same algorithm, but viewed from the other direction. Sometimes this is called "top-down Dynamic Programming". It is no wonder why data qualities such as consistency and integrity have been the central theme in database design for a long time. In context less similarity, the similarity between the two objects is measured by considering the distance of the shortest path connecting the strings in the lexicon through editing steps[3].In general, the string-to-string correction problem is to determine the distance between two strings as measured by the minimum cost sequence of "edit operations," such as insertion, deletion, substitution and transposition needed to change the one string into the other. The proposed algorithm is based on the Longest Common Subsequence [2] with a significant modification for a substantially improved performance. A general benefit of using a metric is that the triangle inequality offers search space reduction during similarity query processing, which enables the system to converge upon the final result quicker. The spelling correction problem is closely related to the city name correction problem. It is, however, different from the automatic string correction problem. The spelling correction problem [1] is generally solved in consultation with the context. Consider the misspelled string "crd". Its correct spelling can be "cord" as in "power cord" or "card" as in "video card"[2].

II. Related Work

Levenshtein Distance:

In information theory and computer science the **Levenshtein distance** is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other. The phrase **edit distance** is often used to refer specifically to Levenshtein distance. It is named after Vladimir Levenshtein who considered this distance in 1965. In general, the string-to-string correction problem [4] is to determine the distance between two strings as measured by the minimum cost sequence of "edit operations," such as insertion, deletion, substitution and transposition needed to change the one string into the other. The string in the lexicon which has the minimum distance from the given incorrect string will be chosen as the correct string for the incorrect one. Hence, the heart of the city name correction problem is at the comparison of two strings for similarity(or dissimilarity). Similarity (or dissimilarity) between two objects in general can be measured either without context or within context. In context less similarity, the similarity between the two objects is measured by considering the distance of the shortest path connecting the strings in the lexicon through editing steps. It is closely related to pair wise string alignments**.** This algorithm is employed on string to string correction of city name errors [5]. The performance of the Levenshtein distance varies as the difference between the two strings in length varies. String to string correction of city name errors are generally solved by the Levenshtein distance (also frequently known as edit distance) and other variations. However, the performance of the Levenshtein distance varies as the difference between the two strings in length varies. For example, the Levenshtein distances from the incorrect name "agleMtn" to unrelated strings "Alta", "Altamont", "Leeton", "Mapleton", "Ogen" and "Salem" are all 5, less than the distance 6 to its correct version "Eagle Mountain".

The Levenshtein distance has several simple upper and lower bounds. These include:

- It is always at least the difference of the sizes of the two strings.
- It is at most the length of the longer string.
- It is zero if and only if the strings are equal.
- If the strings are the same size, the Hamming distance is an upper bound on the Levenshtein distance.
- The Levenshtein distance between two strings is no greater than the sum of their Levenshtein distances from a third string.

Relationship with other distance metrics:

There are other popular measures of edit distance which are calculated using a different set of allowable edit operations. For instance,

- the Damerau–Levenshtein distance allows insertion, deletion, substitution, and the transposition of two adjacent characters;
- the longest common subsequence metric allows only insertion and deletion, not substitution;
- the Hamming distance allows only substitution, hence, it only applies to strings of the same length.

For example, Consider "VISAKHA" as an incorrect city name for "VISAKHAPATNAM" in the city name database. The Levenshtein distance algorithm generates 6 between "VISAKHA" and "VISAKHAPATNAM" as it must insert "PATNAM" to make it same as "VISAKHAPATNAM".

The detailed computation of the Levenshtein distance is shown in Table 1. The value of cell (i, j) of each table represents the cumulative penalty cost to transform the 1..i substring of the vertically shown string (v-string) into the 1..j substring of the horizontally shown string (h-string).

If the two corresponding letters at cell (i, j) are the same, the penalty cost is set to 0 and the value of cell (i, j) will inherit the same value from the cumulative cost at the previous step, i.e., the value of cell (i-1, j-1). Otherwise, one of these three operations will take place: 1) replacement of the current (ith) letter of the v-string by that (jth letter) of the h-string, 2) deletion of the current letter of v-string or 3) insertion of the current letter of h-string. Either way, the current penalty cost should be set to 1. Then, the cumulative penalty cost is determined for the current cell by taking the minimum of the three values: the value of cell (i-1, j) plus 1, that of cell (i, j-1) plus 1 and that of cell (i-1, j-1) plus the current penalty cost set above.

Table 1 : The detailed output of the Levenshtein distance from "VISAKHA" and "VISAKHAPATNAM"

0	**V**	**I**	**S**	**A**	**K**	**H**	**A**	**P**	**A**	**T**	**N**	**A**	**M**
V	**0**	1	2	3	4	5	6	7	8	9	10	11	12
I	1	**0**	1	2	3	4	5	6	7	8	9	10	11
S	2	1	**0**	1	2	3	4	5	6	7	8	9	10
A	3	2	1	**0**	1	2	3	4	5	6	7	8	9
K	4	3	2	1	**0**	1	2	3	4	5	6	7	8
H	5	4	3	2	1	**0**	1	2	3	4	5	6	7
A	6	5	4	3	2	1	**0**	**1**	**2**	**3**	**4**	**5**	**6**

III. DATA CLEANSING USING LCS

An alternative measure to the Levenshtein distance is to use the ratio of the distance to the length of the longer string of the two. This alternative measure successfully suggests "Eagle Mountain" as the correct string for "agle Mtn" in the database. However, this alternative measure prefers "Boulder" to "Butlerville" for an incomplete string "Butler". In response to the problem noted above, an algorithm, called Longest Common Subsequence (LCS) , is presented in this paper to automatically correct incorrect city names in a database using a lexicon without requiring human intervention or a prior knowledge of the context. The proposed algorithm is based on the Longest Common Subsequence [5] with a significant modification for a substantially improved performance. The longest common subsequence (LCS) problem is to find the longest subsequence common to all sequences in a set of sequences. A Subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. For example, the sequence (A,B,D) is a subsequence of (A,B,C,D,E).Given two sequences *X* and *Y*, a sequence *Z* is said to be a *common subsequence* of *X* and *Y*, if *Z* is a subsequence of both *X* and *Y*. The longest common subsequence (LCS) problem is to find the longest subsequence common to all sequences in a set of sequences. A Subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. For example, the sequence (A,B,D) is a subsequence of (A,B,C,D,E).Given two sequences *X* and *Y*, a sequence *Z* is said to be a *common subsequence* of *X* and *Y*, if *Z* is a subsequence of both *X* and *Y* [6].

For example, Let us think of character strings as sequences of characters.

Given two strings X and Y, the Longest Common Subsequence of X and Y is a longest sequence Z that is a subsequence of both X and Y. For example, let X = (ABRACADABRA) and let Y = (YABBADABBADOO). Then the Longest Common Subsequence is Z = (ABADABA).

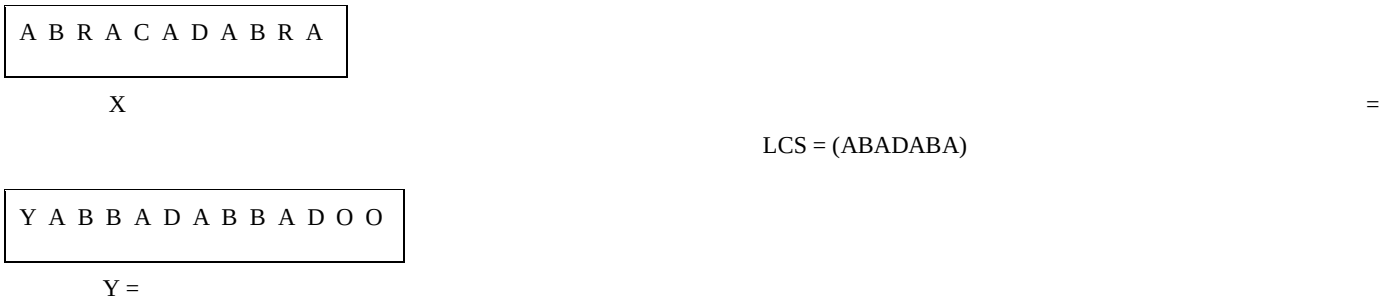

Fig 1 : An example of the LCS of two strings X and Y

The Longest Common Subsequence Problem (LCS) is the following. Given two sequences X = ($x_{1,}$……,x_m) and Y = ($y_{1,}$……,y_n) determine the length of their longest common subsequence, and more generally the sequence itself. Note that the subsequence is not necessarily unique. For example, LCS of (ABC) and (BAC) is either (AC) or (BC).

Last characters match:

Suppose xi = yj . For example: Let Xi = <ABCA> and let Yj =<DACA>. Since both end in 'A', it is easy to see that the LCS must also end in 'A'. (We will leave the formal proof as an exercise, but intuitively this is proved by contradiction. If the LCS did not end in 'A', then we could make it longer by adding 'A' to its end.) Also, there is no harm in assuming that the last two characters of both strings will be matched to each other, since matching the last 'A' of one string to an earlier instance of 'A' of the other can only limit our future options. Since the 'A' is the last character of the LCS, we may find the overall LCS by (1) removing 'A' from both sequences, (2) taking the LCS of Xi−1 = <ABC> and Yj−1 = <DAC> which is <AC>, and (3) adding 'A' to the end. This yields <ACA> as the LCS. Therefore, the length of the final LCS is the length of lcs(Xi−1, Yj−1) + 1 (see Fig. 2), which provides us with the following rule: if (xi = yj) then lcs(i, j) = lcs(i − 1, j − 1) + 1. We compute both options and take the one that gives us the longer LCS (see Fig. 3). If ($x_i \neq y_j$) then LCS(*i,j*) = max(LCS(*i-1,j*) ,(*i,j-1*))

Combining these observations we have the following recursive formulation:

- LCS(i,j) = 0 if i = 0 or j = 0
- LCS(i,j) = LCS(i-1,j-1) + 1 if i,j> 0 and $x_i = y_j$
- LCS(i,j) = max(LCS(i-1,j), LCS(i,j-1)) if i,j> 0 and $x_i \neq y_j$

Extracting the LCS:

The algorithms given so far compute only the length of the LCS, not the actual sequence. The remedy is common to many other DP algorithms. Whenever we make a decision, we save some information to help us recover the decisions that were made. We then work backwards, unraveling these decisions to determine all the decisions that led to the optimal solution. In particular, the algorithm performs three possible actions:

addXY : Add xi(= yj) to the LCS ('↖' in Fig. 4(b)) and continue with lcs[i − 1, j − 1]

skipX: Do not include xi to the LCS ('↑' in Fig. 4(b)) and continue with lcs[i − 1, j]

skipY : Do not include yj to the LCS ('←' in Fig. 4(b)) and continue with lcs[i, j − 1]

An updated version of the bottom-up computation with these added hints is shown in the code block below and Fig. 4(b).

How do we use the hints to reconstruct the answer? We start at the the last entry of the table, which corresponds to lcs(m, n). In general, suppose that we are visiting the entry.

Table 2: The detailed output of the LCS from "BACDB" and "BDCB"

0	**B**	**D**	**C**	**B**
B	**1**	**1**	1	1
A	1	**1**	1	1
C	1	1	**2**	2
D	1	2	**2**	2
B	1	2	2	**3**

Pseudo code of the lcs algorithm:

- Take two sequences to find out LCS.
- LCS function uses "zeroth element" which is convenient to define zero prefixes.
- All the prefixes are placed in a table in the first row and in the first column.
- This table is used to store the LCS sequence for each step of the calculation.
- when empty sequence is compared with a non empty sequence, the longest common subsequence is always an empty sequence.
- Next compare the elements in each sequence.
- If element is same in two sequences then that element is appended to the upper left sequence.
- If it is doesn't match then use second property and then LCS gets the longest of two sequences.
- The final result contains all the longest subsequences common to both sequences.

IV. Implementation and Results

There are four Components/Modules in my architecture diagram, and the functionality of these modules are as follows

- User Interface
- LCS Algorithm
- File Selection for input
- Database of 1000 records

The user will then interact with the system and ask to upload the file for selection of the input. The input file will be comprised of incorrect city names and then this file will be made to compare with the city names in the database of 1000 records. And the matching will be carried out by implementing LCS algorithm. And the application is implemented in basic java and on windows platform. End user can implement this application on any windows platform.

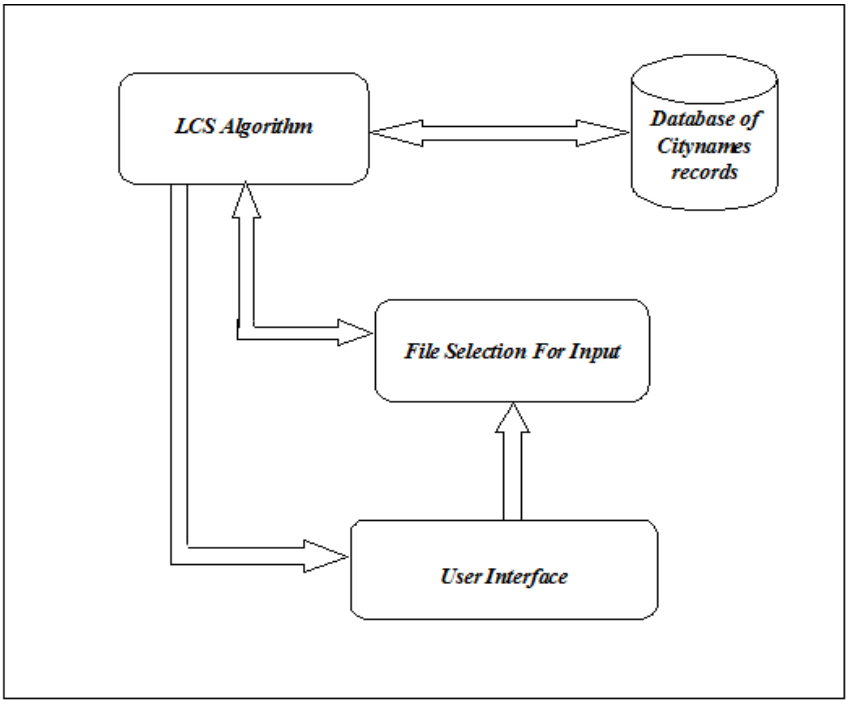

Fig 4 : Architecture for LCS based Data Cleansing Application

V. RESULTS

We now present the result of the experiment conducted on the Levenshtein distance, Longest Common Subsequece (LCS) within the City names database. The two similarity / dissimilarity algorithms were tested using city names within the City names database. There are 1000 city names records in the database, as a first step of data cleansing ,the user enters the incorrect input file as an input to the system, and then this in corrected input file will be compared to the city names in the database and then gives you the optimal solution. And the entire application is implemented in basic java and on Windows Platform. And with the precision of 90% accuracy is achieved. In such a way that the no.of records or data cleansed will give the accuracy.

VI. CONCLUSIONS

Data Cleansing is the process of detecting, correcting, corrupted inaccurate records from a record set, table or database. Data Cleansing is the term refers to identifying incomplete, incorrect, inaccurate, irrelavant, etc... parts of the data and then replacing, modifying or deleting this dirty data. It deals with detecting and removing errors and inconsistencies from data. This paper is mainly focused on automatic correction of incorrect city names in spatial data mining. This is nothing but data cleansing. Consists of removal of noisy data from the database with the help of the Longest Common Subsequence algorithm. We are going to implement this project making use Java programming language on Windows platform. Longest Common Subsequence algorithm mains to give you the best optimal solution which the existing algorithm Levenshtein Distance algorithm fails to give the best solution. In my application, I am considering a dataset of 1000 records and these data records are cleansed to give you the optimal solution. Making use of this application the end user will obtain an accuracy of > 90 % for the data cleansing operations.

References

1. S. Cucerzan and E. Brill. Spelling Correction as an iterative process that exploits the collective knowledge of web users. In EMNLP 2004, pages 293-300, 2004.
2. M. A. Alvarez and S. Lim.A graph modeling of semantic similarity between words. In 1st IEEE Int'l Conf. on Semantic Computing, pages 355-362. IEEE, 2007.
3. J. Otero, J. Grana, and M. Vilares. Contextual spelling correction. Lecture notes in computer science, 4739:290–296, 2005.
4. P. Zezula, G. Amato, V. Dohnal, and M. Batko. Similarity Search – The Metric Space Approach, volume 32. Springer, 2006.
5. G. Navarro. A guided tour to approximate string matching. ACM Computing Surveys, 33(1):31-88, 2001.
6. M. T. Goodrich and R. Tamassia. Algorithm Design: Foundations, Analysis, and Internet examples. Wiley, 2001.

