Levenshtein Automata

Mithali Shashidhar
PES1201700190
CSE Department
PES University
Bangalore, India
mithalishashidhar8@gmail.com

Ishwar Choudhary
PES1201700189
CSE Department
PES University
Bangalore, India
ishwarc404@gmail.com

Aditya Vinod Kumar
PES1201700138
CSE Department
PES University
Bangalore, India
ads.vinodk@gmail.com

***Abstract*—This project focusses on the implementation of the Levenshtein automata to compute the edit distance between two strings as well to be able to return all the strings that are in a an edit distance from an input string.**

I. Introduction

The Levenshtein distance can be defined as 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. It is named after Vladimir Levenshtein, who discovered this equation in 1965.

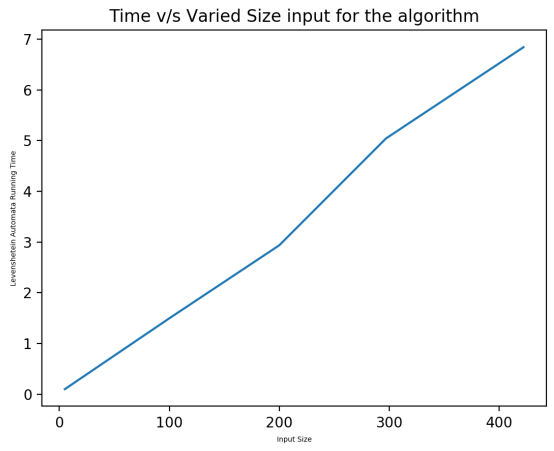
The basic insight behind Levenshtein automata is that it's possible to construct a Finite state automaton that recognises exactly the set of strings within a given Levenshtein distance of a target word. We can then feed in any word, and the automaton will accept or reject it based on whether the Levenshtein distance to the target word is at most the distance specified when we constructed the automaton. Further, due to the nature of FSAs, it will do so in O(n) time with the length of the string being tested.

II. Implementation of the Automata

The most crucial part of the entire implementation is the construction of the NFA and its transitions. We're basically constructing the transitions ,as well as denoting the correct set of final states. State labels are tuples, rather than strings, with the tuples using the same notation we described above. Because this is an NFA, there can be multiple active states. Between them, these represent the possible interpretations of the string processed so far.

III. ANALYSIS OF THE ALGORITHM

The algorithm as mentioned can run in O(n) time.



A plot which shows how the almost linear time increase with the increase in input size for the Levenshtein Automata

IV. Other algorithms to compute edit distance

A. LCS - Longest Common Subsequence

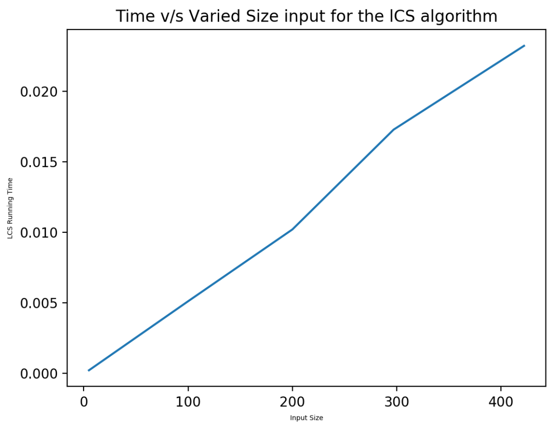
The longest common subsequence (LCS) problem consists in finding the longest subsequence common to two (or more) sequences. It differs from problems of finding common substrings: unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences.

The LCS distance between strings X (of length n) and Y (of length m) is n + m - 2 |LCS(X, Y)| min = 0 max = n + m

LCS distance is equivalent to Levenshtein distance when only insertion and deletion is allowed (no substitution), or when the cost of the substitution is the double of the cost of an insertion or deletion.

Our LCS algorithm implements the dynamic programming approach, which has a space requirement O(m.n), and computation cost O(m.n).

The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the `diff` utility, and has applications in computational linguistics and bioinformatics.



A plot which shows how the almost linear time increase with the increase in input size for the LCS Algorithm.

B. Dynamic Programming Approach

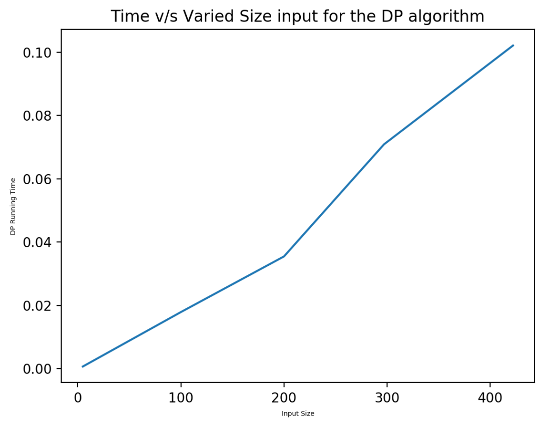
In this approach, all three operations(insertion, deletion and substitution) are allowed and are used to transform one string to another and the number of such operations required to do so would be the edit distance.

The algorithm for this approach is as follows:

- f last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
- Else (If last characters are not same), we consider all operations on 'str1', consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.
    - Insert: Recur for m and n-1
    - Remove: Recur for m-1 and n
    - Replace: Recur for m-1 and n-1

From the aforementioned algorithm, we can observe that there are many subproblems that are solved repetitively, therefore fulfilling the primary requirement for it being a DP Problem. We therefore, compute these values as save it in a DP Table for future use, thereby improving its efficiency.

However, The time complexity of above solution is exponential. In worst case, we may end up doing O(3^m) operations. The worst case happens when none of characters of two strings match. On a more average case, it has a O(m.n) time complexity.



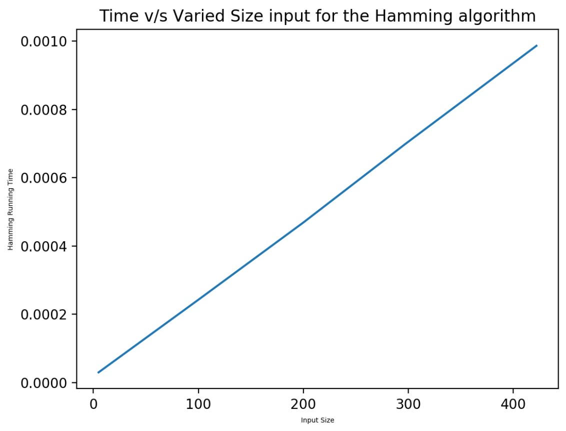
Graph which shows how the almost linear time increase with the increase in input size for the DP Algorithm.

C. *Hamming's Algorithm*

Hamming's algorithm calculates the hamming distance between two strings. The constraint imposed on this algorithm however, is that the two strings must be of equal length and only substitutions are allowed, unlike the two aforementioned algorithms. Therefore, Hamming Distance can be defined as the distance between two strings of equal length is the number of positions at which the corresponding character are different.

This problem can be solved with a simple approach in which we traverse the strings and count the mismatch at corresponding position.

The time complexity of such an algorithm is O(n).



A plot which shows how the almost linear time increase with the increase in input size for the DP Algorithm.

For Hamming distance of two binary numbers, we can simply return count of set bits in XOR of two numbers.

V. APPLICATOIONS

1. String Searching
2. String Matching
3. Auto-correct and Autocompletion

CONCLUSIONS

Construction of Levenshtein automata is very efficient for dealing with large strings since any matching involved is O(n) time complexity when compared to the others with O(n^2). This tends to be extremely useful when we are dealing with very large inputs, with several applications such as autocomplete, spellcheck, intersection of DFA etc.

ACKNOWLEDGMENT

We would like to thank Prof. Channa Bankapur for providing his invaluable guidance and suggestions throughout the course and for making learning algorithms a fun experience. we would also like to thank the teaching assistants for their support.

REFERENCES

1. https://www.geeksforgeeks.org
2. https://pythogalleryn-graph-.com.
3. https://medium.com/python-pandemonium/data-visualization-in-python-bar-graph-in-matplotlib-f1738602e9c4
4. https://en.wikipedia.org/wikiLongest_common_subsequence_problem
5. http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata
6. https://en.wikipedia.org/wiki/Powerset_construction