Writeup 2: *(Levenshtein distance)*

Shahroz Imtiaz[1], Guillermo Saavedra[2],

***Abstract*—You should complete the assignment in groups of two. This assignment is indented to be a fun and creative exercise. Pick an algorithm that we have not discussed in class and write a one page paper about it. Use this latex document as your template.**

I. Introduction

Levenshtein distance, sometimes referred to as edit distance, is a metric used to determine the dissimilarity between two given strings. Informally, this means finding the least amount of edits to get from string1 to string2. An edit being an insertion, deletion, replacement. It was originally thought of by a soviet mathematician named Vladimir Levenshtein, who considered this distance in 1965 [1]. One of the applications that the algorithm can be used for is in spell checking by providing the words with the least Levenshtein distance. Other applications also include being able to determine the similarities between two DNA sequences. [1].

II. Description

A. Levenshtein Algorithm

The idea is to traverse over both strings starting from either the left or right end. Let's start on the right. If the last characters are the same, move left one character. Else if the last characters are not the same, consider the three edits, recursively compute the three edits, and take the minimum cost of the three possible edits. For inserting, recur on the same character from string 1 but one character over to the left for string 2. For remove, recur on one character over to the left for string 1 and the same character for string 2. For replace, recur on one character over to the left for both strings [2].

Pseudo code:

min(int x, int y,int z) {
return lowest of the three inputs
}

editDistance(String stringOne, String stringTwo, int pointerOne, int pointerTwo {

if pointerOne is zero then return pointerTwo

if pointerTwo is zero then return pointerOne

if the character from stringOne at pointerOne equals the character in stringTwo at pointerTwo then move over to the left one character in both strings

else add one plus the min cost of an edit and use the min function and recursion on:

Insert: Keep pointerOne the same. Move pointerTwo to the left by one.

Remove: Move pointerOne to the left by one. Keep pointerTwo the same.

Replace: Move pointerOne to the left by one. Move pointerTwo to the left by one.

}
Runtime: $O(3^m)$

III. Conclusion

Conclude with your general thoughts on the algorithm and how it might be improved.

This algorithm is really cool because of it's real life applications for things such as spell checking and DNA sequence matching, but as the algorithm sits currently, it's far too slow as it is exponential in worst case run-time.
A way to improve on the algorithm's run-time is to realize that many sub-problems are solved over and over again and as we learned in class, this is the basis for dynamic programming. To avoid the repetition of solving the same sub-problems over and over again, we can use a temporary 2D array to store the results of sub-problems when we first solve them and just use those results over and over again. The size of the 2D array would be the length of the first string times the length of the second string. We can start on the upper left corner of the 2D array and on each cell we can either move horizontally or vertically depending on whether an insert, delete or substitution is needed. We would need to add one for every operation that we do on the string. At each step we have the current min edit distance up to that given point. If we follow this routine the matrix will end up having the final edit distance in the bottom right cell [3].

Runtime: $O(mxn)$ where m and n are the lengths of the string

References

[1] *Levenshtein Distance*. 14 Apr. 2019, en.wikipedia.org/wiki/Levenshtein_distance
[2] Edit Distance — DP-5. GeeksforGeeks, 8 Jan. 2019, www.geeksforgeeks.org/edit-distance-dp-5/.
[3] Babar, Nikhil. The Levenshtein Algorithm. *Cuelogic Technologies Pvt. Ltd.*, 3 Apr. 2019,www.cuelogic.com/blog/the-levenshtein-algorithm.