Refined Levenshtein

Yi Lu

November 2023

Contents

0	1	2	...
1	0	0	...
2	0	0	...
⋮	⋮	⋮	⋱

Table 1: dp initialiaztion

1 Introduction

An algorithm based on the Levenshtein distance function that can determine whether one string can be edited into another of equal length with a given number of edits. The Levenshtein distance function algorithm is implemented based on dynamic programming. If we want to calculate whether it's possible to reach the other string with a specified number of substitutions, insertions, and deletions, we just need to store the edit history at each position in the matrix. However, if implemented as such, it may result in a lot of redundant computations, and it may not be conducive to GPU implementation. Therefore, I propose a static type of implementation to address this issue.

This function is used in [1]. And we can obtain a code table with 90 score of compatibility and lower code rate $(1:5)$ compared with $(1:6)$.

2 Implementation

2.1 Original Levenshtein function

If we denote s_1 and s_2 as two different strings, the original function is used to calculate their edit distance as following relationship [2]:

$$lev(s_1, s_2) = \begin{cases} len(s_1) & \text{if } len(s_2) == 0 \\ len(s_2) & \text{if } len(s_1) == 0 \\ lev(s_1[1:], s_2[1:]) & \text{if } s_1[0] == s_2[0] \\ 1 + min \begin{cases} lev(s_1[1:], s_2) \\ lev(s_1, s_2[1:]) \\ lev(s_1[1:], s_2[1:]) \end{cases} & \text{otherwise} \end{cases}.$$

If we denote dp as the corresponding dynamic programming matrix and n as the length of these two strings, the shape of dp is

$$(n+1) \times (n+1).$$

Additionally, the first row and the first column need to be initialized as table1 to represent the number of insertions and deletions required to transform an empty string to the respective positions i and j,

Then for i-rows and j-columns, we can repeat following update function

$$dp[i][j] = \begin{cases} dp[i-1][j-1] & \text{, if } s_1[i] == s_2[j]. \\ 1 + min \begin{cases} dp[i-1][j] \\ dp[i][j-1] \\ dp[i-1][j-1] \end{cases} & \text{, otherwise.} \end{cases}$$

The final result is $dp[n-1][n-1]$.

2.2 Implementation for refined Levenshtein function

Here is the implementation of Python [3].

If we denote the substitution limit as n_s and the insertion or deletion limit as n_{indel} (since in the case of comparing equal-length strings, insertion is equivalent to deletion). We denote dynamic matrix as rdp. Therefore, the maximum possible value for all edit records is

$$R = (n_s + 1) \times (n_{indel} + 1)^2.$$

We use a ternary tuple to store the counts of substitutions, insertions, and deletions in the following order:

$$rdp[i][j][k] = (S_{ijk}, I_{ijk}, D_{ijk}), 0 \leq S_{ijk} \leq n_s, 0 \leq I_{ijk}, D_{ijk} \leq n_{indel}.$$

Encode this tuple through a one-to-one mapping into an index,

$$d_k = D_{ijk} + n_{indel} \times I_{ijk} + n_{indel}^2 \times Sijk \leftrightarrow rdp[i][j][k].$$

To save memory, here, we use $K = \lceil \frac{R}{8} \rceil$ bits of int8 data to record all possible edit records for the current step. So the shape of rdp is

$$(n+1) \times (n+1) \times K.$$

A k-th numerical representation, where each bit corresponds to the presence (1) or absence (0) of the corresponding edit record.

For updating the dynamic programming matrix, we only need to iterate over k (We next focus on $s_1[i]! = s_2[j]$). By using the edit records corresponding to k, we can deduce whether the truth values at the respective positions can be obtained through one step of the corresponding edit. Specifically, suppose the tuple corresponding to k is (a, b, c), we only need to affirm index k_s responding to $(a-1, b, c)$ in $rdp[i-1][j-1]$, index k_i responding to $(a, b-1, c)$ in $rdp[i][j-1]$ or index k_d responding to $(a, b, c-1)$ in $rdp[i-1][j]$ is 1 or not. If there exists 1 then let $rdp[i][j][k] = 1$ else we don't change anything.

3 Analysis

3.1 Space complexity

Although we mentioned using a dynamic programming matrix to implement the entire algorithm earlier, we did not fully utilize all positions in the matrix.

Considering that the dynamic programming is at most related to the previous two steps, and each step only needs to consider the number of insertions and deletions, the minimum required storage space can be calculated as:

$$[3 \times (2 \times n_{indel} + 1)] \times K \sim O(n_s \cdot (n_{indel})^3).$$

We should notice that n_s, n_{indel} can't be larger than n.

3.2 Time complexity

Regarding time complexity, we only provide an upper bound estimate for coordinates that contribute to effective computation. The actual number of coordinates that are computed is:

$$n \times (2 \times n_{indel} + 1).$$

The loop iterations required for each coordinate are:

$$3 \times R.$$

Overall, the maximum complexity is:

$$3 \times n \times R \times (2 \times n_{indel} + 1) \sim O(n_s \cdot (n_{indel})^2 \cdot n).$$

Whether in terms of time complexity or space complexity, it may seem that n_{indel} is high. However, in practical use,n_{indel} is often considered to be a small constant value, while n is large. This implies that the upper bound of the reachability algorithm is $O(n)$.

References

[1] The Mammoth International Contest On OMICS Sciences. `https://micos.cngb.org/zh-hans/`

[2] Levenshtein distance. `https://en.wikipedia.org/wiki/Levenshtein_distance`

[3] Code for refined Levenshtein function. `https://github.com/ylu1997/ylu1997.github.io/blob/main/Only3000/Refined_Levenshtein.py`