# Fast, memory efficient Levenshtein algorithm

By [Sten Hjelmqvist](http://www.codeproject.com/script/Membership/View.aspx?mid=1905096), 26 Mar 2012

# Introduction

The Levenshtein distance is the difference between two strings. I use it in a web crawler application to compare the new and old versions of a web page. If it has changed enough, I update it in my database.

## Description

The original algorithm creates a matrix, where the size is StrLen1\*StrLen2. If both strings are 1000 chars long, the resulting matrix is 1M elements; if the strings are 10,000 chars, the matrix will be 100M elements. If the elements are integers, it will be 4\*100M == 400MB. Ouch!

This version of the algorithm uses only 2\*StrLen elements, so the latter example would give 2\*10,000\*4 = 80 KB. The result is that, not only does it use less memory but it's also faster because the memory allocation takes less time. When both strings are about 1K in length, the new version is more than twice as fast.

## Example

The original version would create a matrix[6+1,5+1], my version creates two vectors[6+1] (the yellow elements). In both versions, the order of the strings is irrelevant, that is, it could be matrix[5+1,6+1] and two vectors[5+1].

## The new algorithm

### Steps

|  |  |
| --- | --- |
| Step | Description |
| 1 | Set n to be the length of s. ("GUMBO") Set m to be the length of t. ("GAMBOL") If n = 0, return m and exit. If m = 0, return n and exit. Construct two vectors, v0[m+1] and v1[m+1], containing 0..m elements. |
| 2 | Initialize v0 to 0..m. |
| 3 | Examine each character of s (i from 1 to n). |
| 4 | Examine each character of t (j from 1 to m). |
| 5 | If s[i] equals t[j], the cost is 0. If s[i] is not equal to t[j], the cost is 1. |
| 6 | Set cell v1[j] equal to the minimum of: a. The cell immediately above plus 1: v1[j-1] + 1. b. The cell immediately to the left plus 1: v0[j] + 1. c. The cell diagonally above and to the left plus the cost: v0[j-1] + cost. |
| 7 | After the iteration steps (3, 4, 5, 6) are complete, the distance is found in the cell v1[m]. |

This section shows how the Levenshtein distance is computed when the source string is "GUMBO" and the target string is "GAMBOL":

#### Steps 1 and 2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | v0 | v1 |  |  |  |  |
|  |  | G | U | M | B | O |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| G | 1 |  |  |  |  |  |
| A | 2 |  |  |  |  |  |
| M | 3 |  |  |  |  |  |
| B | 4 |  |  |  |  |  |
| O | 5 |  |  |  |  |  |
| L | 6 |  |  |  |  |  |

#### Steps 3 to 6, when i = 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | v0 | v1 |  |  |  |  |
|  |  | G | U | M | B | O |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| G | 1 | 0 |  |  |  |  |
| A | 2 | 1 |  |  |  |  |
| M | 3 | 2 |  |  |  |  |
| B | 4 | 3 |  |  |  |  |
| O | 5 | 4 |  |  |  |  |
| L | 6 | 5 |  |  |  |  |

#### Steps 3 to 6, when i = 2

SWAP(v0,v1): If you look in the code you will see that I don't swap the content of the vectors but I refer to them.

Set v1[0] to the column number, e.g. 2.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | v0 | v1 |  |  |  |
|  |  | G | U | M | B | O |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| G | 1 | 0 | 1 |  |  |  |
| A | 2 | 1 | 1 |  |  |  |
| M | 3 | 2 | 2 |  |  |  |
| B | 4 | 3 | 3 |  |  |  |
| O | 5 | 4 | 4 |  |  |  |
| L | 6 | 5 | 5 |  |  |  |

#### Steps 3 to 6, when i = 3

SWAP(v0,v1).

Set v1[0] to the column number, e.g. 3.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | v0 | v1 |  |  |
|  |  | G | U | M | B | O |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| G | 1 | 0 | 1 | 2 |  |  |
| A | 2 | 1 | 1 | 2 |  |  |
| M | 3 | 2 | 2 | 1 |  |  |
| B | 4 | 3 | 3 | 2 |  |  |
| O | 5 | 4 | 4 | 3 |  |  |
| L | 6 | 5 | 5 | 4 |  |  |

#### Steps 3 to 6, when i = 4

SWAP(v0,v1).

Set v1[0] to the column number, e.g. 4.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | v0 | v1 |  |
|  |  | G | U | M | B | O |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| G | 1 | 0 | 1 | 2 | 3 |  |
| A | 2 | 1 | 1 | 2 | 3 |  |
| M | 3 | 2 | 2 | 1 | 2 |  |
| B | 4 | 3 | 3 | 2 | 1 |  |
| O | 5 | 4 | 4 | 3 | 2 |  |
| L | 6 | 5 | 5 | 4 | 3 |  |

#### Steps 3 to 6, when i = 5

SWAP(v0,v1).

Set v1[0] to the column number, e.g. 5.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | v0 | v1 |
|  |  | G | U | M | B | O |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| G | 1 | 0 | 1 | 2 | 3 | 4 |
| A | 2 | 1 | 1 | 2 | 3 | 4 |
| M | 3 | 2 | 2 | 1 | 2 | 3 |
| B | 4 | 3 | 3 | 2 | 1 | 2 |
| O | 5 | 4 | 4 | 3 | 2 | 1 |
| L | 6 | 5 | 5 | 4 | 3 | 2 |

#### Step 7

The distance is in the lower right hand corner of the matrix, v1[m] == 2. This corresponds to our intuitive realization that "GUMBO" can be transformed into "GAMBOL" by substituting "A" for "U" and adding "L" (one substitution and one insertion = two changes).

## Improvements

If you are sure that your strings will never be longer than 2^16 chars, you could use ushort instead of int, if the strings are less than 2^8 chars, you could use byte. I guess, the algorithm would be even faster if we use unmanaged code, but I have not tried it.