WIKIPEDIA
The Free Encyclopedia
WIKIPEDIA

Levenshtein distance

This is an old revision of this page, as edited by Pouya (talk | contribs) at 21:40, 29 January 2005 (→The algorithm: i -> i1). The present address (URL) is a permanent link to this revision, which may differ significantly from the current revision.

In information theory, the **Levenshtein distance** or **edit distance** between two strings is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution. It is named after the Russian scientist Vladimir Levenshtein, who considered this distance in 1965. It is useful in applications that need to determine how similar two strings are, such as spell checkers.

For example, the Levenshtein distance between "kitten" and "sitting" is 3, since these three edits change one into the other, and there is no way to do it with less than three edits:

1. kitten
2. sitten
3. sittin
4. sitting

It can be considered a generalization of the Hamming distance, which is used for strings of the same length and only considers substitution edits. There are also further generalizations of the Levenshtein distance that consider, for example, exchanging two characters as an operation.

The algorithm

A commonly-used algorithm for computing the Levenshtein distance involves the use of an $(n + 1) \times (m + 1)$ matrix, where n and m are the lengths of the two strings. Here is pseudocode for a function *LevenshteinDistance* that takes two strings, *str1* of length *lenStr1*, and *str2* of length *lenStr2*, and computes the Levenshtein distance between them:

```
int LevenshteinDistance ( char str1[ 1..lenStr1 ], char str2[ 1..lenStr2 ] )
   // d is a table with lenStr1+1 rows and lenStr2+1 columns
   declare int d[ 0..lenStr1, 0..lenStr2 ]
   // i1 and i2 are used to iterate over str1 and str2
   declare int i1, i2, cost

   for i1 from 0 to lenStr1
       d[ i1, 0 ] := i1
   for i2 from 0 to lenStr2
       d[ 0, i2 ] := i2

   for i1 from 1 to lenStr1
```


META-1042
IPR2025-01248

```
        for i2 from 1 to lenStr2
            if str1[ i1 ] = str2[ i2 ] then cost := 0
                                       else cost := 1
            d[ i1, i2 ] = minimum(
                                d[ i1 - 1, i2     ] + 1,     // insertion
                                d[ i1    , i2 - 1 ] + 1,     // deletion
                                d[ i1 - 1, i2 - 1 ] + cost   // substitution
                                 )

    return d[ lenStr1, lenStr2 ]
```

Possible improvements

Possible improvements to this algorithm include:

- We can adapt the algorithm to use less space, O(m) instead of O(mn), since it only requires that the previous row and current row be stored at any one time.
- We can store the number of insertions, deletions, and subtitutions separately, or even the positions at which they occur, which is always `j`.
- We can give different penalty costs to insertion, deletion and substitution.
- The initialization of `d[i,0]` can be moved inside the main outer loop.
- This algorithm parallelizes poorly, due to a large number of data dependencies. However, all the `costs` can be computed in parallel, and the algorithm can be adapted to perform the `minimum` function in phases to eliminate dependencies.

Proof of correctness

The invariant is that we can transform the initial segment `s[1..i]` into `t[1..j]` using a minimum of `d[i,j]` operations. This invariant holds since:

- It is initially true on row and column 0 because `s[1..i]` can be transformed into the empty string `t[1..0]` by simply dropping all `i` characters. Similarly, we can transform `s[1..0]` to `t[1..j]` by simply adding all `j` characters.
- The minimum is taken over three distances, each of which is feasible:
  - If we can transform `s[1..i]` to `t[1..j-1]` in `k` operations, then we can simply add `t[j]` afterwards to get `t[1..j]` in `k+1` operations.
  - If we can transform `s[1..i-1]` to `t[1..j]` in `k` operations, then we can do the same operations on `s[1..i]` and then remove the original `s[i]` at the end in `k+1` operations.
  - If we can transform `s[1..i-1]` to `t[1..j-1]` in `k` operations, we can do the same to `s[1..i]` and then do a substitution of `t[j]` for the original `s[i]` at the end if necessary, requiring `k+cost` operations.
- The operations required to transform `s[1..n]` into `t[1..m]` is of course the number required to transform all of `s` into all of `t`, and so `d[n,m]` holds our result.

This proof fails to validate that the number placed in `d[i,j]` is in fact minimal; this is more difficult to show, and involves an argument by contradiction in which we assume `d[i,j]` is smaller than the minimum of the three, and use this to show one of the three is not minimal.

Upper and lower bounds

The Levenshtein distance has several simple upper and lower bounds that are useful in applications which compute many of them and compare them. 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 identical.
- If the strings are the same size, the Hamming distance is an upper bound on the Levenshtein distance; otherwise the Hamming distance plus the difference in sizes is an upper bound.
- If the strings are called `s` and `t`, the number of characters found in `s` but not in `t` is a lower bound.

Sample implementations

Haskell

```
min3 :: Int->Int->Int->Int
min3 x y z = min x (min y z)


cost :: Char->Char->Int
cost a b
    | a == b = 0
    | otherwise = 1


sumCosts :: String->Char->Int
sumCosts [] a = 0
sumCosts (s:ss) a = cost s a + sumCosts ss a


editDistance :: String->String->Int
editDistance [] [] = 0
editDistance s [] = sumCosts s '-'
editDistance [] t = sumCosts t '-'
editDistance (s:ss) (t:ts) = min3 (cost s t + editDistance ss ts)
                                  (cost s '-' + editDistance ss (t:ts))
                                  (cost '-' t + editDistance (s:ss) ts)
```

Scheme

Uses srfi-25 and srfi-42

```
(define add1 (lambda (x) (+ x 1)))

(define levenshtein-distance
  (lambda (s1 s2)
    (let* ((width (add1 (string-length s1)))
           (height (add1 (string-length s2)))
           (d (do ((d (make-array (shape 0 height 0 width) 0))
                   (x 0 (add1 x)))
                  ((= x width)
                   (do ((y 0 (add1 y)))
                       ((= y height))
                     (array-set! d y 0 y))
                   d)
                (array-set! d 0 x x))))
      (do-ec (: x (string-length s1)) (: y (string-length s2))
             (begin
               (array-set!
                d (add1 y) (add1 x)
```

```
                (min
                 (add1 (array-ref d y (add1 x)))
                 (add1 (array-ref d (add1 y) x))
                 (+ (array-ref d y x)
                    (if (eqv? (string-ref s1 x)
                              (string-ref s2 y))
                        0
                        1))))))
      (array-ref d (sub1 height) (sub1 width)))))
```

External links

- Levenshtein Distance, in Three Flavors, by Michael Gilleland (http://www.merriampark.com/ld.htm)
- NIST's Dictionary of Algorithms and Data Structures: Levenshtein Distance (http://www.nist.gov/dads/HTML/Levenshtein.html)
- CSE 590BI, Winter 1996 Algorithms in Molecular Biology (http://www.cs.washington.edu/education/courses/590bi/96wi/) The algorithms from lectures 2, 3 and 4 are based on the Levenshtein distance but implement a different scoring function. The Haskell example was based on these notes.
- Levenshtein Distance - visualized (http://www-igm.univ-mlv.fr/~lecroq/seqcomp/node2.html)
- Distance between strings - Levenshtein and Hamming (http://www.cut-the-knot.org/do_you_know/Strings.shtml)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=10279684"