« lcfirst

- Manual de PHP
- Referencia de funciones
- Procesamiento de texto
- Strings
- Funciones de strings

Change language: Spanish

Submit a Pull Request Report a Bug

levenshtein

(PHP 4 >= 4.0.1, PHP 5, PHP 7, PHP 8)

levenshtein — Cálculo de la distancia Levenshtein entre dos strings

Descripción ¶

levenshtein(string `$str1`, string `$str2`): int
levenshtein(
    string `$str1`,
    string `$str2`,
    int `$cost_ins`,
    int `$cost_rep`,
    int `$cost_del`
): int

La distancia Levenshtein se define como el número mínimo de caracteres que se tienen que sustituir, insertar o borrar para transformar `str1` en `str2`. La complejidad del algoritmo es `O(m*n)`, donde `n` y `m` son la longitud de `str1` y `str2` (bastante bueno en comparación con similar_text(), la cual es O(max(n,m)**3), pero sigue siendo costoso).

En su forma más simple la función tomará sólo los dos strings como parámetros y calculará sólo el número de operaciones de insertar, reemplazar y eliminar, necesarias para transformar `str1` en `str2`.

Una segunda variante tomará tres parámetros adicionales que definen el costo de las operaciones de insertar, reemplazar y eliminar. Esto es más general y adaptable que la primera variante, pero no tan eficiente.

Parámetros ¶

`str1`

Uno de los strings a ser evaluados para la distancia Levenshtein.

`str2`

Uno de los strings a ser evaluados para la distancia Levenshtein.

`cost_ins`

Define el costo de inserción.

`cost_rep`

Define el costo de reemplazo.

`cost_del`

Define el costo de eliminación.

Valores devueltos ¶

Esta función devuelve la distancia Levenshtein entre los dos strings argumentos ó -1, si uno de los strings argumentos es mayor que el límite de 255 caracteres.

Ejemplos ¶

Ejemplo #1 Ejemplo de levenshtein()

```php
<?php
// palabra de entrada mal escrita
$input = 'carrrot';

// array de palabras contra las cuales verificar
$words  = array('apple','pineapple','banana','orange',
                'radish','carrot','pea','bean','potato');

// no se ha encontrado la distancia más corta, aun
$shortest = -1;

// bucle a través de las palabras para encontrar la más cercana
foreach ($words as $word) {

    // calcula la distancia entre la palabra de entrada
    // y la palabra actual
    $lev = levenshtein($input, $word);

    // verifica por una coincidencia exacta
    if ($lev == 0) {

        // la palabra más cercana es esta (coincidencia exacta)
        $closest = $word;
        $shortest = 0;

        // salir del bucle, se ha encontrado una coincidencia exacta
        break;
    }

    // si esta distancia es menor que la siguiente distancia
    // más corta o si una siguiente palabra más corta aun no se ha encontrado
    if ($lev <= $shortest || $shortest < 0) {
        // establece la coincidencia más cercana y la distancia más corta
        $closest  = $word;
        $shortest = $lev;
    }
}

echo "Input word: $input\n";
if ($shortest == 0) {
    echo "Exact match found: $closest\n";
} else {
    echo "Did you mean: $closest?\n";
}

?>
```

El resultado del ejemplo sería:

```
Input word: carrrot
Did you mean: carrot?
```

Ver también ¶

- soundex() - Calcula la clave soundex de un string
- similar_text() - Calcula la similitud entre dos strings
- metaphone() - Calcula la clave metaphone de un string

+ add a note

User Contributed Notes 26 notes

up
down
74
luciole75w at no dot spam dot gmail dot com ¶
8 years ago

The levenshtein function processes each byte of the input string individually. Then for multibyte encodings, such as UTF-8, it may give misleading results.

Example with a french accented word :
- levenshtein('notre', 'votre') = 1
- levenshtein('notre', 'nôtre') = 2 (huh ?!)

You can easily find a multibyte compliant PHP implementation of the levenshtein function but it will be of course much slower than the C implementation.

Another option is to convert the strings to a single-byte (lossless) encoding so that they can feed the fast core levenshtein function.

Here is the conversion function I used with a search engine storing UTF-8 strings, and a quick benchmark. I hope it will help.

```
<?php
// Convert an UTF-8 encoded string to a single-byte string suitable for
// functions such as levenshtein.
//
// The function simply uses (and updates) a tailored dynamic encoding
// (in/out map parameter) where non-ascii characters are remapped to
// the range [128-255] in order of appearance.
//
// Thus it supports up to 128 different multibyte code points max over
// the whole set of strings sharing this encoding.
//
function utf8_to_extended_ascii($str, &$map)
{
    // find all multibyte characters (cf. utf-8 encoding specs)
    $matches = array();
    if (!preg_match_all('/[\xC0-\xF7][\x80-\xBF]+/', $str, $matches))
        return $str; // plain ascii string

    // update the encoding map with the characters not already met
    foreach ($matches[0] as $mbc)
        if (!isset($map[$mbc]))
            $map[$mbc] = chr(128 + count($map));
```

```
    // finally remap non-ascii characters
    return strtr($str, $map);
}

// Didactic example showing the usage of the previous conversion function but,
// for better performance, in a real application with a single input string
// matched against many strings from a database, you will probably want to
// pre-encode the input only once.
//
function levenshtein_utf8($s1, $s2)
{
    $charMap = array();
    $s1 = utf8_to_extended_ascii($s1, $charMap);
    $s2 = utf8_to_extended_ascii($s2, $charMap);

    return levenshtein($s1, $s2);
}
?>

Results (for about 6000 calls)
- reference time core C function (single-byte) : 30 ms
- utf8 to ext-ascii conversion + core function : 90 ms
- full php implementation : 3000 ms
```

up
down
9
Johan Gennesson php at genjo dot fr ¶
6 years ago

```
Please, be aware that:

<?php
// Levenshtein Apostrophe (U+0027 &#39;) and Right Single Quotation Mark (U+2019 &#8217;)
echo levenshtein("'", "’");
?>

will output 3!
```

up
down
21
paulrowe at iname dot com ¶
14 years ago

```
[EDITOR'S NOTE: original post and 2 corrections combined into 1 -- mgf]

Here is an implementation of the Levenshtein Distance calculation that only uses a one-dimensional
array and doesn't have a limit to the string length. This implementation was inspired by maze
generation algorithms that also use only one-dimensional arrays.

I have tested this function with two 532-character strings and it completed in 0.6-0.8 seconds.

<?php
/*
* This function starts out with several checks in an attempt to save time.
*   1.  The shorter string is always used as the "right-hand" string (as the size of the array is
based on its length).
*   2.  If the left string is empty, the length of the right is returned.
*   3.  If the right string is empty, the length of the left is returned.
*   4.  If the strings are equal, a zero-distance is returned.
*   5.  If the left string is contained within the right string, the difference in length is
```

```
returned.
*   6.  If the right string is contained within the left string, the difference in length is
returned.
* If none of the above conditions were met, the Levenshtein algorithm is used.
*/
function LevenshteinDistance($s1, $s2)
{
  $sLeft = (strlen($s1) > strlen($s2)) ? $s1 : $s2;
  $sRight = (strlen($s1) > strlen($s2)) ? $s2 : $s1;
  $nLeftLength = strlen($sLeft);
  $nRightLength = strlen($sRight);
  if ($nLeftLength == 0)
    return $nRightLength;
  else if ($nRightLength == 0)
    return $nLeftLength;
  else if ($sLeft === $sRight)
    return 0;
  else if (($nLeftLength < $nRightLength) && (strpos($sRight, $sLeft) !== FALSE))
    return $nRightLength - $nLeftLength;
  else if (($nRightLength < $nLeftLength) && (strpos($sLeft, $sRight) !== FALSE))
    return $nLeftLength - $nRightLength;
  else {
    $nsDistance = range(1, $nRightLength + 1);
    for ($nLeftPos = 1; $nLeftPos <= $nLeftLength; ++$nLeftPos)
    {
      $cLeft = $sLeft[$nLeftPos - 1];
      $nDiagonal = $nLeftPos - 1;
      $nsDistance[0] = $nLeftPos;
      for ($nRightPos = 1; $nRightPos <= $nRightLength; ++$nRightPos)
      {
        $cRight = $sRight[$nRightPos - 1];
        $nCost = ($cRight == $cLeft) ? 0 : 1;
        $nNewDiagonal = $nsDistance[$nRightPos];
        $nsDistance[$nRightPos] =
          min($nsDistance[$nRightPos] + 1,
              $nsDistance[$nRightPos - 1] + 1,
              $nDiagonal + $nCost);
        $nDiagonal = $nNewDiagonal;
      }
    }
    return $nsDistance[$nRightLength];
  }
}
?>
```

up
down
9

engineglue at gmail dot com ¶

10 years ago

I really like [the manual's] example for the use of the levenshtein function to match against an array. I ran into the need to specify the sensitivity of the result. There are circumstances when you want it to return false if the match is way out of line. I wouldn't want "marry had a little lamb" to match with "saw viii" simply because it was the best match in the array. Hence the need for sensitivity:

```
<?php
    function wordMatch($words, $input, $sensitivity){
        $shortest = -1;
        foreach ($words as $word) {
```

```
            $lev = levenshtein($input, $word);
            if ($lev == 0) {
                $closest = $word;
                $shortest = 0;
                break;
            }
            if ($lev <= $shortest || $shortest < 0) {
                $closest  = $word;
                $shortest = $lev;
            }
        }
        if($shortest <= $sensitivity){
            return $closest;
        } else {
            return 0;
        }
    }

    $word = 'PINEEEEAPPLE';

    $words  = array('apple','pineapple','banana','orange',
                    'radish','carrot','pea','bean','potato');

    echo wordMatch($words, strtolower($word), 2);
?>
```

up
down
6
dale3h ¶
14 years ago

Using PHP's example along with Patrick's comparison percentage function, I have come up with a function that returns the closest word from an array, and assigns the percentage to a referenced variable:

```
<?php
  function closest_word($input, $words, &$percent = null) {
    $shortest = -1;
    foreach ($words as $word) {
      $lev = levenshtein($input, $word);

      if ($lev == 0) {
        $closest = $word;
        $shortest = 0;
        break;
      }

      if ($lev <= $shortest || $shortest < 0) {
        $closest  = $word;
        $shortest = $lev;
      }
    }

    $percent = 1 - levenshtein($input, $closest) / max(strlen($input), strlen($closest));

    return $closest;
  }
?>
```

Usage:

```
<?php
  $input = 'carrrot';
  $words = array('apple','pineapple','banana','orange',
                 'radish','carrot','pea','bean','potato');

  $percent = null;
  $found = closest_word($input, $words, $percent);

  printf('Closest word to "%s": %s (%s%% match)', $input, $found, round($percent * 100, 2));
?>
```

I found that lowercasing the array prior to comparing yields a better comparison when the case is not of importance, for example: comparing a user-inputted category to a list of existing categories.

I also found that when the percentage was above 75%, it was usually the match that I was looking for.

up
down
7

carey at NOSPAM dot internode dot net dot au ¶
16 years ago

I have found that levenshtein is actually case-sensitive (in PHP 4.4.2 at least).

```
<?php
$distance=levenshtein('hello','ELLO');
echo "$distance";
?>
```

Outputs: "5", instead of "1". If you are implementing a fuzzy search feature that makes use of levenshtein, you will probably need to find a way to work around this.

up
down
10

dschultz at protonic dot com ¶
22 years ago

It's also useful if you want to make some sort of registration page and you want to make sure that people who register don't pick usernames that are very similar to their passwords.

up
down
2

yhoko at yhoko dot com ¶
6 years ago

Note that this function might cause problems when working with multibyte charactes like in UTF-8. Example:

```
<?php
print( similar_text( 'hä', 'hà' ) ); // Returns 2 where only 1 character matches
?>
```

up
down
6

justin at visunet dot ie ¶
17 years ago

```
<?php

    /**************************************************************
    * The below func, btlfsa, (better than levenstien for spelling apps)
    * produces better results when comparing words like haert against
```

```
     * haart and heart.
     *
     * For example here is the output of levenshtein compared to btlfsa
     * when comparing 'haert' to 'herat, haart, heart, harte'
     *
     * btlfsa('haert','herat'); output is.. 3
     * btlfsa('haert','haart'); output is.. 3
     * btlfsa('haert','harte'); output is.. 3
     * btlfsa('haert','heart'); output is.. 2
     *
     * levenshtein('haert','herat'); output is.. 2
     * levenshtein('haert','haart'); output is.. 1
     * levenshtein('haert','harte'); output is.. 2
     * levenshtein('haert','heart'); output is.. 2
     *
     * In other words, if you used levenshtein, 'haart' would be the
     * closest match to 'haert'. Where as, btlfsa sees that it should be
     * 'heart'
     */

    function btlfsa($word1,$word2)
    {
        $score = 0;

        // For each char that is different add 2 to the score
        // as this is a BIG difference

        $remainder  = preg_replace("/[".preg_replace("/[^A-Za-z0-9\']/",'
',$word1)."]/i",'',$word2);
        $remainder .= preg_replace("/[".preg_replace("/[^A-Za-z0-9\']/",'
',$word2)."]/i",'',$word1);
        $score      = strlen($remainder)*2;

        // Take the difference in string length and add it to the score
        $w1_len  = strlen($word1);
        $w2_len  = strlen($word2);
        $score  += $w1_len > $w2_len ? $w1_len - $w2_len : $w2_len - $w1_len;

        // Calculate how many letters are in different locations
        // And add it to the score i.e.
        //
        // h e a r t
        // 1 2 3 4 5
        //
        // h a e r t     a e        = 2
        // 1 2 3 4 5   1 2 3 4 5
        //

        $w1 = $w1_len > $w2_len ? $word1 : $word2;
        $w2 = $w1_len > $w2_len ? $word2 : $word1;

        for($i=0; $i < strlen($w1); $i++)
        {
            if ( !isset($w2[$i]) || $w1[$i] != $w2[$i] )
            {
                $score++;
            }
        }
```

```
        return $score;
    }

    // *********************************************************
    // Here is a full code example showing the difference

    $misspelled = 'haert';

    // Imagine that these are sample suggestions thrown back by soundex or metaphone..
    $suggestions = array('herat', 'haart', 'heart', 'harte');

    // Firstly order an array based on levenshtein
    $levenshtein_ordered = array();
    foreach ( $suggestions as $suggestion )
    {
        $levenshtein_ordered[$suggestion] = levenshtein($misspelled,$suggestion);
    }
    asort($levenshtein_ordered, SORT_NUMERIC );

    print "<b>Suggestions as ordered by levenshtein...</b><ul><pre>";
    print_r($levenshtein_ordered);
    print "</pre></ul>";

    // Secondly order an array based on btlfsa
    $btlfsa_ordered = array();
    foreach ( $suggestions as $suggestion )
    {
        $btlfsa_ordered[$suggestion] = btlfsa($misspelled,$suggestion);
    }
    asort($btlfsa_ordered, SORT_NUMERIC );

    print "<b>Suggestions as ordered by btlfsa...</b><ul><pre>";
    print_r($btlfsa_ordered);
    print "</pre></ul>";

?>
```

up
down
4

Chaim Chaikin ¶

11 years ago

As regards to Example #1 above, would it not be more efficient to first use a simple php == comparison to check if the strings are equal even before testing the word with levenshtein().

Something like this:

```
<?php
// input misspelled word
$input = 'carrrot';

// array of words to check against
$words  = array('apple','pineapple','banana','orange',
                'radish','carrot','pea','bean','potato');

// no shortest distance found, yet
$shortest = -1;

// loop through words to find the closest
foreach ($words as $word) {
```

```
    // check for an exact match
    if ($input == $word) {

        // closest word is this one (exact match)
        $closest = $word;
        $shortest = 0;

        // break out of the loop; we've found an exact match
        break;
    }

    // calculate the distance between the input word,
    // and the current word
    $lev = levenshtein($input, $word);

    // if this distance is less than the next found shortest
    // distance, OR if a next shortest word has not yet been found
    if ($lev <= $shortest || $shortest < 0) {
        // set the closest match, and shortest distance
        $closest  = $word;
        $shortest = $lev;
    }
}

echo "Input word: $input\n";
if ($shortest == 0) {
    echo "Exact match found: $closest\n";
} else {
    echo "Did you mean: $closest?\n";
}

?>
```

up
down
1

june05 at tilo-hauke dot de ¶
17 years ago

```
//levenshtein for arrays
function array_levenshtein($array1,$array2)
        {   $aliases= array_flip(array_values(array_unique(array_merge($array1,$array2))));
            if(count($aliases)>255) return -1;
            $stringA=''; $stringB='';
            foreach($array1 as $entry) $stringA.=chr($aliases[$entry]);
            foreach($array2 as $entry) $stringB.=chr($aliases[$entry]);
            return levenshtein($stringA,$stringB);
        }

// e.g. use array_levenshtein to detect special expressions in unser-inputs

echo array_levenshtein(split(" ", "my name is xxx"), split(" ","my name is levenshtein"));

//output: 1
```

up
down
1

fgilles at free dot fr ¶
21 years ago

Exempla of use for a forum: users can't post messages too much uppercased

```
<?php
if ((strlen($subject)>10) and ( ( levenshtein ($subject, strtolower ($subject) / strlen ($subject)
) > .3 ) ){
    $subject = strtolower($subject);
}
?>
```

up
down
3
"inerte" is my hotmail.com username ¶
19 years ago

I am using this function to avoid duplicate information on my client's database.

After retrieving a series of rows and assigning the results to an array values, I loop it with foreach comparing its levenshtein() with the user supplied string.

It helps to avoid people re-registering "John Smith", "Jon Smith" or "Jon Smit".

Of course, I can't block the operation if the user really wants to, but a suggestion is displayed along the lines of: "There's a similar client with this name.", followed by the list of the similar strings.

up
down
3
WiLDRAGoN ¶
7 years ago

Some small changes allow you to calculate multiple words.

```
<?php

$input = array();
$dictionary  = array();
foreach ($input as $output) {
    $shortest = -1;
    foreach ($dictionary as $word) {
        $lev = levenshtein($output, $word);
        if ($lev == 0) {
            $closest = $word;
            $shortest = 0;
        }
        if ($lev <= $shortest || $shortest < 0) {
            $closest  = $word;
            $shortest = $lev;
        }
    }
    echo "Input word: $output\n";
    if ($shortest == 0) {
        echo "Exact match found: $closest\n";
    } else {
        echo "Did you mean: $closest?\n";
    }
}

?>
```

up
down
1

atx dot antrax at gmail dot com ¶

14 years ago

I have made a function that removes the length-limit of levenshtein function and ajust the result with similar_text:

```
<?php
function _similar($str1, $str2) {
    $strlen1=strlen($str1);
    $strlen2=strlen($str2);
    $max=max($strlen1, $strlen2);

    $splitSize=250;
    if($max>$splitSize)
    {
        $lev=0;
        for($cont=0;$cont<$max;$cont+=$splitSize)
        {
            if($strlen1<=$cont || $strlen2<=$cont)
            {
                $lev=$lev/($max/min($strlen1,$strlen2));
                break;
            }
            $lev+=levenshtein(substr($str1,$cont,$splitSize), substr($str2,$cont,$splitSize));
        }
    }
    else
    $lev=levenshtein($str1, $str2);

    $porcentage= -100*$lev/$max+100;
    if($porcentage>75)//Ajustar con similar_text
    similar_text($str1,$str2,$porcentage);

    return $porcentage;
}
?>
```

up
down
2

bisqwit at iki dot fi ¶

20 years ago

At the time of this manual note the user defined thing
in levenshtein() is not implemented yet. I wanted something
like that, so I wrote my own function. Note that this
doesn't return levenshtein() difference, but instead
an array of operations to transform a string to another.

Please note that the difference finding part (resync)
may be extremely slow on long strings.

```
<?php

/* matchlen(): returns the length of matching
* substrings at beginning of $a and $b
*/
function matchlen(&$a, &$b)
{
  $c=0;
  $alen = strlen($a);
  $blen = strlen($b);
```

```
    $d = min($alen, $blen);
    while($a[$c] == $b[$c] && $c < $d)
      $c++;
    return $c;
  }

  /* Returns a table describing
  * the differences of $a and $b */
  function calcdiffer($a, $b)
  {
    $alen = strlen($a);
    $blen = strlen($b);
    $aptr = 0;
    $bptr = 0;

    $ops = array();

    while($aptr < $alen && $bptr < $blen)
    {
      $matchlen = matchlen(substr($a, $aptr), substr($b, $bptr));
      if($matchlen)
      {
        $ops[] = array('=', substr($a, $aptr, $matchlen));
        $aptr += $matchlen;
        $bptr += $matchlen;
        continue;
      }
      /* Difference found */

      $bestlen=0;
      $bestpos=array(0,0);
      for($atmp = $aptr; $atmp < $alen; $atmp++)
      {
        for($btmp = $bptr; $btmp < $blen; $btmp++)
        {
          $matchlen = matchlen(substr($a, $atmp), substr($b, $btmp));
          if($matchlen>$bestlen)
          {
            $bestlen=$matchlen;
            $bestpos=array($atmp,$btmp);
          }
          if($matchlen >= $blen-$btmp)break;
        }
      }
      if(!$bestlen)break;

      $adifflen = $bestpos[0] - $aptr;
      $bdifflen = $bestpos[1] - $bptr;

      if($adifflen)
      {
        $ops[] = array('-', substr($a, $aptr, $adifflen));
        $aptr += $adifflen;
      }
      if($bdifflen)
      {
        $ops[] = array('+', substr($b, $bptr, $bdifflen));
        $bptr += $bdifflen;
      }
```

```
    $ops[] = array('=', substr($a, $aptr, $bestlen));
    $aptr += $bestlen;
    $bptr += $bestlen;
  }
  if($aptr < $alen)
  {
    /* b has too much stuff */
    $ops[] = array('-', substr($a, $aptr));
  }
  if($bptr < $blen)
  {
    /* a has too little stuff */
    $ops[] = array('+', substr($b, $bptr));
  }
  return $ops;
}


Example:

$tab = calcdiffer('T?m? on jonkinlainen testi',
                  'T?m? ei ole mink??nlainen testi.');
$ops = array('='=>'Ok', '-'=>'Remove', '+'=>'Add');
foreach($tab as $k)
  echo $ops[$k[0]], " '", $k[1], "'\n";

Example output:

Ok 'T?m? '
Remove 'on jonki'
Add 'ei ole mink??'
Ok 'nlainen testi'
Add '.'
```

up
down
1
mcreuzer at r-world dot com ¶
17 years ago

```
I am using the Levenshtein distance to SORT my search results.

I have a search page for peoples names. I do a SOUNDEX() search on the name in mysql. MySQL
SOUNDEX() will perform the "fuzzy" search for me.

I then calculate the Levenshtein distance between the search term and the actual name found by the
SOUNDEX() search. This will give me a score on how close my results are to the search string.

I can the sort my results for display listing the closest results first.

<?php
// PHP CODE INCLUDING DB LOOKUPS HERE
    usort($searchresults, "finallevenshteinsortfunction");

function finallevenshteinsortfunction($a, $b)
{
    if(($a['levenshtein'] > $b['levenshtein']) || ( $a['levenshtein'] == $b['levenshtein'] &&
strnatcasecmp( $a['Last_Name'], $b['Last_Name']) >= 1) ){ return $a['levenshtein'];} // Ok... The
levenstein is greater OR with the same levenshtein, the last name is alphanumerically first
    elseif($a['levenshtein'] == $b['levenshtein']){ return '0';} // The levenstein matches
    elseif($a['levenshtein'] < $b['levenshtein']){ return -$a['levenshtein'];}
```

```
    else{die("<!-- a horrable death -->");}
}
?>
```

up
down
0

peratik at gmail dot com ¶
5 years ago

python get_close_matches equivalent:

```
function get_close_matches($str, $arr) {
    $closest = 1000;
    $word = false;
    foreach($arr as $w) {
        $po = levenshtein($str, $w);
        if ($po<$closest) {
            $closest = $po;
            $word = $w;
        }
    }
    return $word;
}
```

up
down
-1

qbolec ¶
7 years ago

For the rare occassions, where you want:
1. multibyte UTF-8 characters
2. linear memory consumption (that is O(n+m) , not O(n*m))
3. learn the string which is the longest common subsequence
4. reasonable (that is O(n*m)) time complexity
Consider this implementation:

```
<?php
class Strings
{
  public static function len($a){
    return mb_strlen($a,'UTF-8');
  }
  public static function substr($a,$x,$y=null){
    if($y===NULL){
      $y=self::len($a);
    }
    return mb_substr($a,$x,$y,'UTF-8');
  }
  public static function letters($a){
    $len = self::len($a);
    if($len==0){
      return array();
    }else if($len == 1){
      return array($a);
    }else{
      return Arrays::concat(
        self::letters(self::substr($a,0,$len>>1)),
        self::letters(self::substr($a,$len>>1))
      );
    }
  }
  private static function lcs_last_column(array $A,array $B){
```

```
    $al=count($A);
    $bl=count($B);
    $last_column = array();
    for($i=0;$i<=$al;++$i){
      $current_row = array();
      for($j=0;$j<=$bl;++$j){
        //$a[0,$i) vs $b[0,$j)
        if($i==0 || $j == 0){
          $v = 0;
        }else if($A[$i-1]===$B[$j-1]){
          $v = 1 + $last_row[$j-1];
        }else{
          $v = max($last_row[$j],$current_row[$j-1]);
        }
        $current_row[] = $v;
      }
      $last_column[] = $current_row[$bl];
      $last_row = $current_row;
    }
    return $last_column;
  }
  public static function lcs($a,$b){
    $A = self::letters($a);
    $B = self::letters($b);
    $bl=count($B);
    if($bl==0){
      return '';
    }else if($bl==1){
      return FALSE===array_search($B[0],$A,true)?'':$B[0];
    }
    $left=self::lcs_last_column($A,array_slice($B,0,$bl>>1));

$right=array_reverse(self::lcs_last_column(array_reverse($A),array_reverse(array_slice($B,$bl>>1))));

    $best_i = 0;
    $best_lcs = 0;
    foreach($left as $i => $lcs_left){
      $option = $lcs_left + $right[$i];
      if($best_lcs < $option){
        $best_lcs = $option;
        $best_i = $i;
      }
    }
    return
      self::lcs(self::substr($a,0,$best_i), self::substr($b,0,$bl>>1)).
      self::lcs(self::substr($a,$best_i), self::substr($b,$bl>>1));
  }
?>
```

This is a classic implentation in which several tricks are used:
1. the strings are exploded into multi-byte characters in O(n lg n) time
2. instead of searching for the longest path in a precomputed two-dimensional array, we search for the best point which lays in the middle column. This is achieved by spliting the second string in half, and recursively calling the algorithm twice. The only thing we need from the recursive call are the values in the middle column. The trick is to return the last column from each recursive call, which is what we need for the left part, but requires one more trick for the right part - we simply mirror the strings and the array so that the last column is the first column. Then we just find the row which maximizes the sum of lenghts in each part.
3. one can prove that the time consumed by the algorithm is proportional to the area of the (imaginary) two-dimensional array, thus it is O(n*m).

up
down
0
gzink at zinkconsulting dot com ¶
18 years ago
Try combining this with metaphone() for a truly amazing fuzzy search function. Play with it a bit, the results can be plain scary (users thinking the computer is almost telepathic) when implemented properly. I wish spell checkers worked as well as the code I've written.

I would release my complete code if reasonable, but it's not, due to copyright issues. I just hope that somebody can learn from this little tip!
up
down
0
genialbrainmachine at dot IHATESPAM dot tiscali dot it ¶
19 years ago
I wrote this function to have an "intelligent" comparison between data to be written in a DB and already existent data. Not ony calculating distances but also balancing distances for each field.

```
<?php
/*
This function calculate a balanced percentage distance between an array of strings
"$record" and a compared array "$compared", balanced through an array of
weights "$weight". The three arrays must have the same indices.
For an unbalanced distance, set all weights to 1.
The used formula is:
percentage distance = sum(field_levenshtein_distance * field_weight) / sum(record_field_length *
field_weight) * 100
*/
function search_similar($record, $weights, $compared, $precision=2) {
         $field_names = array_keys($record);
         # "Weighted length" of $record and "weighted distance".
         foreach ($field_names as $field_key) {
                 $record_weight += strlen($record[$field_key]) * $weights[$field_key];
                 $weighted_distance += levenshtein($record[$field_key],$compared[$field_key]) *
$weights[$field_key];
                 }
         # Building the result..
         if ($record_weight) {
             return round(($weighted_distance / $record_weight * 100),$precision);
            } elseif ((strlen(implode("",$record)) == 0) && (strlen(implode("",$compared)) == 0)) {
// empty records
            return round(0,$precision);
            } elseif (array_sum($weights) == 0) { // all weights == 0
            return round(0,$precision);
            } else {
            return false;
            }
         /*
         Be very careful distinguising 0 result and false result.
         The function results 0 ('0.00' if $precision is 2 and so on) if:
         - $record and $compared are equals (even if $record and $compared are empty);
         - all weights are 0 (the meaning could be "no care about any field").
         Conversely, the function results false if $record is empty, but the weights
         are not all 0 and $compared is not empty. That cause a "division by 0" error.
         I wrote this kind of check:

         if ($rel_dist = search_similar(...)) {
             print $rel_dist;
```

```
            } elseif ($rel_dist == "0.00") { // supposing that $precision is 2
            print $rel_dist;
            } else {
            print "infinite";
            }

         */
         }
?>
```

up
down
0

jlsalinas at gmx dot net ¶

19 years ago

Regarding the post by fgilles on April 26th 2001, I suggest not to use levenshtein() function to test for over-uppercasing unless you've got plenty of time to waste in your host. ;) Anyhow, I think it's a useful feature, as I get really annoyed when reading whole messages in uppercase.

PHP's levenshtein() function can only handle up to 255 characters, which is not realistic for user input (only the first paragraph oh this post has 285 characters). If you choose to use a custom function able to handle more than 255 characters, efficiency is an important issue.

I use this function, specific for this case, but much faster:

```
function ucase_percent ($str) {
    $str2 = strtolower ($str);

    $l = strlen ($str);
    $ucase = 0;

    for ($i = 0; $i < $l; $i++) {
        if ($str{$i} != $str2{$i}) {
            $ucase++;
        }
    }

    return $ucase / $l * 100.0;
}
```

I think 10% is enough for written English (maybe other languages like German, which use more capital letters, need more). With some sentencies in uppercase (everybody has the right to shout occasionally), 20% would be enough; so I use a threshold of 30%. When exceeded, I lowercase the whole message.

Hope you find it useful and it helps keeping the web free of ill-mannered people.

up
down
0

Anonymous ¶

20 years ago

For spell checking applications, delay could be tolerable if you assume the typist got the first two or three chars of each word right. Then you'd only need to calc distances for a small segment of the dictionary. This is a compromise but one I think a lot of spell checkers make.
For an example of site search using this function look at the PHP manual search button on this page. It appears to be doing this for the PHP function list.

up
down
0

ad1n at dc dot uba dot ar ¶

22 years ago

One application of this is when you want to look for a similar match instead of an exact one. You can sort the results of checking the distances of a word to a dictionary and sort them to see which were the more similar ones. Of course it will be a quite resourse consuming task anyway.

up
down
-1

dinesh AT dinsoft DOT net ¶

16 years ago

Here is a string resynch function:

```
<?php
// Trouve les operations a effectuer pour modifier $b en $a en exploitant leurs similitudes (Finds the operations required to change $b to $a)
// Identique a la fonction Resynch Compare de Hex Workshop
//
// Parametres:
//      $a Premiere chaine (cible, target)
//      $b Seconde chaine (source)
//      $l Nombre d'octets devant correspondre pour etre consides comme un bloc similaire (number of matching bytes required)
//      $s Distance maximale dans laquelle les blocs similaires sont cherches (search window)
//
// Retourne:
//      Array
//              Array
//                      [0] Operation: + Add , - Del , / Replace, = Match
//                      [1] Source offset
//                      [2] Source count
//                      [3] Target offset
//                      [4] Target count
//
function str_resynch($a,$b,$l=32,$s=2048) {
        $r=array();
        for($i=0,$c=strlen($a),$cc=strlen($b),$ii=0,$z=$s-1,$z2=($z<<1)+1; $i<$c; $i++) {
                $d=$i-$z;
                $d=($d<$ii)?substr($b,$ii,$z2-$ii+$d):substr($b,$d,$z2);

                $p=strpos($d,$a{$i});
                $n=0;
                while ($p!==FALSE) {
                        $m=1;
                        $bi=$i;
                        $bp=$p;
                        $p+=$ii;
                        while ((++$i<$c) && (++$p<$cc)) {
                                if ($a{$i}!=$b{$p}) break;
                                $m++;
                        }
                        if ($m<$l) {
                                $i=$bi;
                                $n=$bp+1;
                                $p=@strpos($d,$a{$i},$n);
                        }
                        else {
                                $i--;
                                $r[]=array($bi,$bp+$ii,$m); // offset a, offset b, Count
                                $ii=$p;
                                break;
```

```
                        }
                }
        }

        if (!count($r)) return ($cc)?array('/',0,$c,0,$cc):array(array('+',0,$c,0,0));

        $o=array();
        $bi=0;
        $bp=0;
        for($i=0,$m=count($r);$i<$m;$i++) {
                if ($r[$i][0]!=$bi) {
                        if ($r[$i][1]!=$bp) {
                                // Replace
                                $o[]=array('/',$bi,$r[$i][0]-$bi,$bp,$r[$i][1]-$bp);
                                $bi=$r[$i][0];
                                $bp=$r[$i][1];
                        }
                        else {
                                // Insertion
                                $o[]=array('+',$bi,$r[$i][0]-$bi,$bp,0);
                                $bi=$r[$i][0];
                        }
                }
                elseif ($r[$i][1]!=$bp) {
                        // Delete
                        $o[]=array('-',$bi,0,$bp,$r[$i][1]-$bp);
                        $bp=$r[$i][1];
                }

                // Match
                $o[]=array('=',$r[$i][0],$r[$i][2],$r[$i][1],$r[$i][2]);
                $bi+=$r[$i][2];
                $bp+=$r[$i][2];
        }

        if ($c!=$bi) {
                if ($cc!=$bp) $o[]=array('/',$bi,$c-$bi,$bp,$cc-$bp);
                else $o[]=array('+',$bi,$c-$bi,$bp,0);
        }
        elseif ($cc!=$bp) $o[]=array('-',$bi,0,$bp,$cc-$bp);

        return $o;
}
?>
```

up
down
-1
luka8088 at gmail dot com ¶
14 years ago

Simple levenshtein function without string length limit ...

```
<?php

function levenshtein2($str1, $str2, $cost_ins = null, $cost_rep = null, $cost_del = null) {
    $d = array_fill(0, strlen($str1) + 1, array_fill(0, strlen($str2) + 1, 0));
    $ret = 0;

    for ($i = 1; $i < strlen($str1) + 1; $i++)
        $d[$i][0] = $i;
```

```
    for ($j = 1; $j < strlen($str2) + 1; $j++)
        $d[0][$j] = $j;

    for ($i = 1; $i < strlen($str1) + 1; $i++)
        for ($j = 1; $j < strlen($str2) + 1; $j++) {
            $c = 1;
            if ($str1{$i - 1} == $str2{$j - 1})
                $c = 0;
            $d[$i][$j] = min($d[$i - 1][$j] + 1, $d[$i][$j - 1] + 1, $d[$i - 1][$j - 1] + $c);
            $ret = $d[$i][$j];
        }

    return $ret;
}

?>
```

+ add a note

- Funciones de strings
  - addcslashes
  - addslashes
  - bin2hex
  - chop
  - chr
  - chunk_split
  - convert_uudecode
  - convert_uuencode
  - count_chars
  - crc32
  - crypt
  - echo
  - explode
  - fprintf
  - get_html_translation_table
  - hebrev
  - hex2bin
  - html_entity_decode
  - htmlentities
  - htmlspecialchars_decode
  - htmlspecialchars
  - implode
  - join
  - lcfirst
  - levenshtein
  - localeconv
  - ltrim
  - md5_file
  - md5
  - metaphone
  - money_format
  - nl_langinfo
  - nl2br
  - number_format
  - ord
  - parse_str
  - print
  - printf
  - quoted_printable_decode
  - quoted_printable_encode

- quotemeta
- rtrim
- setlocale
- sha1_file
- sha1
- similar_text
- soundex
- sprintf
- sscanf
- str_contains
- str_ends_with
- str_getcsv
- str_ireplace
- str_pad
- str_repeat
- str_replace
- str_rot13
- str_shuffle
- str_split
- str_starts_with
- str_word_count
- strcasecmp
- strchr
- strcmp
- strcoll
- strcspn
- strip_tags
- stripcslashes
- stripos
- stripslashes
- stristr
- strlen
- strnatcasecmp
- strnatcmp
- strncasecmp
- strncmp
- strpbrk
- strpos
- strrchr
- strrev
- strripos
- strrpos
- strspn
- strstr
- strtok
- strtolower
- strtoupper
- strtr
- substr_compare
- substr_count
- substr_replace
- substr
- trim
- ucfirst
- ucwords
- utf8_decode
- utf8_encode
- vfprintf
- vprintf

- vsprintf
- wordwrap

- Deprecated
  - convert_cyr_string
  - hebrevc

- My PHP.net
- Contact
- Other PHP.net sites
- Privacy policy
- View Source