An Algorithm for Computing the Invariant Distance from Word Position

Sergio Luján-Mora[1]

Departamento de Lenguajes y Sistemas Informáticos, Universidad de Alicante,
Campus de San Vicente del Raspeig,
Ap. Correos 99 – E-03080 Alicante, Spain
{slujan, mpalomar}@dlsi.ua.es

Abstract. There are many problems involving string matching. The string matching bases in a number of similarity or distance measures, and many of them are special cases or generalisations of the Levenshtein distance. In this paper, we focus on the problem of evaluating an invariant distance from word position between two strings: e.g., the strings "Universidad de Alicante" and "Alicante, Universidad de" refer to the same concept, so the distance ought to be low, in order to consider them as very similar strings.

Keywords: String Matching, String Similarity, Approximate String Matching, Edit Distance, Pattern Matching

[1] Phone: (+34) 965 90 34 00 ext. 2514   Fax: (+34) 965 90 93 26

An Algorithm for Computing the Invariant Distance from Word Position

Sergio Luján-Mora

Departamento de Lenguajes y Sistemas Informáticos, Universidad de Alicante,
Campus de San Vicente del Raspeig,
Ap. Correos 99 – E-03080 Alicante, Spain
{slujan, mpalomar}@dlsi.ua.es

Abstract. There are many problems involving string matching. The string matching bases in a number of similarity or distance measures, and many of them are special cases or generalisations of the Levenshtein distance. In this paper, we focus on the problem of evaluating an invariant distance from word position between two strings: e.g., the strings "Universidad de Alicante" and "Alicante, Universidad de" refer to the same concept, so the distance ought to be low, in order to consider them as very similar strings.

1 Introduction

The problem of determining the differences between two sequences of symbols has been studied extensively [1, 3, 4, 5, 6, 10, 12]. Algorithms for the problem have numerous applications, including approximate string matching, spelling error detection and correction system, phonetic string matching, file comparison tools, and the study of genetic evolution [7, 9, 13, 14].

The string matching bases in a number of similarity or distance measures, and many of them are special cases or generalisations of the Levenshtein distance [8]. This distance is not useful when we want to compute the similarity between two strings of words: although edit distance is robust to spelling variants, it can be completely useless when permutations of words occur.

Consider the seven strings in Table 1. They belong in three classes, $\{s_1, s_2, s_3, s_4\}$, $\{s_5, s_6\}$ and $\{s_7\}$. Each of these classes represents a separate institution and we would like to consider them as similar strings. In Section 2, we present the Levenshtein distance and show how this distance is useless when a permuted word order exists. In Section 3 we outline the *invariant distance from word position* as a mechanism for overcoming the problem.

Table 1. Strings

	Length
s_1: Universidad de Alicante	23
s_2: Universitat d'Alacant	21
s_3: University of Alicante	22
s_4: Alicante University	19

s_5: Ciencias, Universidad de Valencia	33
s_6: Universitat de València, Ciències	33
s_7: Universidad Politécnica de Valencia	35

2 String Similarity

The similarity between any two strings can be evaluated by the *edit distance* or *Levenshtein distance* (LD) [8]. This distance has been traditionally used in approximate-string searching and spelling error detection and correction. The LD of strings x and y is defined as the minimal number of simple editing operations that are required to transform x into y. The simple editing operations considered are: the insertion of a character, the deletion of a character, and the substitution of one character with another, with costs $\delta(\lambda, \sigma)$, $\delta(\sigma, \lambda)$, and $\delta(\sigma_1, \sigma_2)$, that are functions of the involved character(s). Extended Levenshtein distances also consider transposing two adjacent characters. In the examples of this paper, we have taken a unitary cost function for all the operations and for all of the characters.

The LD of two strings m and n in length, respectively, can be calculated by a dynamic programming algorithm. The algorithm requires $\Theta(mn)$ time and space, although refinements of the algorithm require only Θ(ND) time and space [11], where N is the sum of m and n and D is the distance between the strings.

The Levenshtein distance between $x[1{:}m]$ and $y[1{:}n]$, $LD(m, n)$, can be calculated by the next recurrence relation [5]:

$$LD(i,j) = \begin{cases} 0, & \text{if } i=0 \text{ and } j=0 \\ LD(0,j-1)+\delta(\lambda,y_j), & \text{if } i=0 \text{ and } j>0 \\ LD(i-1,0)+\delta(x_i,\lambda), & \text{if } i>0 \text{ and } j=0 \\ LD(i-1,j-1), & \text{if } x_i = y_j \\ min\begin{Bmatrix} LD(i,j-1)+\delta(\lambda,y_j), \\ LD(i-1,j)+\delta(x_i,\lambda), \\ LD(i-1,j-1)+\delta(x_i,y_j) \end{Bmatrix}, & \text{if } x_i \neq y_j \end{cases} \tag{1}$$

Table 2 shows the pairwise edit distances[2] (LD) for the strings of Table 1. As it is a symmetric matrix, the lower part of the matrix is not presented. As we can see, the strings s_1, s_2, and s_3 (*type of institution*, *name*) have a low LD between them: 6 (s_1-s_2), 5 (s_1-s_3), and 7 (s_2-s_3). However, if we consider the previous strings and the string s_4 (*name*, *type of institution*), the distances between them increase very much: 19 (s_4-s_1),

[2] All the strings are converted to lower case before the distance is calculated.

18 (s_4-s_2), and 18 (s_4-s_3). The difference is due to word permutation: the four strings represent the same institution and have similar words, but the string s_4 has a different word order. If two strings contain the same words but with a permuted word order (variant forms of the same term), the LD will be large. Therefore, the LD is not a useful distance when a permuted word order exists.

Table 2. LD matrix

	s_1	s_2	s_3	s_4	s_5	s_6	s_7
s_1	0	6	5	19	17	14	19
s_2		0	7	18	20	15	21
s_3			0	18	22	16	21
s_4				0	23	25	28
s_5					0	22	16
s_6						0	19
s_7							0

3 Invariant Distance from Word Position

The problem we have shown in the previous section can be resolved if we use a different representation of the strings. The conventional representation of a string is a sequence of characters. A more useful representation is to think of a string as a set of words, where each word is a sequence of characters (letters and digits). Using this representation, the *invariant distance from word position* (IDWP) is calculated. This distance is based on the *approximate word matching* referred to in [2].

In Table 3 we can see the strings shown in Table 1 broken up into words. The words have been converted to lower case and the punctuation (commas and apostrophes) has also been removed.

Table 3. Strings

	Number of words	Words
s_1: Universidad de Alicante	3	universidad, de, alicante
s_2: Universitat d'Alacant	3	universitat, d, alacant
s_3: University of Alicante	3	university, of, alicante
s_4: Alicante University	2	alicante, university
s_5: Ciencias, Universidad de Valencia	4	ciencias, universidad, de, valencia
s_6: Universitat de València, Ciències	4	universitat, de, valencia, ciencies
s_7: Universidad Politécnica de Valencia	4	universidad, politecnica, de, valencia

3.1 The algorithm

To calculate the IDWP of two strings, they are broken up into words. The idea is to pair up the words so that the sum of the LD is minimized. If the strings contain different number of words, the cost of each word in excess is the length of the word.

It is almost always the case that *IDWP(x, y) < LD(x, y)*, although this is not always true. For instance, for the strings "*abc def*" and "*a bcd ef*", the LD and IDWP values are 3 and 4 respectively.

In Table 4 we show the IDWP algorithm. Basically the core of the algorithm is the recursive method *matching*, that is shown in Table 5. This method calculates the best matching of the words by means of a *branch and bound* scheme.

Table 4. IDWP algorithm

```
Input:
S: Array of words (s1...sm)
m: Integer
T: Array of words (t1...tn)
n: Integer
Output:
Idwp: Integer
Variables:
i, j, aux: Integer
D: Matrix (m + 1) x (n + 1) of Integer

* Fulfil D with zeros:
For i = 0 To m
  For j = 0 To n
    D[i][j] = 0
  Next
Next

* Fulfil the first column ant the first row of D with the length of si and ti
respectively:
For i = 1 To m
  D[i][0] = |si|
Next
For j = 1 To n
  D[0][j] = |tj|
Next

* Mark the words that matches between them:
For i = 1 To m
  For j = 1 To n
    If D[i][j] = 0 Then
      For aux = 0 To m
        D[aux][j] = -1
      Next
```

```
      For aux = 0 To n
        D[i][aux] = -1
      Next
    End If
  Next
Next

* Call the matching method:
Idwp = -1
matching(D, m, n, 1, 0, Idwp)
```

Table 5. Matching method

```
Input:
D: Matrix (m + 1) x (n + 1) of Integer
m: Integer
n: Integer
i: Integer
cost: Integer
Idwp: Integer
Output:
Idwp: Integer
Variables:
j, aux: Integer

If  i <= m Then
  If D[i][0] <> -1 Then

* The word i of S has not yet been matched:
    For j = 1 To n

* If the word j of T has not yet been matched:
      If D[i][j] <> -1 And D[0][j] > 0 Then
        aux = cost + D[i][j]
        If aux < Idwp Or Idwp = -1 Then

* Mark the word j of T as matched:
          D[0][j] = -D[0][j]
          matching(D, m, n, i + 1, aux, Idwp)

* Remove the mark of word j of T:
          D[0][j] = -D[0][j]
        End If
      End If
    Next

* The method consider the word i of S is not matched, so its length is
```

```
added to the final distance:
        aux = cost + D[i][0]
        If aux < Idwp Or Idwp = -1 Then
          matching(D, m, n, i + 1, aux, Idwp)
        End If
      Else

    * The word i of S has already been matched, the method goes on to the
next word:
        matching(D, m, n, i + 1, cost, Idwp)
      End If
    Else

    * All the words of S have been matched; if any word of T has not been
matched, its length is added to the final distance:
      aux = cost
      For j = 1 To n
        If D[0][j] > 0 Then
          aux = aux + D[0][j]
        End If
      Next
      If aux < Idwp Or Idwp = -1 Then
        Idwp = aux
      End If
    End If
```

Table 5 shows the pairwise IDWP for the strings of Table 1. As it is a symmetric matrix, the lower part of the matrix is not presented.

Table 5. IDWP matrix

	s_1	s_2	s_3	s_4	s_5	s_6	s_7
s_1	0	5	5	5	15	17	16
s_2		0	6	5	17	15	20
s_3			0	2	20	19	21
s_4				0	20	19	21
s_5					0	3	8
s_6						0	10
s_7							0

In Table 6 is shown LD matrix versus IDWP matrix. As we can see, the greater changes between this two distances are between s_4 and $\{s_1, s_2, s_3\}$ and between s_5 and s_6. The IDWP between the strings of $\{s_1, s_2, s_3, s_4\}$ and $\{s_5, s_6\}$ are more homogenous than LD. IDWP is lower or equal than LD in all the cases, except s_1-s_6 and s_3-s_6.

Table 6. LD matrix vs. IDWP matrix

	s_1	s_2	s_3	s_4	s_5	s_6	s_7

s_1	0	6 / 5	5 / 5	19 / 5	17 / 15	14 / 17	19 / 16
s_2		0	7 / 6	18 / 5	20 / 17	15 / 15	21 / 20
s_3			0	18 / 2	22 / 20	16 / 19	21 / 21
s_4				0	23 / 20	25 / 19	28 / 21
s_5					0	22 / 3	16 / 8
s_6						0	19 / 10
s_7							0

References

1. A.V. Aho, D.S. Hirschberg, J.D. Ullman. Bounds on the Complexity of the Longest Common Subsequence Problem. *Journal of the ACM*, 23(1):1-12, 1976.
2. J.C. French, A.L. Powell, E. Schulman. Applications of Approximate Word Matching in Information Retrieval. In Forouzan Golshani, Kia Makki, editors, *Proceedings of the Sixth International Conference on Information and Knowledge Management (CIKM 1997)*, pages 9-15, Las Vegas (USA), November 1997.
3. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
4. D.S. Hirschberg. Algorithms for the Longest Common Subsequence Problem. *Journal of the ACM*, 24(4):664-675, 1977.
5. D.S. Hirschberg. Serial Computations of Levenshtein Distances. In A. Apostolico, Z. Galil, editors, *Pattern Matching Algorithms*. Oxford University Press, 1997.
6. J.W. Hunt, T.G. Szymanski. A Fast Algorithm for Computing Longest Common Subsequences. *Communications of the ACM*, 20(5):350-353, 1977.
7. K. Kukich. Techniques for Automatically Correcting Words in Text. *Computing Surveys*, 24(4):377-440, 1992.
8. V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10:707-710, 1966.
9. R. Lowrance, R.A. Wagner. An Extension of the String-to-String Correction Problem. *Journal of the ACM*, 22(2):177-183, 1975.
10. W.J. Masek, M.S. Paterson. A Faster Algorithm for Computing String Edit Distances. *Journal of Computer and Systems Sciences*, 20(1):18-31, 1980.
11. E. Myers. An O(ND) Difference Algorithm and its Variations. *Algorithmica*, 1(2):251-266, 1986.
12. N. Nakatsu, Y. Kambayashi, S. Yajima. A Longest Common Subsequence Algorithm Suitable for Similar Text Strings. *Acta Informatica*, 18:171-179, 1982.
13. W. Tichy. The String-to-String Correction Problem with Block Moves. *ACM Transactions on Computer Systems*, 2(4):309-321, 1984.
14. R.A. Wagner, M.J. Fischer. The String-to-String Correction Problem. *Journal of the ACM*, 21(1): 168-173, 1974.