ELSEVIER

Information Processing Letters 54 (1995) 93-96

Information Processing Letters

An improved algorithm for computing the edit distance of run-length coded strings

H. Bunke [a,*], J. Csirik [b]

[a] *Institut für Informatik und Angewandte Mathematik, Länggassstrasse 51, 3012 Bern, Switzerland*
[b] *Department of Applied Commputer Science, University of Szeged, Arpad ter 2, H-6720 Szeged, Hungary*

Communicated by M.J. Atallah; received 7 November 1994; revised 22 December 1994

Keywords: Algorithms; Approximate string matching; String edit distance

1. Introduction

Comparing strings of symbols is an interesting problem from both the theoretical and practical point of view [6]. In this paper we focus on string distance computation based on a set of edit operations. The algorithm of Wagner and Fischer [9] is usually referred to as the standard solution to this problem. It is based on dynamic programming and has a time complexity of $O(n \cdot m)$, where n and m give the lengths of the two strings to be compared. Two faster algorithms for the string edit distance problem having a time complexity of $O(n^2/\log n)$ and $O(d \cdot m)$, were described in [5] and [8], respectively; here it is assumed that d is the edit distance of the two strings, and $n \geqslant m$. Other algorithms for approximate string matching are reported in [2,3,7,11].

Depending on the particular application, it can be an advantage to use a special representation or coding method for the strings to be compared. One well-known method that has been widely used, for example in image processing, is run-length coding. Here, one does not explicitly list all individual symbols in a string, but considers runs of identical consecutive symbols and gives only one representative symbol, together with its multiplicity, for each run. For example, the run-length coding representation of the string $a_1a_1a_1a_1a_1a_2a_2a_2a_3a_3a_3a_3$ is $a_1^5a_2^3a_3^4$ or, equivalently, $(a_1,5)(a_2,3)(a_3,4)$. This coding scheme can result in significant memory and access time savings if the strings under consideration consist of long runs of identical consecutive symbols.

Recently, an algorithm for computing the edit distance of run-length coded strings was proposed [1]. Clearly, a brute force method to solve this problem is to first reconstruct the standard, full-length representation of the strings under consideration from the run-length code, and then to apply one of the known distance computation algorithms. However, this approach needs additional effort for string decoding. Also, it does not utilize the lower data rate of the run-length code that may potentially lead to a speed-up of the distance computation. In the earlier algorithm of Bunke and Csirik [1], the edit matrix of Wagner and Fischer's algorithm was split into blocks, each block corresponding to a pair of runs of identical symbols in the two strings to be compared. It was shown, under a particular cost function, that the algorithm has a time complexity of $O(k \cdot l)$ if all runs in the two strings are of the same length, with $k \cdot l$ being number of blocks

* Corresponding author. Email: bunke@iam.unibe.ch.

SSDI 0020-0190(95)00005-4

in the edit matrix. This means that there is, independently of the size of a block, only a constant number of operations required for each block. For the general case, however, where the runs in the two strings to be matched are of different length, the earlier algorithm has the same time complexity as that of Wagner and Fischer's algorithm. In this paper, we propose a different approach. Our new algorithm will be also based on a division of the edit matrix into blocks. However, no subdivision of these blocks will ever be required. It will be shown that it is sufficient, in order to get the edit distance of two run-length coded strings, to compute for each block only its last row and column in the edit matrix. The new algorithm is restricted, however, to the special cost function under which the cost of any insertion and deletion is equal to 1, and the cost of any substitution is equal to 2. The algorithm described in [1] can additionally handle the case where all edit operations have unit cost.

String edit distance computation for run-length coded strings under the cost function considered in this paper is a special case of the set-set longest common subsequence problem for multisets. In [10] an algorithm for this problem was proposed that has the same time complexity as the method described in this paper. The method in the present paper, however, is derived in a more straightforward manner for the particular case of run-length coded strings.

2. Some properties of string edit distance

We consider strings $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \ldots b_m$ over a finite alphabet V. The empty string is denoted by ε. An edit operation is a pair (a, b), also denoted by $a \to b$, where $a, b \in V \cup \{\varepsilon\}$, $(a, b) \neq (\varepsilon, \varepsilon)$, $a \neq b$. String B results from string A through the edit operation $a \to b$ if $A = xay$ and $B = xby$ for some strings x and y over V. We call $a \to b$ a substitution if $a \neq \varepsilon$ an $b \neq \varepsilon$, a deletion if $b = \varepsilon$, and an insertion if $a = \varepsilon$. A sequence E of edit operations will be called an edit sequence. Let $E = e_1, e_2, \ldots, e_k$ be an edit sequence. We say that B is derivable from A via E if there is a sequence $D_0, D_1, \ldots, D_k$ of strings such that $A = D_0$, $B = D_k$ and D_i results from D_{i-1} through e_i, $i = 1, \ldots, k$.

A cost function c is a function assigning a non-negative real number $c(a \to b)$ to each edit operation $a \to b$. The cost of a sequence $E = e_1, e_2, \ldots, e_k$ of edit operations is defined by $c(E) = \sum_{i=1}^{k} c(e_i)$ and the edit distance of strings A and B by $d(A, B) = \min\{c(E) \mid B \text{ is derivable from } A \text{ via } E\}$. Throughout this paper, we will restrict our considerations to the cost function $c(a \to \varepsilon) = c(\varepsilon \to a) = 1$, $c(a \to b) = 2$ for any $a, b \in V, a \neq b$. Notice that under this cost function, string edit distance computation is equivalent to finding the longest common subsequence of A and B [9].

First we recall that, when matching two strings, the edit distance will either increase or decrease by the value of 1 if we extend one of the two strings by one symbol.

Lemma 1. *Let $a, b \in V$ be two symbols and $x, y \in V^*$ two strings. Then $|d(xa, y) - d(x, y)| = 1$ and $|d(x, yb) - d(x, y)| = 1$.*

The proof follows immediately from a simple case analysis.

Lemma 2. *Let $a \in V$ be a symbol and $x, y \in V^*$ two strings. Then $d(xa, ya) = d(x, y)$.*

Proof. From Wagner and Fischer's algorithm it follows that $d(xa, ya) = \min\{d(x, y), d(xa, y) + 1, d(x, ya) + 1\}$ and from Lemma 1 we know that $d(x, y) \leqslant d(xa, y) + 1$ and $d(x, y) \leqslant d(x, ya) + 1$. □

Lemma 3. *Let $a \in V$ be a symbol and $x, y \in V^*$ two strings. Then $d(xa^n, ya^n) = d(x, y)$ for any $n \geqslant 0$.*

This lemma is an immediate consequence of Lemma 2. The next lemma follows from Lemma 1.

Lemma 4. *Let $a, b \in V$ with $a \neq b$ be two symbols and $x, y \in V^*$ two strings. Then $d(xa, yb) = \min\{d(xa, y) + 1, d(x, yb) + 1\}$.*

Lemma 5. *Let a, b, x, y be the same as in Lemma 4. Then $d(xa^n, yb^m) = \min\{d(xa^n, y) + m, d(x, yb^m) + n\}$ for any $n, m \geqslant 0$.*

Proof. By Lemma 4 we have $d(xa^n, yb^m) = \min\{d(xa^n, yb^{m-1}) + 1, d(xa^{n-1}, yb^m) + 1\}$. Applying Lemma 4 again to both $d(xa^n, yb^{m-1})$ and $d(xa^{n-1}, yb^m)$ and resubstituting, we get

		$b_1^{m_1}$	$b_1^{m_2}$		$b_l^{m_l}$
	$D_{0,0}$	$D_{0,1}$	$D_{0,2}$	...	$D_{0,l}$
$a_1^{n_1}$	$D_{1,0}$	$D_{1,1}$	$D_{1,2}$	...	$D_{1,l}$
$a_2^{n_2}$	$D_{2,0}$	$D_{2,1}$	$D_{2,2}$	...	$D_{2,l}$
	⋮	⋮	⋮		⋮
$a_k^{n_k}$	$D_{k,0}$	$D_{k,1}$	$D_{k,2}$	...	$D_{k,l}$

Fig. 1. Subdivision of the edit matrix.

$d(xa^n, yb^m) = \min\{d(xa^n, yb^{m-2}+2),\ d(xa^{n-1}, yb^{m-1})+2,\ d(xa^{n-2}, yb^m)+2\}$. Repeating this process of expansion and resubstitution, we get

$$
\begin{aligned}
&d(xa^n, yb^m)\\
&\quad = \min\{d(x,y)+m+n, d(xa,y)+m+n-1,\\
&\qquad d(xa^2,y)+m+n-2,\ldots,d(xa^n,y)+m,\\
&\qquad d(x,yb)+m+n-1, d(x,yb^2)+m+n-2,\\
&\qquad \ldots, d(x,yb^m)+n\}.
\end{aligned}
$$

Applying Lemma 1, we conclude that the minimum of $d(x,y)+m+n, d(xa,y)+m+n-1,\ldots,d(xa^n,y)+m$ is equal to $d(xa^n,y)+m$. Similarly, the minimum of $d(x,y)+m+n, d(x,yb)+m+n-1,\ldots,d(x,yb^m)+n$ is equal to $d(x,b^m)+n$. This concludes the proof. □

3. The algorithm

Let $A = (a_1,n_1)(a_2,n_2)\ldots(a_k,n_k)$ and $B = (b_1,m_1)(b_2,m_2)\ldots(b_l,m_l)$ with $a_i, b_i \in V$, $n = n_1+n_2+\cdots+n_k$, $m = m_1+m_2+\cdots+m_l$, $a_i \neq a_{i+1}$, $b_j \neq b_{j+1}$, $i = 1,\ldots,k-1$, $j = 1,\ldots,l-1$, be two run-length coded strings that are to be compared. We will call (a_i,n_i) and (b_j,m_j) the ith run of A and the jth run of B, respectively. Our proposed algorithm will be based on a subdivision of the edit matrix D into submatrices $D_{i,j}$, $i = 0,1,\ldots,k$, $j = 0,1,\ldots,l$ as shown in Fig. 1. Similar to Wagner and Fischer's algorithm, the submatrix $D_{i,j}$ holds string distance values that arise during the computation of the distance between the ith run of A and the jth run of B.

From the results derived in the last section it follows that it is sufficient to compute for each submatrix $D_{i,j}$ only the last row and column in order to derive $d(A,B)$. The elements of the last row and column of the submatrix $D_{i,j}$ can be directly computed, by means of a constant number of operations, from the elements of the last row of the submatrix $D_{i-1,j}$ and the last column of $D_{i,j-1}$. We call the last column and row of a submatrix $D_{i,j}$ its output column and output row. As there is no need to compute any of the interior elements, the algorithm needs only $O(n_i+m_j)$ operations for any of the submatrices $D_{i,j}$, instead of $O(n_i \cdot m_j)$ operations required by Wagner and Fischer's algorithm. Also, there is no need to split any of the submatrices $D_{i,j}$ into smaller pieces as its is required in the earlier algorithm for run-length coded string matching [1].

The description of the proposed algorithm uses two three-dimensional arrays, $outcol(i,j;r)$ and $outrow(i,j;s)$, $i = 1,\ldots,k$, $j = 1,\ldots,l$, $r = 0,1,\ldots,n_i$, $s = 0,1,\ldots,m_j$ where we store the elements of the output row and output column of submatrix $D_{i,j}$, respectively. In an implementation of the algorithm we can use the two-dimensional array $D(i,j)$ because $outcol(i,j;r)$ is equivalent to the matrix element $D(I,J)$ of Wagner and Fischer's algorithm, and $outrow(i,j;s)$ is equivalent to $D(I',J')$, where $I = n_1+\cdots+n_{i-1}+r$, $J = m_1+\ldots+m_j$, $I' = n_1+\cdots+n_i$, $J' = m_1+\cdots+m_{j-1}+s$.

In Fig. 2 the algorithm is shown in pseudo code. For notational convenience, we use the following abbreviations: $N_0 = M_0 = 0$; $N_i = \sum_{\nu=1}^{i} n_\nu$ for $i = 1,\ldots,k$; $M_j = \sum_{\nu=1}^{j} m_\nu$, for $j = 1,\ldots,l$.

Theorem 6. *After termination of procedure* rlc_string_match, *we have*

$$outcol(k,l;n_k) = outrow(k,l;m_l) = d(A,B).$$

The proof is by induction on the lengths of A and B using the lemmas of Section 3.

Theorem 7. *Algorithm* rlc_string_match *has a time and space complexity of* $O(ln+km)$.

The proof follows directly from the pseudo code of the algorithm.

4. Concluding remarks

If it is desired not only to compute the edit distance of a pair of run-length coded strings, but also to get the optimal sequence of edit operations that transform

```
procedure rlc_string_match(A, B)

input: two run-length coded strings A and B
output: the edit distance d(A, B)

method:
/* initialization*/
1.  for i = 1 to k do /* initial column */
2.    for r = 0 to n_i do outcol(i, 0; r) = N_{i-1} + r;
3.  for j = 1 to l do /* initial row */
4.    for s = 0 to m_j do outcol(0, j; s) = M_{j-1} + s;
    /* main loop */
5.  for i = 1 to k do
6.    for j = 1 to l do
        /* outputcolumn */
7.      outcol(i, j; 0) = outrow(i - 1, j; m_j);
8.      for r = 1 to n_i do
9.        if a_i = b_j then
10.         if r ≤ m_j then
              outcol(i, j; r) = outrow(i - 1, j; m_j - r)
11.         else /* r > m_j */
              outcol(i, j; r) = outcol(i, j - 1; r - m_j)
12.       else /* a_i ≠ b_j */
13.         outcol(i, j; r) = min{outcol(i, j; 0) + r,
                                  outcol(i, j - 1; r) + m_j};
        /* outputrow */
14.     outrow(i, j; 0) = outcol(i, j - 1; n_i);
15.     for s = 1 to m_j do
16.       if a_i = b_j then
17.         if s ≤ n_i then
              outrow(i, j; s) = outcol(i, j - 1; n_i - s)
18.         else /* s > n_i */
              outrow(i, j; s) = outrow(i - 1, j; s - n_i)
19.       else /* a_i ≠ b_j */
20.         outrow(i, j; s) = min{outrow(i, j; 0) + s,
                                  outrow(i - 1, j; s) + n_i}
    }
21. end rlc_string_match
```

Fig. 2. Procedure rlc_string_match.

one string into the other, we can augment algorithm rlc_string_match by pointers similar to Wagner and Fischer's algorithm. If these pointers are not needed, the space complexity of the algorithm can be reduced to $O(n)$ similarly to the method proposed in [4].

It is an open question if an algorithm similar to rlc_string_match exists for other cost functions. As an example, for cost function $c(a \to \varepsilon) = c(\varepsilon \to a) = c(a \to b) = 1$, $a \neq b$, it seems that the elements $outcol(i, j; r)$ and $outrow(i, j; s)$ cannot be computed in constant time. Instead, the statements equivalent to 13 and 20 of rlc_string_match need to determine, in each execution cycle, the minimum of a set of elements, the size of which is proportional to the length of the runs. As a consequence, the complexity of the algorithm increases by a factor of n_i and m_j, respectively, resulting in $O(nm)$ time.

References

[1] H. Bunke and J. Csirik, An algorithm for matching run-length coded strings, *Computing* **50** (1993) 297-314.

[2] Z. Galil and R. Giancarlo, Data structures and algorithms for approximate string matching, *J. Complexity* **4** (1988) 33-72.

[3] Z. Galil and K. Park, An improved algorithm for approximate string matching, *SIAM J. Comput.* **19** (6) (1990) 989-999.

[4] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, *Comm. ACM* **18** (6) (1975) 341-343.

[5] W.J. Masek and M.S. Paterson, A faster algorithm for comparing string-edit distances, *J. Comput. System Sci.* **20** (1) (1980) 18-31.

[6] D. Sankoff and J.B. Kruskal, *Time Warps, String Edits, and Macromolecules; The Theory and Practice of Sequence Comparison* (Addison-Wesley, Reading, MA, 1983).

[7] J. Tarhio and E. Ukkonen, Approximate Boyer-Moore string matching, *SIAM J. Comput.* **22** (2) (1993) 243-260.

[8] E. Ukkonen, Algorithms for approximate string matching, *Inform. and Control* **64** (1985) 100-118.

[9] R.A. Wagner and M.J. Fischer, The string-to-string correction problem, *J. ACM* **21** (1) (1974) 168-173.

[10] B.-F. Wang, G.-H. Chen and K. Park, On the set LCS and set-set LCS problems, *J. Algorithms* **14** (1953) 466-477.

[11] S. Wu and U. Manber, Fast text searching allowing errors, *Comm. ACM* **35** (10) (1992) 83-91.