Edit distance of run-length coded strings

*H. Bunke, †J. Csirik

*Institut für Informatik, Universität Bern, Switzerland
†Department of Applied Computer Science, University of Szeged, Hungary

Abstract

We give an algorithm for measuring the similarity of run-length coded strings. In run-length coding, not all individual symbols in a string are listed. Instead, one run of identical consecutive symbols is coded by giving one representative symbol together with its multiplicity. If the strings under consideration consist of long runs of identical symbols, significant reductions in memory and access time can be achieved by run-length coding. Our algorithm determines the minimum cost sequence of edit operations needed to transform one string into another. It uses as basic data structure an edit matrix similar to the classical algorithm of Wagner and Fischer [1]. However, depending on the particular pair of strings to be compared, only a part of this edit matrix usually needs to be computed. In the worst case, our algorithm has a time complexity of $O(n \cdot m)$, where n and m give the lengths of the strings to be compared. In the best case, the time complexity is $O(k \cdot l)$, where k and l are the numbers of runs of identical symbols in the two strings under comparison.

Introduction

String matching is a problem that constantly receives attention in different areas of science and engineering. In information processing, for example, there are applications such as text editing, text retrieval or file difference checking that require the comparison of strings of symbols. In biology, there is a need to compare sequences of acids and proteins in order to answer questions regarding the evolution of organisms. Other applications of string matching include the analysis of human speech, or object recognition in images. Besides practical applications, string matching has been intensively studied in the context of theoretical computer science.

There are many different classes of string matching problems. In this paper we focus on approximate string matching, i.e. on string distance computation based on a set of edit operations. We are interested in measuring the similarity of pairs of strings via the minimum cost sequence of edit operations needed to transform one string into the other. The first reference for this type of string matching problem is [2]. Later the edit distance was extensively studied and a close relation to the longest common subsequence problem was given. The method of Wagner and Fischer is regarded as the standard reference for the dynamic programming solution of this problem. The time complexity of this method is $O(n \cdot m)$. Masek and Paterson improved this bound to $O(n^2/\log n)$, using the "Four Russians" method [3]. Other algorithms for computing the edit distance of two strings on the basis of their longest common subsequence have been given in [4]-[6].

Depending on the particular application, it can be an advantage to use a particular representation or coding method for the strings to be compared. One well known and widely used method 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. On the other hand, for a run of length one, run-length coding requires more memory than the standard representation.

In this paper we consider the problem of distance computation of run-length coded strings. 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 this paper we give a string distance computation algorithm that operates directly on the run-length representations of the strings under consideration.

An extensive survey on string matching, including string distance computation algorithms, has recently been given [7]. An earlier reference that includes various applications in molecular biology and human speech understanding is [8]. Applications of string matching in computer vision and pattern recognition have been reported in [9]-[10]. An experimental study in the reduction of the computational cost in template matching that results from run-length coding has been described in [11].

Basic definitions

We are given two strings $A = a_1a_2\ldots a_n$ and $B = b_1b_2\ldots b_m$ over a finite alphabet V. The problem we would like to solve is the computation of the edit distance of A and B when A and B are given in run-length coded form.

Let us denote the empty string by ε. An edit operation is a pair $(c_i, c_j) \neq (\varepsilon, \varepsilon), c_i, c_j \in V \cup \{\varepsilon\}$. String B results from string A through the edit operation (c_i, c_j) (our notation for this will be $A \longrightarrow B$ via (c_i, c_j)) if $A = D_1c_iD_2, B = D_1c_jD_2$ for some strings D_1, D_2 over V. We call (c_i, c_j) a replacement if $c_i \neq \varepsilon, c_j \neq \varepsilon$, a delete if $c_j = \varepsilon$ and an insert if $c_i = \varepsilon$.

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 will say that B is derivable from A via E if there is a sequence of strings $D_0, D_1, \ldots, D_k$ such that $A = D_0, B = D_k$ and for $1 \leq i \leq k$ $D_{i-1} \longrightarrow D_i$ via e_i. Naturally, B is always derivable from A via a sequence consisting of n deletes and m inserts.

A cost function δ is a function assigning a non-negative real number to each edit operation (c_i, c_j). We can define the cost of a sequence $E = e_1, e_2, \ldots, e_k$ by

$$\delta(E) = \sum_{i=1}^{k} \delta(e_i).$$

Now, the edit distance δ of strings A and B can be defined by

$$\delta(A, B) = \min\{\delta(E) | B \text{ is derivable from } A \text{ via } E\}.$$

An algorithm for computing the edit distance $\delta(A, B)$ was given by Wagner and Fischer [1]; their solution uses a simple dynamic programming approach. Next, we briefly review this algorithm.

Let $A(i, j) = a_ia_{i+1}\ldots a_j$ and $B(i, j) = b_ib_{i+1}\ldots b_j$. For short, we shall write A_i for $a_1a_2\ldots a_i, B_j$ for $b_1b_2\ldots b_j$ and $\delta_{i,j}$ for $\delta(A_i, B_j)$. Wagner and Fischer's algorithm constructs a $(n+1)\times(m+1)$ edit matrix $D = (d_{i,j})$ with indices running from 0 to n and from 0 to m. The first row and column are simply given by

$$d_{0,0} = 0, \quad d_{0,j} = \delta(\varepsilon, B_j), \quad d_{i,0} = \delta(A_i, \varepsilon)$$

and all others elements $d_{i,j}$ are equal to $\delta_{i,j}$. Wagner and Fischer proved that

$$\begin{aligned} d_{i,j} \;=\; & \min(d_{i-1,j-1} + \delta(a_i, b_j), \\ & d_{i-1,j} + \delta(a_i, \varepsilon), d_{i,j-1} + \delta(\varepsilon, b_j)) \end{aligned} \qquad (0)$$

for $1 \leq i \leq n$, $1 \leq j \leq m$, and used this equation to compute the elements in the edit matrix. Clearly, $\delta(A, B) = d_{n,m}$ and the algorithm uses $O(n \cdot m)$ elementary steps and $O(n \cdot m)$ space.

Throughout this paper, we will restrict our considerations to cost functions where

$$\delta(a, a) = 0;\ \delta(a, \varepsilon) = \delta(\varepsilon, a) = 1; a \in V$$

and we will analyse the two most often used subcases

$$1) \qquad \delta(a, b) = 1;\ a, b \in V; a \neq b \qquad (1)$$

$$2) \qquad \delta(a, b) = 2;\ a, b \in V; a \neq b. \qquad (2)$$

General idea and computation of the first submatrix

Let

$$\begin{aligned} & A = (a_1, n_1)(a_2, n_2)\ldots(a_k, n_k) \quad \text{and} \\ & B = (b_1, m_1)(b_2, m_2)\ldots(b_l, m_l) \quad \text{with} \end{aligned} \qquad (3)$$

$a_i, b_i \in V;\ n = n_1 + n_2 + \ldots + n_k;\ m = m_1 + m_2 + \ldots + m_l;\ a_i \neq a_{i+1};\ b_j \neq b_{j+1};\ i = 1, \ldots, k-1;\ j = 1, \ldots, l-1$

		$b_1^{m_1}$	$b_2^{m_2}$	$\cdots$	$b_l^{m_l}$
	$D_{0,0}$	$D_{0,1}$	$D_{0,2}$	$\cdots$	$D_{0,l}$
$a_1^{n_1}$	$D_{1,0}$	$D_{1,1}$	$D_{1,2}$	$\cdots$	$D_{1,l}$
$a_2^{n_2}$	$D_{2,0}$	$D_{2,1}$	$D_{2,2}$	$\cdots$	$D_{2,l}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$a_k^{n_k}$	$D_{k,0}$	$D_{k,1}$	$D_{k,2}$	$\cdots$	$D_{k,l}$

Figure 1: Subdivision of the edit matrix

be two run-length coded strings that are to be compared. We will call (a_i, n_i) and (b_j, m_j) the i-th run of A and the j-th 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 i-th run of A and the j-th run of B.

The submatrix $D_{0,0}$ is a 1×1 matrix holding the value 0. The submatrices $D_{0,1}, D_{0,2}, \ldots, D_{0,l}$ are of dimension $1 \times m_1, 1 \times m_2, \ldots, 1 \times m_l$ with values

$$\begin{aligned} &D_{0,j} = (t_j + 1, t_j + 2, \ldots, t_j + m_j); \\ &t_j = m_1 + m_2 + \ldots + m_{j-1}; \\ &j = 1, \ldots, l. \end{aligned} \tag{4}$$

Similarly, the submatrices $D_{1,0}, D_{2,0}, \ldots, D_{k,0}$ are of dimension $n_1 \times 1, n_2 \times 1, \ldots, n_k \times 1$ with values

$$\begin{aligned} &D_{i,0} = (s_i + 1, s_i + 2, \ldots, s_i + n_i)_t; \\ &s_i = n_1 + n_2 + \ldots + n_{i-1}; \\ &i = 1, \ldots, k. \end{aligned} \tag{5}$$

These submatrices $D_{0,0}$, $D_{0,j}$ and $D_{i,0}$ hold the initial values similar to Wagner and Fischer's algorithm.

For a concrete example, let $V = \{a, b, c\}$,

$A = (a, 8)(b, 6)(a, 3)(c, 4)(b, 5)$,

$B = (a, 12)(b, 4)(c, 7)(b, 9)$.

Then, the decomposition of the edit matrix into submatrices and the initial values are shown in Fig. 2.

		a^{12}	b^4	c^7	b^9
	0	1...12	13...16	17...23	24...32
a^8	1 ⋮ 8	$D_{1,1}$	$D_{1,2}$		$D_{1,4}$
b^6	9 ⋮ 14	$D_{2,1}$			
a^3	15 ⋮ 17				
c^4	18 ⋮ 21				
b^5	22 ⋮ 26	$D_{5,1}$			$D_{5,4}$

Figure 2: An example of the subdivision of the edit matrix

For any submatrix $D_{i,j}, i = 1, \ldots, k$; $j = 1, \ldots l$ of an edit matrix D we will call the element at the right lower corner the final element of $D_{i,j}$. The last row and last column of $D_{i,j}$ will be called the output row and output column of $D_{i,j}$, respectively. Furthermore, the final element of $D_{i-1,j-1}$ together with the output (last) row of $D_{i-1,j}$ will be called the input row of $D_{i,j}$, and the final element of $D_{i-1,j-1}$ together with the output (last) column of $D_{i,j-1}$ will be called the input column of $D_{i,j}$. Formally, with the notations of (4) and (5) the input row, input column, output row and output column of $D_{i,j}$ are given by

$(d_{s_i,t_j}, d_{s_i,t_j+1}, \ldots, d_{s_i,t_j+m_j})$

$(d_{s_i,t_j}, d_{s_i+1,t_j}, \ldots, d_{s_i+n_i,t_j})$

$(d_{s_i+n_i,t_j+1}; d_{s_i+n_i,t_j+2}, \ldots, d_{s_i+n_i,t_j+m_j})$

$(d_{s_i+1,t_j+m_j}, d_{s_i+2,t_j+m_j}, \ldots, d_{s_i+n_i,t_j+m_j})$, respectively,

where $i = 1, \ldots, k$; $j = 1, \ldots, l$.

For example, the matrix $D_{1,1}$ in Fig. 2 has the input row $(0, 1, 2, \ldots, 12)$ and the input column $(0, 1, 2, \ldots, 8)$.

The overall goal of our computation is the final element of D, i.e. $d_{n,m}$, as it is equal to the desired distance $\delta(A, B)$. An important observation for our algorithm is that under certain circumstances to be analysed in more detail below, it is possible to calculate the output row and output column of any submatrix $D_{i,j}$ directly from

		a_1	a_1	a_1	...	a_1
	0	1	2	3	...	m_1
a_1	1	0	1	2	...	m_1-1
a_1	2					$\vdots$
$\vdots$	$\vdots$					
a_1	n_1	n_1-1	n_1-2	n_1-3	... 1 0 1	... m_1-n_1

Figure 3: Computation of submatrix $D_{1,1}$; $a_1 = b_1$; cost functions (1) and (2), $m_1 \geq n_1$

		b_1	b_1	b_1	...		b_1
	0	1	2	3	...		m_1
a_1	1	1	2	3	...		m_1
a_1	2	2	2	3	...		m_1
$\vdots$							$\vdots$
a_1	n_1	n_1	n_1	n_1	...n_1	n_1+1...	m_1

Figure 4: Computation of submatrix $D_{1,1}$; $a_1 \neq b_1$; cost function (1)

its input row and input column, without the need for an explicit computation of the interior elements of $D_{i,j}$. First, we consider the computation of the submatrix $D_{1,1}$. The input row of this submatrix is $D_{0,0}$ together with $D_{0,1}$, and the input column is $D_{0,0}$ together with $D_{1,0}$. Knowing the input row and input column, the output row and output column of $D_{1,1}$ can be given directly. It depends only on the lengths n_1 and m_1 of the first runs in A and B, respectively, and on whether $a_1 = b_1$ or $a_1 \neq b_1$. This can readily be seen in Figs 3 and 4. Figure 3 corresponds to the case where $a_1 = b_1$, and in Fig. 4, we have $a_1 \neq b_1$. The values in Figs 3 and 4 are based on the cost function (1). In Figs 3 and 4, we assume $n_1 \leq m_1$. The case $n_1 > m_1$ is similar.

For cost function (2) and $a_1 = b_1$, we get the same result as in Fig. 3. The submatrix $D_{1,1}$ for cost function (2) and $a_1 \neq b_1$ is shown in Fig. 5, again assuming $n_1 \leq m_1$. We conclude that in any case the output row and output column of $D_{1,1}$ can be inferred directly from the input row and input column, without the necessity to compute any interior element of $D_{1,1}$.

		b_1	b_1	b_1		b_1
	0	1	2	3	...	m_1
a_1	1	2	3	4	...	m_1+1
a_1	2	3				
$\vdots$						
a_1	n_1	n_1+1	...			n_1+m_1

Figure 5: Computation of submatrix $D_{1,1}$; $a_1 \neq b_1$; cost function (2)

Computation of the other submatrices

Under the two cost functions considered in this paper, any element in an edit matrix will be a non-negative integer. For the first submatrix $D_{1,1}$, both the input row and input column are strictly monotonically increasing sequences of integers of the form $(0, 1, \ldots, m_1)$ and $(0, 1, \ldots, n_1)$, respectively. In general, however, the input row and input column of any of the other submatrices can be of different form. For example, the output row and output column in Fig. 3, which are of different form, correspond to the input row of $D_{2,1}$ and input column of $D_{1,2}$.

For our algorithm, we split up the input rows and input columns of any submatrix into sequences of integers according to the following three different basic types:

- A monotonically increasing cost sequence of the form $(s, s+1, \ldots, s+t)$; $s \geq 0$; $t \geq 1$. Such a sequence will be denoted by (inc, t), where inc indicates the type increasing and t is the length of the considered sequence. The starting value s of the sequence is not important for our considerations.
- A monotonically decreasing cost sequence of the form $(s, s-1, \ldots, s-t); s, t \geq 1; s \geq t$. A sequence of this form will be denoted by (dec, t), similarly to an increasing sequence.
- A sequence of constant cost values $(s, s, \ldots, s)$ with $t+1 \geq 1$ occurrences of the value $s \geq 0$. Such a sequence will be represented by (eq, t).

Given these basic types, any input or output row or column of a submatrix $D_{i,j}$, $i = 1, \ldots, k$; $j = 1, \ldots, l$ will be represented by the sequence of its basic types

$$(\text{type}1, t_1), (\text{type}2, t_2), \ldots, (\text{type}N, t_N) \qquad (6)$$

where type$i \in \{inc, dec, eq\}$; $t_i \geq 1$; $i = 1, \ldots, N$.

For example, the last row of the matrix in Fig. 3 will be represented by

$$((dec, n_1), (inc, m_1 - n_1))$$

and the last row in Fig. 4 by

$$((eq, n_1), (inc, m_1 - n_1)).$$

As a possible generalization of the procedure for the calculation of $D_{1,1}$, one could think of giving rules for directly computing the output row and output column of any of the other submatrices $D_{i,j}, i = 1, \ldots, k;\ j = 1, \ldots, l;\ (i,j) \neq (1,1)$. However, this is not feasible as many such rules may be necessary. Instead, we further subdivide each submatrix $D_{i,j}$ into a number of smaller submatrices according to the subsequences of basic type of its input row and input column. Below, we give rules for the processing of these smaller submatrices. Similarly as in Section 3, the output row and output column of each of these smaller submatrices can be immediately determined from its input row and input column without computing any of the interior elements, given the corresponding parts of the strings A and B. Since there are three basic types of sequences of cost values, we have to consider a total of nine different cases for both $a_i = b_j$ and $a_i \neq b_j$. First, we assume a cost function according to (1). Let $a_i^{l_1}$ and $b_j^{l_2}$ be two subsequences of A and B, respectively, that are to be compared. We analyse only the first subcases in details.

Case A: $a_i = b_j$

A11. Input column: $((inc, l_1))$
Input row: $((inc, l_2))$
If $l_1 \leq l_2$, then
output column: $((dec, l_1))$
output row: $((dec, l_1), (inc, l_2 - l_1))$
If $l_1 > l_2$, then
output column: $((dec, l_2), (inc, l_1 - l_2))$
output row: $((dec, l_2))$

In either case, the element in the left upper corner of the submatrix $D_{i+1,j+1}$ is given by $d_{i+1,j+1} = d_{i,j} + |l_1 - l_2|$.

Case B: $a_i \neq b_j$

B11. Input column: $((inc, l_1))$
Input row: $((inc, l_2))$
If $l_1 \leq l_2$, then
output column: $((eq, l_1))$
output row: $((eq, l_1), (inc, l_2 - l_1))$
If $l_1 > l_2$, then
output column: $((eq, l_2), (inc, l_1 - l_2))$
output row: $((eq, l_2))$

A similar case analysis can be made for the cost function (2). Here, any input or output row or column consists of only increasing or decreasing basic cost sequences. Equal cost runs do not occur. We give the full analysis in Case C and omit it in Case D.

Case C: $a_i = b_j$

C11. Input column: $((inc, l_1))$
Input row: $((inc, l_2))$
If $l_1 \leq l_2$, then
output column: $((dec, l_1))$
output row: $((dec, l_1), (inc, l_2 - l_1))$
If $l_1 > l_2$, then
output column: $((dec, l_2), (inc, l_1 - l_2))$
output row: $((dec, l_2))$

C12. Input column: $((inc, l_1))$
Input row: $((dec, l_2))$
Output column: $((inc, l_1))$
Output row: $((dec, l_2))$

C21. Input column: $((dec, l_1))$
Input row: $((inc, l_2))$
Output column: $((dec, l_1))$
Output row: $((inc, l_2))$

C22. Input column: $((dec, l_1))$
Input row: $((dec, l_2))$
If $l_1 \leq l_2$, then
output column: $((inc, l_1))$
output row: $((inc, l_1), (dec, l_2 - l_1))$
If $l_1 > l_2$, then
output column: $((inc, l_2), (dec, l_1 - l_2))$
output row: $((inc, l_2))$

Case D: $a_i \neq b_j$

We can see that in all cases the output vectors contain only increasing or decreasing parts; equal runs do not occur. Thus, we really have only the subcases discussed above.

Overall algorithm

We are now ready to give the overall algorithm for distance computation of run-length coded strings. Given A and B according to (3), the initial matrices $D_{0,0}, D_{0,1}, D_{0,2}, \ldots, D_{0,l}$ and $D_{1,0}, D_{2,0}, \ldots D_{k,0}$ can be

computed according to (4,5). Then, we compute the output row and output column of $D_{1,1}$ as described in section 3. Next, we continue with the computation of $D_{1,2}$. Its input row and input column are known from the computation of $D_{0,1}, D_{0,2}$ and $D_{1,1}$. The input row of $D_{1,2}$ consists of only one sequence of cost values of basic type. If its input column also consists of only one sequence of basic type (this is the case, for example, if $n_1 \leq m_1$; see case A11 in Section 4), then we can immediately determine the output row and output column of $D_{1,2}$ according to the case analysis given is Section 4. If the input column of $D_{1,2}$ consists of more than one basic cost sequence, then we subdivide $D_{1,2}$ into submatrices, each of which will be handled in exactly the same way as $D_{1,2}$. Having determined the output row and output column of $D_{1,2}$, we continue with $D_{1,3}, D_{1,4}, \ldots, D_{1,l}$.

After the computation of the output rows and output columns of these matrices, we know the input rows for each of $D_{2,1}, D_{2,2}, \ldots, D_{2,l}$ and we also know the input column for $D_{2,1}$. Thus, we proceed with $D_{2,1}$ and then with $D_{2,2}, D_{2,3}, \ldots, D_{2,l}$ splitting each submatrix that is encountered into smaller submatrices depending on the number of cost sequences of basic type occurring at any of the input rows or input columns. This process is repeated until the output row and column of $D_{k,l}$ are known.

Let us turn to our example in Fig. 2, using the cost function according to (1). Here, the input row for $D_{1,1}$ is of type $((inc, 12))$ and the input column is of type $((inc, 8))$. As both letters are the same, we have to use A11, subcase $l_1 \leq l_2$. Hence, the last row is of type $((dec, 8)(inc, 4))$, and the last column is $((dec, 8))$. Now the input column of $D_{1,2}$ is $((dec, 8))$, and the input row is $((inc, 4))$. The letters are different, so we have to use B31. Hence, the output column is $((dec, 8))$ (=input column of $D_{1,3}$), and the output row is $((inc, 4))$. Accordingly, the output columns of $D_{1,3}$ and $D_{1,4}$ are the same, while the output row of $D_{1,3}$ is $((inc, 7))$ and that of $D_{1,4}$ is $((inc, 9))$.

We continue now with submatrices $D_{2,1}, D_{2,2} \ldots, D_{2,4}$. The input column of $D_{2,1}$, is $((inc, 6))$, and the input row is $((dec, 8), (inc, 4))$. The letters are different. To get the outputs, we have determine the intermediate column where the basic parts of the sequence of the input row meet. Hence, the input column in the next step is $((inc, 6))$ and the input row is $(dec, 8)$. We have to use B13. The output row will be $((dec, 8))$ and the output column $((inc, 6))$. The latter is the input column of the second part of $D_{2,1}$. The input row is $((inc, 4))$. We have to use B11, subcase $l_1 > l_2$. Then, the last column is $((eq, 4), (inc, 2))$ and the last row is $((eq, 4))$. Thus, the output row of $D_{2,1}$ is $((dec, 8), (eq, 4))$, and the output column is $((eq, 4), (inc, 2))$.

Now we should continue with $D_{2,2}$. Here the input column has two basic parts, and so we have to divide up the computation. So far we have used a row-by-row algorithm. However, the algorithm is much easier to program if we compute the upper part of $D_{2,2}$ and then continue with the upper parts of further submatrices $D_{2,3}$ and $D_{2,4}$. The computation of the remaining submatrices is done in a similar way.

Computational complexity, remarks, and summary

The algorithm is easy to program if we use pointers showing the border rows and columns of all submatrices, and add further pointers if an output vector is divided up into more than one basic part.

We have to consider all submatrices $D_{i,j}$, and so a lower bound of the computational complexity is $\Omega(k \cdot l)$. In the worst case, we end up, after processing some submatrices, with a subdivision of the complete edit matrix into subblocks of size 1×1. This results in a worst time complexity of $O(n \cdot m)$. Generally, the actual runtime of the algorithm depends on the lengths of the runs. The longer the runs in A and B, the faster the algorithm.

However, for cost function (2) and the special case where all runs in A and B are of equal length, i.e. $n_1 = n_2 = \ldots = n_k = m_1 = m_2 = \ldots = m_l$, the actual time complexity reaches the lower bound. This can easily be seen from cases C and D in Section 4, as only non-quadratic matrices will be split into smaller submatrices. In order to obtain the final element of $D_{k,l}$ holding the string distance $\delta(A, B)$ we actually have to compute only the final element and the basic type of the output row and output column of each submatrix $D_{i,j}; i = 1, \ldots, k;\ j = 1, \ldots, l$. However, these entities can be immediately inferred from the final element of $D_{i-1,j-1}$ and the input row and input column of $D_{i,j}$. Hence, there is no need to explicitly compute all values in the input and output rows and columns, and the total number of operations to obtain $\delta(A, B)$ is $O(k \cdot l)$.

One can naturally ask whether an easier computation is possible without the restriction that all runs in A and B are of equal length. The first idea would be to use a dynamic programming-type approach for submatrices, similarly to the basic algorithm of Wagner and Fisher. More precisely, let $D_{i,j}$ be some submatrix in D with input vectors of lengths l_1 (column) and l_2 (row). Let $d_{i,j}$ and $d_{i+1,j+1}$ be the final elements of $D_{i-1,j-1}$ and $D_{i,j}$, respectively, and let $d_{i,j+1}$ and $d_{i+1,j}$ be the last elements of input row and input column of the $D_{i,j}$, respectively. Then, a simple generalization of (0) would give

$$d_{i+1,j+1} = \min(d_{i,j} + \max(l_1, l_2), d_{i,j+1} + l_1, d_{i+1,j} + l_2) \quad (7)$$

if the letters are different, and

$$d_{i+1,j+1} = \min(d_{i,j} + |l_1 - l_2|, d_{i,j+1} + l_1, d_{i+1,j} + l_2) \quad (8)$$

otherwise.

Now, we can see that this approach can not be used even for the easier cost function (2). It can be proved in a somewhat tiresome way that formula (7) is valid, i.e. we could use the classical dynamic programming approach for different letters. However, (8) does not hold. We can see this in our example, at submatrix $D_{3,1}$. This is as follows (using full notation)

						a						
	14	13	12	11	10	9	8	7	6	6	6	6
	15	14	13	12	11	10	9	8	7	6	6	6
a	16	15	14	13	12	11	10	9	8	7	6	6
	17	16	15	14	13	12	11	10	9	8	7	6

Here, formula (8) is clearly not valid. The reason for this is simply the fact that the input row consists of different basic types of cost sequences, and this will influence the right bottom corner of the matrix. Naturally, we cannot use only (7) as a speeding-up method, because this formula does not give us the basic parts necessary for computations at the submatrices with the same letters, as we have seen.

In summary, we have given an algorithm for edit distance computation of run-length coded strings. It determines the minimum cost sequence of edit operations needed to transform one string into another. The algorithm uses as basic data structure an edit matrix similar to the one of the algorithm by Wagner and Fisher [1]. However, depending on the particular strings to be compared, only a part of this edit matrix needs to be computed. We have discussed two special cases of the cost function where (1) all deletions, insertions, and substitutions have cost equal to one, and (2) deletions and insertions have cost equal to one while substitutions have cost equal to two. In the worst case, our algorithm has a time complexity of $O(n \cdot m)$, where n and m give the lengths of the two strings to be compared. In the best case, the time complexity is $O(k \cdot l)$ where k and l are the numbers of runs of identical symbols in the two strings under comparison.

References

[1] Wagner, R. A./Fischer, M. J.: The string-to-string correction problem. Journal of the ACM, Vol. 21, No. 1, 1974, 168-173.

[2] Levensthtein, V. I.: Binary codes capab le of correcting deletions, insertions, and reversals. Cybernetics and Control Theory, Vol. 10, No. 8, 1966, 707-710.

[3] Masek, W. J./Paterson, M.S.: A faster algorithm for comparing string-edit distances. Journal of Computer and System Sciences, Vol. 20, No 1, 1980, 18-31.

[4] Hunt, J.W./Szymanski, T.G.: A fast algorithm for computing longest common subsequences. CACM, Vol. 20, No. 5, 1977, 350-353.

[5] Myers, E.W.: An $O(ND)$ difference algorithm and its variations. Algorithmica, Vol. 1, 1986, 251-266.

[6] Ukkonen, E.: Algorithms for approximate string matching. Inform. and Control, Vol. 64, 1985, 100-118.

[7] Aho, A.V.: Algorithms for finding patterns in strings. In J. van Leeuwen (ed.): Handbook of theoretical computer science. Elsevier Science Publishers B. V., 1990, 255-300.

[8] Sankoff, D./Kruskal, J.B. (eds.): Time warps, string edits, and macromolecules; the theory and practice of sequence comparsion. Addison Wesley Publ. Co., Reading, Ma., 1983.

[9] Maes, M.: Polygonal shape recognition using string matching techniques. Pattern Recognition, Vol. 24, No. 5, 1991, 433-440.

[10] Wang, Y.P./Pavlidis, T.: Optimal correspondences of string subsequences. In Baird, H. (ed.): SSPR 90, Preproceedings International Association for Pattern Recognition Workshop on Syntactic and Structural Pattern Recognition, Murray Hill, New Jersey, 1990, 460-479.

[11] Margalit, A./Rosenfeld, A.: Reducing the expected computational cost of template matching using run length representation. Pattern Recognition Letters 11, 1990, 255-265.