Parametric string matching and its application to pattern recognition

H. Bunke [*] J. Csirik [†]

Abstract

String matching is a useful concept in pattern recognition that is constantly receiving attention from both theoretical and practical points of view. In this paper we propose a generalized version of the string matching algorithm by Wagner and Fischer [1]. It is based on a parametrization of the edit cost. We assume constant cost for any delete and insert operation, but the cost for replacing a symbol is given as a parameter r. For any two given strings A and B, our algorithm computes the edit distance of A and B in terms of the parameter r. We give the new algorithm and study some of its properties. Its time complexity is $O(n^2 \cdot m)$, where n and m are the lengths of the two strings to be compared and $n \leq m$. We also discuss potential applications of the new string distance to pattern recognition. Finally, we present some experimental results.

key words: string matching, inference of edit cost, dynamic programming, symbolic clustering, symbolic nearest-neighbor classification.

1 Introduction

A large variety of different methods in pattern recognition have been developed during recent years. The majority of these approaches, however, are based on the same principle. An unknown pattern to be recognized is compared, or matched, with stored models in order to determine that individual model or class of models which is most similar to the unknown pattern. From an overall point of view, one can distinguish between statistical and numerical methods in pattern recognition on the one hand, and symbolic methods on the other hand. In the statistical approach, the patterns under study are represented by means of feature vectors, and pattern similarity is based on a numerical distance function or on concepts from probability theory. Symbolic approaches mainly rely on symbolic data structures for the representation of the models and the unknown patterns to be recognized. Here, the similarity of patterns is expressed in terms of the similarity of the data structures used for pattern representation.

Common data structures for symbolic pattern representation are strings, trees and graphs. There is a trade-off between computational efficiency and expressive power. The more complex a data structure is, the easier it is to represent the properties of the patterns under study. On the other hand, a high degree of complexity usually implies high computational costs of the operations carried out on the underlying data structures.

[*]Institut für Informatik und angewandte Mathematik, Universität Bern, Switzerland
[†]Department of Applied Computer Science, University of Szeged, Hungary

Strings of symbols over a finite alphabet are very simple data structures for pattern representation. Because of the low computational complexity of many operations on strings, this formalism has received much attention in the past. For measuring the similarity of patterns that are represented by strings, one needs the notion of string similarity. The first algorithms for computing the distance, or similarity, of two strings was proposed long ago [2]. Subsequently, other authors have studied the string distance computation problem [1]. Generally, one introduces a set of basic edit operations with costs associated with each edit operation and defines the distance of two strings as the minimum cost sequence of edit operations needed to transform one string into the other. The algorithm of Wagner and Fischer [1] is usually referred to as the standard solution to the 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.

It has also been shown that string distance is closely related with the problem of finding the longest common subsequence of two strings. For this problem, a number of algorithms with a lower computational complexity than that of Wagner and Fischer's algorithm have been published [3]-[5]. The asymptotically fastest algorithm for the original string distance problem is that of Masek and Patterson [6], having a time complexity of $O(n^2/\log n)$, where again n and m are the lengths of the two strings to be compared, and $n \geq m$. A recent survey on algorithms for string matching has been given in [7]. Another article that reviews and compares a number of different string matching algorithms is [8].

There are a number of interesting applications of string matching in the discipline of pattern recognition. One class of problems that has received much attention is the recognition of two-dimensional shapes [9]-[11]. Character recognition has been addressed in [12]. Recently it was shown that string matching is useful for a number of medical applications [13]. A bar code reading procedure based on string matching has been described in [14]. A reference covering applications of string matching in speech recognition, molecular biology and other fields is [15].

One advantage of the use of string matching in pattern recognition is that there is no inference or learning procedure of the models or prototype patterns required. That is, given a finite number of models from one or more classes, their string representation can be directly used for matching with the string representation of the unknown patterns. This is a great advantage over syntactic or artificial intelligence-based approaches, where a grammar, a set of rules, logical clauses or similar representations have to be inferred from the sample set of patterns. However, a serious problem that remains in string matching is the proper adjustment of the costs of the elementary edit operations. The distance of any two strings usually depends critically on these costs. Changing the edit costs only slightly may result in a drastic change in recognition performance. For example, let

$x = baacb$,

$y = acba$,

$z = cacba$

and the cost for inserting or deleting any symbol be equal to one. Now, if the cost for replacing any symbol is $r = 2$, then $d(x, y) = 3$ and $d(x, z) = 4$. This means that x is more similar to y than to z. However, if $r = 0.5$, then $d(x, y) = 2.5$ and $d(x, z) = 2$. If y and z are representatives of two different classes and x is an unknown pattern, then the classification result depends critically on the replacement cost r. We conclude that proper adjustment of the basic edit operation costs is one of the major keys to successfully solving string matching-based pattern recognition problems.

Despite a large body of work on string matching, motivated by theoretical and practical issues, the problem of edit cost adjustment has not yet been adequately addressed in the literature. Usually, one has at hand a sample set of prototype patterns and has to adjust the edit costs in such a way that some recognition performance criterion is optimized, on the basis of the sample set. Thus a number of experiments are usually performed until a combination of basic edit costs has been found that satisfies the given optimization criterion in an acceptable way. However, this procedure of trial and error may require testing of a large number of different edit costs. Furthermore, after an acceptable combination has been found, it is not clear whether or not a better one exists. Thus, this commonly practised procedure of blindly searching for edit costs is somewhat unsatisfactory.

In this paper we propose a generalized version of the string matching algorithm by Wagner and Fischer. It is based on a parametrization of the edit costs. For the purpose of simplification, we use only one parameter, r, representing replace, or substitution, cost. Insert and delete costs are kept constant. For any two strings, our new algorithm computes their edit distance in terms of the parameter r. If we are given a set of sample patterns and some optimization criterion, then we propose to use our new parametric distance algorithm on the sample set. As a result, we get a number of intervals for the parameter r. In each of these intervals, the optimization criterion will take the same value. Hence, we can restrict the search for the optimum value of the parameter r to these intervals and do not have to test all possible values of r. It will be shown that the number of intervals is usually much smaller that the number of all possible edit cost values. This results in a faster and more systematic way for finding optimal edit costs for string matching.

2 Preliminaries

We are given two strings $A = a_1 a_2 \dots a_n$ and $B = b_1 b_2 \dots b_m$ over a finite alphabet V. The problem we would like to solve is the computation of the edit distance of A and B.

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 $A = D_1 c_i D_2, B = D_1 c_j D_2$ 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 $E = e_1, e_2, \dots, e_k$ be an edit sequence. We will say that B is derivable from A via E if there is a sequence of strings $D_0, D_1, \dots, D_k$ 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 $E = e_1, e_2, \dots, e_k$ by

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

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

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

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 $A(i,j) = a_i a_{i+1} \dots a_j$ and $B(i,j) = b_i b_{i+1} \dots b_j$. For short, we shall write A_i for $a_1 a_2 \dots a_i$, B_j for $b_1 b_2 \dots b_j$ and $\delta_{i,j}$ for $\delta(A_i, B_j)$. Wagner and Fischer's algorithm constructs an $(n+1) \times (m+1)$ 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

$$d_{i,j} = \min(d_{i-1,j-1} + \delta(a_i, b_j), d_{i-1,j} + \delta(a_i, \varepsilon), d_{i,j-1} + \delta(\varepsilon, b_j)) \tag{0}$$

for $1 \le i \le n$, $1 \le j \le m$, 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.

In this paper, we make the following assumptions regarding our cost function

$$\delta(v_i, v_j) = \begin{cases} r, 0 < r \le 2 & \text{if } v_i \neq v_j;\ v_i, v_j \in V. \\ 0 & \text{if } v_i = v_j;\ v_i, v_j \in V. \end{cases}$$

$$\delta(\varepsilon, v_i) = \delta(v_i, \varepsilon) = 1;\ v_i \in V.$$

That is, we (arbitrarily) fix the cost for insertion and deletion to be constantly equal to one. But the replacement costs r may vary between zero and two. Clearly, the value of zero is a natural lower bound for the cost of any basic edit operation. The upper bound equal to two is justified by the observation that the distance of two arbitrary strings will be the same for any $r \ge 2$ as the string matching algorithm always prefers an insertion together with a deletion over a replacement, independently of the concrete value of r as long as $r \ge 2$.

3 The parametric string matching algorithm

As data structures, we use a cost matrix of dimension $(n+1) \times (m+1)$, similar to the algorithm by Wagner and Fischer [1]. In each position of this matrix, we store a list consisting of elements $[k, (x, y)]$. If $D(i,j)$ contains a list element $[k, (x, y)]$, then $\delta(A_i, B_j) = k$ for $r \in (x, y)$, where r gives the replacement cost and $0 \le x < y \le 2$. Generally, k is a function of the replacement cost r. Furthermore, the algorithm uses a stack. Each element on the stack is of the form [*initial-position, actual-interval*] where *initial-position* gives the position (i,j), $1 \le i \le n$, $1 \le j \le m$ in the cost matrix where we have to start the recomputation of the matrix elements for the actual interval (x, y) of the replacement cost.

Next, we give an informal description of the algorithm. Each position in the 0-th row and column of the cost matrix consists of a one-element list. The lists in the first row are $([0,(0,2)]), ([1,(0,2)]), \dots, ([m,(0,2)])$. Similarly, in the first column we have $([0,(0,2)]), ([1,(0,2)]), \dots, ([n,(0,2)])$.

After we have assigned the appropriate values to the 0-th row and column of the cost matrix, the dynamic programming formula (0) is used to compute all other elements of the matrix. If the minimum of the three expressions is the same for all values of r between 0 and 2, then we simply take this value as the cost. If, for different values of r, different parts of (0) give the minimum, then we have to divide up the following computations. As the three parts in (0) are all linear functions, say f_1, f_2, f_3, it is easy to find the subdivision of the interval $(0, 2)$ into subintervals $(0, x), (x, y)$ and $(y, 2), 0 \leq x \leq y \leq 2$ such that f_1 is the minimum in $(0, x), f_2$ is the minimum in (x, y), and f_3 is the minimum in $(y, 2)$. In the following part of the edit matrix, we will restrict ourselves to $r \in (0, x)$ and we will compute the further cost values under the assumption that f_1 is the minimum of f_1, f_2 and f_3. However, we will mark any element of the matrix where such a division has been made, and return to it to compute the cost values for $r \in (x, y)$ and $r \in (y, 2)$, too. In the further course of the algorithm, if, for any of the intervals $(0, x), (x, y)$ or $(y, 2)$, the minimization in (0) gives different results for different subintervals, then we further split the actual interval. This procedure of splitting a cost interval and computing the cost values individually for each subinterval is continued as long as necessary.

A pseudocode description of the algorithm is given below. The algorithm consists of three procedures. The main procedure is **parametric_distance**. This procedure calls **stringmatch**, and **stringmatch** calls **make-entry**. The backpointers in lines 18, 21 and 26 have the same meaning as in the algorithm by Wagner and Fischer [1]. They can be used after termination of the algorithm to get the actual minimum cost sequence of edit operations for each of the considered intervals.

```
    main procedure parametric_distance $(A, B)$

input: two strings $A = a_1 \ldots a_n$ and $B = b_1 \ldots b_m$ to be matched
output: the cost matrix $D(0 : n, 0 : m)$, where $D(n, m)$ contains
        $\delta(A, B)$ as a function of the replacement cost $r$

method:

1. initialize stack with [initial-position, actual-interval]=[(1,1), (0,2)];

   /* first column and row of cost matrix */
2. $D(0,0) = ([0, (0,2)])$;
3. for $i = 1$ to $n$, do $D(i,0) = ([i, (0,2)])$;
4. for $j = 1$ to $m$, do $D(0,j) = ([j, (0,2)])$;

   /* initialization of rest of cost matrix */
5. for $i = 1$ to $n$, do
6.     for $j = 1$ to $m$, do $D(i,j)$ =NIL; /*NIL is the empty list */

   /* main loop */
7. while stack is not empty, do
       begin
8.         pop [initial-position, actual-interval] from stack;
```

```
9.          stringmatch (initial-position, actual-interval)
        end
end parametric-distance
```

procedure stringmatch (initial-position, actual-interval)

```
input: the initial position of the partial cost matrix to be computed and the actual interval of
       the replacement cost to be considered. Let
       initial-position = $(I, J)$ and
       actual-interval = $(X, Y)$.
output: updated cost matrix containing the edit distance for the actual interval of the
        replacement cost

method:

    /* initial row */
10. for $j = J$ to $m$, do make_entry ($I, j$, actual_interval);

    /* remaining part of cost matrix */
11. for $i = I + 1$ to $n$, do
12.     for $j = 1$ to $m$, do make_entry ($i, j$, actual_interval);
end stringmatch
```

procedure make_entry (i, j, actual_interval)

```
input: a row index $i$ and column index $j$ of the cost matrix, and the interval of the
       replacement cost actually considered. Let
       actual-interval = $(X, Y)$
output: updated cost matrix at position $(i, j)$ for replacement cost from the actual interval

method:
13. $f_1 = k_1 + r$, where $[k_1, (x_1, y_1)]$ is the last list element in $D(i-1, j-1)$;
14. $f_2 = k_2 + 1$, where $[k_2, (x_2, y_2)]$ is the last list element in $D(i-1, j)$;
15. $f_3 = k_3 + 1$, where $[k_3, (x_3, y_3)]$ is the last list element in $D(i, j-1)$;
16. if $f_l = \min(f_1, f_2, f_3)$ is uniquely determined over actual-interval,
    then begin /* no subdivision of actual_interval required */
17.             add $[f_l, (X, Y)]$ as last list element to $D(i, j)$;
18.             set backpointer from $[f_l, (X, Y)]$ to each $[k_s, (X_l, Y_l)]$ in
                $D(i-1, j-1), D(i-1, j)$ or $D(i, j-1)$
                with $f_s = f_l, s = 1, 2, 3$;
        end

    else /* subdivision required */
19.     if $f_l$ =min$(f_1, f_2, f_3)$ over the interval $(X, Z)$, and $\min(f_1, f_2, f_3)$
            over $(Z, Y)$ is different from $f_l$, and
            $X < Z < Y$
            then begin
20.                 add $[f_l, (X, Z)]$ as last element to $D(i, j)$;
21.                 set backpointer from $[f_l, (X, Z)]$ to each $[k_s, (X_l, Y_l)]$ in
                    $D(i-1, j-1), D(i-1, j)$, or $D(i, j-1)$
                    with $f_s = f_l, s = 1, 2, 3$;
22.                 push $[(i, j), (Z, Y)]$ on stack ;
23.                 actual_interval = $(X, Z)$
                end
    /* joining of adjacent intervals with identical costs */
24. if last and second last list element in $D(i, j)$ are of the
       form $[k, (x, z)]$ and $[k, (z, y)]$, respectively,
25. then begin replace them by one element $[k, (x, y)]$;
26.         set backpointer from $[k, (x, y)]$ to the same element
            as backpointer from $[k, (x, z)]$ pointed to
      end
end make-entry
```

4 An example

Let $V = \{a, b, c\}$, $A = baacb$ and $B = acba$. The final result of the algorithm for parametric string distance computation is shown in Fig. 1. For easier graphical representation, all backpointers have been omitted in Fig. 1.

It is easy to see that until $D(2, 2)$ all elements are unique. For example, at $D(1, 1)$ we have $f_1 = r, f_2 = f_3 = 2$ and so $f_1 = \min(f_1, f_2, f_3)$ for $r \in (0, 2)$. Similarly, at $D(2, 1)$, for example, $f_1 = 1, f_2 = 1 + r, f_3 = 3$ and so $f_1 = \min(f_1, f_2, f_3)$ for $r \in (0, 2)$. At $D(2, 2)$ we have $f_1 = 2r, f_2 = 2+r, f_3 = 2$. So, $f_1 = \min(f_1, f_2, f_3)$ if $r \in (0, 1)$ and $f_2 = \min(f_1, f_2, f_3)$ if $r \in (1, 2)$. We split up the computation at $D(2, 2)$ and continue with $r \in (0, 1)$. But the position $(2, 2)$ together with the interval $(1, 2)$ will be pushed onto the stack.

For actual-interval $= (0, 1)$, we will get at $D(2, 3)$ $f_1 = f_2 = 1 + 2r = \min(f_1, f_2, f_3)$ and at $D(2, 4)$ $f_1 = 2 = \min(f_1, f_2, f_3)$. The computation is continued until $D(5, 4)$, where the interval $(0, 1)$ will be further split into the subintervals $(0, 2/3)$ and $(2/3, 1)$.

Then we return to $D(2, 2)$ and recompute all subsequent matrix elements for $r \in (1, 2)$. As one can easily verify, no further split of the interval (1,2) will be necessary. However, a number of adjacent intervals having identical costs will be joined. For example, at $D(2, 4)$ we get for $r \in (1, 2)$ $f_1 = 2 = \min(f_1, f_2, f_3)$ and so we replace the last two list elements $[2, (0, 1)]$ and $[2, (1, 2)]$ by $[2, (0, 2)]$. Similar join operations occur at $D(3, 1), D(3, 2), D(4, 1), D(4, 2), D(5, 1), D(5, 2)$ and $D(5, 3)$. At $D(5, 4)$ we can join the interval $(2/3, 1)$ with $(1, 2)$ and get the list element $[3, (2/3, 2)]$. From $D(5, 4)$, we conclude

$$d(A, B) = \begin{cases} 1 + 3r & \text{if } r \in (0, 2/3) \\ 3 & \text{if } r \in (2/3, 2) \end{cases}$$

		a	c	b	a
	$[0,(0,2)]$	$[1,(0,2)]$	$[2,(0,2)]$	$[3,(0,2)]$	$[4,(0,2)]$
b	$[1,(0,2)]$	$[r,(0,2)]$	$[1+r,(0,2)]$	$[2,(0,2)]$	$[3,(0,2)]$
a	$[2,(0,2)]$	$[1,(0,2)]$	$[2r,(0,1)]$ $[2,(1,2)]$	$[1+2r,(0,1)]$ $[3,(1,2)]$	$[2,(0,1)]$ $[2,(1,2)]$
a	$[3,(0,2)]$	$[2,(0,1)]$ $[2,(1,2)]$	$[1+r,(0,1)]$ $[1+r,(1,2)]$	$[3r,(0,1)]$ $[2+r,(1,2)]$	$[1+2r,(0,1)]$ $[3,(1,2)]$
c	$[4,(0,2)]$	$[3,(0,1)]$ $[3,(1,2)]$	$[2,(0,1)]$ $[2,(1,2)]$	$[1+2r,(0,1)]$ $[3,(1,2)]$	$[4r,(0,1)]$ $[4,(1,2)]$
b	$[5,(0,2)]$	$[4,(0,1)]$ $[4,(1,2)]$	$[3,(0,1)]$ $[3,(1,2)]$	$[2,(0,1)]$ $[2,(1,2)]$	$[1+3r,(0,2/3)]$ $[3,(2/3,1)]$ $[3,(1,2)]$

Fig. 1: An example

5 Some properties of the algorithm

Theorem 1 *a) The algorithm for parametric distance calculation is complete over the interval (0,2) of replacement cost. That is, after termination each cost matrix element $D(i,j)$, $0 \le i \le n$,$0 \le j \le m$ holds a list ($[k_1,(x_1,y_1)]$, $[k_2,(x_2,y_2)]$, $\ldots$, $[k_L,(x_L,y_L)]$ such that*

$$\bigcup_{i=1}^{L}(x_i,y_i) = (0,2).$$

b) The algorithm is correct. That is, for any $1 \le i \le n, 1 \le j \le m$, if we have an element $[k,(x,y)]$ in $D(i,j)$, then $\delta(A_i,B_j)=k$ for $r \in (x,y)$.

Proof: a) By means of lines 2-4, the interval of any element in the 0-th row and column is (0,2). Furthermore, we set the actual interval equal to (0,2) at stack initialization in line 1. Later, whenever an interval (x,y) is split into two subintervals (x,z) and (z,y), then the union of these subintervals gives the original interval (x,y) (see lines 19-23). The subinterval (x,z) will be processed immediately after the split (lines 10-12), while (z,y) will be remembered on the stack and processed later (see lines 7-9). Thus, after termination of the algorithm, the union of all subintervals in any matrix element covers the complete interval (0,2).

b) Each of the intervals $(x_1,y_1),(x_2,y_2)$ and (x_3,y_3) in lines 13-15 is either identical with the actual interval (x,y) or contains (x,y) as a subinterval, so each of the values k_1,k_2 and k_3 gives the minimum cost for the actual interval at $D(i-1,j-1)$, $D(i-1,j)$ and $D(i,j-1)$, respectively. Therefore, the determination of the minimum in lines 13-16 corresponds to the determination of the minimum in (0) and the correctness of the algorithm immediately follows from the correctness of Wagner and Fischer's algorithm [1]. □

From this theorem, we immediately get the following corollary.

Corollary 2 *At any position $D(i,j), 1 \le i \le n, 1 \le j \le m$ in the cost matrix, we have the distance between A_i and B_j in the following form:*

$$D(i,j) = \delta(A_i, B_j) \quad = \quad \begin{array}{lll} x_1 + y_1 \cdot r & if & 0 = r_0 \leq r \leq r_1 \\ x_2 + y_2 \cdot r & if & r_1 \leq r \leq r_2 \\ \vdots & & \vdots \\ x_k + y_k \cdot r & if & r_{k-1} \leq r \leq r_k = 2 \end{array} \tag{5.1}$$

where $0 = r_0 < r_1 < r_2 < \ldots < r_{k-1} < r_k = 2$ *relates to rational numbers and* $x_1, \ldots, x_k, y_1, \ldots, y_k$ *are non-negative integers.*

□

In the following, we consider any element $D(i,j)$ in the cost matrix after termination of the algorithm.

Lemma 3 *For* $i = 1, \ldots, k-1$, *the equation*

$$x_i + y_i \cdot r_i = x_{i+1} + y_{i+1} \cdot r_i$$

holds.

□

This lemma follows immediately from the observation that the minimum cost function in the interval (r_{i-1}, r_i) and the minimum cost function in (r_i, r_{i+1}) must have the same value at the common point r_i.

Lemma 4 *For* $i = 1, \ldots, k-1$, *we have* $y_i \neq y_{i+1}$.

Proof: If $y_i = y_{i+1}$, then $x_i = x_{i+1}$ because of Lemma 3. Hence, the cost function in (r_{i-1}, r_i) is identical with the cost function in (r_i, r_{i+1}). However, this is not possible because adjacent intervals with identical cost functions will be merged by means of the statements in lines 24-26.

□

Lemma 5 *For* $i = 1, \ldots, k-1$, *the inequality* $y_i > y_{i+1}$ *holds.*

Proof: As one can see from the graphical illustration in Fig. 2, y_i and y_{i+1} are the slopes of the cost functions $x_i + y_i r$ and $x_{i+1} + y_{i+1} r$, respectively. From Lemma 3, we know that $x_i + y_j r_i = x_{i+1} + y_{i+1} r_i$, and from Lemma 4 $y_i \neq y_{i+1}$. Now, if $y_i < y_{i+1}$, then $x_{i+1} + y_{i+1} \cdot r$ cannot be the minimum edit cost in (r_i, r_{i+1}). As this is a contradiction, we conclude $y_i > y_{i+1}$.

□

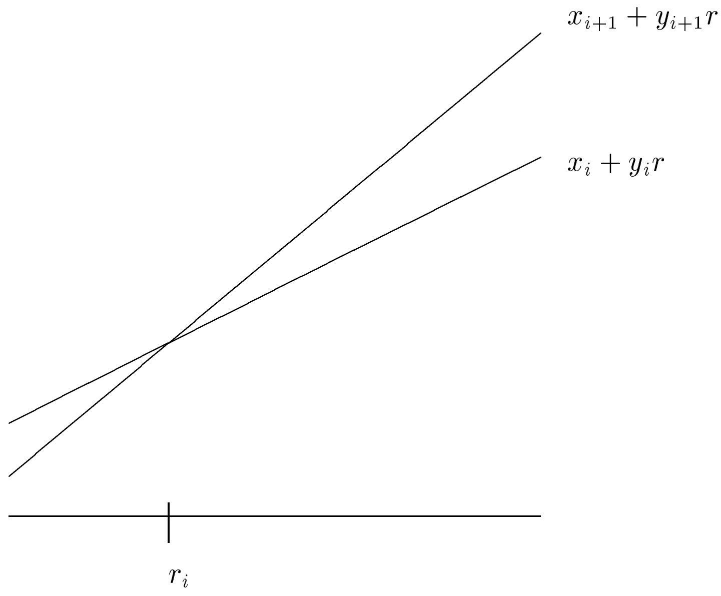

Fig. 2: Graphical illustration of edit cost function

As any y_i, $1 \leq i \leq k$ gives the number of substitutions of the optimal edit sequence for $r \in (r_{i-1}, r_i)$, we get the following corollary.

Corollary 6 *$y_i \leq \min(n, m)$, where n and m are the lengths of the two strings to be compared.*

This leads us to the following theorem, which gives us an upper bound on the number of subintervals, i.e. list elements, to be considered in each matrix position.

Theorem 7 *For the parametric distance of any two strings of length n and m, respectively, given in the form of Corollary 2, the number of subintervals is less than or equal to $\min(n, m)$.*

The proof follows immediately from Corollary 2, Lemma 5 and Corollary 6.

Theorem 8 *The time complexity of the algorithm for parametric string distance computation is $O(n^2 m)$, where n and m are the lengths of the two strings to be compared, respectively, and $n = \min(n, m)$.*

Proof: For each subinterval, we have to do a constant number of operations at any of the matrix positions. There are $O(n \cdot m)$ different matrix positions to be considered. The number of different subintervals is $O(n)$. This results in a time complexity of $O(n^2 m)$.
□

The next theorem shows that there is no conflict concerning the backpointers when two adjacent intervals with identical cost functions are merged, as the two elements have their backpointers pointing to the same element.

Theorem 9 *If two list elements $[k,(x,y)]$ and $[k,(y,z)]$ are joined by means of the statements in lines (24) and (25) of the algorithm, then they have their backpointers pointing to the same list element.*

Proof: For the proof, we consider two adjacent intervals (x,y) and (y,z) with y as their common border. Let $[R_1,(x,y)]$ and $[S_1,(y,z)]$ be the last two list elements in $D(i-1,j-1)$. Similarly, let $[R_2,(x,y)]$ and $[S_2,(y,z)]$ be the last two list elements in $D(i-1,j)$, and $[R_3,(x,y)]$ and $[S_3,(y,z)]$ the last two list elements in $D(i,j-1)$. All of R_i and S_i are linear functions of the replacement cost r, $i = 1,2,3$.

Assume that $D(i,j)$ contains $[R,(x,y)]$. Now, we consider what happens when a new list element $[S,(y,z)]$ is added to $D(i,j)$. According to our assumption, we have $R = S$ and merge the intervals (x,y) and (y,z) into (x,z), i.e. we replace the elements $[R,(x,y)]$ and $[R,(y,z)]$ in $D(i,j)$ by $[R,(x,z)]$.
Case 1: $R = R_i$ and $S = S_i$ for $i \in \{1,2,3\}$. Thus, the backpointers from $[R,(x,y)]$ and $[S,(y,z)]$ point to the same matrix position $D(i-1,j-1), D(i-1,j)$ or $D(i,j-1)$. An example for the matrix position $D(i-1,j)$, i.e. $i = 2$, is shown in Fig. 3. As $R = S$, we have $R_i = S_i$ and this means that $[R_i,(x,y)]$ and $[S_i,(y,z)]$ have been joined into the element $[R_i,(x,z)]$ by the algorithm before. Therefore, the backpointers from $[R,(x,y)]$ and $[S,(y,z)]$ point to the same element $[R_i,(x,z)]$.

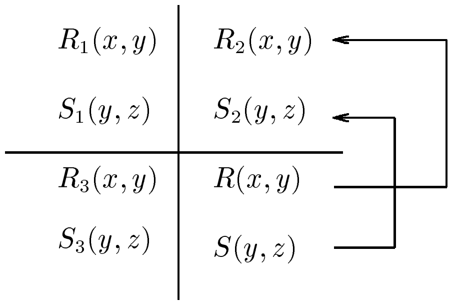

Fig. 3.: The positions $D(i-1,j-1)$, $D(i-1,j)$, $D(i,j-1)$ and $D(i,j)$ of the cost matrix with their second last list elements and an example of the backpointers.

Case 2: The backpointers from$[R,(x,y)]$ and $[S,(y,z)]$ point to different matrix positions.
2 a) Backpointer from $[R,(x,y)]$ to $D(i-1,j)$ and from $[S,(y,z)]$ to $D(i,j-1)$.
In this case, we have $R = R_2 + 1$ and $S = S_3 + 1$, and thus
(1) $R_2 = S_3$ for the whole interval (x,z).
As backpointers always point to minimum values, we know that
(2) $R_2 \leq R_3$ for the whole interval (x,y), and
(3) $S_3 \leq S_2$ for the whole interval (y,z).
As all of R_i, S_i, R and S, $i \in \{1,2,3\}$ are linear functions of r, we have at the common interval border y
(4) $R_i(y) = S_i(y)$ and $R(y) = S(y)$, $i \in \{1,2,3\}$.
From (2) and (4), we conclude (see Fig. 4)
(5) $R_3(z) \leq R_2(z)$

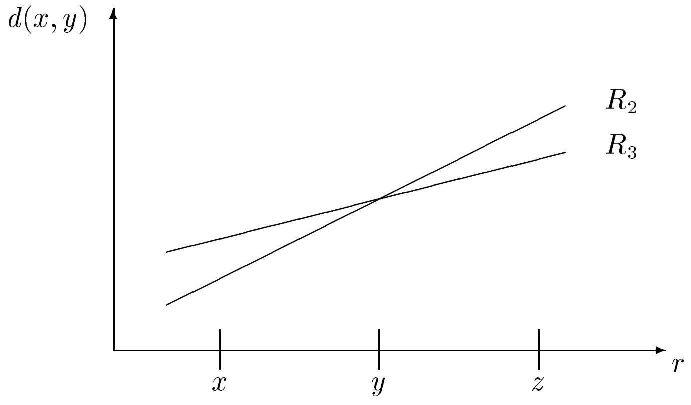

Fig. 4: Illustration for the proof of Theorem 9. If $R_2 \leq R_3$ at x, then $R_3 \leq R_2$ at z.

Combining (1), (3) and (5), we obtain

(6) $R_3(z) \leq R_2(z) = S_3(z) \leq S_2(z)$

As S_2 is minimum in (y,z), we know that

(7) $S_2(z) \leq R_2(z)$

and as S_3 is minimum in (y,z), we know that

(8) $S_3(z) \leq R_3(z)$.

From (6), (7) and (8), we obtain

(9) $R_2(z) = R_3(z) = S_2(z) = S_3(z)$

and from (4) and (9), we finally conclude

(10) $R_2 = R_3 = S_2 = S_3$ for the complete interval (x,z).

Therefore, R_2 and S_2 and also R_3 and S_3 have been joined by the algorithm before and there were pointers from $[R,(x,y)]$ to both $[R_2,(x,z)]$ and $[R_3,(x,z)]$, and from $[S,(y,z)]$ to both $[R_2,(x,z)]$ and $[R_3,(x,z)]$. Thus, the backpointers from $[R,(x,y)]$ point to exactly the same elements as the backpointers from $[S,(y,z)]$.

2 b) Backpointer from $[R,(x,y)]$ to $D(i-1,j)$

and from $[S,(y,z)]$ to $D(i-1,j-1)$ and $S = S_1$ (i.e. $a_i = b_i$).

As $R = R_2 + 1$ and $R = S$, we have

(11) $S_1 = R_2 + 1$ for the whole interval (x,z).

Because backpointers always point to minimum values, we know that

(12) $R_2 + 1 \leq R_1$ for the interval (x,y), and

(13) $S_1 \leq S_2 + 1$ for the interval (y,z).

At the common interval border y, we know that

(14) $R_2(y) + 1 = S_1(y) = R_1(y) = S_2(y) + 1$.

From (12) and (14), we conclude, similar to (5)

(15) $R_1(z) \leq R_2(z) + 1$.

Combining (11), (13) and (15), we obtain
(16) $R_1(z) \leq R_2(z) + 1 = S_1(z) \leq S_2(z) + 1$.
As S_1 is minimum in (y, z), we know that
(17) $S_1(z) \leq R_1(z)$
and as S_2 is minimum in (y, z), we know that
(18) $S_2(z) \leq R_2(z)$.
From (16), (17) and (18), we obtain
(19) $R_1(z) = S_1(z) = R_2(z) + 1 = S_2(z) + 1$
and from (19) and (14), we finally conclude
(20) $R_1 = S_1 = R_2 + 1 = S_2 + 1$ for the complete interval (x, z).
Therefore, R_1 and S_1, and also R_2 and S_2 have been joined by the algorithm before and the backpointers from $[R, (x, y)]$ and $[S, (y, z)]$ go to the same list elements.

2 c) Backpointer from $[R, (x, y)]$ to $D(i-1, j)$ and from $[S, (y, z)]$ to $D(i-1, j-1)$ and $S = S_1 + r$ (i.e. $a_i \neq b_i$). If we substitute R_1 by $R_1 + r$ and S_1 by $S_1 + r$, then this case can be treated in exactly the same way as case 2b.

All remaining cases are equivalent to one of the cases analyzed above.

□

6 The combination of different string distances

In this section, we are interested in combining different string distances. Such a combination will be important for the applications considered in section 7. As a parametric string distance according to (5.1) can be considered as a function of r, we sometimes use the term distance function instead of string distance from now on.

Let $D = \{d_1(r), d_2(r), \ldots, d_K(r)\}$ be a set of distance functions where, for $i = 1, \ldots, K$,

$$\begin{array}{rcll} d_i(r) & = & x_1^{(i)} + y_1^{(i)} r & if \quad 0 = r_0^{(i)} \leq r \leq r_1^{(i)} \\ & & x_2^{(i)} + y_2^{(i)} r & if \quad r_1^{(i)} \leq r \leq r_2^{(i)} \\ & & \vdots & \\ & & x_{L_i}^{(i)} + y_{L_i}^{(i)} r & if \quad r_{L_i-1}^{(i)} \leq r \leq r_{L_i}^{(i)} = 2 \end{array}$$

Then, the set of *critical points* of D is completely defined by the following two rules:

1. Any interval border $r_j^{(i)}$ is a critical point; $i = 1, \ldots, K$; $j = 0, \ldots, L_i$.
2. Any point of intersection between two different linear pieces of any two different $d_i(r)$ and $d_j(r)$ is a critical point. Thus, if

$$x_k^{(i)} + y_k^{(i)} r = x_l^{(j)} + y_l^{(j)} r$$

then r is a critical point; $i, j = 1, \ldots, K$; $i \neq j$; $k \in \{1, \ldots, L_i\}$; $l \in \{1, \ldots, L_j\}$; $(x_k^{(i)}, y_k^{(i)}) \neq (x_l^{(i)}, y_l^{(i)})$.

Given a set D of distance functions and the set $\{r_0, r_1, \ldots, r_M\}$ of critical points of D, where $0 = r_0 < r_1 < \ldots < r_M = 2$, we call (r_j, r_{j+1}) a *basic interval* of D, $j = 0, \ldots, M-1$. We will also call any interval $(r_j^{(i)}, r_{j+1}^{(i)}), i = 1, \ldots, K; j = 1, \ldots, L_i - 1$ a basic interval of the distance function $d_i(r)$.

Example 10

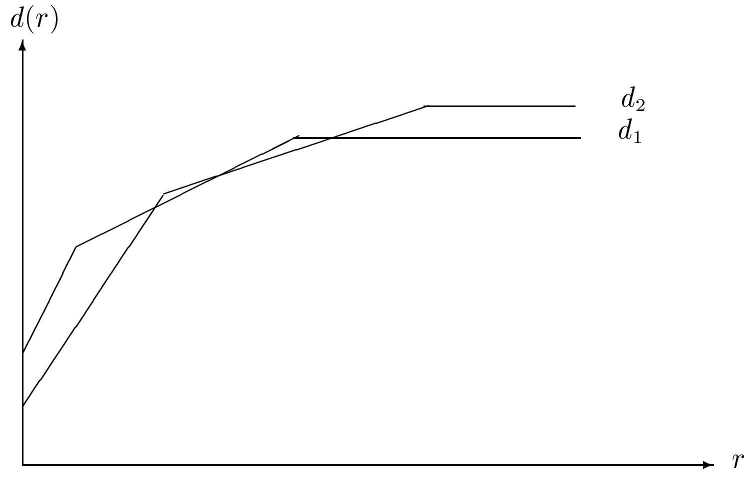

Fig. 5: Graphical illustration of critical points and basic intervals.

$$\begin{array}{rcll} \textit{Let } d_1(r) & = & 8 + 40r & 0 \le r \le 0.2, \\ & & 14 + 10r & 0.2 \le r \le 1, \\ & & 24 & 1 \le r. \\ d_2(r) & = & 4 + 40r & 0 \le r \le 0.4, \\ & & 18 + 5r & 0.4 \le r \le 1.6, \\ & & 26 & 1.6 \le r. \end{array}$$

Then, the set of criticial points of $D = \{d_1(r), d_2(r)\}$ is given by $\{0, 0.2, 0.33, 0.4, 0.8, 1.0, 1.2, 1.6, 2.0\}$. The basic intervals of D are $(0, 0.2), (0.2, 0.33), (0.33, 0.4)$, $(0.4, 0.8)$, $(0.8, 1.0)$, $(1.0, 1.2), (1, 2, 1.6), (1.6, 2.0)$. A graphical illustration is given in Fig. 5.

$\square$

Lemma 11 *Let $D = \{d_1(r), d_2(r), \ldots, d_K(r)\}$ be a set of distance functions, each having at most L basic intervals. Then, D has at most LK^2 basic intervals.*

Proof: We iteratively consider $D_1 = \{d_1(r)\}, D_2 = D_1 \cup \{d_2(r))\}, \ldots, D_K = D_{K-1} \cup \{d_k(r)\}$. According to our assumption, D_1 has at most L basic intervals. When we add $d_2(r)$, we get at most L additional intervals from the new interval borders $r_1^{(2)}, r_2^{(2)}, \ldots, r_{L-1}^{(2)}$. Each line segment of $d_2(r)$ with constant slope can intersect $d_1(r)$ at most two times. As there are L such line segments, we get at most $2L$ more basic intervals. Hence, there are at most $4L$ basic intervals for D_2.

Generally, when adding $d_i(r)$ to D_{i-1} to construct D_i we generate at most L new intervals due to L new interval borders $r_1^{(i)}, r_2^{(i)}, \ldots, r_L^{(i)}$. Each line segment of $d_i(r)$ of constant slope can intersect each $d_1(r), \ldots, d_{i-1}(r)$ at most two times, $i = 2, \ldots, K$. This leads to the following recursion formula

$Z_1 = 1$

$Z_i = Z_{i-1} + 1 + 2(i-1) = Z_{i-1} + 2i - 1;\ i = 2, \ldots, K,$

where LZ_i is the number of basic intervals of D_i. It can be easily proven that

$Z_i = i^2$.

So the number of basic intervals of D_K is at most LK^2.

□

This Lemma gives us an upper bound on the number of basic intervals. As we shall see in section 8, however, the number of basic intervals actually achieved in practical applications can be much smaller.

7 Potential applications of the parametric string distance in pattern recognition

In this section, we give some examples showing how the proposed parametric string distance can be applied to pattern recognition problems. Generally, we assume that a string is an abstract representation of a pattern where the individual symbols of the string correspond to certain features of the pattern. For example, a string may represent a sequence of contour elements of a two-dimensional shape, or a sequence of amplitude values of a one-dimensional speech signal. In the following, we consider two sample sets of strings from different classes, $C_1 = \{x_1, \ldots, x_N\}$ and $C_2 = \{y_1, \ldots, y_M\}$. What we want is to find a suitable value r of the replacement cost such that certain classification or clustering criteria are satisfied.

There are criteria where the goal is to minimize or maximize a sum of string distances. An example is the minimization of the average intra-class distance, i.e. the average distance within C_1 and C_2. Here, we want to minimize

$$A = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} d(x_i, x_j) + \sum_{i=1}^{M-1} \sum_{j=i+1}^{M} d(y_i, y_j). \tag{7.1}$$

A similar criterion is the maximization of the average inter-class distance, i.e. the average distace between an element from C_1 and an element from C_2. In this case, we want to maximize

$$B = \sum_{j=1}^{M} \sum_{i=1}^{N} d(x_i, y_j). \tag{7.2}$$

In neither case does the application of our proposed parametric string distance make sense because, trivially, any sum of string distances will be minimized or maximized by defining $r = 0$ or $r \geq 2$, respectively.

However, if we take a combination of A and B, say

$$E = A/B \tag{7.3}$$

and aim at the minimization of E, then it is no longer trivial for which value of r the minimum is achieved. [1] In a brute force approach, we could calculate E for any possible value of r and then select the value of r that minimizes E. Using the proposed parametric string distance, however, it is sufficient to consider only the critical points, i.e. the borders of the basic intervals of the set of distance functions $d(x_i, x_j), d(y_i, y_j)$ and $d(x_i, y_j)$ according to (7.1) and (7.2).

Lemma 12 *Let D be the set of distance functions according to (7.1) and (7.2). If a minimum of E occurs within a basic interval of D, then this minimum also occurs at at least one of the borders of this interval.*

Proof: Consider any of the basic intervals (r_i, r_{i+1}). At r_i we have

$$E(r_i) = \frac{\sum_j x_j + y_j r_i}{\sum_j \overline{x}_j + \overline{y}_j r_i} = \frac{A}{B}$$

and at r_{i+1}

$$E(r_{i+1}) = \frac{A+C}{B+D},$$

where $A, B, C, D, \geq 0$.
For any r inside the interval, i.e. $r_i < r < r_{i+1}$, we have

$$E(r) = \frac{A + \frac{1}{K}C}{B + \frac{1}{K}D} = \frac{KA + C}{KB + D} \; for\ some\ K > 1.$$

Case 1: $E(r_i) \leq E(r_{i+1})$.
It follows that $AD \leq BC$, or $KAB + AD \leq KAB \leq BC$, or $E(r_i) \leq E(r)$. Furthermore, $E(r) \leq E(r_{i+1})$ as $(BC - AD) \leq K(BC - AD)$, or $KAD + BC \leq KBC + AD$, or $KAB + KAD + BC + CD \leq KAB + KBC + AD + CD$.

Case 2: $E(r_i) > E(r_{i+1})$.
This case is analogous to case 1. We get here $E(r_i) > E(r) > E(r_{i+1})$.

$\square$

A criterion which is in some sense similar to the minimization of E in (7.3) is the maximization of

$$F = \frac{d(x,y)}{S_1 + S_2} \tag{7.4}$$

where

$$S_1 = \sum_{i=1}^{N} d(x, x_i) \; , S_2 = \sum_{i=1}^{M} d(y, y_i) \tag{7.5}$$

and x and y are the class centers of C_1 and C_2, respectively. Intuitively, the maximization of F means the selection of a value of r that results in a large distance between the class

[1] No normalization with respect to the size of C_1 and C_2 has been attempted in this paper. Such a normalization can be easily added if required.

centers x and y while keeping the distances between the elements in the same class small. A suitable choice for the class centers is to select x and y such that

$$\sum_{i=1}^{N} d(x,x_i) = \min\{\sum_{i=1}^{N} d(x_j,x_i) | j = 1,\dots,N\},$$

$$\sum_{i=1}^{M} d(y,y_i) = \min\{\sum_{i=1}^{M} d(y_j,y_i) | j = 1,\dots,M\}. \tag{7.6}$$

Using the same argument as for the minimization of E in (7.3), we can conclude that it is sufficient to evaluate F only at the borders of the basic intervals when searching for the maximum.

More precisely, we compute the set D of all parametric distances $d(x,y), d(x,x_i)$ and $d(y,y_i)$ for one pair $(x,y) \in C_1 \times C_2$ according to (7.4) and (7.5), and construct the basic intervals of D. Then, we minimize $F = F(x,y)$ with respect to r. Finally, the global minimum is obtained by considering all pairs $(x,y) \in C_1 \times C_2$. According to Lemma 9, we need to consider only basic interval borders when searching for the minimum of $F(x,y)$ with respect to r as in (7.4) we have the quotient of two sums of distances.

Another meaningful criterion for adjustment of the replacement cost is the minimization of the error rate when reclassifying C_1 and C_2, using only one class representative x and y for C_1 and C_2, respectively. Thus, we check all possible class representatives $x \in C_1$ and $y \in C_2$ and aim at the minimization of

$$\begin{aligned} G &= |\{x_i | x_i \in C_1 \wedge d(x_i,y) < d(x_i,x)\}| + \\ &\quad |\{y_i | y_i \in C_2 \wedge d(y_i,x) < d(y_i,y)\}| \end{aligned} \tag{7.7}$$

In this case, again, we can avoid an exhaustive search over all possible values of r. Instead, we first calculate the set of all necessary parametric string distances $d(x_i,x_j), d(x_i,y_j)$ and $d(y_i,y_j)$ and construct the corresponding basic intervals. In contrast with the mimization of E in (7.3) and F in (7.4), however, where we evaluated our optimization function at the basic interval borders, we have to select one and only one representative value of r within each basic interval. We may take, for example

$$r = \frac{r_i + r_{i+1}}{2} \tag{7.8}$$

as the representative value for the basic interval (r_i, r_{i+1}).

It is easy to see that one representative value from each interval is sufficient. In evaluating the expression in (7.7), we have to compare pairs of distances and determine which one is larger than the other. As each distance function within one basic interval corresponds to a straight line (i.e. it is a linear function) and no two such functions cross each other, we can select an arbitrary point within one interval for this comparison.

Lemma 13 *Let r_i be the common border of any two basic intervals (r_{i-1}, r_i) and (r_i, r_{i+1}). Then, it will always hold that $G(r) \leq G(r_i)$ for any $r, r_{i-1} < r \leq r_i$, or $G(r) \leq G(r_i)$ for any $r, r_i \leq r < r_{i+1}$.*

Proof.: Consider any two string distance functions $d_1(r_i)$ and $d_2(r_i)$ at r_i. Clearly, either $d_1(r_i) = d_2(r_i)$ or $d_1(r_i) \neq d_2(r_i)$.

If $d_1(r_i) \neq d_2(r_i)$, then $G(r_i)$ will be equal to $G(r)$ for any $r_{i-1} < r < r_i$ and $r_i < r < r_{i+1}$ (see Fig. 6a).

If $d_1(r_i) = d_2(r_i)$ and d_1 is different from d_2 for $r_{i-1} < r < r_i$ and $r_i < r < r_{i+1}$, then $G(r_i) = G(r) - 1$, i.e. $G(r_i) < G(r)$ for either $r_{i-1} < r < r_i$ or $r_i < r < r_{i+1}$, depending on the slope of d_1 and d_2 (see Fig. 6b).

If $d_1(r_i) = d_2(r_i)$ and d_1 is equal to d_2 for either $r_{i-1} < r < r_i$, $r_i < r < r_{i+1}$ or $r_{i-1} < r < r_{i+1}$, then, clearly, $G(r_i) = G(r)$ for $r_{i-1} < r < r_i$, $r_i < r < r_{i+1}$ or $r_{i-1} < r < r_{i+1}$.

$\square$

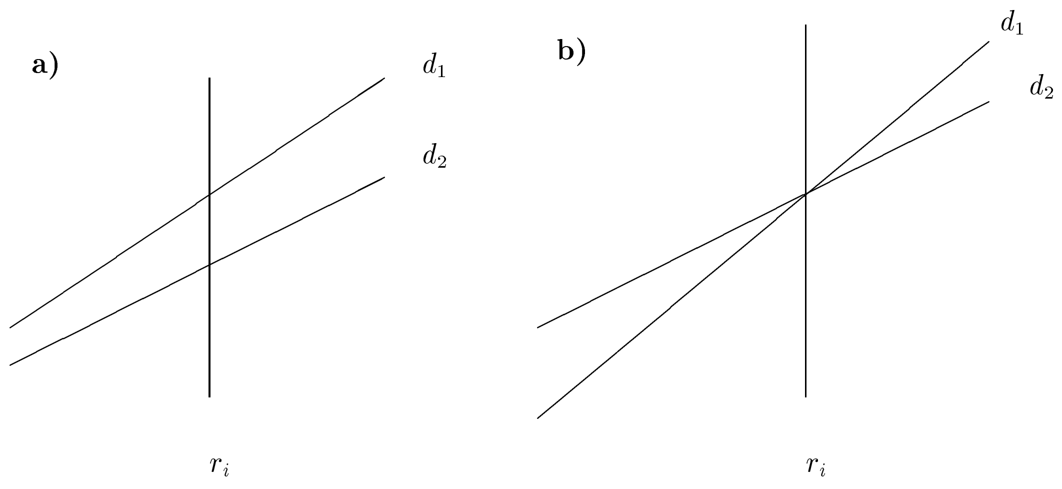

Fig. 6: Graphical illustration for Lemma 13

From this Lemma, we can conclude that any maximum or minimum of G that occurs at a basic interval border also occurs in one of the neighboring intervals, but not conversely, in general. Hence, we can restrict the search for a minimum of G to one representative point from each basic interval.

Next, we consider the problem of adjusting the substitution cost such that the nearest neighbor of as many elements as possible from C_1 is also from C_1, and the nearest neighbor of as many elements as possible from C_2 is also from C_2. Thus, we want to maximize

$$\begin{aligned} H &= |\{x_i | x_i, x \in C_1 \wedge d(x_i, x) = \min\{d(x_i, z) | z \neq x_i \wedge z \in C_1 \cup C_2\}\}| + \\ & \quad |\{y_i | y_i, y \in C_2 \wedge d(y_i, y) = \min\{d(y_i, z) | z \neq y_i \wedge z \in C_1 \cup C_2\}\}| \end{aligned} \tag{7.9}$$

This corresponds to the minimization of the error rate when reclassifying C_1 and C_2 according to the leave-one-out method. For the maximization of H, we can use the same argument as for the minimization of G. That is, it is sufficient to evaluate H at only one representative point from each basic interval of the combined distance function.

We can generalize the minimization of H by using the k-nearest neighbors instead of the nearest neighbor for a suitably chosen k. Furthermore, all considerations of this section can be generalized to K different classes $C_1, C_2, \ldots, C_K$. In all these cases, we can restrict the search to one representative point from each interval or the interval borders, depending on the particular optimization criterion that is used.

Clearly, there are more clustering or classification problems to which the parametric string distance can perhaps be applied. We do not intend to list them completely here. Rather, we have chosen a few prototypical examples to give a general feeling of the applicability of the proposed parametric string distance.

8 Experimental results

We performed a computer implementation of the algorithm for parametric string distance computation and ran some preliminary experiments to investigate the number of basic intervals that are obtained. In these experiments, we randomly generated strings belonging to either pattern class 1 or 2. Then, we calculated all intra- and inter-class parametric distances and computed the basic intervals.

The results of three similar experiments are given in Table 1. There is one row in Table 1 for each experiment. Columns 1 and 2 of Table 1 show the number of strings in class 1 and 2 used in one experiment, respectively. The length of these strings varies between 5 and 14 symbols. The maximum number of different intervals of one parametric distance function, i.e. the maximum value of k according to Corollary 2, is given in column 3. Column 4 shows the number of different inter-class parametric distances computed. (This number is equal to the number in column 1 times the number in column 2.) Similarly, column 5 contains the number of different intra-class parametric distances for both class 1 and 2. Finally, columns 6 and 7 show the number of basic intervals obtained as the result of this experiment. Theoretically, each number in column 6 (or 7) is bounded by the product of the value in column 3 and the squared value in column 4 (or 5). It can be concluded from the table that the actual number of subintervals obtained is much smaller. Only one representative value of r from each interval has to be considered for each of the pattern recognition problems given in section 7. As an example, the maximum number of subintervals that are to be taken into account for the classification problem 1 described in the previous section is given in column 8. (It is the sum of the values in columns 6 and 7.)

1	2	3	4	5	6	7	8
3	3	1	9	6	2	2	4
10	10	3	100	90	25	18	43
20	20	4	400	380	45	25	70

Table 1: Result of computer experiment; see text.

9 Summary and conclusions

In this paper, we have proposed a new parametric string distance measure. It is based on a generalization of the well-known algorithm by Wagner and Fischer [1]. For our new string distance, we assume constant cost for any delete and insert operation. However, the replacement cost is given as a parameter. Hence, for any two given strings A and B our algorithm computes their edit distance in terms of the parameter r.

The new algorithm can be easily implemented. It has a time complexity of $O(n^2 \cdot m)$, where n and m give the lengths of the two strings under consideration and $n = \min(n, m)$. In [16], it has been shown that the space complexity of the algorithm by Wagner and Fischer [1] can be reduced from $O(n \cdot m)$ to $O(n)$ as only one predecessor row is needed when computing the elements of the cost matrix in a row-by-row fashion. The same idea is applicable to our algorithm and yields a reduction of the space complexity from $O(n^2 \cdot m)$ to $O(n \cdot m)$.

There are numerous potential applications of the proposed parametric distance in pattern recognition. Generally, the problem of edit cost adjustment based on a sample set of patterns and a given optimization criterion has not received much attention in the past. Our parametric string distance can be considered as a step towards the solution of this problem.

Given a set of string distances, we can easily determine the basic intervals of the replacement cost r. Now, if we have to solve the task of finding a value of r that optimizes a certain classification or clustering criterion, then we can restrict the search for such an optimum to one representative value of r from each basic interval. (More precisely, depending on the concrete optimization task, we either select the values at the common boundaries of the basic intervals, or choose one value inside each basic interval as a representative.) Theoretically, the number of basic intervals is quadratically bounded by the number of different string distances to be considered. However, it has been demonstrated by means of computer experiments that the actual number of basic intervals is potentially much smaller than the theoretical upper bound. This means a great reduction in computation time compared to the brute force approach where one searches through all possible values of the replacement cost r to find the maximum or minimum of the given optimization criterion.

In this paper, only the replacement cost r has been regarded as a parameter. As a generalization, one could also introduce either the delete or insert cost of a symbol as a parameter. Even more generally, one can consider different costs for an edit operation on different symbols. For example, the cost of replacing the symbol a by b can be defined differently from the cost of replacing b by c, or the insertion of a can have a different cost from the insertion of b. For an alphabet of size N, there are $n = N^2 + N - 1$ parameters in the most general case. Given $n > 1$ parameters, our n-parametric string distance computation algorithm will be principally the same as the algorithm described in this paper. However, it has to deal with cost functions in an n-dimensional space and, rather than determining the intersection of two straight lines, it has to compute the intersection between hyperplanes. The time and space complexity will increase by a factor of n for each additional parameter. Obviously, there is a trade-off between computational effort and versatility as for many applications it could be a great advantage if the cost of more than one type of edit operation could be given as a parameter. However, a more detailed analysis and a study of applications of the n-parametric edit distance of strings is beyond the scope of this paper and is left to future research.

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] Levenshtein, V. I.: Binary codes capable of correcting deletions, insertions, and reversals. Cybernetics and Control Theory, Vol. 10, No. 8, 1966, 707-710.

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

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

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

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

[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] Abe, K., Sugita, N.: Distances between strings of symbols-Review and remarks. Proc. 6th ICPR, Munich, 1982, 172-174.

[9] Tsai, W.H./Yu, S.S.: Attributed string matching with merging for shape recognition, IEEE Trans. PAMI 7, 1985, 453-462.

[10] Gorman, J.W/Mitchell, O.R./Kuhl, F.: Partial shape recognition using dynamic programming, IEEE Trans. PAMI 10, 1988, 257-266.

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

[12] Fu, K.S./Lu, S.-Y.: A clustering procedure for syntactic patterns, IEEE Trans. SMC 7, 1977, 734-742.

[13] Gregor, J.: Aspects of data-driven inference and dynamic programming analysis of pattern structure in strings, PhD Thesis, Laboratory of Image Analysis, Institute of Electronic Systems, University of Aalborg, Denmark, 1991.

[14] 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.

[15] 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.

[16] Hirschberg, D. S.: A linear space algorithm for computing maximal common subsequences, Comm. ACM, Vol. 18, No. 6, 1975, 341-343.