Edit Distance of Regular Languages

Horst Bunke
Department of Computer Science, University of Bern
Neubrückstr. 10, CH-3012 Bern, Switzerland

Phone: +41-31-6314451
Fax: +41-31-6313965
Email: bunke@iam.unibe.ch

Abstract

The edit distance of a pair of strings, and of a string and a language are well known concepts that have various applications in optical character recognition (OCR) and document image analysis (DIA). In the present paper, a generalization is proposed, viz., the edit distance of two regular languages. A method for the computation of the edit distance of two regular languages is introduced and its correctness is shown. Also, applications in the areas of OCR and DIA are discussed.

Keywords: *edit distance, regular language, finite state automaton, graph search, optical character recognition, document image analysis, postprocessing.*

1 Introduction

String edit distance is an important concept that has many applications in areas such as optical character recognition (OCR) and document image analysis (DIA). Particular applications include the postprocessing of OCR results [1, 2], and the comparison of OCR output with ground truth data [3]. The edit distance of two strings is the length of the shortest sequence of edit operations that transform one string into the other. Often a particular cost is assigned to each edit operation. Then the edit distance turns into the minimum cost taken over all sequences of edit operations that transform a given word into another. Various algorithms for string edit distance computation have been described in the literature [4, 5, 6, 7].

Recently, an algorithm has been introduced that computes the edit distance of a string and a regular language [8]. This algorithm has been formulated as the restriction of the error-correcting version of Earley's parsing algorithm [9] to regular languages. Similar algorithms have been previously proposed by other authors [10, 11].

In the present paper, we go a step further and describe a method for the computation of the edit distance of two regular languages. This method includes, as special cases, the edit distance of two strings, and the edit distance of a string and a regular language. The edit distance of two languages L_1 and L_2 is defined as the distance between the pair of words $x \in L_1$ and $y \in L_2$ that are closest to each other. It will be shown that the edit distance computation problem can be transformed into a graph search problem, which can be solved by known techniques.

In [12, p. 272], a method for the computation of the edit distance of a pair of networks was described. A network is a special case of finite state automaton without cycles. This means that a network represents only a finite number of words, while regular languages as considered in this paper may be infinite.

In the present paper, we will also discuss potential applications of the edit distance of languages to OCR and DIA. However, the focus will be on the underlying theory. The main con-

tribution of the present paper is the procedure for edit distance computation and the proof of its correctness.

2 Preliminaries

In this paper, we consider a finite alphabet Σ. A *word* over Σ is a sequence $a_1 \dots a_n$ of symbols where $a_i \in \Sigma$ for $i = 1, \dots, n$. The *empty word* is denoted by ε. By definition, a word $a_1 \dots a_n$ with $n = 0$ is equal to ε. The set of all words over alphabet Σ is denoted by Σ^*.

2.1 String edit distance

An *edit operation* is a pair (a, b) with $a, b \in \Sigma \cup \{\varepsilon\}, ab \neq \varepsilon$. Sometimes, we'll also write $(a \to b)$ instead of (a, b). The edit operation (a, b) is called an *insertion* if $a = \varepsilon$, a *deletion* if $b = \varepsilon$, and a *substitution* if $a \neq \varepsilon \neq b$. An edit operation is a basic step in transforming, i.e. editing, a word into another word. The meaning of the operations $(\varepsilon, b), (a, \varepsilon)$, and (a, b) is to insert b, to delete a, and to substitute a by b, respectively. To model the fact that certain edit operations are more likely than others, a *cost* $c(a \to b)$ is assigned to each edit operation $(a \to b)$. We generally assume $c(a \to b) \geq 0$, and $c(a \to b) = 0$ if $a = b$. An *edit sequence* S is a sequence of edit operations, $S = ((a_1, b_1), \dots, (a_n, b_n)), n \geq 1$. The cost of an edit sequence S, $C(S)$, is definded as $C(S) = \sum_{i=1}^{n} c(a_i, b_i)$. The *edit distance* $d(x, y)$ between two words $x, y \in \Sigma^*$ is defined as the minimum cost taken over all edit sequences that transform x into y. That is, $d(x, y) = \min\{C(S) | S$ is a sequence of edit operations transforming x into $y\}$.

Several algorithms have been proposed for the computation of the edit distance of two words [4, 5, 6, 7]. The considerations in this paper will be based on Wagner & Fischer's algorithm [4]. This algorithm uses a recursive dynamic programming procedure to compute the edit distance between two words $x = a_1 \dots a_n$ and $y = b_1 \dots b_m$. It is based on the following relations

$$d(\varepsilon, \varepsilon) = 0;$$

$$d(\varepsilon, b_1 \dots b_i) = d(\varepsilon, b_1 \dots b_{i-1}) + c(\varepsilon \to b_i) \quad \text{for } i = 1, \dots, m;$$

$$d(a_1 \dots a_i, \varepsilon) = d(a_1 \dots a_{i-1}, \varepsilon) + c(a_i \to \varepsilon) \quad \text{for } i = 1, \dots, n;$$

$$d(a_1 \dots a_i, b_1 \dots b_j) = \min \begin{cases} d(a_1 \dots a_i, b_1 \dots b_{j-1}) + c(\varepsilon \to b_j) \\ d(a_1 \dots a_{i-1}, b_1 \dots b_j) + c(a_i \to \varepsilon) \\ d(a_1 \dots a_{i-1}, b_1 \dots b_{j-1}) + c(a_i \to b_j) \end{cases} \quad \text{for } i = 1, \dots, n; j = 1, \dots, m.$$

For the actual computation of $d(x, y)$, a matrix $D(i, j)$ with rows $i = 0, 1, \dots, n$ and columns $j = 0, 1, \dots, m$ is used where $d(a_1 \dots a_i, b_1 \dots b_j)$ is stored in $D(i, j)$.

In this paper, we assume a slightly different, but equivalent, representation of the matrix $D(i, j)$. Instead of $D(i, j)$, a graph $\mathcal{E}(\S, \dagger)$ is used. We'll call this graph the *edit graph* of x and y. This graph consists of nodes and labeled edges. The nodes are simply the set $\{(i, j) | i = 0, 1, \dots, n; j = 0, 1, \dots, m\}$. We call $(0, 0)$ and (n, m) the *initial* and *final node* in $\mathcal{E}(\S, \dagger)$, respectively. The edges of $\mathcal{E}(\S, \dagger)$ consist of three subsets, namely,

$$\begin{aligned} I &= \{((i, j-1), (i, j)) \mid i = 0, 1, \dots, n; j = 1, \dots, m\}, \\ D &= \{((i-1, j), (i, j)) \mid i = 1, \dots, n; j = 0, 1, \dots, m\}, \text{ and} \\ S &= \{((i-1, j-1), (i, j)) \mid i = 1, \dots, n; j = 1, \dots, m\}. \end{aligned}$$

Here an element $((i, j), (k, l))$ denotes an edge from node (i, j) to (k, l). The edges in I, D, and S correspond to all insertions, deletions, and substituions, respectively, that can be applied in an edit sequence that transforms x into y. In $\mathcal{E}(\S, \dagger)$, any edge $((i, j-1), (i, j))$ in I is labeled with (ε, y_j). Similarly, an edge $((i-1, j), (i, j))$ in D is labeled with (x_i, ε), and an edge $((i-1, j-1), (i, j))$ in S is labeled with (x_i, y_j). Thus any edge label represents an edit operation. A *path* in $\mathcal{E}(\S, \dagger)$ – and also in the alignment graph introduced in the next section – is a sequence of nodes $(n_1, \dots, n_k)$ such that there are edges (n_i, n_{i+1}) for $i = 1, \dots, k-1$. Alternatively, the path $(n_1, \dots, n_k)$ may also be represented by the sequence of its edge labels, i.e. by $(l_1, \dots, l_{k-1})$ where l_i is the label of edge (n_i, n_{i+1}) for $i = 1, \dots, k-1$.

If any edge with label (a, b) in $\mathcal{E}(\S, \dagger)$ gets assigned the cost $c(a \to b)$, then the edit distance $d(x, y)$ is equal to the minimum cost taken over all paths in $\mathcal{E}(\S, \dagger)$ that lead from the initial to the final node, assuming that the cost of a path in $\mathcal{E}(\S, \dagger)$ is given by the sum of the costs of the individual edges on this path. Apparently, the Wagner & Fischer algorithm is a procedure that determines the minimum cost path from the initial to the final node in $\mathcal{E}(\S, \dagger)$ by means of dynamic programming.

2.2 Finite state automata

In this paper, we assume that the regular languages under consideration are given by finite state automata. For the following notation, see [13], for example.

A *finite state automaton* (fsa) is a 5-tuple, $A = (Q, \Sigma, \delta, q_0, F)$ where

- Q is the finite set of *states*,
- Σ is a finite alphabet, called *input alphabet*,
- $\delta : Q \times \Sigma \to Q$ is a partial function, the so-called *transition function*,
- $q_0 \in Q$ is the *initial* state,
- $F \subseteq Q$ is the set of *final* states.

The transition function δ can be extended from single symbols to complete words by defining $\delta(q, \varepsilon) = q$ and $\delta(q, xa) = \delta(\delta(q, x), a)$ for any $x \in \Sigma^*, a \in \Sigma$, and $q \in Q$.

In this paper we'll exclusively consider *deterministic* fsa's, as defined above. This means that for state $q \in Q$ and symbol $a \in \Sigma$ the state $\delta(q, a)$ is uniquely determined. However, we don't require that an fsa is complete. That is, the transition function δ may be undefined for certain states $q \in Q$ and symbols $a \in \Sigma$.

The *language* of an fsa A, L_A, is the set of words that lead from the initial to a final state, i.e.

$$L_A = \{x \mid x \in \Sigma^*, \delta(q_0, x) \in F\}.$$

Furthermore, we define the set of words leading to state q, $L_A(q)$, as

$$L_A(q) = \{x \mid x \in \Sigma^*, \delta(q_0, x) = q\}.$$

It is well known that the class of languages of fsa's is identical to the class of regular languages. Also, any nondeterministic fsa can be transformed into an equivalent deterministic fsa [13].

The aim of this paper is to describe a procedure for the computation of the edit distance of two regular languages. Let L_1 and L_2 be two (finite or infinite) sets of

procedure align(A, B)

input: two fsa's, $A = (P, \Sigma, \delta_A, p_0, F_A)$ and $B = (Q, \Sigma, \delta_B, q_0, F_B)$

output: the alignment graph of A and B, $\mathcal{A}(\mathcal{A}, \mathcal{B})$, consisting of a set of nodes, N, and a set of labeled edges, E.

method:

/* generation of nodes */
$N = P \times Q$;

/* generation of edges */
/* initialization */
$E = \emptyset$;

/* step 1: insertion */
for all $p \in P$ {
  for all $q \in Q$ {
    for all transitions $\delta_B(q, a) = q'$ in B {
      add an edge with label (ε, a) from (p, q) to (p, q') }}};

/* step 2: deletion */
for all $p \in P$ {
  for all $q \in Q$ {
    for all transitions $\delta_A(p, a) = p'$ in A {
      add an edge with label (a, ε) from (p, q) to (p', q) }}};

/* step 3: substitution */
for all $p \in P$ {
  for all $q \in Q$ {
    for all transitions $\delta_A(p, a) = p'$ in A {
      for all transitions $\delta_B(q, b) = q'$ in B {
        add an edge with label (a, b) from (p, q) to (p', q') }}}}
end align

Figure 1: This is the procedure align(A, B) in pseudo code.

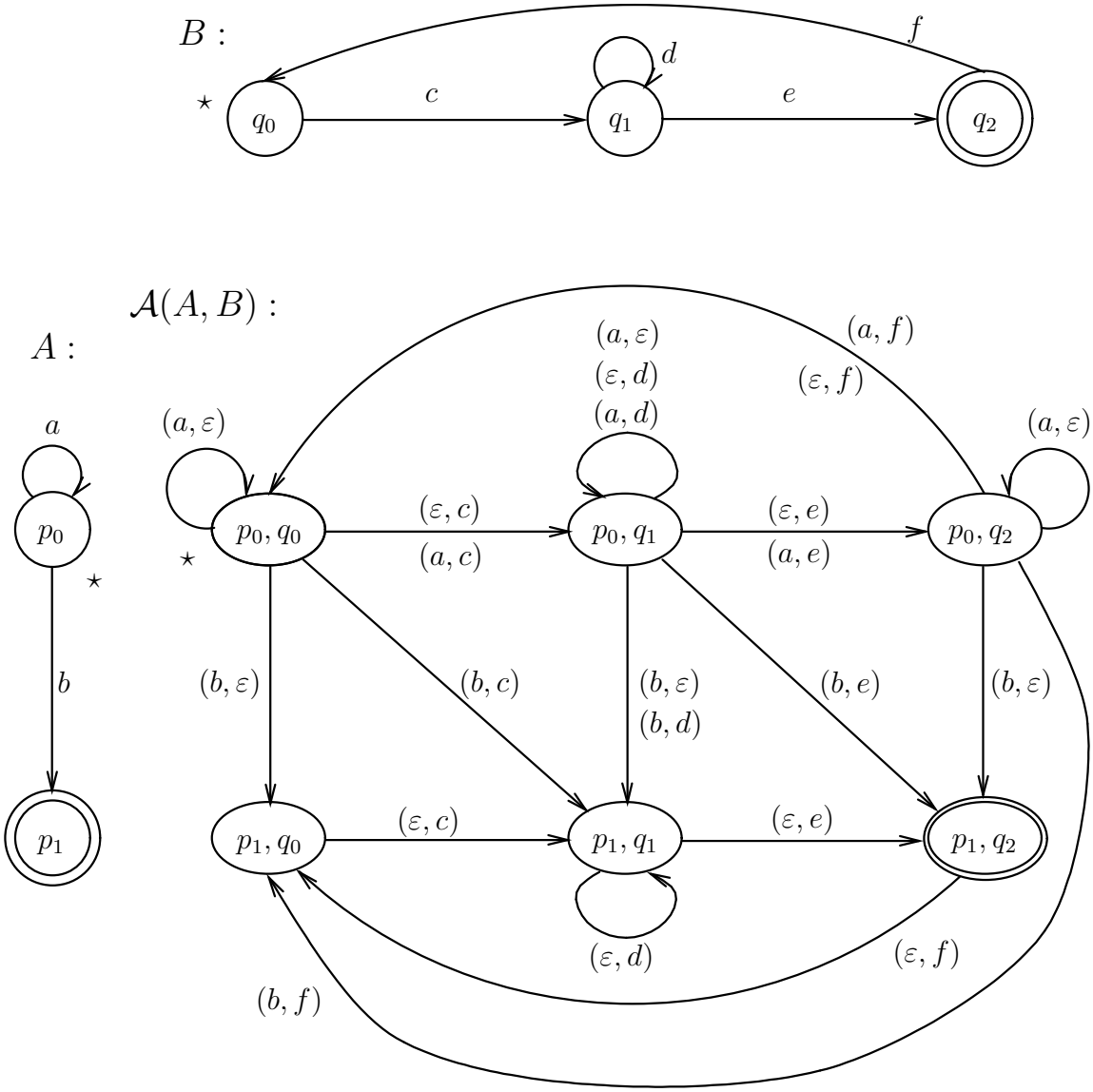

Figure 2: This example shows two fsa's, A and B, and their alignment graph $\mathcal{A}(\mathcal{A}, \mathcal{B})$.

words. Then their edit distance, $d(L_1, L_2)$, is given by

$$d(L_1, L_2) = \min\{d(x, y) \mid x \in L_1, y \in L_2\}.$$

3 Distance computation

Let $A = (P, \Sigma, \delta_A, p_0, F_A)$ and $B = (Q, \Sigma, \delta_B, q_0, F_B)$ be two fsa's. The *alignment graph* of A and B, $\mathcal{A}(\mathcal{A}, \mathcal{B})$, is constructed according to the procedure described in Fig. 2.2. The node (p_0, q_0) in $\mathcal{A}(\mathcal{A}, \mathcal{B})$ will be called *initial*, and all nodes $(p, q) \in F_A \times F_B$ *final*.

As an example, consider the two fsa's A and B, and their alignment graph shown in Fig. 2. (In the graphical representation in Fig. 2, initial states are marked with the $\star$-symbol and final states are indicated by double-circles.)

Generally, for any pair of words $x \in L_A(p), y \in L_B(q)$ for some $(p, q) \in P \times Q$, there will be more than one path $((a_1, b_1), \ldots, (a_n, b_n))$ from (p_0, q_0) to (p, q) in $\mathcal{A}(\mathcal{A}, \mathcal{B})$ such that $a_1 \ldots a_n = x$ and $b_1 \ldots b_n = y$. In Fig. 2, for example, for $x = b$ and y=ce, there are five such paths, namely, $((b, \varepsilon), (\varepsilon, c), (\varepsilon, e))$, $((b, c), (\varepsilon, e))$, $((\varepsilon, c), (b, \varepsilon), (\varepsilon, e))$, $((\varepsilon, c), (b, e))$ and $((\varepsilon, c), (\varepsilon, e), (b, \varepsilon))$. We denote the set of all those paths by $\mathcal{P}(\S, \dagger; \surd, \text{II})$. That is,

$$\begin{aligned}\mathcal{P}(\S, \dagger; \surd, \text{II}) = \{&((\dashv_\infty, \lfloor_\infty), \ldots, (\dashv_\backslash, \lfloor_\backslash))| \\ &a_1 \ldots a_n = x, b_1 \ldots b_n = y, \\ &\delta(p_0, x) = p, \delta(q_0, y) = q\}.\end{aligned}$$

The next lemma now immediately follows from the construction procedure *align(A,B)*.

Lemma 3.1 *Let $\mathcal{A}(\mathcal{A}, \mathcal{B})$ be an alignment graph and $((a_1, b_1), \ldots, (a_n, b_n)) \in \mathcal{P}(\S, \dagger; \surd, \text{II})$. Then*

a) $a_1 \ldots a_n \in L_A(p)$,

b) $b_1 \ldots b_n \in L_B(q)$,

c) $(a_1 \to b_1), \ldots, (a_n \to b_n)$ *is a sequence of edit operations transforming $a_1 \ldots a_n$ into $b_1 \ldots b_n$.*

This Lemma tells us that any path in the alignment graph $\mathcal{P}(\S, \dagger; \surd, \text{II})$ from (p_0, q_0) to some node (p, q) represents a sequence of edit operations that transform a word $x \in L_A(p)$ into a word $y \in L_B(q)$.

Theorem 3.1 *Let A and B be two fsa's, $p \in P, q \in Q, x \in L_A(p)$, and $y \in L_B(q)$. Then any path $((a_1, b_1), \ldots, (a_n, b_n))$ in the edit graph $\mathcal{E}(\S, \dagger)$ from the initial to the final node is contained in $\mathcal{A}(\mathcal{A}, \mathcal{B})$.*

Proof: We proceed by induction on the lengths of x and y.

a) Assume $x = \varepsilon$. For any $y = b_1 \ldots b_n$, there exists only one path in the edit matrix $\mathcal{E}(\S, \dagger)$, namely, $((\varepsilon, b_1), (\varepsilon, b_2), \ldots, (\varepsilon, b_n))$. Clearly, this path is contained in $\mathcal{A}(\mathcal{A}, \mathcal{B})$ as $p = p_0$ and for $i = 1, \ldots, n$ there are states $q_0, q_1, \ldots, q_n$ such that $\delta_B(q_{i-1}, b_i) = q_i$. Consequently, the edges $(\varepsilon, b_1), (\varepsilon, b_2), \ldots, (\varepsilon, b_n)$ are generated by step 1 of algorithm *align(A,B)*.

A similar argument holds true for $y = \varepsilon$. Notice that there is no edge in $\mathcal{E}(\S, \dagger)$ if $x = y = \varepsilon$. Thus, the proposition also holds for this case.

b) Now let's assume that the proposition holds for the following pairs of words for some $i \geq 0, j \geq 0$:
$(x, y') = (a_1 \ldots a_i, b_1 \ldots b_{j-1})$,
$(x', y) = (a_1 \ldots a_{i-1}, b_1 \ldots b_j)$,
$(x', y') = (a_1 \ldots a_{i-1}, b_1 \ldots b_{j-1})$.
According to Wagner & Fischer's algorithm, the set of paths in $\mathcal{E}(\S, \dagger)$ from the initial to the final node is the union of three subsets, namely,

(I) S_1, the set of paths from the initial to the final node in $\mathcal{E}(\S, \dagger')$, extended by the edge (ε, b_j)

(II) S_2, the set of paths from the initial to the final node in $\mathcal{E}(\S', \dagger)$, extended by the edge (a_i, ε)

(III) S_3, the set of paths from the initial to the final node in $\mathcal{E}(\S', \dagger')$, extended by the edge (a_i, b_j)

We notice that the edges (ε, b_j), (a_i, ε) and (a_i, b_j) in (I), (II), and (III) are inserted by steps 1,2, and 3 of *align(A,B)*. Therefore, the proposition holds true for the pair (x, y). This concludes the proof.

□

Let $\mathcal{A}(\mathcal{A},\mathcal{B})$ be an alignment graph and $((a_1,b_1),\ldots,(a_n,b_n)) \in \mathcal{P}(\S,\dagger;\surd,\Pi)$ for some $p \in P, q \in Q; n \geq 1$. The cost of the path $((a_1,b_1),\ldots,(a_n,b_n))$ is given by

$$C[((a_1,b_1),\ldots,(a_n,b_n))] = \sum_{i=1}^{n} c(a_i \rightarrow b_i)$$

Let $\Pi^*(x,y;p,q)$ denote the path in $\mathcal{P}(\S,\dagger;\surd,\Pi)$ that has the minimum cost.[1] That is,

$$C[\Pi^*(x,y;p,q)] = \min\{C[\Pi]|\Pi \in \mathcal{P}(\S,\dagger;\surd,\Pi)\}. \quad (1)$$

Lemma 3.2 *Let $p \in P, q \in Q, x \in L_A(p), y \in L_B(q)$. Then $d(x,y) = C[\Pi^*(x,y;p,q)]$.*

Proof: By definition, $d(x,y)$ is equal to the minimum cost taken over all sequences of edit operations that transform x into y. From Wagner & Fischer's algorithm we know that the sequence of edit operations with minimum cost corresponds to the minimum cost path in $\mathcal{E}(\S,\dagger)$. By Theorem 3.1 this path is contained in $\mathcal{A}(\mathcal{A},\mathcal{B})$. Moreover, its cost must be equal to $\min\{C[\Pi]|\Pi \in \mathcal{P}(\S,\dagger;\surd,\Pi)\} = \mathcal{C}[\diamond^*(\S,\dagger;\surd,\Pi)]$ (see (1)). Otherwise, because of Lemma 3.1, if there was another path in $\mathcal{P}(\S,\dagger;\surd,\Pi)$ with a smaller cost, then this other path would correspond to the minimum cost sequence of edit operations transforming x into y. Thus, $d(x,y)$ would not correspond to the edit distance of x and y.

□

Generally, for each pair of states $p \in P$ and $q \in Q$, there exists a set of pairs of words $x \in L_A(p)$ and $y \in L_B(q)$. For each such pair of words, there exists a set of paths in $\mathcal{P}(\S,\dagger;\surd,\Pi)$. Each path in each of these sets has a different cost, in general. Let $\Pi^*(p,q)$ denote the path with minimum cost among all these paths. That is,

$$C[\Pi^*(p,q)] = \min\{C[\Pi^*(x,y;p,q)]| \; x \in L_A(p), y \in L_B(q)\}. \quad (2)$$

[1] For a given alignment graph $\mathcal{A}(\mathcal{A},\mathcal{B})$ and $\mathcal{P}(\S,\dagger;\surd,\Pi)$ the minimum cost path $\Pi^*(x,y;p,q)$ is not necessarily unique.

Lemma 3.3 *Let $p \in P, q \in Q$. Then $d(L_A(p),L_B(q)) = C[\Pi^*(p,q)]$.*

Proof: By definition, $d(L_A(p),L_B(q)) = \min\{d(x,y)|x \in L_A(p), y \in L_B(q)\}$. By Lemma 3.2, the right-hand side of this equation is equal to $\min\{C[\Pi^*(x,y;p,q)]|x \in L_A(p), y \in L_B(q)\}$ which is equal to $C[\Pi^*(p,q)]$ by (2).

□

Theorem 3.2 *Let A and B be two fsa's. Then*

$$d(L_A,L_B) = \min\{C[\Pi^*(p,q)]| \; p \in F_A, q \in F_B\}.$$

Proof: From Lemma 3.3 we know that $\min\{C[\Pi^*(p,q)]|p \in F_A, q \in F_B\} = \min\{d(L_A(p),L_B(q))|p \in F_A, q \in F_B\}$. By definition, the right-hand side if this equation is equal to $\min\{d(x,y)|p \in F_A, q \in F_B, x \in L_A(p), y \in L_B(q)\}$, which is equal to $d(L_A,L_B)$.

□

Theorem 3.2 suggests to compute the edit distance $d(L_A,L_B)$ of two regular languages L_A and L_B in two steps. First, we construct the alignment graph $\mathcal{A}(\mathcal{A},\mathcal{B})$, and then we search a minimum cost path from (p_0,q_0) to a terminal node $(p,q) \in F_A \times F_B$ in $\mathcal{A}(\mathcal{A},\mathcal{B})$. Computing the minimum cost path can be done by the standard graph search algorithm as described, for example, in Chapter 4 of [14].

If we assume costs equal to one for any edit operation, i.e. $c(u \rightarrow u) = 0, c(u \rightarrow v) = 1$ for $u,v \in \Sigma \cup \{\varepsilon\}; u \neq v; uv \neq \varepsilon, \Sigma = \{a,b,c,d,e,f\}$, then the cost of the minimum path from the initial to the final node in Fig. 2 is equal to 2. In this example, we have $d(L_A,L_B) = 2 = d(b,ce) = d(ab,ce)$. It can be concluded from $\mathcal{A}(\mathcal{A},\mathcal{B})$ that there are two minimum cost edit sequences for $x = b$ and $y = ce$, namely $((b,c),(\varepsilon,e))$, and $((\varepsilon,c),(b,e))$. For $x = ab$ and $y = ce$ there is one such edit sequence, namely $((a,c),(b,e))$. For any minimum cost path in $\mathcal{A}(\mathcal{A},\mathcal{B})$ the words x and y with $d(x,y) = d(L_A,L_B)$ as well as their corresponding edit sequence can be easily reconstructed. This property holds true for any pair of fsa's and their corresponding alignment graph.

Theorem 3.3 *Let* $A = (P, \Sigma, \delta_A, p_0, F_A)$ *and* $B = (Q, \Sigma, \delta_B, q_0, F_B)$ *be two fsa's and* $N_A = |P|, N_B = |Q|$. *Then the edit distance* $d(L_A, L_B)$ *can be computed in time* $O(N_A N_B \log(N_A N_B))$.

Proof: As we are dealing with deterministic fsa's, the number of transitions in A and B is $O(N_A)$ and $O(N_B)$, respectively. Thus the alignment graph consists of $N_A N_B$ nodes and $O(N_A N_B)$ edges. Obviously, its construction takes $O(N_A N_B)$ time. From Chapter 4 of [14] we know that finding the minimum cost path from the initial to some final node in a graph with n nodes and e edges can be done in $O(e \log n)$ steps. Substituting $n = N_A N_B$ and $e = O(N_A N_B)$ concludes the proof.

□

In many applications, the edit costs $c(a \rightarrow b)$ are drawn from some restricted domain, for example, the integers between 0 and M for some M. In this case, the graph search algorithm can be implemented such that it runs in time $O(N_A N_B \log M)$. This is an interesting alternative to the $O(N_A N_B \log(N_A N_B))$ version of Theorem 3.3. For details of the algorithm, see Chapter 4 of [14].

4 Applications

In this section we discuss some potential applications of the proposed edit distance to OCR and DIA.

The first application is postprocessing in OCR. One of the main problems in both machine and hand printed character recognition is segmentation. Any pure bottom-up segmentation technique is highly prone to errors. Therefore, it is common to consider several concurrent segmentation hypotheses. Moreover, multiple character hypotheses are usually provided for each possible character location. The structure of all these hypotheses can be favorably represented by means of a graph. An example from the area of numeral string recognition is shown in Fig. **??** (see [15]). Given such a graph, each path from the initial to the final node represents one possible word hypothesis (in the present case, a sequence of digits). In order to overcome both segmentation and recognition errors, the set of all word hypotheses, i.e. the set of all paths in the graph, is matched against a dictionary of legal words. Notice that a graph like the one in Fig. **??** is nothing but a fsa. The language generated by this fsa is the set of all word hypotheses. Thus matching the set of word hypotheses against a dictionary is equivalent to matching the language of the fsa against the dictionary.

Notice that one of the standard data structures for dictionaries is a trie. Yet a trie is nothing but a special type of fsa. The language accepted by this fsa is the set of all dictionary words represented by the trie. Thus finding the word among a set of hypotheses that is most similar to a word in a dictionary can be solved by determining the edit distance of two regular languages, each of which is represented by a fsa. Moreover, if the text to be recognized consists of a number of consecutive words then the fsa representing the dictionary can be generalized such that it accepts any sequence of dictionary words. Such a generalization can be easily accomplished by adding transitions from the final states back to the inital state.

In the interpretation of engineering drawings, the set of legal denotations of graphical symbols is often a regular language. For example, the set of denotations of a resistor in a particular class of circuit diagrams may be a string starting with character "R", followed by an integer number consisting of up to three digits, followed by a slash, followed by a single digit. An instance is "R123/0". The set of such denotations can be directly represented by a fsa, and OCR postprocessing can be done by matching all hypotheses of the denotation string, represented in a similar way as in Fig. **??**, against the corresponding fsa. Notice, moreover, that many concepts, which occur in various forms, such as dates, intervals of numbers, or priority digits can be represented by fsa's [8]. Thus OCR postprocessing can be done by means of edit distance computation of regular languages.

There are a number of applications in DIA where a grammar has been used to model the structure of documents [16, 17, 18]. For those applications, regular grammars are often sufficient. They can be represented by fsa's. Rather than using a parser in order to match the items extracted from an input document with the fsa, we propose to apply our algorithm for computing the edit distance of regular languages. One of the two languages to be matched is given

by the fsa representing the document model, while the other can be a structure similar to Fig. **??** that takes into regard various hypotheses arising from different alternatives in segmentation and/or labeling. In the most simple case, if there are no ambiguities, this structure is just a simple string of items extracted from the document. In general, such a schema can be regarded an error correcting parser that can simultaneously handle multiple concurrent inputs. It can be concluded that such an approach is more flexible and better theoretically founded than the methods currently used.

If the structure of a document is represented by a grammar, then the edit distance of languages may be a useful concept for off-line analysis during system design. Assume that the given task is to classify a document into several categories $C_1, C_2, C_3, \ldots$ each of which is modeled by a fsa. Then we can determine, in an off-line analysis by means of the method proposed in this paper, all distances $d(C_i, C_j), i \neq j$. If we find out that, for example, $d(C_i, C_j) < d(C_i, C_k)$ then we have gained some a priori knowledge that class C_i is perhaps more likely to be misclassified as C_j than as C_k. Such an off-line analysis is also very useful if the structure of new documents is to be defined. A concrete application of such an off-line analysis is check classification as described in [8]. Here the similarity of the coding lines of different types of checks can be analyzed using the edit distance of regular languages. By means of such an analysis, "critical" classes that have a small edit distance to other classes can be identified. Also, whenever new coding line formats are to be introduced, they can be defined such that the edit distance to existing coding lines is large.

In some of the applications sketched above, the language generated by the fsa under consideration is finite, due to the fact that there are neither loops nor cycles in the fsa's. Examples are trie structures representing dictionaries and word hypotheses graphs such as the one shown in Fig. **??**. If the underlying fsa's are loop- and cycle-free, there will be no loops nor cycles in the alignment graph. In this case, a particular efficient version of the graph search algorithm can be used; see Chapter 4 of [14]. It has a time complexity of only $O(N_A N_B)$, where N_A and N_B denote the number of states of the two underlying automata. For the extreme case where both automata represent just a single word, our algorithm thus runs in time no more than that of Wagner & Fischer [4]. This means that we don't loose any efficiency through the increase in flexibility.

5 Discussion and conclusions

In this paper, a method for computing the edit distance of regular languages was introduced and its correctness was proven. The method is flexible and includes, as special cases, the computation of the edit distance of a pair of strings, and of a string and a regular language. In the general case, the computational complexity of the method is $O(N_A N_B \log(N_A N_B))$ where N_A and N_B denote the number of states of the finite state automata that define the underlying languages. The price for the increase in generality — i.e. matching a pair of languages rather than strings — is just a (moderate) additional factor, $\log(N_A N_B)$, compared to the standard algorithm [4].

For practical applications, a number of optimizations can be included in the method. For example, it is sufficient in the construction of the alignment graph to keep just the edge with minimum cost between a pair of nodes. Thus, there will be at most one edge between two nodes in the alignment graph. Moreover, if it is sufficient to determine $d(L_1, L_2)$ and no more than one representative pair $x \in L_1, y \in L_2$ such that $d(x, y) = d(L_1, L_2)$ then we can safely eliminate all loops in the alignment graph. In applications including several languages $L_1, L_2, L_3, \ldots$ we are perhaps interested in the distance $d(L_i, L_j)$ only if it is below a certain threshold T. In such a case, we can terminate the search through the alignment graph as soon as T is exceeded.

It is an open question whether there exists a direct generalization of a dynamic programming procedure like the one in [4] – presumably with time complexity $O(N_A N_B)$ – to find $d(L_1, L_2)$. For automata without loops and cycles, such a generalization is straightforward (see the discussion in Section 4). But in the presence of cycles, it is clear that the minimum cost path can't be found in a single dynamic programming sweep over the alignment graph.

There are several applications of edit dis-

tance of regular languages in the areas of OCR and DIA. Moreover, the proposed method can be expected a useful tool in an experimental software environment for OCR and DIA research, which can be used for various string matching problems.

Concrete applications of the proposed method are currently under investigation.

Acknowledgment: The main body of this paper was completed when the author was visiting the Faculty of Education at Kagawa University, Japan. The author wants to thank T. Yamasaki for his support. Also, many thanks go to J. Csirik, T. Nartker and S. Rice for enlightening and exciting discussions on string matching.

References

[1] Srihari, S.N. (ed.): Computer Text Recognition and Error Correcting, Tutorial, IEEE Comp. Soc. Press, Silver Springs, MD, 1985

[2] Kukich, K.: Techniques for automatically correcting words in text, ACM Comp. Surv., Vol. 24, No. 4, 1992, 377-439

[3] Rice, S.V., Kanai, J., Nartker, T.A.: A difference algorithm for OCR-generated text, in Bunke, H. (ed.): Advances in Structural and Syntactic Pattern Recognition, World Scientific Publ. Co., 1993, 333-341

[4] Wagner, R.A., Fischer, M.F.: String-to-string correcting problem, JACM, Vol. 21, No. 1, 1974, 168-173

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

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

[7] Bunke, H.: A fast algorithm for finding the nearest neighbor of a word in a dictionary, Proc. 2nd ICDAR, Tsukuba City, 1992, 632-637

[8] Bunke, H., Liviero, R.: Classification and postprocessing of documents using an error-correcting parser, Proc. 3rd ICDAR, Montreal, 1995, 222-226

[9] Aho, A.V.,Peterson, T.G.: A minimum distance error-correcting parser for context-free languages, SIAM J. Computing 1, 1972, 305-312

[10] Wagner, A.: Order-n correction for regular languages, CACM, Vol. 17, No. 5, 1974, 265-268

[11] Myers, E.W., Miller, W.: Approximate matching of regular expressions, Bulletin of Math. Biology, Vol. 51, No. 1, 1989, 5-37

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

[13] Hopcroft, J., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation, Addison Wesley, 1979

[14] Mehlhorn, K.: Graph Algorithms and NP-Completeness, Springer Verlag, 1984

[15] Ha, T.M., Niggeler, D., Bunke, H.: A system for segmenting and recognising totally unconstrained handwritten numeral strings, Proc. 3rd ICDAR, Montreal, 1995, 1003-1009

[16] Viswanathan, M.: Analysis of scanned documents – a syntactic approach, in Baird, H., Bunke, H., Yamamoto, K. (eds.): Structured Document Image Analysis, Springer, 1992, 115-136

[17] Conway, A.: Page grammars and page parsing – a syntactic approach to document layant recognition, Proc. 2nd ICDAR, Tsukuba City, 1993, 761-764

[18] Azokly, A., Ingold, R.: A language for document generic layout description and its use for segmentation into regions, Proc. 3rd ICDAR, Montreal, 1995, 1123-1126