Computing the edit distance of a regular language ⋆

Stavros Konstantinidis

Department of Mathematics and Computing Science, Saint Mary's University, Halifax, Nova Scotia, B3H 3C3 Canada

Abstract

The edit distance (or Levenshtein distance) between two words is the smallest number of substitutions, insertions, and deletions of symbols that can be used to transform one of the words into the other. In this paper we consider the problem of computing the edit distance of a regular language (the set of words accepted by a given finite automaton). This quantity is the smallest edit distance between any pair of distinct words of the language. We show that the problem is of polynomial time complexity. In particular, for a given finite automaton A with n transitions, over an alphabet of r symbols, our algorithm operates in time $O(n^2r^2q^2(q+r))$, where q is either the diameter of A (if A is deterministic), or the square of the number of states in A (if A is nondeterministic). Incidentally, we also obtain an upper bound on the edit distance of a regular language in terms of the automaton accepting the language.

Key words: algorithm, finite automaton, constraint system, edit distance, Levenshtein distance, regular language.

⋆ Research supported by grant R220259 of the Natural Sciences and Engineering Research Council of Canada. A short version of this paper appeared in the electronic proceedings of the 2005 IEEE Information Theory Workshop on Coding and Complexity [5].

Email address: `s.konstantinidis@smu.ca` (Stavros Konstantinidis).

1 Introduction

The problem of measuring the distance, or generally the difference, between words and languages (sets of words) is important in various applications of information processing such as error control in data communications, bioinformatics, speech recognition and spelling correction. The languages of interest are usually regular languages (also called constraint systems), that is, languages defined by finite automata (edge-labeled finite state graphs), but they could be non regular as well. Well-known measures of the difference between two words are the edit (or Levenshtein) distance and the Hamming distance. The edit distance between two words is the smallest number of substitutions, insertions, and deletions of letters required to transform one of the words to the other. Typical problems pertaining to difference measures for words and languages are the following.

(1) *The word difference problem:* Compute the edit distance between two given words. The problem can be solved using a dynamic programming algorithm – see [14], for instance.
(2) *The error-correction problem:* Given a language of valid (or correct) words, correct a given word to some word of the language that is the least different to the given word. This problem presupposes an agreed measure of word difference such as the edit distance. For (arbitrary) regular languages and for the measure of edit distance the problem was first addressed in [16]. In [10] it is addressed for cases where the language in question is not even regular. In [8] and [3] the problem is considered for regular languages and for general word difference measures defined by weighted automata – in fact in these papers the authors show how to compute a pair of words from two given regular languages such that these words are the least different. In coding theory of course, the problem has been addressed extensively for various instances of the word difference measure – this measure is usually implicitly specified by the communications channel that is capable of changing words [12]. We also note that a related problem between a word and a hypertext has been considered in [1].
(3) *The error-detection capability problem:* Compute the distance (also referred to as self-distance or inner distance) of a given language. This quantity is simply the minimum distance between any pair of distinct words in the language. When the language in question is viewed as a code, the value of the distance represents the maximum number of errors that the code can detect. For example, it is well known that a block code (set of equal length words) can detect up to m bit substitution errors if and only if the Hamming distance of the code is greater than m. A similar observation exists for the case of the edit distance [6]. In [4], the authors show how to compute the Hamming distance of a given regular

language. We also note that in some sense this and the previous problem are related as follows when it comes to the cases of the Hamming or Levenshtein distance. A block code can detect m errors if and only if it can correct $\lfloor m/2 \rfloor$ errors. Hence, in these cases, the problem here can be called the error-correction capability problem.

In this work, we address the third problem for the case of the edit distance of a regular language. In this case, the value of the edit distance represents the maximum number m, say, of substitution, insertion and deletion errors in the words of the language that are guaranteed to be *detected*. This means that no word of the language can be transformed to another different word of the language if up to m errors are used. We show that the problem is of polynomial time complexity. In particular, for a given finite automaton A with n transitions, over an alphabet of r symbols, our algorithm operates in time $O(n^2 r^2 q^2 (q + r))$, where q is either the diameter of A (if A is deterministic), or the square of the number of states in A (if A is nondeterministic).

The paper is organized as follows. In the next section we provide the basic notation about words, automata, and edit strings. An edit string is a special word whose symbols are called edit operations. It has been used to define formally the edit distance between ordinary words. Here we also consider languages and finite automata of edit strings as tools for reasoning about the distance of a language. In Section 3, we obtain a few lemmata that will be used to prove the correctness of the main result of the paper. We believe that these lemmata might be of interest in their own right. For example, we show an infinitely-often tight upper bound on the edit distance of a regular language, which depends on the automaton accepting the language. Section 4, contains the main result of the paper, that is, a polynomial time algorithm to compute the edit distance of a given regular language. The time complexity of the algorithm is higher when the language is given via a nondeterministic automaton. Section 5 contains a few concluding remarks and proposes possible directions for future research.

2 Basic background and notation

2.1 Cardinality, alphabet, word, language

For any set S, we use the expression $|S|$ to denote the *cardinality* of the set S. An *alphabet* is a finite nonempty set Δ whose elements are called *symbols*. A *word* or *string* (over Δ) is a finite sequence $a_1 \cdots a_n$ such that each a_i is in Δ. The *length* of a word w is denoted by $|w|$. The *empty word* is the word of length zero. For any two words w_1, w_2 the expression $w_1 w_2$ denotes the word that results by concatenating w_1 and w_2. A *language* is a set of words. The language of all words is denoted by Δ^*.

Notational convention.
A word w of length n can be viewed as a mapping from the index set $\{1, \ldots, n\}$ into Δ such that $w(i)$ is *the symbol of w at position i*. If $i > |w|$, we agree to write that $w(i) = \perp$, where $\perp$ is a symbol not in the alphabet Δ. With this convention, it follows that any two words u and v are different if and only if $u(i) \neq v(i)$ for some positive integer i.

2.2 Finite automaton, computation, diameter, regular language

A (nondeterministic) *finite automaton* is a quintuple $A = (\Delta, S, s_0, F, T)$ such that Δ is the alphabet, S is the (finite and nonempty) set of states, $s_0 \in S$ is the start state, $F \subseteq S$ is the set of final states, and T is the set of transitions, which we denote with expressions of the form paq such that p, q are states and a is a symbol in Δ. The automaton A is said to be deterministic if, for any two transitions of the form paq_1, paq_2, it is the case that $q_1 = q_2$. A *computation* of A is denoted by an expression of the form $p_0a_1p_1 \cdots a_np_n$, for some $n \geq 0$, where each $p_{i-1}a_ip_i$ is a transition of A. When A is viewed as a directed edge-labeled graph, a computation is simply a labeled path of A. The *diameter of A*, denoted by $\mathrm{diam}(A)$, is the largest number of states in a computation $p_0a_1p_1 \cdots a_np_n$ for which p_0 is the start state and no state occurs more than once, that is, $i \neq j$ implies $p_i \neq p_j$. Obviously the following inequalities about $\mathrm{diam}(A)$ hold:

$$1 \leq \mathrm{diam}(A) \leq |S|.$$

If, in the computation $p_0a_1p_1 \cdots a_np_n$, the state p_0 is the start state and p_n is a final state then the word $a_1 \cdots a_n$ is accepted by A. The *language accepted* by the automaton A is the set of all words accepted by A, and is denoted by $L(A)$. The *size* of A, denoted with $|A|$, is the quantity $|S| + |T|$, which is the number of states plus the number of transitions. A language L is called *regular*, or a *constraint system*, if there is a finite automaton accepting L [7,17]. If A and A' are two finite automata then the expression

$$A \cap A'$$

denotes the finite automaton that results using the standard cross product construction such that $L(A \cap A') = L(A) \cap L(A')$ – see [17]. Moreover, $|A \cap A'| = O(|A||A'|)$.

2.3 Ordinary word, edit string, weight, edit distance

In this paper we shall use a fixed, but arbitrary, alphabet Σ of *ordinary* symbols, and the alphabet E of the *(basic) edit operations* that depends on Σ. The *empty word* over Σ is denoted by λ, that is, $\lambda w = w\lambda = w$, for all words w. The alphabet E consists of all symbols (x/y) such that $x, y \in \Sigma \cup \{\lambda\}$ and at least one of x and y is in Σ. If (x/y) is in E and x is not equal to y then we call (x/y) an *error* [2]. We write (λ/λ) for the empty word over the alphabet E. We note that λ is used as a formal symbol in the elements of E. For example, if a and b are in Σ then $(a/\lambda)(a/b) \neq (a/a)(\lambda/b)$. The elements of E^* are called *edit strings* [2], or alignments [8]. The *input* and *output* parts of an edit string $h = (x_1/y_1)\cdots(x_n/y_n)$ are the words (over Σ) $x_1 \cdots x_n$ and $y_1 \cdots y_n$, respectively. We write $\mathrm{inp}(h)$ for the input part and $\mathrm{out}(h)$ for the output part of h. The expression $\mathrm{weight}(h)$ denotes the number of errors in h.

The *edit (or Levenshtein) distance* between two words u and v, denoted by $\mathrm{dist}(u, v)$, is the smallest number of errors (substitutions, insertions and deletions of symbols) that can transform u to v. More formally,

$$\mathrm{dist}(u, v) = \min\{\mathrm{weight}(h) \mid h \in E^*,\ \mathrm{inp}(h) = u,\ \mathrm{out}(h) = v\}.$$

For example, for $\Sigma = \{a, b\}$, we have that $\mathrm{dist}(ababa, babbb) = 3$ and the edit string

$$h = (a/\lambda)(b/b)(a/a)(b/b)(a/b)(\lambda/b)$$

is a minimum weight edit string that transforms *ababa* to *babbb*. In words, h says that we can use the deletion (a/λ), the substitution (a/b), and the insertion (λ/b) to transform *ababa* to *babbb*. We note the following fact from [6], for all words x, u, v, y:

$$\mathrm{dist}(xuy, xvy) = \mathrm{dist}(u, v).$$

If L is a language containing at least two words then the edit distance of L is

$$\mathrm{dist}(L) = \min\{\mathrm{dist}(u, v) \mid u, v \in L \text{ and } u \neq v\}.$$

Definition 1 *Let L be any set of at least two words. We say that an edit string h* realizes the edit distance of L *if* $\mathrm{weight}(h) = \mathrm{dist}(L)$ *and* $\mathrm{inp}(h) \neq \mathrm{out}(h)$ *and* $\mathrm{inp}(h), \mathrm{out}(h) \in L$.

3 Intermediate lemmata

In this section we obtain a few lemmata that will be used to prove the correctness of the main result of the paper. We believe that these lemmata might be of interest in their own right. Moreover the following two paragraphs provide useful technical tools.

Argument used in proofs.
In the proofs of the various statements we shall use the following argument about an automaton A with s states. If $P = p_0 a_1 p_1 \cdots a_k p_k$ is a computation of A and the number of states $k+1$ appearing in P exceeds s, or $\mathrm{diam}(A)$ when p_0 is the start state, then at least two states p_i and p_j, with $i < j$, in P must be equal. Then we can define a shorter computation P' of A that also starts with p_0 and ends with p_k, by removing the part $p_i a_i \cdots p_{j-1} a_j$ of P. Moreover, for the words $w_1 = a_1 \cdots a_{i-1}$, $w = a_i \cdots a_j$ and $w_2 = a_{j+1} \cdots a_k$ we have that w is nonempty and both of the words $w_1 w w_2$ and $w_1 w_2$ are formed in computations of A. Of course, this argument is well-known, but we include it here for completeness.

The automaton $A \cap_E A$.
Given two automata A_1, A_2 and a subset D of E, the finite automaton $A_1 \cap_D A_2$ [3] accepts all edit strings h that transform a word of $L(A_1)$ into a word of $L(A_2)$ using only the edit operations in D. Moreover, we have that $|A_1 \cap_D A_2| = O(|A_1||A_2|)$. The automaton $A_1 \cap_D A_2$ is the trim part of the automaton C constructed as follows – the trim part of C results by removing all states of C that are not reachable from the start state and cannot reach a final state. The states of C are the pairs (p, q), where p, q are states of A_1, A_2, respectively. The start state of C is the pair consisting of the start states of A_1 and A_2, and the final states are all pairs (p, q) such that p, q are final states of A_1, A_2, respectively. The transitions of C are of the form $(p_1, q_1)x/y(p_2, q_2)$ such that (i) x/y is in D; (ii) either $p_1 x p_2$ is a transition of A_1, or $p_1 = p_2$ and $x = \lambda$; and (iii) either $q_1 y q_2$ is a transition of A_2 or $q_1 = q_2$ and $y = \lambda$. We note that a graph (called regular expression edit graph) similar to $A_1 \cap_E A_2$ is defined in [9] between a word and a finite automaton.

In our considerations, we shall use the finite automaton $A \cap_E A$ such that

$$L(A \cap_E A) = \{h \in E^* \mid \mathrm{inp}(h), \mathrm{out}(h) \in L(A)\}.$$

Obviously, if A has s states then $A \cap_E A$ has at most s^2 states.

Lemma 2 *Let A be a deterministic finite automaton accepting at least two words. There are distinct words u, v in $L(A)$ and an index $j \le \mathrm{diam}(A)$ such that $u(j) \ne v(j)$ and* $\mathrm{dist}(L(A)) = \mathrm{dist}(u, v)$.

Proof. Suppose that $\mathrm{dist}(L(A)) = \mathrm{dist}(u, v)$ for some distinct words u, v in $L(A)$. We choose the pair (u, v) to be minimal as follows: if (u_1, v_1) is another such pair then $|u| + |v| \leq |u_1| + |v_1|$. Let j be the smallest index for which $u(j) \neq v(j)$ and assume that $j > \mathrm{diam}(A)$. We show that this assumption leads to a contradiction. First note that $j = |z| + 1$, where the word z is the common prefix of length $j - 1$ of u and v. Let u' and v' be the words defined by $u = zu'$ and $v = zv'$. Then $u' \neq v'$ and $\mathrm{dist}(u, v) = \mathrm{dist}(u', v')$. As $|z| \geq \mathrm{diam}(A)$, the path of A on which the word z is formed, which starts from the start state, contains more than $\mathrm{diam}(A)$ states. This implies that the path contains a repeated state and, therefore, there are words w_1, w, w_2 such that $|w| > 0$ and $z = w_1ww_2$ and the words w_1w_2u' and w_1w_2v' are distinct and in $L(A)$. Moreover, $\mathrm{dist}(w_1w_2u', w_1w_2v') = \mathrm{dist}(u', v') = \mathrm{dist}(L(A))$. This, however, contradicts the choice of (u, v) being minimal. Hence, $j \leq \mathrm{diam}(A)$. □

Unfortunately, the above proof cannot be used if the automaton A is nondeterministic. This is because there can be two different paths of A on which the prefix z is formed and, therefore, the pairs of repeated states in these paths are not identical, in general. Thus, the fact that w_1ww_2u' and w_1ww_2v' are in $L(A)$ does not imply that also w_1w_2u' and w_1w_2v' are in $L(A)$. The next lemma deals with nondeterministic automata and shows that, in this case, the largest value of j is higher – see also the counterexample of Section 4 showing that these largest values cannot be improved.

Lemma 3 *Let A be a (nondeterministic) finite automaton accepting at least two words. There are distinct words u, v in $L(A)$ and an index $j \leq s^2$ such that $u(j) \neq v(j)$ and $\mathrm{dist}(L(A)) = \mathrm{dist}(u, v)$, where s is the number of states in A.*

Proof. For the sake of contradiction, assume that for all pairs of distinct words u, v in $L(A)$ with $\mathrm{dist}(u, v) = \mathrm{dist}(L(A))$ it is the case that, for any index j, $u(j) \neq v(j)$ implies $j > s^2$. For any edit string f for which the input and output parts are different, we denote with i_f the smallest value of an index i for which $(\mathrm{inp}(f))(i) \neq (\mathrm{out}(f))(i)$. Now let h be an edit string that realizes the edit distance of $L(A)$. Moreover, h is chosen to be of minimal length $|h|$. By the assumption, it is the case that $i_h > s^2$. There are distinct words x and y, and alphabet symbols $a_1, \ldots, a_{i_h-1}$ such that

$$\mathrm{inp}(h) = a_1 \cdots a_{i_h-1}x \quad \text{and} \quad \mathrm{out}(h) = a_1 \cdots a_{i_h-1}y$$

and $\mathrm{dist}(L(A)) = \mathrm{dist}(x, y)$ and $x(1) \neq y(1)$. Let g be an edit string that realizes the edit distance of $\{x, y\}$. Define the edit string

$$h' = (a_1/a_1) \cdots (a_{i_h-1}/a_{i_h-1})\, g.$$

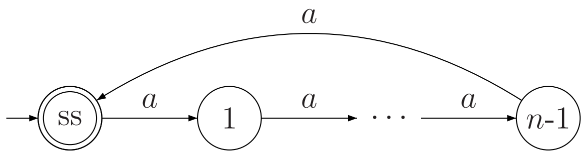

Fig. 1. The finite automaton C_n. The start state is ss. In pictures for automata, final states are indicated with double lines.

One can verify that h' realizes the edit distance of $L(A)$. Now the prefix $(a_1/a_1)\cdots(a_{i_h-1}/a_{i_h-1})$ of h' appears in some computation, call it P, of $A\cap_E A$. This computation involves i_h states and, as $i_h > s^2$, there is a repeated state in P. This implies that there is a shorter computation, say P', from the first to the last state of P and, therefore, there is an edit string h'' that is shorter than h and realizes the edit distance of $L(A)$. But this contradicts the assumption about the minimality of h. □

Thus, when the automaton A is nondeterministic the upper bound on the index j is larger. In the next section, in the context of discussing the complexity of the algorithm for computing the edit distance, we demonstrate that the two upper bounds on the index j cannot be improved, asymptotically at least.

In the next lemma we show that the edit distance of a regular language is upper bounded by the diameter of any automaton accepting the language. This upper bound cannot be improved, in general. For example, in Figure 1, we show a sequence of automata (C_n) such that $\mathrm{diam}(C_{n-1}) < \mathrm{diam}(C_n)$ and $L(C_n)$ consists of the words $\lambda, a^n, a^{2n}, \ldots$. Moreover, $\mathrm{dist}(L(C_n)) = \mathrm{diam}(C_n) = n$.

Lemma 4 *For any finite automaton A accepting at least two words, we have that* $\mathrm{dist}(L(A)) \leq \mathrm{diam}(A)$.

Proof. Let $(u_1, u_2, \ldots)$ be an ordering of $L(A)$ such that $|u_i| \leq |u_{i+1}|$ for all i. The word u_2 can be written as $u_1'z$, for some words u_1' and z with $|u_1'| = |u_1|$. Obviously,

$$\mathrm{dist}(u_1, u_1'z) \leq \mathrm{dist}(u_1, u_1') + \mathrm{dist}(u_1', u_1'z) \leq |u_1'| + |z|.$$

Hence, $\mathrm{dist}(L(A)) \leq |u_2|$. If $|u_2| < \mathrm{diam}(A)$, the statement is true. So assume that $|u_2| \geq \mathrm{diam}(A)$. Then there is an accepting computation of A for u_2 of the form

$$P = p_0 a_1 p_1 \cdots a_i p_i \cdots p_{k-1} a_k p_k \cdots a_n p_n$$

such that $n+1 > \mathrm{diam}(A)$ and k is the first index for which $p_k = p_i$, for some $i < k$. Then there are words w_1, w, w_2 such that w is nonempty, $|w_1 w| = k$, $w_1 w w_2 = u_2$ and $w_1 w_2$ is in $L(A)$ (in fact $w_1 w_2$ must be equal to u_1). Moreover, it follows that $\mathrm{dist}(L(A)) \leq \mathrm{dist}(w_1 w w_2, w_1 w_2) = |w|$. Now, as all states

$p_0, \ldots, p_{k-1}$ are distinct and p_0 is the start state, we have that $k \le \mathrm{diam}(A)$. This implies $|w| \le \mathrm{diam}(A)$ and, therefore, $\mathrm{dist}(L(A)) \le \mathrm{diam}(A)$ as required. □

Next we show the existence of a polynomial size automaton accepting all edit strings h for which the input part of h and the output part of h differ at position k, for some given positive integer k. In the expression for the size of this automaton we show explicitly the contribution of the size $|\Sigma|$ of the alphabet Σ because the alphabet is not related to the given parameter k in any way.

Lemma 5 *For any positive integer k, we can construct a finite automaton T_k of size $\Theta(k^2|\Sigma|^2 + k|\Sigma|^3)$ such that T_k accepts all edit strings h for which $(\mathrm{inp}(h))(k) \ne (\mathrm{out}(h))(k)$.*

Proof. The states of T_k and their associated meanings are defined as follows.

(1) $[i, j]$ for $0 \le i < k$ and $0 \le j < k$: means that the automaton has seen exactly i input symbols and exactly j output symbols.
(2) $[ak, j]$ for $a \in \Sigma$ and $0 \le j < k$: means that the automaton has seen at least k input symbols and exactly j output symbols, and the k-th input symbol was a.
(3) $[i, ak]$ for $a \in \Sigma$ and $0 \le i < k$: means that the automaton has seen at least k output symbols and exactly i input symbols, and the k-th output symbol was a.
(4) $[k, k]$: means that the automaton has seen at least k input and at least k output symbols and the k-th input symbol was different from the k-th output symbol.

The start state is $[0, 0]$ and the final states are $[ak, j]$, $[i, ak]$, and $[k, k]$ for all $i, j < k$ and $a \in \Sigma$. Of course T_k accepts an edit string h if and only if T_k starts at state $[0, 0]$ and ends its computation on h at one of the final states. This implies that h is accepted if and only if $(\mathrm{inp}(h))(k) \ne (\mathrm{out}(h))(k)$, as required. The correctness of the construction is established by defining the transitions of T_k such that the meaning of the states is preserved. This is done next – see also Fig. 2. It is assumed that $0 \le i, j \le k - 1$ and $a, b, c \in \Sigma$.

(1) $[i, j](\lambda/a)[i, j + 1]$, if $j < k - 1$. This defines $k(k - 1)|\Sigma|$ transitions.
(2) $[i, j](\lambda/a)[i, ak]$, if $j = k - 1$. This defines $k|\Sigma|$ transitions.
(3) $[i, j](a/\lambda)[i + 1, j]$, if $i < k - 1$. This defines $k(k - 1)|\Sigma|$ transitions.
(4) $[i, j](a/\lambda)[ak, j]$, if $i = k - 1$. This defines $k|\Sigma|$ transitions.
(5) $[i, j](a/b)[i + 1, j + 1]$, if $i, j < k - 1$. This defines $(k - 1)^2|\Sigma|^2$ transitions.
(6) $[i, j](a/b)[ak, j + 1]$, if $i = k - 1$ and $j < k - 1$. This defines $(k - 1)|\Sigma|^2$ transitions.
(7) $[i, j](a/b)[i + 1, bk]$, if $i < k - 1$ and $j = k - 1$. This defines $(k - 1)|\Sigma|^2$ transitions.

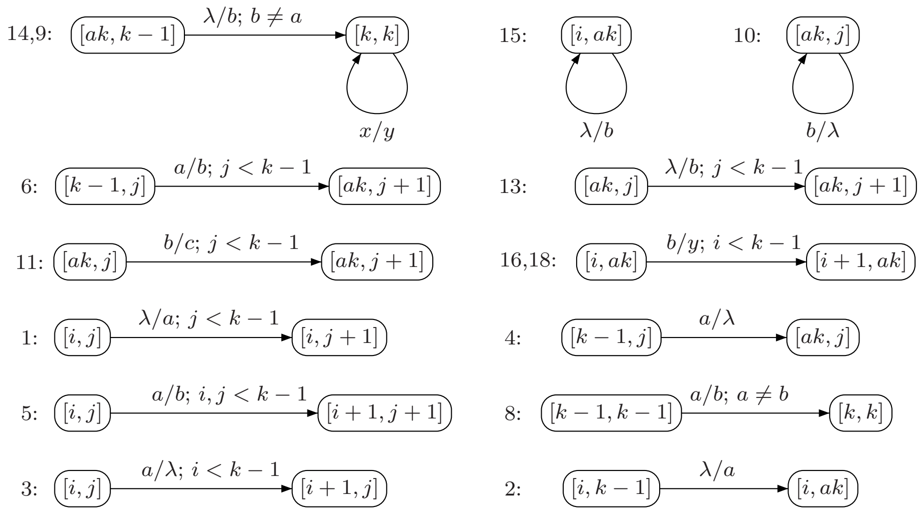

Fig. 2. Transitions of T_k (see proof of Lemma 5).

(8) $[i,j](a/b)[k,k]$, if $i = j = k-1$ and $a \neq b$. This defines $|\Sigma|^2 - |\Sigma|$ transitions.
(9) $[k,k](x/y)[k,k]$ with $x, y \in \Sigma \cup \{\lambda\}$ and $xy \neq \lambda$. This defines $|\Sigma|^2 + 2|\Sigma|$ transitions.
(10) $[ak,j](b/\lambda)[ak,j]$. This defines $k|\Sigma|^2$ transitions.
(11) $[ak,j](b/c)[ak,j+1]$, if $j < k-1$. This defines $(k-1)|\Sigma|^3$ transitions.
(12) $[ak,j](b/c)[k,k]$, if $j = k-1$ and $a \neq c$. This defines $|\Sigma|^2(|\Sigma|-1)$ transitions.
(13) $[ak,j](\lambda/b)[ak,j+1]$, if $j < k-1$. This defines $(k-1)|\Sigma|^2$ transitions.
(14) $[ak,j](\lambda/b)[k,k]$, if $j = k-1$ and $a \neq b$. This defines $|\Sigma|^2(|\Sigma|-1)$ transitions.
(15) $[i,ak](\lambda/b)[i,ak]$. This defines $k|\Sigma|^2$ transitions.
(16) $[i,ak](b/c)[i+1,ak]$, if $i < k-1$. This defines $(k-1)|\Sigma|^3$ transitions.
(17) $[i,ak](b/c)[k,k]$, if $i = k-1$ and $a \neq b$. This defines $|\Sigma|^2(|\Sigma|-1)$ transitions.
(18) $[i,ak](b/\lambda)[i+1,ak]$, if $i < k-1$. This defines $(k-1)|\Sigma|^2$ transitions.
(19) $[i,ak](b/\lambda)[k,k]$, if $i = k-1$ and $a \neq b$. This defines $|\Sigma|^2(|\Sigma|-1)$ transitions.

The claim about the size of T_k follows if one calculates the total number of transitions – this number exceeds the number of states of T_k. □

4 Computing the distance in polynomial time

In this section we present the main result of the paper, that is, a polynomial time algorithm to compute the edit distance of a given regular language. The time complexity of the algorithm is higher when the language is given via a nondeterministic automaton.

In the first place one could approach the problem as follows. Let A be the given finite automaton, and let B be the automaton $A \cap_E A$ (see Section 3) accepting all edit strings for which the input and output parts are in $L(A)$. If we are able to construct an automaton T accepting all edit strings for which the input and output parts are distinct, then we can compute the automaton $B \cap T$ that accepts all edit strings for which the input and output parts are distinct and in $L(A)$. When $B \cap T$ is treated as a weighted graph such that the weight of a transition is either 0 or 1, and it is 1 when the label of the transition is an error, then the edit distance of $L(A)$ is simply the weight of the shortest path from the initial to a final state in $B \cap T$. The question here is whether the required automaton T exists. We believe that T does not exist. For the sake of completeness, however, we note that one can construct a finite-state transducer T realizing all pairs of words (u, v) with $u \neq v$, but when viewed as an automaton, T does not accept *all* possible edit strings h whose parts are distinct as desired.

Although the idea described in the preceding paragraph fails, we can build on that idea using the results of the previous section and arrive at the desired algorithm. In the algorithm we shall use the following result from [4].

Lemma 6 *There is a linear time $\Theta(|G|)$ algorithm that takes as input a weighted graph G whose weights on the edges are in $\{0, 1\}$, and a vertex p of G, and computes, for each vertex of G, the length of the shortest path to the vertex from p.*

Theorem 7 *The following problem is computable in polynomial time.*

Input: *A (deterministic or nondeterministic) finite automaton A accepting at least two words.*
Output: *The edit distance* $\mathrm{dist}(L(A))$.

Proof. Let $q(A)$ be the quantity $\mathrm{diam}(A)$ or s^2, depending on whether A is deterministic or not, where s is the number of states in A. We have the following algorithm.

Input = some finite automaton A;
dist = $\mathrm{diam}(A)$;
$B = A \cap_E A$;
$m = q(A)$;
for each $j = 1, \ldots, m$
{
  $G = B \cap T_j$;
  weight = $\mathrm{ShortestPathWeight}(G)$;
  if (weight < dist) dist = weight;

}
Output = dist;

The function call ShortestPathWeight(G) treats the automaton G as a weighted graph and computes the shortest path to every state from the start state. The weight of a transition $p(x/y)q$ of G, where (x/y) is an edit operation and p, q are states, is 1 if $x \neq y$, or 0 if $x = y$. The algorithm initializes dist to the maximum possible value of $\mathrm{dist}(L(A))$ – see Lemma 4 – and then updates dist according to the equation

$$\mathrm{dist}(L(A)) = \min\{\mathrm{weight}(h) \mid h \in L(A \cap_E A) \cap L(T_j),\ \text{for some } j \leq q(A)\}.$$

By the lemmata of Section 3, we know that $\mathrm{dist}(L(A)) = \mathrm{dist}(u, v)$ for some words u, v in $L(A)$ with $u(j) \neq v(j)$ and $j \leq q(A)$. Equivalently, $\mathrm{dist}(L(A))$ is equal to $\mathrm{weight}(h)$ for some smallest weight edit string h such that $h \in L(A \cap_E A)$ and $(\mathrm{inp}(h))(j) \neq (\mathrm{out}(h))(j)$, for some $j \leq q(A)$. This establishes that the algorithm operates correctly.

We turn now to the time complexity of the algorithm. The function call $\mathrm{diam}(A)$ that returns the diameter of A operates in time $O(|A|)$ using a plain depth first search algorithm that computes, for each state, the length of the shortest (unweighted) path to the state from the start state. The automaton B can be computed in time $O(|A|^2)$ and the automaton G in time $O(|A|^2|T_j|)$. By Lemma 6, the function ShortestPathWeight(G) operates in time $O(|A|^2|T_j|)$. Hence, using also Lemma 5, the dominating term in the time complexity is

$$\sum_{j=1}^{q(A)} |A|^2(j^2|\Sigma|^2 + j|\Sigma|^3) = O\left(|A|^2|\Sigma|^2 q(A)^2(q(A) + |\Sigma|)\right).$$

□

The question that arises here is whether the quantity $m = q(A)$, which is the upper bound of the loop of the algorithm, can be reduced asymptotically. In turn, the question is whether the upper bounds on the index j that appear in the lemmata of Section 3 can be improved asymptotically. Next we show that this is not possible. Let us establish the following notation for a finite automaton A accepting at least two words:

- $\mathrm{MinInd}(A)$ denotes the smallest value of an index j for which there are words u, v in $L(A)$ such that $u(j) \neq v(j)$ and $\mathrm{dist}(u, v) = \mathrm{dist}(L(A))$.

The upper bound $\mathrm{diam}(A)$ of Lemma 2 on the index $\mathrm{MinInd}(A)$ cannot be improved. More specifically there is a sequence (A_n) of deterministic finite automata such that $\mathrm{diam}(A_{n-1}) < \mathrm{diam}(A_n)$ and, for all $n \geq 2$, $\mathrm{MinInd}(A_n) = \mathrm{diam}(A_n) = n$. The automaton A_n is shown in Figure 3.

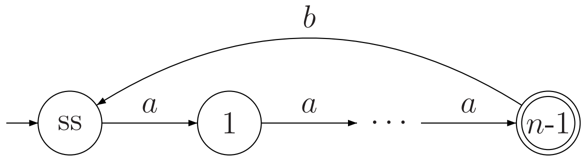

Fig. 3. The deterministic finite automaton A_n.

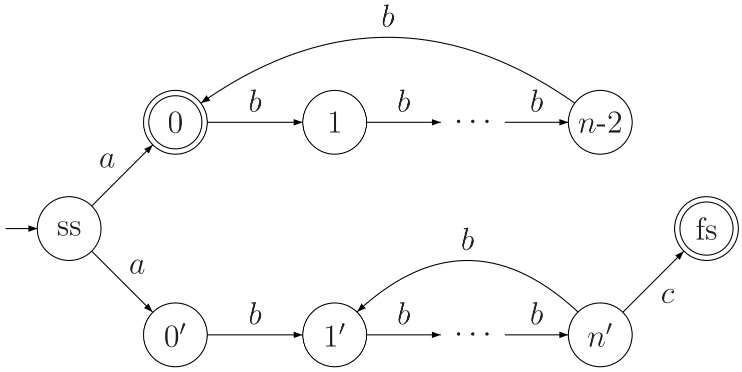

Fig. 4. The nondeterministic finite automaton B_n.

Similarly, the upper bound s^2 of Lemma 3 on the index MinInd(A) cannot be improved, asymptotically. More specifically, there is a sequence (B_n) of nondeterministic finite automata such that $s_{n-1} < s_n$, where s_n is the number of states in B_n, and $\mathrm{MinInd}(B_n) = \Theta(s_n^2)$, as $n \to \infty$. The automaton B_n is shown in Figure 4. It should be clear that $s_n = 2n + 2$ and the language $L(B_n)$ consists of all words of the forms $ab^{i(n-1)}$ and $ab^{n+jn}c$, for all $i, j \geq 0$. Moreover, for $n \geq 3$, $\mathrm{dist}(L(B_n)) = 1$ and $\mathrm{dist}(ab^{i(n-1)}, ab^{(j+1)n}c) = 1$ if and only if $i(n-1) = (j+1)n$ if and only if $i = mn$ and $j + 1 = m(n-1)$ for some $m \geq 1$. Then, for $m = 1$, the words ab^{n^2-n} and $ab^{n^2-n}c$ differ at position $n^2 - n + 2$, which implies that $\mathrm{MinInd}(B_n) = n^2 - n + 2 = \Theta(s_n^2)$.

5 Discussion

We have shown that the problem of computing the edit distance of a given regular language is of polynomial time complexity. We have restricted our attention to the case of the unweighted edit distance because this case is closely related to the concept of error detection as discussed in the introduction. However, the methods can be applied even in the case where the edit distance is weighted such that the weights on the errors are positive numbers. In this case, however, the function call ShortestPathWeight(G) in the algorithm of Theorem 7 has to invoke Dijkstra's algorithm instead of the algorithm of Lemma 6. This change would increase the time complexity of the algorithm – which of course would remain polynomial.

The first question that arises, which we pose as an open problem, is whether there is another algorithm for our edit distance problem whose time complexity is asymptotically lower. It appears that this might be possible for certain special cases at least. For example, when the given automaton A is a trellis [15], that is, an automaton with a single final state accepting only words of the same length n, then $n = \mathrm{diam}(A) - 1$ and in typical applications n is much smaller than $|A|$.

The next question that arises is whether the results extend to the case of other difference measures for words, in particular, those defined by weighted automata [8,3]. In this case, the edit strings that one can use to transform words are restricted to only those permitted by the weighted automaton. We note that, for the case of the plain edit distance considered here, two of the arguments that were used in the proofs are (i) $\mathrm{dist}(xuy, xvy) = \mathrm{dist}(u, v)$, for all words x, u, v, y; and (ii) if h is any edit string that is used to transform words then also the edit string $(a_1/a_1)\cdots(a_n/a_n)h$ is permitted for transforming words.

Acknowledgements

The author is grateful to Nicolae Sântean for providing fruitful comments on this paper, and to the referees of this paper for helping improve the presentation and suggesting the references [1] and [9].

References

[1] Amir, A., Lewenstein, M., and Lewenstein, N. (2000), Pattern Matching in Hypertext, *Journal of Algorithms* **35**, 82–99.

[2] Kari, L., and Konstantinidis, S. (2004), Descriptional complexity of error/edit systems, *J. Automata, Languages and Combinatorics* **9**, 293–309.

[3] Kari, L., Konstantinidis, S., Perron, S., Wozniak, G., and Xu, J. (2003), Finite-state error/edit-systems and difference-measures for languages and words. Tech. Report 2003-01, Dept. Math. and Computing Sci., Saint Mary's University, Canada, pp 10. Available electronically at `http://www.stmarys.ca/academic/science/compsci/`

[4] Kari, L., Konstantinidis, S., Perron, S., Wozniak, G., and Xu, J. (2004), Computing the Hamming distance of a regular language in quadratic time, *WSEAS Transactions on Information Science & Applications* **1**, 445–449.

[5] Konstantinidis, S. (2005), Computing the Levenshtein distance of a regular language, *in* "Proceedings (electronic) of 2005 IEEE Information Theory

Workshop on Coding and Complexity, 29th Aug. - 1st Sept, 2005, Rotorua, New Zealand", 113–116.

[6] Levenshtein, V. I. (1966), Binary codes capable of correcting deletions, insertions, and reversals, *Soviet Physics Dokl.* **10**, 707–710.

[7] Marcus, B. H., Roth, R. M., and Siegel, P. H., Constrained systems and coding for recording channels. In [11], pp. 1635–1764.

[8] Mohri, M. (2003), Edit-distance of weighted automata: general definitions and algorithms, *International Journal of Foundations of Computer Science* **14**(6), 957–982.

[9] Myers, E. W., and Miller, W. (1989), Approximate matching of regular expressions, *Bulletin of Mathematical Biology* **51**(1), 5–37.

[10] Pighizzini, G. (2001), How hard is computing the edit distance? *Inform. and Comput.* **165**, 1–13.

[11] Pless V. S., and Huffman W. C., Eds. (1998), *Handbook of Coding Theory*, Elsevier.

[12] S. Roman, S. (1992), *Coding and Information Theory.* Springer-Verlag, New York.

[13] Rozenberg, G., and Salomaa, A., Eds. (1997), *Handbook of Formal Languages, vol. I.* Springer, Berlin.

[14] Sankoff, D., and Kruskal J., Eds. (1999), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* CSLI Publications.

[15] Vardy, A. Trellis structure of codes. In [11], pp. 1989–2117.

[16] Wagner, R. A. (1974), Order-n correction for regular languages, *Communications of the ACM* **17**(5), 265–268.

[17] Yu, S. Regular languages. In [13], pp 41–110.