How Hard is to Compute the Edit Distance*

Giovanni Pighizzini

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
via Comelico, 39 – 20135 Milano – Italy
pighizzi@ghost.dsi.unimi.it

Abstract. The notion of *edit distance* arises from very different fields, such as self–correcting codes, parsing theory, speech recognition and molecular biology. The edit distance between an input string and a language L is the minimum number of *edit operations* (*substitution* of a symbol in another incorrect symbol, *insertion* of an extraneous symbol, *deletion* of a symbol) needed to change the input string into a sentence of L.
In this paper we study the complexity of computing the edit distance, discovering sharp boundaries between classes of languages for which this function can be efficiently evaluated and classes of languages for which it seems to be difficult to compute. Our main result is a parallel algorithm for computing the edit distance for the class of languages accepted by one–way nondeterministic auxiliary pushdown automata working in polynomial time, a class that strictly contains context–free languages. Moreover, we show that this algorithm can be extended in order to find a sentence of the language with minimum distance with respect to the input string.

1 Introduction

Consider a language $L \subseteq \Sigma^*$ and suppose to get strings which represent instances of L affected by errors due, for example, to typing process, to noise in the communication channel, *etc.* We want to find the original sentences of L from which, more likely, the strings we have received were originated. In order to find such sentences, we adopt a minimum distance criterion, i.e., given an input string x, we are interested in determining a sentence $y \in L$ with minimal *edit distance* from x.

The *edit* or *Levenshtein distance* [AP72, WF74] between two strings is the minimum number of edit operations (i.e., the *substitution* of a symbol by an incorrect symbol, the *insertion* of an extraneous symbol, and the *deletion* of a symbol), that transform a string into another. Thus, fixed a language L, the edit distance of L is the function which associates to any input string x the minimal distance between x and strings belonging to L.

* This work was supported in part by the ESPRIT Basic Research Action No. 6317: "Algebraic and Syntactic Methods in Computer Science (ASMICS 2)" and by the Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST).

Edit distances were initially studied in code theory [Lev66], parsing theory [AP72, Pig92], and speech recognition [OTK76, Ack80]. Interesting combinatorial problems on strings, as for example the computation of longest common subsequences [AG87], can be reduced to edit distances. Very recently they regained a certain popularity due to their connections with problems in molecular biology [EGG90, Kar93].

In this paper, we are interested in establishing general results about the complexity of computing edit distances. In particular, we want to delineate the boundaries among classes of languages for which at least a strong evidence can be given about the "hardness" of edit distances and classes for which edit distances can be efficiently computed. The results we obtain are quite tight. More precisely, in Section 3, we show that the edit distance of any language accepted in logarithmic space by a deterministic Turing machine is computable in polynomial time if and only if P = NP. On the other hand, if we restrict our attention to one–way devices, edit distance becomes feasible. In fact, our main result, proved in Section 4, states that the edit distance of languages accepted by one–way nondeterministic auxiliary pushdown automata (i.e., logspace bounded Turing machines augmented with a pushdown store) working in polynomial time (a class of languages wider than context–free) can be efficiently computed.

In general people is not really interested in computing edit distances, but mainly in finding a sentence with minimal edit distance from the input string. In Section 5, we consider such a problem and we show that in the case of languages accepted by one–way nondeterministic auxiliary pushdown automata working in polynomial time, it can be solved in the same bounds given in Section 4 for computing the edit distance.

This paper is a continuation of a stream of research whose aim is that of identifying properties other than the membership which are easily computable for certain classes of languages (see, e.g., [Huy90, Huy91, ABP93, ÀJ93b]).

For brevity reasons, some of the proofs have been omitted from this version of the paper.

2 Preliminary definitions and results

Given a finite alphabet Σ and the monoid Σ^* of strings over Σ, the binary relation $\vdash$, representing the three kinds of syntax errors considered in this paper can be defined as follows. For x and y in Σ^*, $x \vdash y$ if and only if there exist $v, w \in \Sigma^*$, $a, b \in \Sigma$ such that $x = vaw$, $y = vbw$ and $a \neq b$ (*substitution* of a symbol), or $y = vaw$ and $x = vw$ (*deletion*), or $y = vw$ and $x = vaw$ (*insertion*). Observe that $\vdash$ is symmetric. By $\stackrel{k}{\vdash}$ we denote the composition of $\vdash$ with itself k times. Then $x \stackrel{k}{\vdash} y$ if and only if y can be obtained from x with k errors. The *edit distance* between two strings $x, y \in \Sigma^*$, denoted by $d(x, y)$, is defined as the minimum integer k such that $x \stackrel{k}{\vdash} y$.

For example given $\Sigma = \{a, b, +, *\}$, we have:

$$a + a * a \vdash aa + a * a \vdash aa + ba * a \vdash aa + bb * a \vdash aa + bb * .$$

Then $a + a * a \vdash^{4} aa + bb*$. Moreover, it is possible to verify that this is optimal, i.e., $d(aa + bb*, a + a * a) = 4$.

The following properties of the edit distance can be easily verified:

- $\forall u, v, w, z \in \Sigma^*$: $d(uv, wz) \leq d(u, w) + d(v, z)$;
- $\forall x', x'', y \in \Sigma^*$: $d(x'x'', y) = \min\{d(x', y') + d(x'', y'') \mid y = y'y''\}$.

Given a language $L \subseteq \Sigma^*$ and a string $x \in \Sigma^*$, we define the edit distance $d(x, L)$ between x and L as the minimum of the distances between x and strings belonging to L, i.e., $d(x, L) = \min\{d(x, y) \mid y \in L\}$, with the convention that $d(x, \emptyset) = \infty$. We point out that if L is not empty then $d(x, L)$ is at most $|x| + m_L$, where m_L denotes the length of the shortest string belonging to L. In fact, we can assume that all $|x|$ symbols of x are insertions and all m_L symbols of the shortest string of L have been deleted. From this fact and the trivial inequality $d(x, y) \geq |(|x| - |y|)|$, it turns out that if $y \in L$ is a string with minimal distance from x, i.e., $d(x, L) = d(x, y) = \min\{d(x, z) \mid z \in L\}$, then $|y| \leq 2|x| + m_L$.

When we deal with Turing machines, we will denote by Σ the input alphabet, by Γ the worktape alphabet and by Q the set of states. We always suppose that these sets are pairwise disjoint.

We briefly recall that a *one–way nondeterministic auxiliary pushdown automaton* (1–NAuxPDA) [Bra77] is a nondeterministic Turing machine having a one–way, end–marked, read–only input tape, a pushdown tape, and a two–way, read/write work tape *with a logarithmic space bound.* (For more formal definitions see, e.g, [HU79].) "Space" on an 1–NAuxPDA means space on the work tape only (excluding the pushdown). Without loss of generality, we make the following assumptions about 1–NAuxPDAs: (1) at the start of the computation the pushdown store contains only one symbol Z_0; this symbol is never pushed or popped on the stack; (2) there is only one final state q_f; when the automaton reaches the state q_f the computation stops; (3) the input is accepted if and only if the automaton reaches q_f, the pushdown store contains only Z_0, all the input has been scanned and the worktape is empty; (4) if the automaton moves the input head, then no operations are performed on the stack; (5) every push adds exactly one symbol on the stack; and (6) input head can be kept stationary only after moves that do not depend on the content of currently scanned input square.

We recall that a *surface configuration* of a 1–NAuxPDA M [Coo71] on an input string of length n is a 5–tuple (q, w, j, γ, i) where q is the state of M, w is a string of worktape symbols (the *worktape contents*), j is an integer, $1 \leq j \leq |w|$ (the *worktape head position*), γ is a pushdown symbol (the *stack top*), and i is an integer $1 \leq i \leq n + 1$ (the *input head position*). Throughout the paper, we will denote by $\mathcal{S}_n$ the set of all surface configurations on inputs of length n, and, for any surface configuration A, by i_A the input head position in A. Observe that the size of any surface configuration belonging to $\mathcal{S}_n$ is at most $O(\log n)$.

The initial surface configuration, denoted by A_0, is $(q_0, \sharp, 1, Z_0, 1)$ where q_0 is the initial state, and by our assumptions on 1–NAuxPDAs the only *accepting surface configuration* on input of length n is the tuple $A_f^n = (q_f, \sharp, 1, Z_0, n + 1)$ where $\sharp$ represents the empty worktape.

Given a string $y \in \Sigma^*$ and an integer $t \geq 0$, a pair (A, B) of surface configurations is said to be *realizable on y within t steps* if M from A can reach in at most t steps B ending with its stack at the same height as in A, without popping it below its level in A at any point in this computation, and consuming the input substring y.[2] Of course, if (A, C) is realizable on y' within t' steps and (C, B) is realizable on y'' within t'' steps then (A, B) is realizable on $y'y''$ within $t' + t''$ steps. Moreover, for any surface configuration A, the pair (A, A) is realizable on ϵ in zero steps, and it cannot be realizable on nonempty strings.

Given two surface configurations A, B and an integer $t \geq 0$, by L^t_{AB} we denote the set of all strings $y \in \Sigma^*$ such that the pair (A, B) is realizable on y within t steps. Note that $|y| = i_B - i_A$. Thus, $L^{n^s}_{A_0 A^n_f}$ is the set of strings of length n accepted by a 1–NAuxPDA working in time n^s.

A *family of circuits* is a set $\{C_n \mid n \in N\}$ where C_n is a circuit for inputs of size n. $\{C_n\}$ is logspace uniform if the function $n \to C_n$ is computable on a deterministic Turing machine in logarithmic space. NC^k (AC^k) denotes the class of problems solvable by logspace uniform families of bounded (unbounded) fan–in Boolean circuits of polynomial size and $O(\log^k n)$ depth.

Unbounded fan–in circuits are equivalent to parallel random access machines with both concurrent read and concurrent write (CRCW PRAM). In particular the class AC^1 coincides with the class of functions computed by CRCW PRAM in $O(\log n)$ time, using a polynomial number of processors.

An *arithmetic circuit* is a circuit where the OR (addition) gates and the AND (multiplication) gates are interpreted over a suitable semi–ring. For further notions of parallel computation and arithmetic circuits the reader is referred to [Coo85],[MRK88].

3 Languages whose edit distance is unlike to be easy

The first problem we afford is that of identifying the smallest class of languages for which edit distances are hard to compute. In this section, we give strong evidence of the fact that this class is LOGSPACE. In fact, we prove that if LOGSPACE has polynomial time computable edit distances then P = NP.

Theorem 1. *The following statements are equivalent:*
(1) P = NP;
(2) for all languages in P *the edit distance is computable in polynomial time;*
(3) for all languages in LOGSPACE *the edit distance is computable in polynomial time.*

Proof. (1)⇒(2) For any language $L \in$ P, we consider the problem Π_L of deciding, given a string $x \in \Sigma^*$ and an integer $k \geq 0$, whether or not $d(x, L) \leq k$. Since the edit distance among two strings is computable in time proportional to the product of their lengths [WF74], it is not difficult to see that Π_L can be solved using the following nondeterministic polynomial time algorithm:

[2] This implies that the length of the input substring y is $i_B - i_A$.

input x, k
guess $y \in \Sigma^*$, with $|y| \leq 2|x| + m_L$
if $y \in L$ and $d(x, y) \leq k$ then accept
else reject

At this point, it is possible to devise an algorithm that computes $d(x, L)$ by calling such a machine on input (x, k) for increasing values of k. If P = NP this algorithm works in deterministic polynomial time.

(2)⇒(3) Obvious.

(3)⇒(1) Given a language L accepted by a nondeterministic Turing machine M in polynomial time $p(n)$, we consider the language L_{acc} whose strings represent accepting computations of M. More precisely, L_{acc} contains all strings of the form:

$$x \underbrace{\sharp\omega_0\sharp\omega_1\sharp \ldots \sharp\omega_{p(n)}}_{p(n)+1}, \tag{1}$$

where:

- $x \in \Sigma^*$, $n = |x|$, $\sharp \notin \Sigma \cup \Gamma \cup Q$;
- for all $i = 0, \ldots, p(n)$, ω_i is a string over $\Gamma \cup Q$ of polynomial length $p'(n)$ which encodes a configuration of M;
- ω_0 is the initial configuration of M;
- for all $j = 1, \ldots, p(n)$, ω_{j-1} yields ω_j by a move of M (on input x);
- $\omega_{p(n)}$ is an accepting configuration of M.

It is not difficult to verify that $L_{acc} \in$ LOGSPACE.

Now, we show how to decide membership to L by computing the edit distance for L_{acc}. With every input string $x \in \Sigma^*$ we associate the string

$$\phi(x) = x \underbrace{\sharp\overbrace{\$\ldots\$}^{p'(n)}\sharp\overbrace{\$\ldots\$}^{p'(n)}\sharp\ldots\sharp\overbrace{\$\ldots\$}^{p'(n)}}_{p(n)+1},$$

where $\$ \notin \Sigma \cup \Gamma \cup Q \cup \{\sharp\}$. Thus, given a string $x \in \Sigma^*$ and a string $I_y \in L_{acc}$ representing as in (1) an accepting computation of M on $y \in L$, using properties of the edit distance recalled in Section 2, it turns out that

$$d(\phi(x), I_y) = \min \{d(x, I') + d(\tilde{x}, I'') \mid I'I'' = I_y\},$$

where $\tilde{x} = (\sharp\$^{p'(n)})^{p(n)+1}$, i.e., $\phi(x) = x\tilde{x}$.

Since all \$s in $\tilde{x}$ are not in I'', we have $d(\tilde{x}, I'') \geq p'(n) \cdot (p(n) + 1)$, for any I''. Moreover, I_x can be obtained from $\phi(x)$ substituting each block of $p'(n)$ \$s with a string representing a suitable configuration of M. Thus, it turns out that for $x = y$ the equality holds, and then $d(\phi(x), I_x) = p'(n) \cdot (p(n) + 1)$. On the other hand, if $x \neq y$ then $d(x, I')$ is zero only when I' coincide with x and it is a prefix of y. In this case, it turns out that $I'' = y'\sharp\omega_0\sharp\ldots\sharp\omega_{p(|y|)}$, for some

$y' \in \Sigma^*$, $y' \neq \epsilon$, and the distance $d(\tilde{x}, I'')$ can be computed observing that all symbols of y', $\omega_0, \ldots, \omega_{p(|y|)}$ are not symbols of $\tilde{x}$. Since w.l.o.g. $p(|y|) \geq p(x)$, we have: $d(\tilde{x}, I'') \geq |y'| + \sum_{i=0}^{p(|y|)} |w_i| \geq 1 + p'(n) \cdot (p(n)+1) > p'(n) \cdot (p(n)+1)$. At this point, we can easily conclude that:

$$x \in L \quad \text{if and only if} \quad d(\phi(x), L_{acc}) = p'(n) \cdot (p(n)+1).$$

Thus, membership to L can be decided computing $d(\phi(x), L_{acc})$. □

4 Efficient computation of the edit distance

In [AP72] it is shown that the edit distance of context–free languages can be computed in polynomial time. Moreover, as proved in [Pig92], it can be efficiently computed using a parallel algorithm. In this section we consider a wider class of languages, namely languages accepted by 1–NAuxPDA working in polynomial time, and we exhibit a parallel algorithm for CRCW PRAM that computes edit distances in $O(\log n)$ time, using a polynomial number of processors. So, it turns out that the class of languages accepted in polynomial time by 1–NAuxPDA is the biggest class of languages we know for which edit distances can be efficiently computed.

The algorithm we present consists of defining and evaluating an arithmetic circuit over the commutative semiring $(\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$ (in the following $(\mathbb{N}, \min, +)$). More precisely, given a 1–NAuxPDA M working in polynomial time n^s, and an input string x of length n, let $H = 2n + m_L$ be an upper bound on the length of the string accepted by M with minimal distance from x, and $\mathcal{S}_H$ the set of surface configurations on inputs of length H. We define an arithmetic circuit C_n with three types of nodes:

- *input nodes,* with associated an input value from $\mathbb{N} \cup \{\infty\}$:

$$N_l = \{(A, B, i, j, t) \mid 0 \leq i \leq j \leq n,\ A, B \in \mathcal{S}_H,\ 0 \leq t \leq 1\};$$

- *min nodes,* which compute the minimum among the values of their sons:

$$N_{\min} = \{(A, B, i, j, t) \mid 0 \leq i \leq j \leq n,\ A, B \in \mathcal{S}_H,\ 1 < t \leq H^s\};$$

- *plus nodes,* which computes the sums of the values of their sons:

$$N_+ = \{(A, C, B, i, k, j, t', t'') \mid \\ 0 \leq i \leq k \leq j \leq n,\ A, B, C \in \mathcal{S}_H,\ 0 < t', t'' \leq H^s\}.$$

Connections among nodes are defined as follows. Node $(A, C, B, i, k, j, t', t'') \in N_+$ has exactly two sons, that is nodes $(A, C, i, k, t'), (C, B, k, j, t'') \in N_l \cup N_{\min}$. Node $(A, B, i, j, t) \in N_{\min}$ has three kinds of sons: (1) all nodes of the form $(A, C, B, i, k, j, t', t'') \in N_+$ such that $t' + t'' = t$; (2) all nodes $(A', B', i, j, t-2) \in N_l \cup N_{\min}$ such that A' can be reached from A via a push and B can be reached from B' via a pop of the same symbol; and (3) all nodes $(A, B, i, j, t-1) \in N_l \cup N_{\min}$.

Now, we define the value which will be associated with input nodes of the circuit, in such a way that each node $(A, B, i, j, t) \in N_{\min} \cup N_l$ of the circuit computes the minimum distance between the input substring $x_{i+1} \ldots x_j$ and the language L^t_{AB}. So, the solution of the problem is constructed bottom–up from optimal solutions of subproblems, according to the optimality principle of dynamic programming [Bel57].

For input nodes of the form $(A, B, i, j, 0)$, observing that $L^0_{AB} = \{\epsilon\}$ if $A = B$, and $L^0_{AB} = \emptyset$ otherwise, we define:

$$value(A, B, i, j, 0) = \begin{cases} j - i & \text{if } A = B \\ \infty & \text{otherwise.} \end{cases}$$

The definition for input nodes of the form $(A, B, i, j, 1)$ is more complicated; in fact, L^1_{AB} can be either empty or it can contain the empty string or some element of Σ. For instance, if L^1_{AB} contains one symbol $\sigma \in \{x_{i+1}, \ldots, x_j\}$, then $x_{i+1} \ldots x_j$ can be obtained from L^1_{AB} with $j - i - 1$ insertions. On the other hand, if L^1_{AB} contains some element of Σ, but it does not contain any element of $\{x_{i+1}, \ldots, x_j\}$, then $x_{i+1} \ldots x_j$ can be obtained from L^1_{AB} with one substitution and $j - i - 1$ insertions. More precisely, the value associated with a node $(A, B, i, j, 1)$ is defined as:

$$value(A, B, i, j, 1) = \begin{cases} j - i - 1 & \text{if } A \text{ yields } B \text{ consuming an input} \\ & \quad \text{symbol } \sigma \in \{x_{i+1}, \ldots, x_j\} \\ j - i & \text{if } (A = B) \vee (A \text{ yields } B \text{ without} \\ & \quad \text{consuming any input symbol belonging} \\ & \quad \text{to } \{x_{i+1}, \ldots, x_j\})^3 \\ \infty & \text{otherwise.} \end{cases}$$

The following lemma, whose proof can be given by induction, states the main property of the values computed by the gates of C_n with such inputs:

Lemma 2. *For any $A, B \in \mathcal{S}_H$, $0 \leq i \leq j \leq n$, $0 \leq t \leq H^s$, it holds that:*

$$value(A, B, i, j, t) = d(x_{i+1} \ldots x_j, L^t_{AB}).$$

Finally, recalling that the length of the string of L with minimal distance from x is at most $H = 2n + m_L$, we have:

$$\begin{aligned} d(x, L) &= \min_{m=0,\ldots,H} d(x, L^{m^s}_{A_0 A^m_f}) \\ &= \min_{m=0,\ldots,H} d(x, L^{H^s}_{A_0 A^m_f}) \\ &= \min_{m=0,\ldots,H} value(A_0, A^m_f, 0, n, H^s). \end{aligned}$$

Thus, we can extend circuit C_n with a node performing this operation and producing the output.

[3] This corresponds to two different situations: the pair (A, B) is realizable on the empty string, i.e., A yields B without consuming input symbols, or the pair (A, B) is realizable on some symbol $\sigma \in \Sigma$, but it is not realizable on symbols belonging to the set $\{x_{i+1}, \ldots, x_j\}$.

As a consequence, we are able to prove our main result:

Theorem 3. *The edit distance of languages accepted by one–way nondeterministic auxiliary pushdown automata working in polynomial time is in* AC^1.

Proof. (outline) The edit distance can be computed using the arithmetic circuit C_n above defined. This circuit has polynomial degree and it is defined over a *commutative* semiring. Thus, it can be efficiently evaluated using the technique presented in [MRK88], which consists of $O(\log n)$ applications of a routine called *Phase*, where a single application of *Phase* consists of nothing more complicated than matrix multiplication over the semiring $(\mathbb{N}, \min, +)$. Such a multiplication can be done in constant depth using unbounded fan–in circuits [SV81]. So, it turns out that the whole computation is in AC^1. $\square$

5 How to find the string with minimal edit distance

In Section 4 we showed how to compute the edit distance $d(x, L)$ between a string $x \in \Sigma^*$ and a language $L \subseteq \Sigma^*$ accepted by a 1–NAuxPDA working in polynomial time. Often, however, it is required to exhibit a string $y \in L$ with minimal distance from x. In this section we show how this problem can be solved: we give an algorithm for the computation of the string with minimal distance, which, as that presented in Section 4, can be executed by a CRCW PRAM in $O(\log n)$ time, using a polynomial number of processors.

The idea is that of modifying the evaluation function for gates of C_n, in order to compute optimal substrings during the evaluation process. To achieve this aim, we adopt the following scheme. With each node (A, B, i, j, t) of the circuit C_n we associate a pair (m, w), where $m \in \mathbb{N}$ represents the minimum distance between the input substring $x_{i+1} \ldots x_j$ and the language L^t_{AB}, and $w \in \Sigma^*$ is a string of L^t_{AB} with such a distance, i.e., $d(x_{i+1} \ldots x_j, L^t_{AB}) = d(x_{i+1} \ldots x_j, w)$.

Subsequently, we evaluate circuit C_n over the semiring $R = ((\mathbb{N} \times \Sigma^*) \cup \{\infty\}, \oplus, \otimes, \infty, (0, \epsilon))$, where the operation $\oplus$ selects between two elements of $\mathbb{N} \times \Sigma^*$ the element whose first component is minimal and the operation $\otimes$ adds the first components and concatenates the second ones, i.e., $(m', w') \otimes (m'', w'') = (m' + m'', w'w'')$.

Leaves of C_n are evaluated as follows:

$$value(A, B, i, j, 0) = \begin{cases} (j - i, \epsilon) & \text{if } A = B \\ \infty & \text{otherwise,} \end{cases}$$

$$value(A, B, i, j, 1) = \begin{cases} (j - i - 1, \sigma) & \text{if } A \text{ yields } B \text{ consuming a} \\ & \quad \text{symbol } \sigma \in \{x_{i+1}, \ldots, x_j\} \\ (j - i, \epsilon) & \text{if } A = B \text{ or } A \text{ yields } B \text{ without} \\ & \quad \text{consuming input symbols} \\ (j - i, \sigma) & \text{if } A \text{ yields } B \text{ consuming } \sigma \in \Sigma, \\ & \quad \text{but } A \text{ cannot yield } B \text{ consuming} \\ & \quad \text{a symbol in } \{x_{i+1}, \ldots, x_j\} \\ \infty & \text{otherwise.} \end{cases}$$

However semiring R is not commutative, and so it is not possible to apply the algorithm of [MRK88] to evaluate C_n gates. We show now how to overcome this problem in the case of the alphabet $\Sigma = \{0, 1\}$. The extension to other alphabets is trivial.

We recall that every surface configuration A contains the position i_A of the input head; thus, given a node (A, B, i, j, t) of the circuit, each string $w \in L^t_{AB}$ can contribute to the output string from position i_A to position $i_B - 1$. Moreover, the maximal length $H = 2n + m_L$ of the output, for inputs of length n, is *a priori* known. This will permit us to represent elements of R over a commutative semiring. More precisely, if the value associated with the node (A, B, i, j, t) is the pair (m, w), then we represent it as the integer $h = m \cdot 2^H + w \cdot 2^{i_B}$ (where w here is interpreted as the integer number whose binary representation is the string w). Observe that the binary representation of h is obtained concatenating the binary representation of m and the string $0^{i_A - 1} w 0^{H - i_B + 1}$, whose length is H. In this way, elements of R are represented as integers. It is not difficult to see that, in this representation, semiring operations $\oplus$ and $\otimes$ are substituted by min and $+$ respectively. Thus, the noncommutative semiring R in the case under consideration can be represented by the *commutative* semiring $(\mathbb{N}, \min, +)$, on which we are able to efficiently evaluate C_n as explained in Section 4.

Thus, the following result holds:

Theorem 4. *For any language L accepted by one–way nondeterministic auxiliary pushdown automata in polynomial time, the edit distance and the function which associates with any input string x a sentence of L with minimal distance from x are in* AC^1.

6 Concluding remarks

In this paper we showed that the edit distance of languages accepted in polynomial time by 1–NAuxPDA can be efficiently computed. Our algorithm is given assuming that all edit operations have the same cost. However, it can be easily extended to work also for other cost functions (see [WF74] for the definition of *weighted* edit distance).

The evaluation of the edit distance is an example of optimization problem over the semiring $(\mathbb{N}, \min, +)$. In particular, our results show that edit distances for languages accepted by 1–NAuxPDA working in polynomial time are in the class of functions computable by arithmetic circuits of polynomial size and polynomial degree over the semiring $(\mathbb{N}, \min, +)$. By results of [MRK88] this class is contained in AC^1 and, on the other hand, it can be shown (coding string on integers as outlined in Section 5) that its maximization version contains OptSAC^1 [Vin91, AJ93a], i.e., the class of functions computed by arithmetic circuits of polynomial size and polynomial degree over $(\Sigma^*, \max, \mathrm{concat})$.

References

[ABP93] E. Allender, D. Bruschi, and G. Pighizzini. The complexity of computing maximal word functions. *Computational Complexity*, 3:368–391, 1993.

[Ack80] M. Ackroyd. Isolated word recognition using the weighted Levenshtein distance. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 28:243–244, 1980.

[AG87] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.

[AJ93a] E. Allender and J. Jiao. Depth reduction for noncommutative arithmetic circuits. In *Proc. 25th ACM Symposium on Theory of Computing*, pages 515–522, 1993.

[ÀJ93b] C. Àlvarez and B. Jenner. A very hard log–space counting class. *Theoretical Computer Science*, 107:3–30, 1993.

[AP72] A. Aho and T. Peterson. A minimum distance error–correcting parser for context–free languages. *SIAM J. Computing*, 1:305–312, 1972.

[Bel57] R. Bellman. *Dynamic Programming.* Princeton Univ. Press, 1957.

[Bra77] F. Brandenburg. On one–way auxiliary pushdown automata. In *Proc. 3rd GI Conference*, Lecture Notes in Computer Science 48, pages 133–144, 1977.

[Coo71] S. Cook. Characterization of pushdown machines in terms of time–bounded computers. *Journal of the ACM*, 18:4–18, 1971.

[Coo85] S. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[EGG90] D. Eppstein, Z. Galil, and R. Giancarlo. Efficient algorithms with applications to molecular biology. In R. Capocelli, editor, *Sequences*, pages 59–74. Springer–Verlag, 1990.

[HU79] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computations.* Addison–Wesley, Reading, MA, 1979.

[Huy90] D. Huynh. The complexity of ranking simple languages. *Mathematical Systems Theory*, 23:1–19, 1990.

[Huy91] D. Huynh. Efficient detectors and constructors for simple languages. *International Journal of Foundations of Computer Science*, 2:183–205, 1991.

[Kar93] R. Karp. Mapping the genome: some combinatorial problems arising in molecular biology. In *Proc. 25th ACM Symposium on Theory of Computing*, pages 278–285, 1993.

[Lev66] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics–Doklady*, 10:707–710, 1966.

[MRK88] G. Miller, V. Ramachandran, and E. Kaltofen. Efficient parallel evaluation of straight–line code and arithmetic circuits. *SIAM J. Computing*, 17:687–695, 1988.

[OTK76] T. Okuda, E. Tanaka, and T. Kasai. A method for the correction of garbled words based on the Levenshtein metric. *IEEE Transactions on Computers*, 25:172–178, 1976.

[Pig92] G. Pighizzini. A parallel minimum distance error–correcting context–free parser. In *Theoretical Computer Science – Proceedings of the Fourth Italian Conference*, pages 305–316. World Scientific, 1992.

[SV81] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2:88–102, 1981.

[Vin91] V. Vinaj. Counting auxiliary pushdown automata and semi–unbounded arithmetic circuits. In *Proc. 6th Structure in Complexity Theory*, pages 270–284, 1991.

[WF74] R. Wagner and M. Fisher. The string–to–string correction problem. *Journal of the ACM*, 21:168–173, 1974.