International Journal of Foundations of Computer Science
Vol. 24, No. 7 (2013) 1067–1082

DOI: 10.1142/S0129054113400315

World Scientific
www.worldscientific.com

THE EDIT-DISTANCE BETWEEN A REGULAR LANGUAGE AND A CONTEXT-FREE LANGUAGE

YO-SUB HAN* and SANG-KI KO†
Department of Computer Science, Yonsei University
50, Yonsei-Ro, Seodaemun-Gu, Seoul 120-749, Korea
**emmous@cs.yonsei.ac.kr*
†*narame7@cs.yonsei.ac.kr*

KAI SALOMAA
School of Computing, Queen's University
Kingston, Ontario K7L 3N6, Canada
ksalomaa@cs.queensu.ca

Received 14 December 2012
Accepted 4 May 2013
Communicated by Oscar H. Ibarra and Hsu-Chun Yen

The edit-distance between two strings is the smallest number of operations required to transform one string into the other. The distance between languages L_1 and L_2 is the smallest edit-distance between string $w_i \in L_i$, $i = 1, 2$. We consider the problem of computing the edit-distance of a given regular language and a given context-free language. First, we present an algorithm that finds for the languages an optimal alignment, that is, a sequence of edit operations that transforms a string in one language to a string in the other. The length of the optimal alignment, in the worst case, is exponential in the size of the given grammar and finite automaton. Then, we investigate the problem of computing only the edit-distance of the languages without explicitly producing an optimal alignment. We design a polynomial time algorithm that calculates the edit-distance based on unary homomorphisms.

Keywords: Edit-distance; Levenshtein distance; regular languages; context-free languages.

1. Introduction

The edit-distance between two strings is the smallest number of operations required to transform one string into the other [10, 18]. The edit-distance is often used as a natural measure of string similarity. The problem of finding the edit-distance arises in many areas, such as, computational biology, text processing and speech recognition [13, 14, 16]. Edit-distance can be extended to a similarity measure between

*Corresponding author.

languages [2, 3, 7, 13]. Kari and Konstantinidis [6] introduced error or edit systems (e-systems, for short), which are formal languages over an alphabet of edit operations, which typically consist of deletions, insertions and substitutions. They also studied the descriptional complexity of e-systems.

Different variants of the edit-distance problem have been considered in connection with various applications. For instance, in the the error-correction problem we are given a set S of correct strings and an input string x, and the task is to find the most similar string $y \in S$, that is, the string that minimizes the edit-distance between y and x. If $x = y \in S$, then x has no error. If S is regular, we can use a finite-state automaton (FA) for S to identify the most similar string in S with respect to x [17]. Allauzen and Mohri [1] designed a linear-space algorithm that computes the edit-distance between a string and an FA. Pighizzini [15] considered the edit-distance between a string and a one-way nondeterministic auxiliary pushdown automaton. The problem of calculating the error-detection capability is related to the self-distance of a language L [7]. The self-distance, or inner distance, is the minimum edit-distance between any pair of distinct strings in L. We can use the self-distance as the maximum number of errors that L can identify. Based on this observation, Konstantinidis and Silva [8, 9] introduced the maximal error-detecting capability of a formal language and showed how to compute the maximal error-detecting capability of a regular language for channels involving any types of edit operations as errors.

We investigate the problem of computing the edit-distance between a regular language and a context-free language; namely, the problem of finding a closest pair of strings in the languages based on the edit-distance model. This was an open problem and it can be noted that the edit-distance problem between two context-free languages is known to be undecidable [13]. Based on structural properties of the FA and the pushdown automaton for the given languages, we design an algorithm to compute the edit-distance and the optimal alignment between the languages. We note that the optimal alignment, that is, the optimal sequence of edit operations transforming a string of one language to a string of the other may be exponential in the size of the given FA and pushdown automaton. Also, we design a polynomial time algorithm that computes the edit-distance between a context-free language and a regular language but does not give the optimal alignment between the languages.

In Sec. 2, we define some basic notions. We recall the definition of the edit-distance and the associated algorithmic problems in Sec. 3. Then, we present an efficient algorithm for computing the edit-distance and an optimal alignment between a context-free language and a regular language in Sec. 4. We present a faster algorithm that only computes the optimal cost using unary homomorphisms in Sec. 5.

2. Preliminaries

Here we recall some basic definition and fix notation. For complete background knowledge in automata theory, the reader may refer to textbooks [5, 19].

The size of a finite set S is $|S|$. Let Σ denote a finite alphabet and Σ^* denote the set of all finite strings over Σ. For $m \in \mathbb{N}$, $\Sigma^{\leq m}$ is the set of strings over Σ having length at most m. A language over Σ is a subset of Σ^*. The symbol λ denotes the null string. A (nondeterministic) finite automaton (FA) is specified by a tuple $A = (Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, Σ is an input alphabet, $\delta : Q \times \Sigma \to 2^Q$ is a multi-valued transition function, $s \in Q$ is the start state and $F \subseteq Q$ is a set of final states. If F consists of a single state f, we use f instead of $\{f\}$ for simplicity. When $q \in \delta(p, a)$, we say that state p has an *out-transition* to state q (p is a *source state* of q) and q has an *in-transition* from p (q is a *target state* of p). The transition function δ is extended in the natural way to a function $Q \times \Sigma^* \to 2^Q$. A string $x \in \Sigma^*$ is accepted by A if there is a labeled path from s to a state in F such that this path spells out the string x, namely, $\delta(s, x) \cap F \neq \emptyset$. The language $L(A)$ recognized by A is the set of strings accepted by A.

A (nondeterministic) pushdown automaton (PDA) is specified by a tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where Q is a finite set of states, Σ is a finite input alphabet, Γ is a finite stack alphabet, $\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to 2^{Q \times \Gamma^{\leq 2}}$ is a transition function, $q_0 \in Q$ is the start state, Z_0 is the initial stack symbol and $F \subseteq Q$ is the set of final states. Our definition restricts that each transition of P has at most two stack symbols, that is, each transition can push or pop at most one symbol. We use $|\delta|$ to denote the number of transitions in δ, that is, the sum of the cardinalities of the sets $\delta(q, a, \gamma)$ ($q \in Q$, $a \in \Sigma \cup \{\lambda\}, \gamma \in \Gamma$). We define that the size $|P|$ of P is $|Q| + |\delta|$. The language $L(P)$ recognized by P is the set of strings accepted by P.

A context-free grammar (CFG) G is a four-tuple $G = (V, \Sigma, R, S)$, where V is a set of variables, Σ is a set of terminals, $R \subseteq V \times (V \cup \Sigma)^*$ is a finite set of productions and $S \in V$ is the start variable. Let $\alpha A \beta$ be a string over $V \cup \Sigma$, where $A \in V$ and $A \to \gamma \in R$. Then, we say that A can be rewritten as γ and the corresponding derivation step is denoted $\alpha A \beta \Rightarrow \alpha \gamma \beta$. A production $A \to t \in R$ is a *terminating production* if $t \in \Sigma^*$. The reflexive, transitive closure of $\Rightarrow$ is denoted by $\overset{*}{\Rightarrow}$ and the context-free language generated by G is $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$. We say that a variable $A \in V$ is *nullable* if $A \overset{*}{\Rightarrow} \lambda$.

3. Edit-Distance

The edit-distance between strings x and y is the smallest number of basic operations that transform x to y. Here we use three basic operations: deletion, insertion and substitution of single letters. Given an alphabet Σ, an operation that deletes $a \in \Sigma$ (respectively, inserts $b \in \Sigma$ and substitutes b for a) is denoted as $(a \to \lambda)$ (respectively, $(\lambda \to b)$ and $(a \to b)$). Corresponding to an alphabet Σ, we define the alphabet of edit-operations as $\Omega_\Sigma = \{(a \to b) \mid a, b \in \Sigma \cup \{\lambda\}\}$. When Σ is known from the context we denote Ω_Σ simply as Ω. We call a string $\omega \in \Omega^*$ an *edit string* [6] or an *alignment* [13].

Let h be the mapping $\Omega^* \to \Sigma^* \times \Sigma^*$ defined by setting $h((a_1 \to b_1) \cdots (a_n \to b_n)) = (a_1 \cdots a_n, b_1 \cdots b_n)$. The mapping h is a morphism if catenation of elements

of $\Sigma^* \times \Sigma^*$ is defined componentwise. We say that an edit string $\omega \in \Omega^*$ transforms a string x into a string y if and only if $h(\omega) = (x, y)$.

Example 1. *Choose* $\Sigma = \{a, b, c\}$. *The edit string* $\omega = (a \to \lambda)(b \to b)(\lambda \to c)(c \to c)$ *is an alignment between abc and bcc. Note that* $h(\omega) = (abc, bcc)$.

A cost function $\mathtt{C} : \omega_i \to \mathbb{R}_+$ associates a non-negative cost to each edit operation $\omega_i \in \Omega$. In the natural way, the cost $\mathtt{C}(\omega)$ of an alignment $\omega = \omega_1 \cdots \omega_n$ is then:

$$\mathtt{C}(\omega) = \sum_{i=1}^{n} \mathtt{C}(\omega_i).$$

Definition 1. *The edit-distance* $d(x, y)$ *of two strings* x *and* y *over* Σ *is the minimal cost of an alignment between* x *and* y: $d(x, y) = \min\{\mathtt{C}(\omega) \mid h(\omega) = (x, y)\}$. *We say that an alignment* ω *is optimal if* $d(x, y) = \mathtt{C}(\omega)$.

Next the definition of edit-distance is extended for languages as follows.

Definition 2. *The edit-distance* $d(L, R)$ *between two non-empty languages* $L, R \subseteq \Sigma^*$ *is the minimum edit-distance between a string in* L *and a string in* R:

$$d(L, R) = \min\{d(x, y) \mid x \in L \text{ and } y \in R\}.$$

Mohri [13] proved that the edit-distance between two context-free languages is uncomputable and gave an algorithm to compute the edit-distance between two regular languages. We consider the intermediate case where one of the languages is context-free and the other is regular. We use the Levenshtein distance [10] for edit-distance and, thus, assign cost 1 to all edit operations; namely, $\mathtt{C}(a, a) = 0$ and $\mathtt{C}(a, \lambda) = \mathtt{C}(\lambda, a) = \mathtt{C}(a, b) = 1$ for all $a \neq b \in \Sigma$. Using different constant values for the costs would not change our algorithm significantly.

4. The Edit-Distance Between a Regular Language and a Context-Free Language

We present algorithms that compute the edit-distance between a regular language L and a context-free language R and find an optimal alignment ω such that $\mathtt{C}(\omega) = d(L, R)$. Let $A = (Q_A, \Sigma, \delta_A, s_A, F_A)$ be an FA for L and $P = (Q_P, \Sigma, \Gamma, \delta_P, s_P, Z_0, F_P)$ be a PDA for R. Let

$$m_1 = |Q_A|,\ m_2 = |Q_P|,\ n_1 = |\delta_A| \text{ and } n_2 = |\delta_P|. \tag{1}$$

Recall that in our definition each transition of a PDA has at most two stack symbols; namely, each transition can push or pop at most one symbol. It is well known that any context-free language can be recognized by such a PDA [5].

Our algorithm first constructs a new PDA $\mathcal{A}(A, P)$ (called *alignment PDA*) that recognizes the set of all possible alignments of two strings belonging, respectively, to L and R.

4.1. *Alignment PDA*

Given an FA $A = (Q_A, \Sigma, \delta_A, s_A, F_A)$ and a PDA $P = (Q_P, \Sigma, \Gamma, \delta_P, s_P, Z_0, F_P)$, we construct the alignment PDA $\mathcal{A}(A, P) = (Q_E, \Omega, \Gamma, \delta_E, s_E, Z_0, F_E)$, where $Q_E = Q_A \times Q_P$ is the set of states, $\Omega = \{(a \to b) \mid a, b \in \Sigma \cup \{\lambda\}\}$ is the alphabet of edit operations, $s_E = (s_A, s_P)$ is the start state, and $F_E = F_A \times F_P$ is the set of final states. The transition function δ_E consists of three types of transitions corresponding to the type of edit-operations; *deletion*, *insertion* and *substitution*, and additionally transitions that simulate λ-transitions of the PDA P. For $p' \in \delta_A(p, a)$ and $(q', M') \in \delta_P(q, b, M)$, where $p, p' \in Q_A$, $q, q' \in Q_P$, $a, b \in \Sigma$, $M \in \Gamma$, $M' \in \Gamma^*$, $N \in \Gamma$, we define δ_E to have the following transitions:

- $((p', q), N) \in \delta_E((p, q), (a \to \lambda), N)$, [deletion operation]
- $((p, q'), M') \in \delta_E((p, q), (\lambda \to b), M)$, [insertion operation]
- $((p', q'), M') \in \delta_E((p, q), (a \to b), M)$, [substitution operation]
- $((p, q'), M') \in \delta_E((p, q), \lambda, M)$, if $(q', M') \in \delta_P(q, \lambda, M)$.

Note that we have defined deletion operations for all stack symbols N in Γ. Then, in a deletion operation, the transition does not change the stack. For δ_E, we make $n_1 m_2$ transitions for deletions and $n_2 m_1$ transitions for insertions. For substitutions, we consider all pairs of transitions between A and P and, thus, add $n_1 n_2$ transitions. Therefore, the size of δ_E is $|\delta_E| = n_1 m_2 + n_2 m_1 + n_1 n_2 = O(n_1 n_2)$.

Intuitively, an alignment PDA $\mathcal{A}(A, P)$ was constructed to accept sequences of edit operations that transform a string accepted by the FA A to a string accepted by the PDA P. In light of this, the result of the following lemma is not very surprising, however, for the sake of completeness we include a proof to verify the correctness of the construction.

Lemma 3. *The alignment PDA $\mathcal{A}(A, P)$ accepts an edit string ω if and only if $h(\omega) = (x, y)$, where $x \in L(A)$ and $y \in L(P)$.*

Proof.
($\Longrightarrow$) Since $\mathcal{A}(A, P)$ accepts ω, there exists an accepting computation X_ω of $\mathcal{A}(A, P)$ on ω ending in a state (f_A, f_P) where $f_A \in F_A$, $f_P \in F_P$. We assume that $\omega = \omega_1 \cdots \omega_k$, $\omega_i \in \Omega$, $1 \leq i \leq k$. Denote the sequence of states of $\mathcal{A}(A, P)$ appearing in the computation X_ω just before reading the kth symbol of ω as $C_0, \ldots, C_{k-1}$, and denote the state (f_A, f_P) where the computation ends as C_k. (Note that the computation X_ω may contain also λ-transitions simulating the λ-transitions of P.) Consider the first component ($q_i \in Q_A$) of the state $C_i \in Q_E$, for $0 \leq i \leq k$, and the first component $a_j \in \Sigma \cup \{\lambda\}$ of the edit operation ω_j, for $1 \leq j \leq k$. Because of the construction of $\mathcal{A}(A, P)$, it follows that, when $a_{i+1} \in \Sigma$, we have $q_{i+1} = \delta_A(q_i, a_{i+1})$ and, when $a_{i+1} = \lambda$, we have $q_{i+1} = q_i$, for $0 \leq i \leq k-1$. Note that the transitions of δ_E reading an "insertion symbol" $(\lambda \to b)$ do not change the first components of the states and, similarly, transitions of δ_E simulating a λ-transition of P do not change the first component. Thus, the first components of the states $C_0, \ldots, C_k$ spell out an

accepting computation of A on the string $x = a_1 \cdots a_k$ obtained by concatenating the first components of the edit operations of ω, and we have $x \in L(A)$. Using a similar argument for the string y obtained by concatenating the second components of ω, we can show that the computation X_ω yields an accepting computation of P on y.

($\Longleftarrow$) Let $\omega = (\omega_{L(1)} \to \omega_{R(1)})(\omega_{L(2)} \to \omega_{R(2)}) \cdots (\omega_{L(k)} \to \omega_{R(k)})$ be an alignment of length k for two strings $x = \omega_{L(1)}\omega_{L(2)} \cdots \omega_{L(k)} \in L(A)$ and $y = \omega_{R(1)}\omega_{R(2)} \cdots \omega_{R(k)} \in L(P)$. Let

$$C_0, C_1, \ldots, C_m, \quad m \in \mathbb{N}, \tag{2}$$

be a sequence of configurations of the PDA P that traces an accepting computation $\mathcal{C}_{P,y}$ on the input y. Assuming the state (respectively, topmost stack symbol) of C_i is q_i (respectively, γ_i), the configuration C_{i+1} is obtained from C_i by applying a rule $(q_{i+1}, \gamma_{i+1}) \in \delta_P(q_i, b, \gamma_i)$, where $b \in \Sigma \cup \{\lambda\}$.

Based on the computation $\mathcal{C}_{P,y}$ we want to construct a computation of $\mathcal{A}(A, P)$ on the edit-string ω. Roughly speaking, the second component of $\mathcal{A}(A, P)$ simulates the computation of the PDA P, however, this does not directly yield a computation of $\mathcal{A}(A, P)$ because some of the symbols $\omega_{R(j)}$ may be the empty string that would be ignored in a computation of P. For this reason we modify the computation $\mathcal{C}_{P,y}$ as follows.

Suppose that the computation step $C_i \to C_{i+1}$ consumes input symbol $\omega_{R(j)}$, and $\omega_{R(j+1)} = \cdots \omega_{R(j+h)} = \lambda$, $\omega_{R(j+h+1)} \neq \lambda$. Then in the sequence (2) we add, after the configuration C_i, h identical copies of C_i. We denote the modified configuration sequence as

$$D_0, D_1, \ldots, D_r, \quad r \in \mathbb{N}. \tag{3}$$

For any $0 \leq i \leq r-1$, either $D_{i+1} = D_i$ or D_{i+1} is obtained from D_i in one computation step of P.

From the sequence (3) and an accepting computation of A on x, we obtain a sequence of configurations of $\mathcal{A}(A, P)$ describing an accepting computation on ω by adding the first component states that simulate a finite-state computation of A. For the deletion operations ($\omega_{R(j)} = \lambda$) the state change is simulated just in the first component and the configuration of P remains unchanged. (These are the identical copies of P-configurations that were added in (3).) When $\mathcal{A}(A, P)$ processes an edit-symbol $(\omega_{L(j)} \to \omega_{R(j)})$ corresponding to a substitution operation, the computation step simulates both a state transition of A on $\omega_{L(j)}$ and the computation step of P on input $\omega_{R(j)}$. Finally, the transitions of $\mathcal{A}(A, P)$ on edit-symbols corresponding to an insertion symbol or the transition simulating a λ-move in the second component do not change the state of A appearing in the first component. This means that a sequence of states of A spelling out an accepting computation on x can be combined with the sequence (3) to yield an accepting computation of $\mathcal{A}(A, P)$ on ω. □

4.2. *Computing an optimal alignment from $\mathcal{A}(A, P)$*

Based on Lemma 3, we know that computing the edit-distance between a regular language $L(A)$ and a context-free language $L(P)$ can be solved by finding an optimal alignment in $L(\mathcal{A}(A, P))$. We tackle the problem of searching for an optimal alignment using the PDA $\mathcal{A}(A, P)$. The problem seems similar to the problem of finding the shortest string accepted by a PDA, however, it is not necessarily the case that a shortest string over Ω accepted by $\mathcal{A}(A, P)$ is an optimal alignment even under the Levenshtein distance. This is illustrated by the below example.

Example 2.

$$\begin{array}{cc} a\ b\ c\ \lambda & a\ b\ c \\ \downarrow\downarrow\downarrow\downarrow & \downarrow\downarrow\downarrow \\ \lambda\ b\ c\ d & b\ c\ d \\ \omega_X & \omega_Y\ . \end{array}$$

The two edit strings ω_X and ω_Y are alignments between abc and bcd. Under the Levenshtein distance, $\mathtt{C}(\omega_X) = 2$ and $\mathtt{C}(\omega_Y) = 3$ while the lengths of ω_X and ω_Y over Ω are four and three, respectively. Thus, the longer alignment string ω_X is a better alignment than the shorter alignment ω_Y.

We should consider the edit cost of each edit operation to find an optimal alignment. If we replace the zero cost edit operations ($(a \to a)$ for all $a \in \Sigma$) by λ in an edit-string, then in Example 2 the string ω_X becomes $\omega'_X = (a \to \lambda)(\lambda \to d)$, which is shorter than ω_Y. This leads us to the following observation.

Observation 1. *Let* $\mathtt{s}$ *be a morphism* $\Omega^* \to \Omega^*$ *defined by setting:*

$$\mathtt{s}(a \to b) = \begin{cases} \lambda & \text{if } a = b; \\ (a \to b) & \text{otherwise.} \end{cases}$$

Then $\omega \in \Omega^*$ is an optimal alignment in $L = L(\mathcal{A}(A, P))$ if and only if $\mathtt{s}(\omega)$ is a shortest string in $\mathtt{s}(L)$.

Observation 1 shows that the problem of finding an optimal alignment in $L(\mathcal{A}(A, P))$ becomes the problem of identifying a shortest string after the substitution $\mathtt{s}$.

For an FA A with m_1 states and n_1 transitions, we can find the shortest string in $L(A)$ by computing the shortest path from the start state to a final state based on the single-source shortest-path algorithm in $O((n_1+m_1)\log m_1)$ time [12]. However, we cannot obtain the shortest string accepted by a PDA similarly because of the stack operations. Therefore, it turns out to be useful to convert the PDA into a CFG and compute a shortest string generated by the grammar. We also, first, convert $\mathcal{A}(A, P)$ to an equivalent CFG and, then, obtain an optimal alignment from the resulting grammar. Note that if we apply the substitution function $\mathtt{s}$ in Observation 1 directly to the transitions of $\mathcal{A}(A, P)$, then the problem becomes to find a shortest string in $\mathtt{s}(L(\mathcal{A}(A, P)))$. However, since the $\mathtt{s}$ function replaces all

zero cost edit operations with λ, we cannot, in general, retrieve an optimal alignment between two strings. Instead, we only have the optimal edit cost. Therefore, the `s` function is useful for computing the edit-distance only and not for finding an optimal alignment. We revisit the problem of efficiently computing the edit-distance only in Sec. 5. Here we focus on finding an optimal alignment and which, in the worst-case, need not be polynomial in length. First we recall the following.

Proposition 4 (Hopcroft and Ullman [5]).
Given a PDA $P = (Q, \Sigma, \Gamma, \delta, s, Z_0)$, the triple construction *computes an equivalent CFG* $G = (V, \Sigma, R, S)$, *where the set* V *of variables consists of the special symbol* S, *which is the start symbol, and all symbols of the form* $[pXq]$, *where* $p, q \in Q$ *and* $X \in \Gamma$. *The productions of* G *are as follows:*

(1) *For all states* p, G *has the production* $S \to [sZ_0p]$ *and*
(2) *Let* $\delta(q, a, X)$ *contain the pair* $(r, Y_1Y_2 \cdots Y_k)$, *where*
   (a) a *is either a symbol in* Σ *or* $a = \lambda$.
   (b) k *can be any non-negative number, including zero, in which case the pair is* (r, λ).

   Then for all lists of states $r_1, r_2, \ldots, r_k$, G *has the production*

$$[qXr_k] \to a[rY_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k].$$

Note that G has $|Q|^2 \cdot |\Gamma| + 1$ variables and $|Q|^2 \cdot |\delta|$ productions. Now we examine how to compute an optimal alignment from the PDA $\mathcal{A}(A, P) = (Q_E, \Omega, \Gamma, \delta_E, s_E, Z_0, F_E)$ for an FA A and a PDA P, where $|Q_E| = m_1m_2$ and $|\delta_E| = n_1n_2$. Note that since we assume that each transition in P has at most two stack symbols, a transition in $\mathcal{A}(A, P)$ has also at most two stack symbols. Let $G_{\mathcal{A}(A,P)} = (V, \Sigma, R, S)$ be the CFG computed by the triple construction of Proposition 4. Then, $G_{\mathcal{A}(A,P)}$ has $O((m_1m_2)^2 \cdot |\Gamma|)$ variables and $O((m_1m_2)^2 \cdot (n_1n_2))$ productions. Moreover, each production of $G_{\mathcal{A}(A,P)}$ is of the form $A \to \sigma BC$, $A \to \sigma B$, $A \to \sigma$ or $A \to \lambda$, where $\sigma \in \Sigma$ and $B, C \in V$. We note that $G_{\mathcal{A}(A,P)}$ is similar to a Greibach normal form grammar but has λ-productions and each production has at most three symbols starting with a terminal symbol followed by variables in its right-hand side.

We run a preprocessing step before finding an optimal alignment, which speeds up the computation in practice by reducing the input size. This step eliminates nullable variables from $G_{\mathcal{A}(A,P)}$. The elimination of nullable variables is somehow similar to the elimination of λ-productions. The λ-production elimination removes all λ-productions from a CFG G and yields an equivalent (modulo the string λ) CFG G' without λ-productions [5]. However, at least when using the commonly known straightforward algorithm, the grammar G' may have exponentially more productions than the original grammar G [19]. We note that there exists also a more sophisticated algorithm for eliminating λ-productions that causes only a

linear increase in the size of the grammar [4].[a] For our current purpose, we notice that the productions added in the procedure eliminating λ-productions are, in fact, not needed to generate an optimal alignment in $\mathcal{A}(A, P)$. Thus, we can design a straightforward procedure that simply removes all nullable variables and their appearances in $\mathcal{A}(A, P)$ without adding new productions, and do not need to implement the sophisticated algorithm from [4].

The modified grammar still generates an optimal alignment between $L(A)$ and $L(P)$.

Procedure 1 Elimination of Nullable Variables (`ENV`)

Input: $G_{\mathcal{A}(A,P)} = (V, \Sigma, R, S)$
1: let V_N be a set of all nullable variables in $G_{\mathcal{A}(A,P)}$
2: **if** $S \in V_N$ **then**
3: $V = \{S\}$
4: $R = \{S \to \lambda\}$
5: **else**
6: **for** $B \in V_N$ **do**
7: remove all occurrences of B in R // replace B with λ
8: remove all productions of B from R
9: remove B from V
10: **end for**
11: **end if**

The `ENV` (Elimination of Nullable Variables) procedure simply eliminates nullable symbols and their occurrences from the grammar. Note that all productions of $G_{\mathcal{A}(A,P)}$ have as right side either λ or one terminal symbol over Ω followed by at most two variables. Thus the procedure `ENV` needs to scan through the variables only once to find nullable variables. The use of procedure `ENV` is illustrated in Example 3.

Example 3. *Given a grammar G with the following set R_1 of productions,*

$S \to AB \mid a$	$S \to B \mid a$
$A \to aAA \mid \lambda$	~~$A \to aAA \mid \lambda$~~
$B \to bBA \mid a$	$B \to bB \mid a$
R_1	R_2

we obtain R_2 after `ENV`*. Note that we only remove the nullable variable A and its appearances from G and do not increase the size of the grammar.*

Lemma 5. *Let $G_{\mathcal{A}(A,P)} = (V, \Omega, R, S)$ be a CFG generating the language of $\mathcal{A}(A, P)$ and let $G'_{\mathcal{A}(A,P)}$ be the grammar that the procedure* `ENV` *produces from*

[a] We thank one of the anonymous referees for pointing out this reference.

$G_{\mathcal{A}(A,P)}$. Then, the smallest cost of a string generated by $G'_{\mathcal{A}(A,P)}$ is the cost of an optimal alignment between $L(A)$ and $L(P)$.

Proof. Since $G'_{\mathcal{A}(A,P)}$ is obtained from $G_{\mathcal{A}(A,P)}$ by deleting some productions, the cost of any string generated by $G'_{\mathcal{A}(A,P)}$ cannot be less than the cost of an optimal alignment between $L(A)$ and $L(P)$. Thus, in order to prove the claim it is sufficient to show that $G_{\mathcal{A}(A,P)}$ has a derivation D producing an optimal alignment such that in D all occurrences of a nullable variable derive λ.

Let X be a nullable variable and consider a derivation of $G_{\mathcal{A}(A,P)}$ producing an optimal alignment $S \stackrel{*}{\Rightarrow} \alpha X \beta \stackrel{*}{\Rightarrow} w_\alpha w_X w_\beta$ where w_X (respectively, w_α, w_β) is the string generated from the symbol X (respectively, from the string α and β). Since X is nullable and the derivation is context-free, this means that $G_{\mathcal{A}(A,P)}$ generates also the string $w_\alpha w_\beta$ and we know that $\mathtt{C}(w_\alpha w_\beta) \leq \mathtt{C}(w_\alpha w_X w_\beta)$. □

Algorithm 2 Computing an optimal alignment in $L(G_{\mathcal{A}(A,P)})$

Input: $G_{\mathcal{A}(A,P)} = (V, \Omega, R, S)$

```
1: eliminate all nullable variables by ENV
2: for B → t ∈ R, where t ∈ Ω* and C(t) is minimum among all such t in R do
3:    if B = S then
4:       return t
5:    else
6:       replace all occurrences of B in R with t
7:       remove B from V and its productions from R
8:    end if
9: end for
```

Algorithm 2 describes how to find an optimal alignment in $G_{\mathcal{A}(A,P)}$. This algorithm is a modified version of the algorithm for finding the shortest string in a context-free grammar suggested by McLean and Johnston [11]. We first eliminate from $G_{\mathcal{A}(A,P)}$ nullable variables, which are not needed to derive an optimal alignment, as described in line 1 of Algorithm 2. In general, a procedure eliminating nullable variables takes quadratic time in the size of an input grammar. However, in our case all productions of $G_{\mathcal{A}(A,P)}$ have as right side either λ or one terminal symbol over Ω followed by at most two variables. This helps us to identify all nullable variables of $G_{\mathcal{A}(A,P)}$ by scanning R only once. It follows that the `ENV` procedure takes only linear time for $G_{\mathcal{A}(A,P)}$.

Once we have finished the `ENV` procedure, in the main part of the algorithm, we pick a variable that has an edit string with the smallest cost as a production, say $v \to t$, and replace all occurrences of v with t in R and remove v from V. We repeat this step until the smallest cost edit string is a production for the start symbol S. Lemma 6 guarantees that the algorithm produces an optimal alignment.

Lemma 6. *Let $G_{\mathcal{A}(A,P)} = (V, \Omega, R, S)$ be a CFG with no λ-productions. Let $B \to t \in R$ be a terminating production, where $\mathtt{C}(t)$ is minimal for all terminating productions in R. Let $G'_{\mathcal{A}(A,P)}$ be the grammar obtained from $G_{\mathcal{A}(A,P)}$ by removing all productions for B from R and replacing all occurrences of B by t. Then, the smallest cost of a terminal string generated by $G'_{\mathcal{A}(A,P)}$ is the same as the smallest cost of a terminal string generated by $G_{\mathcal{A}(A,P)}$.*

Proof. Clearly, $L(G'_{\mathcal{A}(A,P)}) \subseteq L(G_{\mathcal{A}(A,P)})$ and $G'_{\mathcal{A}(A,P)}$ cannot generate a terminal string whose cost is less than the smallest cost of a terminal string generated by $G_{\mathcal{A}(A,P)}$. Conversely, let w be a minimum cost terminal string generated by $G_{\mathcal{A}(A,P)}$. If a derivation of w does not use B, then the string w can be generated also by $G'_{\mathcal{A}(A,P)}$. Next, consider a derivation D of w that involves (one or more occurrences of) B. Let D' be the derivation obtained from D by replacing each derivation step using a production $B \to \alpha$ by a derivation step using the production $B \to t$ and let w' be the terminal string generated by D'.

Since $\mathtt{C}(t)$ is minimum among all right-hand sides of productions in R and $G_{\mathcal{A}(A,P)}$ does not have any λ-productions, for any string β that can be derived from α, we have $\mathtt{C}(t) \leq \mathtt{C}(\beta)$ and it follows that $\mathtt{C}(w') \leq \mathtt{C}(w)$. □

We notice that the length of an optimal alignment can be exponential in the size of an input grammar as shown in Example 4. In Example 4, once we eliminate one variable v and update G by the single **for** loop in Algorithm 2, the length of an edit string with the smallest cost is doubled.

Example 4. *Consider a CFG $G = (S, A_1, \ldots A_n\}, \{(a \to b)\}, R, S)$, where R has the productions*

$$\begin{array}{lcl} S & \to & A_1 A_1 \\ A_1 & \to & A_2 A_2 \\ & \vdots & \\ A_{n-1} & \to & A_n A_n \\ A_n & \to & (a \to b)\,. \end{array}$$

The language of G is $\{(a \to b)^{2^n}\}$ and $|G| = O(n)$.

Now we consider the cost for replacing the occurrences of variables in Algorithm 2. Since the grammar $G_{\mathcal{A}(A,P)}$ has no λ-productions, the length of an edit string with the smallest cost is one. Note that a production can have at most one terminal followed by two variables. Therefore, when the algorithm replaces the tth variable, we have an edit string of length at most $2^t - 1$. Next, we consider the number of variable occurrences that are eventually replaced with an edit string. Since there are at most $2|R|$ occurrences of variables in R and $|V|$ variables, we replace $\frac{2|R|}{|V|}$ occurrences of a given variable on average. Therefore, the worst-case

time complexity for finding an optimal alignment is

$$\sum_{t=1}^{|V|}\left(|R| + (2^t - 1)\cdot\frac{2|R|}{|V|}\right) = O\left(\frac{2|R|}{|V|}2^{|V|}\right).$$

We note that using the notations (1) for the grammar $G_{\mathcal{A}(A,P)}$, $|V| = O((m_1m_2)^2 \cdot |\Gamma|)$ and $|R| = O((m_1m_2)^2 \cdot (n_1n_2))$, and we establish the time complexity of Algorithm 2 with respect to m_1, m_2, n_1 and n_2 to be:

$$O\left((m_1m_2)^4 \cdot |\Gamma| \cdot (n_1n_2) + \frac{n_1n_2}{|\Gamma|}\cdot 2^{(m_1m_2)^2\cdot|\Gamma|}\right) = O\left(\frac{n_1n_2}{|\Gamma|}\cdot 2^{(m_1m_2)^2\cdot|\Gamma|}\right). \quad (4)$$

We have established the following:

Theorem 7. *Given a PDA $P = (Q_P, \Sigma, \Gamma, \delta_P, s_P, Z_0, F_P)$ and an FA $A = (Q_A, \Sigma, \delta_A, s_A, F_A)$, we can compute the edit-distance between $L(A)$ and $L(P)$ in $O((n_1n_2)\cdot 2^{(m_1m_2)^2\cdot|\Gamma|})$ worst-case time. Here $m_1 = |Q_A|, m_2 = |Q_P|, n_1 = |\delta_A|$ and $n_2 = |\delta_P|$. Moreover, we can also identity two strings $x \in L(A)$ and $y \in L(P)$ and their alignment with each other in the same runtime.*

5. Edit-Distance and Unary Homomorphism

In the previous section, we have designed an algorithm that computes both the edit-distance and an optimal alignment between a regular language and a context-free language. From Theorem 7, we note that running time of the algorithm may be exponential. Furthermore, the exponential running time seems hard to avoid since it is known from Example 4 that the length of an optimal alignment could be exponential in the size of input the FA and PDA. In this section we examine how to calculate the edit-distance without explicitly computing the corresponding optimal alignment and present a polynomial runtime algorithm for finding the edit-distance between a regular and a context-free language.

Let $\Sigma_U = \{u\}$ be a unary alphabet. We often use non-negative integers $\mathbb{Z}_+$ for the cost function associated with the edit-distance. For example, the Levenshtein distance [10] uses one for all operation costs. From now on, we assume that the cost function is defined over $\mathbb{Z}_+$.

We apply a unary homomorphism to the alignment PDA $\mathcal{A}(A, P)$ obtained from an FA A and a PDA P, and construct a CFG for the resulting unary language. Consider an alphabet $\Omega = \{(a \to b) \mid a, b \in \Sigma \cup \{\lambda\}\}$ of edit operations and let $c_{\mathtt{I}}, c_{\mathtt{D}}$ and $c_{\mathtt{S}}$ be the costs of insertion, deletion and substitution, respectively. Then, we define a morphism $\mathcal{H} : \Omega^* \to \Sigma_U^*$ by setting:

$$\begin{aligned}
&\mathcal{H}(\lambda \to a) = u^{\mathtt{C_I}((\lambda\to a))}, && \text{[insertion]}\\
&\mathcal{H}(a \to \lambda) = u^{\mathtt{C_D}((a\to\lambda))}, && \text{[deletion]}\\
&\mathcal{H}(a \to b) = \begin{cases} u^{\mathtt{C_S}((a\to b))}, & \text{if } a \neq b;\\ \lambda, & \text{if } a = b. \end{cases} && \text{[substitution]}\,.
\end{aligned}$$

Clearly for any alignment $\omega \in \Omega^*$, $\mathtt{C}(\omega) = |\mathcal{H}(\omega)|$, that is, the length of $\mathcal{H}(\omega)$ gives the cost of ω.

By Lemma 3, we know that $\mathcal{A}(A, P)$ accepts all edit strings (alignments) between two strings $x \in L(A)$ and $y \in L(P)$. Note that the cost of an optimal alignment is the edit-distance between $L(A)$ and $L(P)$. We apply the homomorphism $\mathcal{H}$ to $\mathcal{A}(A, P)$ by replacing an edit string ω with a unary string u^i, where $i = \mathtt{C}(\omega)$. In this step, we can reduce the number of transitions in $\mathcal{A}(A, P)$ by applying the homomorphism. For example, when there are multiple transitions like $\delta_E(q_E, (a \to b), M) = (q'_E, M')$, where $(a \to b) \in \Omega$, the unary homomorphism results in only one transition in the resulting PDA $\mathcal{A}(A, P)$, say $\mathcal{H}(\mathcal{A}(A, P))$. Since the number of productions in the grammar $G_{\mathcal{H}(\mathcal{A}(A,P))}$ produced by the triple construction is proportional to the number of transitions in $\mathcal{H}(\mathcal{A}(A, P))$, we can reduce the size of the grammar $G_{\mathcal{H}(\mathcal{A}(A,P))}$, compared with $G_{\mathcal{A}(A,P)}$. Then, an optimal alignment in $L(G_{\mathcal{A}(A,P)})$ becomes the shortest string in $L(G_{\mathcal{H}(\mathcal{A}(A,P))})$ and its length is the edit-distance between $L(A)$ and $L(P)$. We establish the following statement.

Corollary 8. *The edit-distance $d(L(A), L(P))$ of an FA A and a PDA P is the length of the shortest string in $L(G_{\mathcal{H}(\mathcal{A}(A,P))})$.*

Corollary 8 establishes that in order to solve the edit-distance problem it is sufficient to find the shortest string in $L(G_{\mathcal{H}(\mathcal{A}(A,P))})$. Before searching for the shortest string, we run a preprocessing step that eliminates λ-productions from $G_{\mathcal{H}(\mathcal{A}(A,P))}$. Lemma 9 guarantees that the preprocessing does not change the length of the shortest strings generated by the grammar.

Lemma 9. *Given a CFG $G = (V, \Sigma, R, S)$, let G' be a CFG constructed from G by eliminating all nullable variables and their occurrences except for the start symbol. If the start symbol of G is nullable, the grammar G' is chosen to have only the production $S \to \lambda$. Then, a shortest string in $L(G')$ is also a shortest string in $L(G)$.*

Proof. Since G' is obtained from G by eliminating nullable variables, any string generated by G' is generated also by G. Conversely, relying on the context-freeness of the derivations exactly as in the proof of Lemma 5, it is seen that G' generates a string with length equal to the length of the shortest string in $L(G)$. (In fact, G' generates all strings of $L(G)$ having minimal length.) □

Algorithm 3 computes the length of the shortest string in $L(G_{\mathcal{H}(\mathcal{A}(A,P))})$. The algorithm is based on similar ideas as Algorithm 2, however, we significantly improve the running time by using a binary encoding for the unary strings. The complexity of Algorithm 2 is exponential since the length of the shortest string (or optimal alignment) can be exponential. On the other hand, now we are looking only for the length of the shortest string (which gives the edit distance) instead of the actual optimal alignment, and we can encode string lengths in binary. For example, we

Algorithm 3 Computing the length of the shortest string in $L(G_{\mathcal{H}(\mathcal{A}(A,P))})$

Input: $G_{\mathcal{H}(\mathcal{A}(A,P))} = (V, \Sigma_U, R, S)$

1: eliminate all nullable variables by `ENV`
2: encode all right-hand productions by the number of u occurrences in binary representation followed by the remaining variables in order
// e.g. from $A \to uuuBCuu$ to $A \to 101BC$ and now $\Sigma_U = \{0, 1\}$ instead of $\{u\}$
3: **for** $A \to t \in R$, where t is the smallest binary number in R **do**
4: **if** $A = S$ **then**
5: **return** t
6: **else**
7: **for** each production $B \to wxAy$ in R, where w is the binary number part and $x, y \in V^*$ **do**
8: $w' = w + t$ in binary representation
9: update the production as $B \to w'xy$
10: **end for**
11: remove A from V and all A's productions from R
12: **end if**
13: **end for**

use 100000 to denote u^{32}. We need only $O(\log n)$ space to denote a unary string of length n, and this significantly improves also the running time of the algorithm.

Now we consider the complexity of Algorithm 3. In the worst-case, we need to eliminate all variables in V. It takes $O(|R|)$ time to eliminate a variable from G since we need to scan the whole grammar in the worst-case. We need $O((m_1m_2)^4 \cdot (n_1n_2) \cdot |\Gamma|)$ time to remove variables in V in the worst-case since $|V| = (m_1m_2)^2 \cdot |\Gamma|$ and $|R| = O((m_1m_2)^2 \cdot (n_1n_2))$. (Our notations are as in (1).) Next, we consider the time needed for replacing the occurrences of variables with encoded numbers in binary. We should replace all occurrences of variables in the worst-case. The number of occurrences is at most $O((m_1m_2)^2 \cdot (n_1n_2))$ and the size of binary numbers is at most $O((m_1m_2)^2 \cdot |\Gamma|)$. Then, the time needed is again upper bounded by $O((m_1m_2)^4 \cdot (n_1n_2) \cdot |\Gamma|)$. Thus, the worst-case time complexity of Algorithm 3 is $O((m_1m_2)^4 \cdot (n_1n_2) \cdot |\Gamma|)$. Assuming the stack alphabet Γ is fixed, we have the following result.

Theorem 10. *Given a PDA* $P = (Q_P, \Sigma, \Gamma, \delta_P, s_P, Z_0, F_P)$ *and an FA* $A = (Q_A, \Sigma, \delta_A, s_A, F_A)$*, we can compute the edit-distance between* $L(A)$ *and* $L(P)$ *in* $O((m_1m_2)^4 \cdot (n_1n_2))$ *worst-case time, where* $m_1 = |Q_A|, m_2 = |Q_P|, n_1 = |\delta_A|$ *and* $n_2 = |\delta_P|$.

6. Conclusion

Computing the edit-distance between two languages involves finding a pair of strings, each of which is from a different language, with the minimum edit-distance. The edit-distance problem between, respectively, two regular languages or between two context-free languages has been well-studied in the literature [2, 7, 13]. We have considered the "intermediate" problem of finding the edit distance between a regular language and a context-free language. We have proposed an algorithm that finds an optimal alignment between a regular language and a context-free language, however, since the length of an optimal alignment can be exponential in the size of the input (the size of the given FA and PDA), the runtime of the algorithm is necessarily exponential. This also led us to consider the problem of computing only the edit-distance, as opposed to identifying the corresponding string of edit operations, and we have given a polynomial runtime algorithm for the latter problem. Note that the second algorithm only calculates the minimum edit cost between strings in the two languages and does not explicitly yield an optimal alignment.

Acknowledgments

We wish to thank the referees for the careful reading of the paper and many valuable suggestions including the previous results by Ésik and Iván [4], and by McLean and Johnston [11]. As usual, however, we alone are responsible for any remaining sins of omission and commission.

Han and Ko were supported by the Basic Science Research Program through NRF funded by MEST (2010-0009168) and Salomaa was supported by the Natural Sciences and Engineering Research Council of Canada Grant OGP0147224.

References

[1] C. Allauzen and M. Mohri. Linear-space computation of the edit-distance between a string and a finite automaton. In *London Algorithmics* 2008*: Theory and Practice*. College Publications, 2009.

[2] H. Bunke. Edit distance of regular languages. In *Proceedings of* 5*th Annual Symposium on Document Analysis and Information Retrieval*, 113–124, 1996.

[3] C. Choffrut and G. Pighizzini. Distances between languages and reflexivity of relations. *Theoretical Computer Science*, 286(1):117–138, 2002.

[4] Z. Ésik and S. Iván. Büchi context-free languages. *Theoretical Computer Science*, 412(8-10):805–821, 2011.

[5] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 2nd edition, 1979.

[6] L. Kari and S. Konstantinidis. Descriptional complexity of error/edit systems. *Journal of Automata, Languages and Combinatorics*, 9(2-3):293–309, 2004.

[7] S. Konstantinidis. Computing the edit distance of a regular language. *Information and Computation*, 205:1307–1316, 2007.

[8] S. Konstantinidis and P. V. Silva. Maximal error-detecting capabilities of formal languages. *Journal of Automata, Languages and Combinatorics*, 13(1):55–71, 2008.

[9] S. Konstantinidis and P. V. Silva. Computing maximal error-detecting capabilities and distances of regular languages. *Fundamenta Informaticae*, 101(4):257–270, 2010.

[10] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[11] M. J. McLean and D. B. Johnston. An algorithm for finding the shortest terminal strings which can be produced from non-terminals in context-free grammars. In *Combinatorial Mathematics III*, volume 452 of *Lecture Notes in Mathematics*, 180–196. 1975.

[12] M. Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7:321–350, 2002.

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

[14] P. A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach* (*Computational Molecular Biology*). The MIT Press, 2000.

[15] G. Pighizzini. How hard is computing the edit distance? *Information and Computation*, 165(1):1–13, 2001.

[16] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.

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

[18] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21:168–173, 1974.

[19] D. Wood. *Theory of Computation*. Harper & Row, 1987.