Computing the Edit-Distance between a Regular Language and a Context-Free Language[*]

Yo-Sub Han[1], Sang-Ki Ko[1], and Kai Salomaa[2]

[1] Department of Computer Science, Yonsei University
{emmous,narame7}@cs.yonsei.ac.kr
[2] School of Computing, Queen's University
ksalomaa@cs.queensu.ca

Abstract. The edit-distance between two strings is the smallest number of operations required to transform one string into the other. The edit-distance problem for two languages is to find a pair of strings, each of which is from different language, with the minimum edit-distance. We consider the edit-distance problem for a regular language and a context-free language and present an efficient algorithm that finds an optimal alignment of two strings, each of which is from different language. Moreover, we design a faster algorithm for the edit-distance problem that only finds the minimum number of operations of the optimal alignment.

Keywords: Edit-distance, Levenshtein distance, Regular language, Context-free language.

1 Introduction

The edit-distance between two strings is the smallest number of operations required to transform one string into the other [7]. We can use the edit-distance as a similarity measure between two strings; the shorter distance implies that the two strings are more similar. We can compute this by using the bottom-up dynamic programming algorithm [14]. The edit-distance problem arises in many areas such as computational biology, text processing and speech recognition [9,10,12]. This problem can be extended to measure the similarity between languages [3,6,9].

For instance, the error-correction problem is based on the edit-distance problem: Given a set S of correct strings and an input string x, we find the most similar string $y \in S$ to x using the edit-distance computation. If $y = x \in S$, we say that x has no error. We compute the edit-distance between all strings in S and x. However, we can also use a finite-state automaton (FA) for S, which is finite, and obtain the most similar string in S with respect to x [13]. Allauzen and Mohri [1] designed a linear-space algorithm that computes the edit-distance

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

between a string and an FA. Pighizzini [11] considered the case when the language is not regular. The error-detection capability problem is related to the self-distance of a language L [6]. The self-distance or inner distance is the minimum edit-distance between any pair of distinct strings in L. We can use the minimum edit-distance as the maximum number of errors that L (code) can identify.

We examine the problem of computing the edit-distance between a regular language and a context-free language. This was an open problem and the edit-distance problem between two context-free languages is already known as undecidable [9]. We rely on the structural properties of FAs and pushdown automata for both languages and design an efficient algorithm that finds the edit-distance.

In Section 2, we define some basic notions. We formally define the edit-distance and the edit-distance problem in Section 3. Then, we present an efficient algorithm for computing the edit-distance and the optimal alignments between a context-free language and a regular language in Section 4. We also present a faster algorithm that only computes the optimal cost based on the unary homomorphism in Section 5.

2 Preliminaries

Let Σ denote a finite alphabet of characters and Σ^* denote the set of all strings over Σ. The size $|\Sigma|$ of Σ is the number of characters in Σ. A language over Σ is any subset of Σ^*. Given a set X, 2^X denotes the power set of X.

The symbol $\emptyset$ denotes the empty language and the symbol λ denotes the null string. A finite-state automaton (FA) A is specified by a tuple $(Q, \Sigma, \delta, s, F)$, 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. For a transition $q \in \delta(p, a)$ in A, we say that p has an *out-transition* and q has an *in-transition*. Furthermore, p is a *source state* of q and q is a *target state* of p. The transition function δ can be extended to a function $Q \times \Sigma^* \to 2^Q$ that reflects sequences of inputs. A string x over Σ 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 $L(A)$ of an FA A is the set of all strings that are spelled out by paths from s to a final state in F.

A pushdown automaton (PDA) P is specified by a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where Q is a finite set of states, Σ is a finite set of input symbols, Γ is a finite stack alphabet, $\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to 2^{Q \times \Gamma^*}$ is the transition function, $q_0 \in Q$ is the start state, Z_0 is the initial stack symbol and $F \subseteq Q$ is a set of final states. We use $|Q|$ to denote the number of states in Q and $|\delta|$ to denote the number of transitions in δ. Here, we assume that each transition in P has at most two stack symbols; namely, each transition can push or pop at most one symbol. In other words, when some symbol X is on the top of the stack, then either λ or a string of the form YX for some stack symbol Y can stand on the right side of the production. Then, the size $|P|$ of P is $|Q| + |\delta|$.

A context-free grammar G is specified by a tuple $G = (V, \Sigma, R, S)$, where V is a set of variables, $R \subseteq V \times (V \cup \Sigma)^*$ is a finite set of productions and $S \in V$ is the start symbol. Let $\alpha A \beta$ be a string over $V \cup \Sigma$ with A a variable and $A \rightarrow \gamma$ be a production of G. Then, we say that $\alpha A \beta \Rightarrow \alpha \gamma \beta$. The reflexive, transitive closure of $\Rightarrow$ is $\overset{*}{\Rightarrow}$. Then the context-free language defined by G is $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$. We say that a variable $A \in V$ is *nullable* if $A \overset{*}{\Rightarrow} \lambda$.

For complete background knowledge in automata theory, the reader may refer to textbooks [4,15].

3 Edit-Distance

The edit-distance between two strings is the smallest number of operations that transform a string to the other. People use different edit operations depending on the applications. We consider three operations, insertion, deletion and substitution for simplicity. Given an alphabet Σ, let

$$\Omega = \{(a \rightarrow b) \mid a, b \in \Sigma \cup \{\lambda\}\}$$

be a set of edit operations. Namely, Ω is an alphabet of all edit operations for *deletions* $(a \rightarrow \lambda)$, *insertions* $(\lambda \rightarrow a)$ and *substitutions* $(a \rightarrow b)$. We call a string $w \in \Omega^*$ an *edit string* [5] or an *alignment* [9].

Let the morphism h between Ω^* and $\Sigma^* \times \Sigma^*$ be

$$h((a_1 \rightarrow b_1) \cdots (a_n \rightarrow b_n)) = (a_1 \cdots a_n, b_1 \cdots b_n).$$

Example 1. *The following is an alignment example* $w = (a \rightarrow \lambda)(b \rightarrow b)(\lambda \rightarrow c)(c \rightarrow c)$ *for* abc *and* bcc*. Note that* $h(w) = (abc, bcc)$*.*

$$\begin{array}{cccc} a & b & \lambda & c \\ \downarrow & \downarrow & \downarrow & \downarrow \\ \lambda & b & c & c \end{array}$$

Definition 1. *An edit string* w *is a sequence of edit-operations transforming a string* x *into a string* y*, also called an alignment for* x *and* y *if and only if* $h(w) = (x, y)$*.*

We associate a non-negative edit cost to each edit operation $w_i \in \Omega$ as a function $\mathtt{C} : \Omega \rightarrow \mathbb{R}_+$. We can extend the function to the cost $\mathtt{C}(w)$ of an alignment $w = w_1 \cdots w_n$ as follows:

$$\mathtt{C}(w) = \sum_{i=1}^{n} \mathtt{C}(w_i).$$

Definition 2. *The* edit-distance $d(x, y)$ *of two strings* x *and* y *over* Σ *is the minimal cost of an alignment* w *between* x *and* y*:*

$$d(x, y) = \min\{\mathtt{C}(w) \mid h(w) = (x, y)\}.$$

We say that w is *optimal* if $d(x, y) = \mathtt{C}(w)$.

We can extend the edit-distance definition to languages.

Definition 3. *The* edit-distance $d(L, R)$ *between two languages* $L, R \subseteq \Sigma^*$ *is the minimum edit-distance of two strings, one is from* L *and the other is from* R*:*

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

Konstantinidis [6] considered the edit-distance within a regular language L and proposed a polynomial runtime algorithm. Mohri [9] studied the edit-distance of two string distributions given by two weighted automata. Mohri [9] also proved that the edit-distance problem is undecidable for two context-free languages. We consider the case in between: L is regular and R is context-free. In other words, given an FA A and a PDA P, we develop an algorithm that computes the edit-distance of two languages $L(A)$ and $L(P)$.

Since we use the Levenshtein distance [7] for edit-distance, we assign one 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 $a \neq b \in \Sigma$.

4 The Edit-Distance between an RL and a CFL

We present algorithms that compute the edit-distance $d(R, L)$ between a regular language R and a context-free language L and find an optimal alignment w such that $\mathtt{C}(w) = d(R, L)$.

Let $A = (Q_A, \Sigma, \delta_A, s_A, F_A)$ be an FA for R and $P = (Q_P, \Sigma, \Gamma, \delta_P, s_P, Z_0, F_P)$ be a PDA for L. Let $m_1 = |Q_A|, m_2 = |Q_P|, n_1 = |\delta_A|$ and $n_2 = |\delta_P|$. We assume that A has no λ-transitions. We also assume that each transition in P has at most two stack symbols; namely, each transition can push or pop at most one symbol. Note that any context-free language can be recognized by a PDA that pushes or pops at most one symbol in one transition [4].

We first construct a new PDA $\mathcal{A}(A, P)$ (called *alignment PDA*) whose transitions denote all possible edit operations of all pairs of strings between R and L. Then, we compute the shortest string accepted by the alignment PDA, which is the optimal alignment.

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 a set of states,
- $\Omega = \{(a \to b) \mid a, b \in \Sigma \cup \{\lambda\}\}$ is an alphabet of edit operations,
- $s_E = (s_A, s_P)$ is the start state,
- $F_E = F_A \times F_P$ is a set of final states.

The transition function δ_E consists of three types of transitions, each of which performs *deletion*, *insertion* and *substitution*, respectively.

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 be

- $((p', q), N) \in \delta_E((p, q), (a \to \lambda), N)$, [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]
- $((p, q'), M') \in \delta_E((p, q), (\lambda \to \lambda), M)$.

The last type of transitions simulate λ-moves of the original PDA P. 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 the complexity of δ_E, we generate n_1m_2 transitions for deletions and n_2m_1 transitions for insertions. For substitutions, we consider all pairs of transitions between A and P and, thus, add n_1n_2 transitions. Therefore, the size of δ_E is

$$|\delta_E| = n_1m_2 + n_2m_1 + n_1n_2 = O(n_1n_2).$$

Theorem 1. *The alignment PDA $\mathcal{A}(A, P)$ accepts an edit string w if and only if $h(w) = (x, y)$, where $x \in L(A)$ and $y \in L(P)$.*

It follows from Theorem 1 that the edit-distance problem is now to find an optimal alignment in $L(\mathcal{A}(A, P))$. In the next section, we discuss how to find an optimal alignment from an alignment PDA efficiently.

4.2 Computing an Optimal Alignment from $\mathcal{A}(A, P)$

An optimal alignment w between two languages is an alignment with the minimum cost among all possible alignments between any pair of strings from each language. We tackle the problem of searching for an optimal alignment from $\mathcal{A}(A, P)$. The problem seems similar to the problem of finding the shortest string in a PDA. However, it is not necessarily true that a shortest string over Ω in $\mathcal{A}(A, P)$ is an optimal alignment even under the Levenshtein distance. See Example 2.

Example 2

$$\begin{array}{cccc} a & b & c & \lambda \\ \downarrow & \downarrow & \downarrow & \downarrow \\ \lambda & b & c & d \\ & & & \\ & w_X & & \end{array} \qquad \begin{array}{ccc} a & b & c \\ \downarrow & \downarrow & \downarrow \\ b & c & d \\ & & \\ & w_Y & \end{array}$$

The two edit strings w_X and w_Y are alignments between abc and bcd. Under the Levenshtein distance, $\mathtt{C}(w_X) = 2$ and $\mathtt{C}(w_Y) = 3$ while the lengths of w_X and w_Y over Ω are four and three, respectively. Namely, the longer alignment string w_X is a better alignment than the shorter alignment string w. Therefore, the shortest string from $\mathcal{A}(A, P)$ is not necessarily an optimal alignment between $L(A)$ and $L(P)$.

As shown in Example 2, we should consider the edit cost of each edit operation to find an optimal alignment. If we regard the zero cost edit operations ($(a \to a)$ for all $a \in \Sigma$) as λ in w, then $w'_X = (a \to \lambda)(\lambda \to d)$, which is shorter than w_Y. This leads us to the following observation.

Observation 1. *Let* $\mathtt{s}$ *be a substitution of* $\Omega^* \to \Omega^*$ *as follows:*

$$\mathtt{s}(a \to b) = \begin{cases} \lambda & a = b, \\ (a \to b) & \textit{otherwise.} \end{cases}$$

An optimal alignment $w \in \Omega^*$ *in* $L(\mathcal{A}(A, P))$ *is a shortest string in* $\mathtt{s}(L(\mathcal{A}(A, P)))$*.*

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

For an FA A with m_1 states and n_1 transitions, we can find the shortest string that A accepts 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 [8]. However, we cannot obtain the shortest string from a PDA P directly as we have done for an FA before because of the stack operations. Therefore, instead of computing a shortest path in P, we convert P into a context-free grammar and compute a shortest string from the grammar. Recently, Alpoget et al. [2] solved the emptiness test of a PDA by converting a PDA to an equivalent CFG using the standard construction in Proposition 1. We also, first, convert $\mathcal{A}(A, P)$ to an equivalent CFG and, then, obtain an optimal alignment from the grammar. Note that if we apply the substitution function $\mathtt{s}$ in Observation 1 directly on 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 retrieve an optimal alignment between two strings. Instead, we only have the optimal edit cost. Therefore, the $\mathtt{s}$ function is useful for computing the edit-distance only. We revisit the problem of computing the edit-distance in Section 5. Here we focus on finding an optimal alignment.

Given an alignment PDA $\mathcal{A}(A, P)$, let $G_{\mathcal{A}(A,P)}$ be the corresponding CFG that we compute using the following standard construction [4].

Proposition 1 (Hopcroft and Ullman [4]). *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*

1. *The special symbol* S*, which is the start symbol.*
2. *All symbols of the form* $[pXq]$*, where* $p, q \in Q$ *and* $X \in \Gamma$*. The productions of* G *are as follows:*
   (a) *For all states* p*,* G *has the production* $S \to [sZ_0p]$ *and*
   (b) *Let* $\delta(q, a, X)$ *contain the pair* $(r, Y_1Y_2 \cdots Y_k)$*, where*
      i. a *is either a symbol in* Σ *or* $a = \lambda$*.*
      ii. 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 $|Q|^2 \cdot |\delta|$ production rules. Now we study how to compute an optimal alignment in the alignment 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 $|Q_E| = m_1 m_2$ and $|\delta_E| = n_1 n_2$. 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. Then, $G_{\mathcal{A}(A,P)}$ has $O((m_1 m_2)^2 \cdot |\Gamma|)$ variables and $O((m_1 m_2)^2 \cdot (n_1 n_2))$ production rules. Moreover, each product rule is in the form of $A \to \sigma BC$, $A \to \sigma B$, $A \to \sigma$ or $A \to \lambda$, where $\sigma \in \Sigma$ and $B, C \in V$. Remark that $G_{\mathcal{A}(A,P)}$ is similar to a Greibach normal form grammar but has λ-productions and each production rule 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 from $\mathcal{A}(A, P)$, which speeds up the computation in practice by reducing the size of an input. This step eliminates nullable variables from $G_{\mathcal{A}(A,P)}$. The elimination of nullable variables is similar to the elimination of λ-productions. The λ-production elimination is to remove all λ-productions from a CFG G and obtain a new CFG G' without λ-productions where $L(G) \setminus \{\lambda\} = L(G')$ [4]. However, this procedure may introduce new productions in G'. We notice that the new productions generated from removing λ-productions do not help to find an optimal alignment in $\mathcal{A}(A, P)$ and, thus, design a procedure that removes all nullable variables and their appearances in $\mathcal{A}(A, P)$ without adding new production rules. Note that the modified grammar is not equivalent to the original grammar, however, as will be seen in Lemma 1, the modified grammar generates an optimal alignment between $L(A)$ and $L(P)$.

Procedure 1. Elimination of Nullable Variable (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 $\lambda$
 8:         remove all productions of $B$ from $R$
 9:         remove $B$ from $V$
10:     end for
11: end if
```

The ENV (Elimination of Nullable Variable) procedure just eliminates nullable symbols and their occurrences from the grammar. Example 3 gives an example of ENV.

Example 3. *Given a grammar G with the following set P_1 of production rules,*

P_1	P_2
$S \to AB \mid a$	$S \to B \mid a$
$A \to aAA \mid \lambda$	~~$A \to aAA \mid \lambda$~~
$B \to bBA \mid a$	$B \to bB \mid a$

we obtain P_2 after `ENV`*. Note that we only remove nullable variable A and its appearances from G and do not increase the size of G.*

The following statement guarantees that `ENV` preserves the optimal alignment of $L(\mathcal{A}(A,P))$.

Lemma 1. *Given a context-free grammar $G_{\mathcal{A}(A,P)} = (V, \Omega, R, S)$, let $G'_{\mathcal{A}(A,P)}$ be the resulting CFG from $G_{\mathcal{A}(A,P)}$ by* `ENV`*. Then, $G'_{\mathcal{A}(A,P)}$ still produces an optimal alignment between $L(A)$ and $L(P)$.*

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 $B \to t \in R$, where $t \in \Omega^*$ and $\texttt{C}(t)$ 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 $G_{\mathcal{A}(A,P)}$. We first eliminate nullable variables, which do not derive an optimal alignment, from $G_{\mathcal{A}(A,P)}$ as described in line 1 in Algorithm 2. The `ENV` procedure generally takes quadratic time in the size of an input grammar. For $G_{\mathcal{A}(A,P)}$, all production rules in $G_{\mathcal{A}(A,P)}$ have at either λ or one terminal symbol over Ω followed by at most two variables. Therefore, we can identify all nullable variables of $G_{\mathcal{A}(A,P)}$ by scanning R only once. (Only a variable that has a λ-production in its production rule is nullable variable in $G_{\mathcal{A}(A,P)}$.) Thus, the `ENV` procedure takes linear time for $G_{\mathcal{A}(A,P)}$.

Lemma 2. *Let $G_{\mathcal{A}(A,P)} = (V, \Omega, R, S)$ be a context-free grammar with no λ-productions. Let $B \to t$ be a terminating production where $B \in V$, $t \in \Omega^*$ and $\texttt{C}(t)$ is minimal among all right sides of terminating productions of $G_{\mathcal{A}(A,P)}$. 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 in right sides of productions by t. Then the smallest cost terminal string generated by G' has the same cost as the smallest cost terminal string generated by $G_{\mathcal{A}(A,P)}$.*

Once we have finished the ENV procedure, in the main part, 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 start symbol S of $G_{\mathcal{A}(A,P)}$ has an edit string as its production rule. We notice that the length of the optimal alignment can be exponential in the size of an input grammar as shown in Example 4.

Example 4. *Given a CFG $G = (S, A_1, \ldots A_n\}, \{(a \to b)\}, R, S)$, where R is*

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

G generates $(a \to b)^{2^n}$, where $|G| = O(n)$.

In Example 4, once we eliminate one variable v and update the grammar by the single **for** loop in Algorithm 2, the length of an edit string with the smallest cost can be doubled. Now we consider the cost for replacing the occurrences of variables. Since there are no λ-productions, the length of an edit string with the smallest cost starts from one. Note that a production rule can have at most one terminal followed by two variables. Therefore, we have an edit string of length at most $2^t - 1$. Next, we consider the average number of variable occurrences that are eventually replaced with the edit string. Since there are at most $2|R|$ occurrences of variables in the production rules and $|V|$ variables, we replace $\frac{2|R|}{|V|}$ occurrences on average.

Now, the worst-case time complexity for finding an optimal alignment is

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

where $|R|$ is the number of production rules and $|V|$ is the number of variables. Since $|V| = O((m_1 m_2)^2 \cdot |\Gamma|)$ and $|R| = O((m_1 m_2)^2 \cdot (n_1 n_2))$ in $G_{\mathcal{A}(A,P)}$, we establish the time complexity of Algorithm 2 with respect to m_1, m_2, n_1 and n_2 as follows:

$$O((m_1 m_2)^4 \cdot |\Gamma| \cdot (n_1 n_2) + \frac{n_1 n_2}{|\Gamma|} \cdot 2^{(m_1 m_2)^2 \cdot |\Gamma|}) = O(\frac{n_1 n_2}{|\Gamma|} \cdot 2^{(m_1 m_2)^2 \cdot |\Gamma|}), \quad (1)$$

where $|\Gamma|$ is the number of stack symbols.

Theorem 2. *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 $O((n_1 n_2) \cdot 2^{(m_1 m_2)^2})$ worst-case time, where $m_1 = |Q_A|, m_2 = |Q_P|, n_1 = |\delta_A|$ 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 for computing the edit-distance and an optimal alignment between a regular language and a context-free language at the same time. As we have noticed in Theorem 2, the algorithm runs in exponential time since the length of an optimal alignment may be exponential in the size of input FA and PDA. Now we examine how to calculate the edit-distance without computing the corresponding optimal alignment and present a polynomial runtime algorithm for the edit-distance problem.

Let Σ_U be a unary alphabet, say $\Sigma_U = \{u\}$. We often use non-negative integers $\mathbb{Z}_+$ for the cost function in the edit-distance problem. For example, the Levenshtein distance uses one for all operation costs. This motives us to investigate the edit-distance problem and unary context-free grammars. From now on, we assume that the cost function is defined over $\mathbb{Z}_+$.

We use a unary homomorphism the alignment PDA $\mathcal{A}(A, P)$ obtained from an FA A and a PDA P and convert it into the context-free grammar. Let $\mathcal{H} : \{(a \to b) \mid a, b \in \Sigma \cup \{\lambda\}\} \to {\Sigma_U}^*$ be a homomorphism between the edit operations and a unary alphabet $\{u\}$. Let $c_{\mathtt{I}}, c_{\mathtt{D}}$ and $c_{\mathtt{S}}$ be the costs of insertion, deletion and substitution, respectively. Then, we define $\mathcal{H}$ to be

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

If follows from the morphism function that given an alignment w

$$\mathtt{C}(w) = |\mathcal{H}(w)|.$$

By Theorem 1, 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 all edit strings w with unary strings u^i, where $i = \mathtt{C}(w)$. 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 new $\mathcal{A}(A, P)$, say, $\mathcal{H}(\mathcal{A}(A, P))$. Since the number of production rules in $G_{\mathcal{H}(\mathcal{A}(A,P))}$ is proportional to the number of transitions in $\mathcal{H}(\mathcal{A}(A, P))$ by the triple construction, we can reduce the size of the grammar $G_{\mathcal{H}(\mathcal{A}(A,P))}$, compared to $G_{\mathcal{A}(A,P)}$. 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 1. *The edit-distance $d(L(A), L(P))$ of an FA A and a PDA P is the length of the shortest string in $L(G_{\mathcal{H}(\mathcal{A}(A,P))})$.*

$$d(A, P) = \inf\{|L(G_{\mathcal{H}(\mathcal{A}(A,P))})|\}.$$

Corollary 1 shows that the edit-distance problem is now to find the shortest string in $L(G_{\mathcal{H}(\mathcal{A}(A,P))})$. Before searching for the shortest string in $L(G_{\mathcal{H}(\mathcal{A}(A,P))})$, we run a preprocessing step, which is similar to that in Algorithm 2, to improve the algorithm runtime in practice. The preprocessing step is eliminating λ-productions from the grammar. We establish a lemma for justifying this step.

Lemma 3. *Given a context-free grammar $G = (V, \Sigma, R, S)$, 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 is nullable, V and R become $\{S\}$ and $\{S \to \lambda\}$, respectively. Then, the shortest string in $L(G')$ is same as the shortest string in $L(G)$.*

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 rule $B \to wxAy$ 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 rule as $B \to w'xy$
10:       end for
11:       remove $A$ from $V$ and all $A$'s production rules from $R$
12:    end if
13: end for
```

Algorithm 3 describes how to compute the length of the shortest string in $L(G_{\mathcal{H}(\mathcal{A}(A,P))})$. This algorithm is similar to Algorithm 2. However, the main difference is that we use a binary encoding to remove the exponential factor in the running time. The complexity of Algorithm 2 is exponential since the length of the shortest string can be exponential. Since we only look for the length (the edit-distance) of the shortest string instead of the string itself (an optimal alignment), we encode string lengths as binary representation. This helps to keep an exponential length as a linear length of binary number. For example, we use 100000 to denote u^{32}.

Now we consider the complexity of Algorithm 3. In the worst-case, we need to eliminate all variables from the grammar, that means we need to repeat at most $|V| = (m_1 m_2)^2 \cdot |\Gamma|$ times for finding the variable generating the shortest string. We scan the whole grammar to find the variable in $O(|R|)$ time. Therefore, to eliminate the variables, we need $O((m_1m_2)^4 \cdot (n_1n_2) \cdot |\Gamma|)$. Then, now we consider the time for replacing the occurrence of variables with encoded numbers in binary. We should replace all occurrences of variables in the worst-case. The number of occurrences will be at most $O((m_1m_2)^2 \cdot (n_1n_2))$ and the size of binary numbers will be at most $O((m_1m_2)^2 \cdot |\Gamma|)$. Then, we need $O((m_1m_2)^4 \cdot (n_1n_2) \cdot |\Gamma|)$ again. Thus, the worst-case time complexity of Algorithm 3 is $O((m_1m_2)^4 \cdot (n_1n_2) \cdot |\Gamma|)$.

Theorem 3. *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 $n_2 = |\delta_P|$.*

References

1. Allauzen, C., Mohri, M.: 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. Alpoge, L., Ang, T., Schaeffer, L., Shallit, J.: Decidability and Shortest Strings in Formal Languages. In: Holzer, M. (ed.) DCFS 2011. LNCS, vol. 6808, pp. 55–67. Springer, Heidelberg (2011)
3. Bunke, H.: Edit distance of regular languages. In: Proceedings of 5th Annual Symposium on Document Analysis and Information Retrieval, pp. 113–124 (1996)
4. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation, 2nd edn. Addison-Wesley, Reading (1979)
5. Kari, L., Konstantinidis, S.: Descriptional complexity of error/edit systems. Journal of Automata, Languages and Combinatorics 9, 293–309 (2004)
6. Konstantinidis, S.: Computing the edit distance of a regular language. Information and Computation 205, 1307–1316 (2007)
7. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10(8), 707–710 (1966)
8. Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems. Journal of Automata, Languages and Combinatorics 7, 321–350 (2002)
9. Mohri, M.: Edit-distance of weighted automata: General definitions and algorithms. International Journal of Foundations of Computer Science 14(6), 957–982 (2003)
10. Pevzner, P.A.: Computational Molecular Biology: An Algorithmic Approach (Computational Molecular Biology). The MIT Press (2000)
11. Pighizzini, G.: How hard is computing the edit distance? Information and Computation 165(1), 1–13 (2001)
12. Thompson, K.: Programming techniques: Regular expression search algorithm. Communications of the ACM 11, 419–422 (1968)
13. Wagner, R.A.: Order-n correction for regular languages. Communications of the ACM 17, 265–268 (1974)
14. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. Journal of the ACM 21, 168–173 (1974)
15. Wood, D.: Theory of Computation. Harper & Row (1987)