Edit-Distance between Visibly Pushdown Languages

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

[1] Department of Computer Science, Yonsei University
50, Yonsei-Ro, Seodaemun-Gu, Seoul 120-749, Republic of Korea
emmous@yonsei.ac.kr
[2] Department of Computer Science, University of Liverpool
Ashton Street, Liverpool, L69 3BX, United Kingdom
sangkiko@liverpool.ac.uk

Abstract. We study the edit-distance between two visibly pushdown languages. It is well-known that the edit-distance between two context-free languages is undecidable. The class of visibly pushdown languages is a robust subclass of context-free languages since it is closed under intersection and complementation whereas context-free languages are not. We show that the edit-distance problem is decidable for visibly pushdown languages and present an algorithm for computing the edit-distance based on the construction of an alignment PDA. Moreover, we show that the edit-distance can be computed in polynomial time if we assume that the edit-distance is bounded by a fixed integer k.

Keywords: visibly pushdown languages, edit-distance, algorithm, decidability

1 Introduction

The edit-distance between two words is the smallest number of operations required to transform one word into the other [9]. We can use the edit-distance as a similarity measure between two words; the shorter distance implies that the two words are more similar. We can compute this by using the bottom-up dynamic programming algorithm [15]. The edit-distance problem arises in many areas such as computational biology, text processing and speech recognition [11, 13, 14]. This problem can be extended to a problem of computing the similarity or dissimilarity between languages [4, 5, 11].

The edit-distance between two languages is defined as the minimum edit-distance of two words, where one word is from the first language and the other word is from the second language. Mohri [11] considered the problem of computing the edit-distance between two regular languages given by finite-state automata (FAs) of sizes m and n and showed that it is computable in $O(mn \log mn)$ time. He also proved that it is undecidable to compute the edit-distance between two context-free languages [11] using the undecidability of the intersection emptiness of two context-free languages. As an intermediate result, Han et al. [5]

considered the edit-distance between a regular language and a context-free language. Given an FA and a pushdown automaton (PDA) of sizes m and n, they proposed a polynomial-time algorithm that computes the edit-distance between their languages [5].

Here we study the edit-distance problem between two visibly pushdown languages. Visibly pushdown languages are recognizable by visibly pushdown automata (VPAs), which are a special type of pushdown automata for which stack behavior is driven by the input symbols according to a partition of the alphabet. Some literature call these automata *input-driven pushdown automata* [10].

The class of visibly pushdown languages lies in between the class of regular languages and the class of context-free languages. Recently, there have been many results about visibly pushdown languages because of nice closure properties. Note that context-free languages are not closed under intersection and complement and deterministic context-free languages are not closed under union, intersection, concatenation, and Kleene-star. On the other hand, visibly pushdown languages are closed under all these operations. Moreover, language inclusion, equivalence and universality are all decidable for visibly pushdown languages whereas undecidable for context-free languages.

Visibly pushdown automata are useful in processing XML documents [1, 12]. For example, a visibly pushdown automaton can process a SAX representation of an XML document, which is a linear sequence of characters along with a hierarchically nested matching of open-tags with closing tags. Note that the edit-distance between two visibly pushdown languages is undecidable if two languages are defined over different visibly pushdown alphabets since the intersection emptiness is undecidable [8]. Therefore, we always assume that two visibly pushdown languages are defined over the same visibly pushdown alphabet.

We show that the edit-distance between visibly pushdown languages is decidable and present an algorithm for computing the edit-distance. Moreover, the edit-distance can be computed in polynomial time if we assume that the edit-distance is bounded by a fixed integer k.

2 Preliminaries

Here we recall some basic definitions and fix notation. For background knowledge in automata theory, the reader may refer to textbooks [6, 16].

The size of a finite set S is $|S|$. Let Σ denote a finite alphabet and Σ^* denote the set of all finite words over Σ. For $m \in \mathbb{N}$, $\Sigma^{\leq m}$ is the set of words over Σ having length at most m. The ith character of a word w is denoted by w_i for $1 \leq i \leq |w|$, and the subword of a word w that begins at position i and ends at position j is denoted by $w_{i,j}$ for $1 \leq i \leq j \leq |w|$. A language over Σ is a subset of Σ^*. Given a set X, 2^X denotes the power set of X. The symbol λ denotes the null word.

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 word $x \in \Sigma^*$ is accepted by A if there is a labeled path from s to a state in F such that this path spells out the word x, namely, $\delta(s, x) \cap F \neq \emptyset$. The language $L(A)$ is the set of words accepted by A.

A (nondeterministic) pushdown automaton (PDA) is specified by a tuple $P = (Q, \Sigma, \Gamma, \delta, s, Z_0, F)$, 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, $s \in Q$ 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. We define the size $|P|$ of P as $|Q| + |\delta|$. The language $L(P)$ is the set of words accepted by P.

A (nondeterministic) *visibly pushdown automaton* (VPA) [2, 10] is a restricted version of a PDA, where the input alphabet Σ consists of three disjoint classes, Σ_c, Σ_r, and Σ_l. Namely, $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_l$. The class of the input alphabet determines the type of stack operation. For example, the automaton always pushes a symbol onto the stack when it reads a *call symbol* in Σ_c. If the input symbol is a *return symbol* in Σ_r, the automaton pops a symbol from the stack. Finally, the automaton neither uses the stack nor even examine the content of the stack for the *local symbols* in Σ_l. Formally, the input alphabet is defined as a triple $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_l)$, where three components are finite disjoint sets.

A VPA is formally defined by a tuple $A = (\widetilde{\Sigma}, \Gamma, Q, s, F, \delta_c, \delta_r, \delta_l)$, where $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_l$ is the input alphabet, Γ is the finite set of stack symbols, Q is the finite set of states, $s \in Q$ is the start state, $F \subseteq Q$ is the set of final states, $\delta_c : Q \times \Sigma_c \to 2^{Q \times \Gamma}$ is the transition function for the push operations, $\delta_r : Q \times (\Gamma \cup \{\bot\}) \times \Sigma_r \to 2^Q$ is the transition function for the pop operations, and $\delta_l : Q \times \Sigma_l \to 2^Q$ is the local transition function. We use $\bot \notin \Gamma$ to denote the top of an empty stack.

A *configuration* of A is a triple (q, w, v), where $q \in Q$ is the current state, $w \in \Sigma^*$ is the remaining input, and $v \in \Gamma^*$ is the content on the stack. Denote the set of configurations of A by $C(A)$ and we define the single step computation with the relation $\vdash_A \subseteq C(A) \times C(A)$.

- **Push operation:** $(q, aw, v) \vdash_A (q', w, \gamma v)$ for all $a \in \Sigma_c, (q', \gamma) \in \delta_c(q, a), \gamma \in \Gamma, w \in \Sigma^*$ and $v \in \Gamma^*$.
- **Pop operation:** $(q, aw, \gamma v) \vdash_A (q', w, v)$ for all $a \in \Sigma_r, q' \in \delta_r(q, \gamma, a), \gamma \in \Gamma, w \in \Sigma^*$ and $v \in \Gamma^*$; furthermore, $(q, aw, \epsilon) \vdash_A (q', w, \epsilon)$, for all $a \in \Sigma_r, q' \in \delta_r(q, \bot, a)$ and $w \in \Sigma^*$.
- **Local operation:** $(q, aw, v) \vdash_A (q', w, v)$, for all $a \in \Sigma_l, q' \in \delta_l(q, a), w \in \Sigma^*$ and $v \in \Gamma^*$.

An *initial configuration* of a VPA $A = (\widetilde{\Sigma}, \Gamma, Q, s, F, \delta_c, \delta_r, \delta_l)$ is (s, w, ϵ), where s is the start state, w is an input word and ϵ implies an empty stack. A

VPA accepts a word if A arrives at the final state by reading the word from the initial configuration. Formally, we write the language recognized by A_1 as

$$L(A) = \{w \in \Sigma^* \mid (s, w, \epsilon) \vdash_A^* (q, \epsilon, v) \text{ for some } q \in F, v \in \Gamma^*\}.$$

We call the languages recognized by VPAs the *visibly pushdown languages*. The class of visibly pushdown languages is a strict subclass of deterministic context-free languages and a strict superclass of regular languages. While many closure properties such as complement and intersection do not hold for context-free languages, visibly pushdown languages are closed under most operations including other basic operations such as concatenation, union, and Kleene-star.

3 Edit-Distance

The edit-distance between two words is the smallest number of operations that transform a word to the other. People use different edit-operations according to own applications. We consider three basic operations, insertion, deletion and substitution for simplicity. Given an alphabet Σ, let $\Omega = \{(a \to b) \mid a, b \in \Sigma \cup \{\lambda\}\}$ be a set of edit-operations. Namely, Ω is an alphabet of all edit-operations for *deletions* $(a \to \lambda)$, *insertions* $(\lambda \to a)$ and *substitutions* $(a \to b)$. We say that an edit-operation $(a \to b)$ is a *trivial substitution* if $a = b$. We call a word $\omega \in \Omega^*$ an *edit string* [7] or an *alignment* [11].

Let the morphism h between Ω^* and $\Sigma^* \times \Sigma^*$ be $h((a_1 \to b_1) \cdots (a_n \to b_n)) = (a_1 \cdots a_n, b_1 \cdots b_n)$, where $a_i, b_i \in \Sigma \cup \{\lambda\}$ for $1 \leq i \leq n$. For example, a word $\omega = (a \to \lambda)(b \to b)(\lambda \to c)(c \to c)$ over Ω is an alignment of abc and bcc, and $h(\omega) = (abc, bcc)$. Thus, from an alignment ω of two words x and y, we can retrieve x and y using h: $h(\omega) = (x, y)$.

We associate a non-negative edit cost to each edit-operation $\omega_i \in \Omega$ as a function $\mathtt{C} : \omega_i \to \mathbb{R}_+$. We can extend the function to the cost $\mathtt{C}(\omega)$ of an alignment $\omega = \omega_1 \cdots \omega_n$ as follows: $\mathtt{C}(\omega) = \sum_{i=1}^{n} \mathtt{C}(\omega_i)$.

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

Note that we use the Levenshtein distance [9] for edit-distance. Thus, we assign cost 1 to all edit-operations $(a \to \lambda), (\lambda \to a)$, and $(a \to b)$ for all $a \neq b \in \Sigma$. We can extend the edit-distance definition to languages.

Definition 2. *The* edit-distance $d(L_1, L_2)$ *between two languages* $L_1, L_2 \subseteq \Sigma^*$ *is the minimum edit-distance of two words, one is from* L_1 *and the other is from* L_2*:* $d(L_1, L_2) = \min\{d(x, y) \mid x \in L_1 \text{ and } y \in L_2\}$.

Mohri [11] revealed that the edit-distance between two context-free languages is undecidable from the undecidability of the intersection emptiness of two context-free languages. Moreover, he presented an algorithm to compute the edit-distance between two regular languages given by FAs of size m and n in

$O(mn \log mn)$ time. Han et al. [5] considered the intermediate case where one language is regular and the other language is context-free. Given a PDA and an FA for the languages, it is shown that the problem is computable in polynomial time while computing an optimal alignment corresponding to the edit-distance requires an exponential runtime.

4 Edit-Distance between Two VPLs

First we study the decidability of the edit-distance between two visibly pushdown languages. We notice that the edit-distance between two context-free languages is undecidable, whereas the edit-distance between two regular languages can be computed in polynomial time.

Interestingly, it turns out that the edit-distance between two visibly pushdown languages is decidable. Here we show that we can compute the edit-distance between two visibly pushdown languages L_1 and L_2 given by two VPAs by constructing the alignment PDA [5], which accepts all the possible alignments between two languages of length up to k. By setting k to be the upper bound of the edit-distance between given two visibly pushdown languages L_1 and L_2, we can compute the edit-distance between L_1 and L_2 by choosing the minimum-cost alignment accepted by the alignment PDA.

The alignment PDA is first proposed by Han et al. [5] to compute the edit-distance between a regular language and a context-free language. Given an FA A and a PDA P, we can construct an alignment PDA $\mathcal{A}(A, P)$ that accepts all possible alignments that transform a word accepted by the FA A to a word accepted by the PDA P. After we construct an alignment PDA for two languages, we can compute the edit-distance between the two languages by computing the length of the shortest word accepted by the alignment PDA. Furthermore, we can obtain an optimal alignment between two languages by taking the shortest word.

An interesting point to note is that the construction of the alignment PDA does not imply that we can compute the edit-distance. For example, the edit-distance between two context-free languages is undecidable even though it is possible to construct an alignment PDA with two stacks that accepts all possible alignments between two PDAs [11]. This is because we cannot compute the length of the shortest word accepted by a two-stack PDA, which can simulate a Turing machine [6]. Here we start from an idea that we do not need to consider all possible alignments between two visibly pushdown languages to compute the edit-distance and an optimal alignment. If we know the upper bound k of the edit-distance between two visibly pushdown languages, we can compute the edit-distance and an optimal alignment by constructing an alignment PDA that accepts all possible alignments of cost up to k between two visibly pushdown languages. This can be done because the stack operation of VPAs is determined by the input character.

Assume that we have two VPAs A_1 and A_2 over the same alphabet Σ. Then, the alignment PDA $\mathcal{A}(A_1, A_2)$ of A_1 and A_2 simulates all possible alignments

between the two languages $L(A_1)$ and $L(A_2)$. Note that $\mathcal{A}(A_1, A_2)$ simulates two stacks from two VPAs simultaneously by reading each edit-operation. Whenever $\mathcal{A}(A_1, A_2)$ simulates a trivial edit-operation $(a \to a), a \in \Sigma$ which substitutes a character into the same one, the difference in height of two stacks does not change since two VPAs read characters in the same class. The height of two stacks becomes different when $\mathcal{A}(A_1, A_2)$ reads insertions of call (or return) symbols, deletions of call (or return) symbols, or substitutions between two characters in different classes. For example, the height of two stacks becomes different by 1 when $\mathcal{A}(A_1, A_2)$ read an insertion $(\lambda \to a)$ of a call symbol $a \in \Sigma_c$ since A_1 does not change its stack while A_2 is pushing a stack symbol onto its stack. The height of two stacks becomes different the most when $\mathcal{A}(A_1, A_2)$ reads an edit-operation that substitutes a call (resp. return) symbol into a return (resp. call) symbol since A_1 pushes (resp. pops) a stack symbol while A_2 pops (resp. pushes) a stack symbol. Therefore, if the upper bound of the edit-distance between two visibly pushdown languages is k, the maximum height difference between two stacks can be at most $2k$ whenever we simulate an alignment that costs up to k. Note that we can easily compute the upper bound of the edit-distance between two visibly pushdown languages by computing the shortest words from each visibly pushdown language. Let $\mathrm{lsw}(L)$ be the length of the shortest word in L.

Proposition 3. *Let $L \subseteq \Sigma^*$ and $L' \subseteq \Sigma^*$ be the languages over Σ. Then, $d(L, L') \leq \max\{\mathrm{lsw}(L), \mathrm{lsw}(L')\}$ holds.*

Now we give the alignment PDA construction for computing the edit-distance between two visibly pushdown languages. The basic idea of the construction is to remember the top stack symbols from two stacks by using the states of the alignment PDA. Intuitively, we store the information of the top $2k$ stack symbols from both stacks in the states instead of pushing into the stack of the alignment PDA.

Let $A_i = (\tilde{\Sigma}, \Gamma_i, Q_i, s_i, F_i, \delta_{i,c}, \delta_{i,r}, \delta_{i,l})$ for $i = 1, 2$ be two VPAs. We construct the alignment PDA $\mathcal{A}(A_1, A_2) = (Q_E, \Omega, \Gamma_E, s_E, F_E, \delta_E)$, where

- $Q_E = Q_1 \times Q_2 \times \Gamma_1^{\leq 2k} \times \Gamma_2^{\leq 2k}$ is the set of states,
- $\Omega = \{(a \to b) \mid a, b \in \Sigma \cup \{\lambda\}\}$ is the alphabet of edit-operations,
- $\Gamma_E = (\Gamma_1 \cup \{\lambda\}) \times (\Gamma_2 \cup \{\lambda\}) \setminus \{(\lambda, \lambda)\}$ is a finite stack alphabet,
- $s_E = (s_1, s_2, \lambda, \lambda)$ is the start state, and
- $F_E = F_1 \times F_2 \times \Gamma_1^{\leq 2k} \times \Gamma_2^{\leq 2k}$ is the set of final states.

Now we define the transition function δ_E. There are seven cases to consider as follows. The alignment PDA $\mathcal{A}(A_1, A_2)$ reads an edit-operation that

(i) pushes on two stacks simultaneously,
(ii) pushes on the first stack,
(iii) pushes on the second stack,
(iv) pops from two stacks simultaneously,
(v) pops from the first stack,
(vi) pops from the second stack, and

(vii) not perform stack operation.

Assume that $x \in \Gamma_1^{\leq 2k}$ and $y \in \Gamma_2^{\leq 2k}$. Simply, x and y are words over the stack alphabet of A_1 and A_2 whose lengths are at most $2k$. For a non-empty word x over Γ_1, recall that we denote the first character and the last character of x by x_1 and $x_{|x|}$, respectively. We also denote the subword that consists of characters from x_i to x_j by $x_{i,j}$, where $i < j$.

Suppose that there are transitions defined in VPAs A_1 and A_2 as follows:

- $(q', \gamma) \in \delta_{1,c}(q, a_c)$, [push operation in A_1]
- $q' \in \delta_{1,l}(q, a_l)$, [local operation in A_1]
- $q' \in \delta_{1,r}(q, \gamma, a_r)$, [pop operation in A_1]
- $(p', \mu) \in \delta_{2,c}(p, b_c)$, [push operation in A_2]
- $p' \in \delta_{2,l}(p, b_l)$, and [local operation in A_2]
- $p' \in \delta_{2,r}(p, \mu, b_r)$. [pop operation in A_2]

By reading an edit-operation $(a_c \to b_c)$, we define δ_E to operate as follows:

- $(q', p', x\gamma, y\mu) \in \delta_E((q, p, x, y), (a_c \to b_c))$ if $|x| < 2k$ and $|y| < 2k$,
- $((q', p', x_{2,|x|}\gamma, y_{2,|y|}\mu), (x_1, y_1)) \in \delta_E((q, p, x, y), (a_c \to b_c))$ otherwise.

By the above transitions, we simulate the push operations on the stacks of A_1 and A_2 at the same time. We store the information of the top $2k$ stack symbols in the states instead of using "real" stack. If a state already contains the information of $2k$ symbols, we start using the stack by pushing the bottommost pair of stack symbols onto the stack.

We also define δ_E to operate as follows by reading an edit-operation $(a_c \to b_l)$:

- $(q', p', x\gamma, y) \in \delta_E((q, p, x, y), (a_c \to b_l))$ if $|x| < 2k$,
- $((q', p', x_{2,|x|}\gamma, y_{2,|y|}), (x_1, y_1)) \in \delta_E((q, p, x, y), (a_c \to b_l))$ otherwise.

Similarly, we define δ_E for an edit-operation $(a_c \to \lambda)$ as follows:

- $(q', p, x\gamma, \mu) \in \delta_E((q, p, x, y), (a_c \to \lambda))$ if $|x| < 2k$,
- $((q', p, x_{2,|x|}\gamma, y_{2,|y|}), (x_1, y_1)) \in \delta_E((q, p, x, y), (a_c \to \lambda))$ otherwise.

Note that the cases of reading $(a_l \to b_c)$ or $(\lambda \to b_c)$ are completely symmetric to the previous two cases. We also consider the cases when we have to pop at least one of two stacks by reading an edit-operation. First, we define δ_E for the case when we read an edit-operation $(a_r \to b_r)$ that pops from both stacks at the same time.

- $(q', p', \gamma_{\mathsf{top}} x_{1,|x|-1}, \mu_{\mathsf{top}} y_{1,|y|-1}) \in \delta_E((q, p, x, y), (\gamma_{\mathsf{top}}, \mu_{\mathsf{top}}), (a_r \to b_r))$ if $x_{|x|} = \gamma$, $y_{|y|} = \mu$ and the top of the stack is $(\gamma_{\mathsf{top}}, \mu_{\mathsf{top}})$,
- $(q', p', x_{1,|x|-1}, y_{1,|y|-1}) \in \delta_E((q, p, x, y), (a_r \to b_r))$ if $x_{|x|} = \gamma$, $y_{|y|} = \mu$ and the stack is empty.

When we simulate pop operations on $\mathcal{A}(A_1, A_2)$, we remove the last stack symbols stored in the state. Then, we pop off the top of the stack, say $(\gamma_{\mathsf{top}}, \mu_{\mathsf{top}})$, and move the pair to the front of the stored stack symbols in the state.

For the case when we have to pop only from the first stack, we define δ_E to be as follows:

- $(q', p', \gamma_{\mathsf{top}} x_{1,|x|-1}, \mu_{\mathsf{top}} y) \in \delta_E((q, p, x, y), (\gamma_{\mathsf{top}}, \mu_{\mathsf{top}}), (a_r \to b_l))$ if $|x| = 2k$, $|y| < 2k$, and the top of the stack is $(\gamma_{\mathsf{top}}, \mu_{\mathsf{top}})$,
- $(q', p', x_{1,|x|-1}, y) \in \delta_E((q, p, x, y), (a_r \to b_l))$ otherwise.

Note that $x_{|x|} = \gamma$ should hold for simulating pop operations on the first stack. We also define the transitions for the edit-operations of the form $(a_r \to \lambda)$ similarly. Again, the pop operations on the second stack are completely symmetric. Lastly, we define δ_E for the case when we do not touch the stack at all. There are three possible cases as follows:

- $(q', p', x, y) \in \delta_E((q, p, x, y), (a_l \to b_l))$,
- $(q', p, x, y) \in \delta_E((q, p, x, y), (a_l \to \lambda))$, and
- $(q, p', x, y) \in \delta_E((q, p, x, y), (\lambda \to b_l))$.

Now we prove that the constructed alignment PDA $\mathcal{A}(A_1, A_2)$ simulates all possible alignments of cost up to k between $L(A_1)$ and $L(A_2)$.

Lemma 4. *Given two VPAs A_1 and A_2, it is possible to construct the alignment PDA $\mathcal{A}(A_1, A_2)$ that accepts all possible alignments between $L(A_1)$ and $L(A_2)$ of cost up to the constant k.*

Proof. Let us consider the simulations of two VPAs A_1 and A_2 for an alignment ω. Given $h(\omega) = (x, y)$, the VPA A_1 has to simulate a word x while the VPA A_2 is simulating a word y. Suppose that x has $2k$ call symbols and y has k' call symbols, where $k' < 2k$, and no return symbols. Then, the stack contents of A_1 and A_2 should be $\gamma_1\gamma_2 \cdots \gamma_{2k}$ and $\mu_1\mu_2 \cdots \mu_{k'}$, respectively, where $\gamma_i \in \Gamma_1$ for $1 \le i \le 2k$ and $\mu_j \in \Gamma_2$ for $1 \le j \le k'$. See Fig. 1 for illustration of this example.

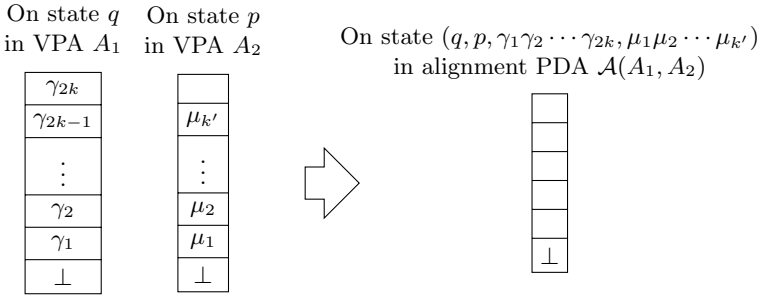

Fig. 1. Illustration of how we store the information of two stacks in the states of the alignment PDA $\mathcal{A}(A_1, A_2)$.

After this, we push the pair (γ_1, μ_1) of two stack symbols in the bottom of two stacks onto the stack of $\mathcal{A}(A_1, A_2)$ if we read an edit-operation $(a \to b)$ where a is a call symbol of A_1. Let us assume that A_1 pushes γ_{2k+1} by reading a and A_2 pushes $\mu_{k'+1}$ by reading b. Then, we push two stack symbols γ_{2k+1}

and $\mu_{k'+1}$ onto the simulated stacks stored in the state. See Fig. 2 to see what happens after reading $(a \to b)$.

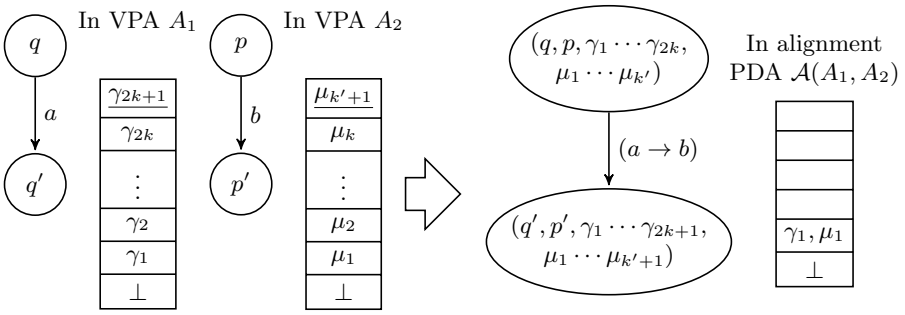

Fig. 2. Illustration of how we store the information of two stacks in the states of the alignment PDA $\mathcal{A}(A_1, A_2)$. Underlined stack symbols γ_{2k+1} and $\mu_{k'+1}$ are pushded by the current transitions of VPAs.

In this way, we can make the stack of $\mathcal{A}(A_1, A_2)$ to be always synchronized. Note that we only push stack symbols onto the stack of $\mathcal{A}(A_1, A_2)$ when we need to push a stack symbol onto the stack where the simulated stack stored in the state is full. Therefore, the stack of $\mathcal{A}(A_1, A_2)$ should be empty if the maximum height of two stacks stored in the state is less than $2k$. When we read an edit-operation that pops a stack symbol from A_1 or A_2, we pop the stack symbols from the stack contents stored in the states. If the height of the stack in the state is $2k$ before we pop a stack symbol, we pop a stack symbol from the top of the real stack of $\mathcal{A}(A_1, A_2)$ and move to the bottom of the simulated stack stored in the state of $\mathcal{A}(A_1, A_2)$.

By storing stack information in the states instead of real stacks, $\mathcal{A}(A_1, A_2)$ accepts alignments where the height difference of two stacks during the simulation can be at most $2k$. We mention that $\mathcal{A}(A_1, A_2)$ also accepts alignments of cost higher than k if the simulation of the alignments does not require the stack height difference to be larger than $2k$.

Now we prove that the alignment PDA $\mathcal{A}(A_1, A_2)$ accepts an alignment ω, where $\mathtt{C}(\omega) \leq k$ if and only if ω is an alignment satisfying $\mathtt{C}(\omega) \leq k$ and $h(\omega) = (x, y)$, where $x \in L(A_1)$ and $y \in L(A_2)$.

($\Longrightarrow$) Since $\mathcal{A}(A_1, A_2)$ accepts an alignment ω, where $\mathtt{C}(\omega) \leq k$, there exists an accepting computation X_ω of $\mathcal{A}(A_1, A_2)$ on ω ending in a state (f_1, f_2) where $f_1 \in F_1, f_2 \in F_2$. We assume that $\omega = w_1 \cdots w_l$, where $w_i \in \Omega$ for $i \leq i \leq l$. Denote the sequence of states of $\mathcal{A}(A_1, A_2)$ appearing in the computation X_ω just before reading the lth symbol of ω as $C_0, \ldots, C_{l-1}$, and denote the state (f_1, f_2) where the computation ends as C_l. Consider the first component $q_i \in Q_1$ of the state $C_i \in Q_E$, for $0 \leq i \leq l$, and the first component $a_j \in \Sigma \cup \{\lambda\}$ of the edit-operation w_j, for $1 \leq j \leq l$. From the construction of $\mathcal{A}(A_1, A_2)$, it follows that

- $(q_{i+1}, \gamma) \in \delta_{A,c}(q, a_{i+1})$, if $a_{i+1} \in \Sigma_c$
- $q_{i+1} \in \delta_{A,l}(q, a_{i+1})$, if $a_{i+1} \in \Sigma_l$
- $q_{i+1} \in \delta_{A,r}(q, \gamma, a_{i+1})$, and if $a_{i+1} \in \Sigma_r$
- $q_{i+1} = q_i$. if $a_{i+1} = \lambda$

for $0 \leq i \leq l-1$. Note that the transitions of δ_E reading an "insertion operation" $(\lambda \to b)$ do not change the first components of the states. Thus, the first components of the state $C_0, \ldots, C_l$ spell out an accepting computation of A_1 on the word $x = a_1 \cdots a_l$ obtained by concatenating the first components of the edit-operations of ω. Using a similar argument for the word y obtained by concatenating the second components of ω, we can show that the computation yields an accepting computation of A_2 on y.
($\Longleftarrow$) Let $\omega = (\omega_{L(1)} \to \omega_{R(1)})(\omega_{L(2)} \to \omega_{R(2)}) \cdots (\omega_{L(l)} \to \omega_{R(l)})$ be an alignment of length l for $x = \omega_{L(1)}\omega_{L(2)} \cdots \omega_{L(l)} \in L(A_1)$ and $y = \omega_{R(1)}\omega_{R(2)} \cdots \omega_{R(l)} \in L(A_2)$. Let $\mathcal{C} = C_0C_1 \cdots C_m, m \in \mathbb{N}$, be a sequence of configurations of the VPA A_1 that traces an accepting computation $X_{A,x}$ on the word x. Assuming that C_i is (q_i, γ_i), the configuration C_{i+1} is obtained from C_i by applying a transition $(q_{i+1}, \gamma_{i+1}) \in \delta_A(q_i, a, \gamma_i)$, where $a \in \Sigma \cup \{\lambda\}$. Note that $\omega_{L(j)}$ or $\omega_{R(j)}$ may be the empty word if $(\omega_{L(j)} \to \omega_{R(j)})$ is an edit-operation representing an insertion or a deletion operation. Suppose that the computation step $C_i \to C_{i+1}$ consumes h empty words. Then in the sequence $\mathcal{C}$ after the configuration C_i, we add $h-1$ identical copies of C_i. Then, we denote the modified sequence of configurations $\mathcal{C}' = C'_0C'_1 \cdots C'_l, l \in \mathbb{N}$. Analogously, let $\mathcal{D} = D'_0D'_1 \cdots D'_l$ be a sequence of configurations of the VPA A_2 that traces an accepting computation $X_{B,y}$ on the word y.

From the sequences $\mathcal{C}'$ and $\mathcal{D}$, we obtain a sequence of configurations of $\mathcal{A}(A_1, A_2)$ describing an accepting computation on the alignment ω. For the deletion operations $(\omega_{L(j)} \to \lambda)$, the state is changed just in the first component and the configuration of A_2 remains unchanged. Recall that we have added the identical copies of configurations for simulating this case. The deletion operations can be simulated symmetrically. For the substitution operations $(\omega_{L(j)} \to \omega_{R(j)})$, the computation step simulates both a state transition of A_1 on $\omega_{L(j)}$ and a state transition of A_2 on $\omega_{R(j)}$.

This implies that two modified sequences of configurations of A_1 and A_2 can be combined to yield an accepting computation of $\mathcal{A}(A_1, A_2)$ on ω. □

Let us consider the size of the constructed alignment PDA $\mathcal{A}(A_1, A_2)$. Let $m_i = |Q_i|, n_i = |\delta_{i,c}| + |\delta_{i,r}| + |\delta_{i,l}|$, and $l_i = |\Gamma_i|$ for $i = 1, 2$.

Note that each state contains an information of a pair of states from A_1 and A_2, and a stack information of at most $2k$ stack symbols of A_1 and A_2. If we represent a stack where the height of the stack is restricted to $2k$ with a word over the stack alphabet Γ_1, we have

$$l'_1 = \sum_{i=0}^{2k} l_1^i = 1 + l_1 \cdot \frac{l_1^{2k} - 1}{l_1 - 1}$$

possible words. Similarly, we define l'_2 to be the number of possible words over Γ_2. Therefore, the number of states in $\mathcal{A}(A_1, A_2)$ is in

$$m_1 m_2 \cdot l'_1 l'_2 = m_1 m_2 \cdot \left(\sum_{i=0}^{2k} l_1^i\right) \cdot \left(\sum_{i=0}^{2k} l_2^i\right).$$

The size of the stack alphabet Γ_E is $l_1 l_2$ since we use all pairs of the stack symbols where the first stack symbol is from Γ_1 and the second stack symbol is from Γ_2. The size of the transition function δ_E is

$$m_1 m_2 \cdot n_1 n_2 \cdot \left(\sum_{i=0}^{2k} l_1^i\right) \cdot \left(\sum_{i=0}^{2k} l_2^i\right).$$

By Lemma 4, we can obtain an alignment PDA from two VPAs A_1 and A_2 to compute the edit-distance $d(L(A_1), L(A_2))$. Recently, Han et al. [5] studied the edit-distance between a PDA and an FA. The basic idea is to construct an alignment PDA from a PDA and an FA and compute the length of the shortest alignment from the alignment PDA. As a step, they present an algorithm for obtaining a shortest word and computing the length of the shortest word from a PDA. For the sake of completeness, we include the following proposition.

Proposition 5 (Han et al. [5]). *Given a PDA $P = (Q, \Sigma, \Gamma, \delta, s, Z_0, F_P)$, we can obtain a shortest word in $L(P)$ whose length is bounded by $2^{m^2 l+1}$ in $O(n \cdot 2^{m^2 l})$ worst-case time and compute its length in $O(m^4 nl)$ worst-case time, where $m = |Q|$, $n = |\delta|$ and $l = |\Gamma|$.*

Now we establish the following runtime for computing the edit-distance between two VPAs.

Theorem 6. *Given two VPAs $A_i = (\Sigma, \Gamma_i, Q_i, s_i, F_i, \delta_{i,c}, \delta_{i,r}, \delta_{i,l})$ for $i = 1, 2$, we can compute the edit-distance between $L(A_1)$ and $L(A_2)$ in $O((m_1 m_2)^5 \cdot n_1 n_2 \cdot (l_1 l_2)^{10k})$ worst-case time, where $m_i = |Q_i|, n_i = |\delta_{i,c}| + |\delta_{i,r}| + |\delta_{i,l}|, l_i = |\Gamma_i|$ for $i = 1, 2$ and $k = \max\{\mathrm{lsw}(L(A_1)), \mathrm{lsw}(L(A_2))\}$.*

If we replace k with the length $2^{|Q|^2 \cdot |\Gamma|+1}$ of a shortest word from a VPA from the time complexity obtained in Theorem 6, we have double exponential time complexity for computing the edit-distance between two VPAs. It is still an open problem to find a polynomial algorithm for the problem or to establish any hardness result. Instead, we can observe that the edit-distance problem for VPAs can be computed in polynomial time if we limit the edit-distance to a fixed integer k.

Corollary 7. *Let A_1 and A_2 be two VPAs and k be a fixed integer such that $d(L(A_1), L(A_2)) \leq k$. Then, we can compute the edit-distance between $L(A_1)$ and $L(A_2)$ in polynomial time.*

As a corollary of Theorem 6, we can also establish the following result. A *visibly counter automaton* (VCA) [3] can be regarded as a VPA with a single stack symbol. It is interesting to see that we can compute the edit-distance between two VCAs in polynomial time when we are given t

Corollary 8. *Given two VCAs A_1, A_2 and a positive integer $k \in \mathbb{N}$ in unary such that $d(L(A_1), L(A_2)) \leq k$, we can compute the edit-distance between $L(A_1)$ and $L(A_2)$ in polynomial time.*

References

1. R. Alur. Marrying words and trees. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, pages 233–242, 2007.
2. R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 202–211, 2004.
3. V. Bárány, C. Löding, and O. Serre. Regularity problems for visibly pushdown languages. In *Proceedings of the 23rd Annual Symposium on Theoretical Aspects of Computer Science*, pages 420–431, 2006.
4. C. Choffrut and G. Pighizzini. Distances between languages and reflexivity of relations. *Theoretical Compututer Science*, 286(1):117–138, 2002.
5. Y.-S. Han, S.-K. Ko, and K. Salomaa. The edit-distance between a regular language and a context-free language. *International Journal of Foundations of Computer Science*, 24(7):1067–1082, 2013.
6. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 2nd edition, 1979.
7. L. Kari and S. Konstantinidis. Descriptional complexity of error/edit systems. *Journal of Automata, Languages and Combinatorics*, 9:293–309, 2004.
8. J. Leike. VPL intersection emptiness. Bachelor's Thesis, University of Freiburg, 2010.
9. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.
10. K. Mehlhorn. Pebbling mountain ranges and its application to DCFL-recognition. In *Automata, Languages and Programming*, volume 85, pages 422–435. 1980.
11. M. Mohri. Edit-distance of weighted automata: General definitions and algorithms. *International Journal of Foundations of Computer Science*, 14(6):957–982, 2003.
12. B. Mozafari, K. Zeng, and C. Zaniolo. From regular expressions to nested words: Unifying languages and query execution for relational and xml sequences. *Proceedings of the VLDB Endowment*, 3(1-2):150–161, 2010.
13. P. A. Pevzner. *Computational molecular biology - an algorithmic approach*. MIT Press, 2000.
14. K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.
15. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21:168–173, 1974.
16. D. Wood. *Theory of Computation*. Harper & Row, 1987.

Appendix

Context-free grammar (CFG). 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 word 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 $\overset{*}{\to}$ and the context-free language generated by G is $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\to} w\}$. We say that a variable $A \in V$ is *nullable* if $A \overset{*}{\to} \lambda$.

Proposition 3. *Let $L \subseteq \Sigma^*$ and $L' \subseteq \Sigma^*$ be the languages over Σ. Then,*

$$d(L, L') \leq \max\{\mathrm{lsw}(L), \mathrm{lsw}(L')\}$$

holds.

Proof. Assume that $\mathrm{lsw}(L) = m$ and $\mathrm{lsw}(L') = n$ where $n \leq m$. It is easy to see that the edit-distance between two shortest words can be at most m since we can substitute all characters of the shortest word of length n with any subsequence of the longer word and insert the remaining characters. □

Proposition 5. (Han et al. [5]) *Given a PDA $P = (Q, \Sigma, \Gamma, \delta, s, Z_0, F_P)$, we can obtain a shortest word in $L(P)$ whose length is bounded by $2^{m^2 l+1}$ in $O(n \cdot 2^{m^2 l})$ worst-case time and compute its length in $O(m^4 nl)$ worst-case time, where $m = |Q|$, $n = |\delta|$ and $l = |\Gamma|$.*

Proof. Recall that we can convert a PDA into a CFG by the triple construction [6]. Let us denote the CFG obtained from P by G_P. Then, G_P has $|Q|^2 \cdot |\Gamma| + 1$ variables and $|Q|^2 \cdot |\delta|$ productions. Moreover, each production of G_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 $A, B, C \in V$. Since we want to compute the shortest word from G_P, we can remove the occurrences of all *nullable* variables from G_P. Then, we pick a variable A that generates the shortest word $t \in \Sigma^*$ among all variables and replace its occurrence in G_P with t. We can compute the shortest word of $L(P)$ by iteratively removing occurrences of such variables. We describe the algorithm in Algorithm 1.

Since a production of G_P has at most one terminal followed by two variables, the length of the word to be substituted is at most $2^m - 1$ when we replace mth variable. Since we replace at most $|Q|^2 \cdot |\Gamma|$ variables to have the shortest word, the length of the shortest word in $L(P)$ can be at most $2^{|Q|^2 \cdot |\Gamma|+1}$. 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 a shortest word is $O(n \cdot 2^{m^2 l})$. We also note that we can compute only the length of the shortest word in $O(m^4 nl)$ worst-case time by encoding a shortest word to be substituted with a binary number. □

Algorithm 1: ShortestLength(P)

Input: A PDA $P = (Q, \Sigma, \Gamma, \delta, s, Z_0, F_P)$
Output: $\mathrm{lsw}(L(P))$
convert P into a CFG $G_P = (V, \Sigma, R, S)$ by the triple construction
eliminate all nullable variables
for $B \to t \in R$, where $t \in \Sigma^*$ and $|t|$ is minimum among all such t in R **do**
  if $B = S$ **then**
    return $|t|$
  else
    replace all occurrences of A in R with t
    remove A from V and its productions from R
  end if
end for

Theorem 6. *Given two VPAs $A_i = (\Sigma, \Gamma_i, Q_i, s_i, F_i, \delta_{i,c}, \delta_{i,r}, \delta_{i,l})$ for $i = 1, 2$, we can compute the edit-distance between $L(A_1)$ and $L(A_2)$ in $O((m_1m_2)^5 \cdot n_1n_2 \cdot (l_1l_2)^{10k})$ worst-case time, where $m_i = |Q_i|, n_i = |\delta_{i,c}| + |\delta_{i,r}| + |\delta_{i,l}|, l_i = |\Gamma_i|$ for $i = 1, 2$ and $k = \max\{\mathrm{lsw}(L(A_1)), \mathrm{lsw}(L(A_2))\}$.*

Proof. In the proof of Lemma 4, we have shown that we can construct an alignment PDA $\mathcal{A}(A_1, A_2) = (Q_E, \Omega, \Gamma_E, s_E, F_E, \delta_E)$ that accepts all possible alignments between two VPAs A_1 and A_2 of length up to k. From Proposition 5, we can compute the edit-distance in $O(m^4nl)$ time, where $m = |Q_E|$, $n = |\delta_E|$ and $l = |\Gamma_E|$. Recall that

$$m = m_1m_2 \cdot \left(\sum_{i=0}^{2k} l_1^i\right) \cdot \left(\sum_{i=0}^{2k} l_2^i\right), \quad n = m_1m_2 \cdot n_1n_2 \cdot \left(\sum_{i=0}^{2k} l_1^i\right) \cdot \left(\sum_{i=0}^{2k} l_2^i\right),$$

and $l = l_1l_2$. Note that

$$\left(\sum_{i=0}^{2k} l_1^i\right) \in O(l_1^{2k}) \text{ and } \left(\sum_{i=0}^{2k} l_2^i\right) \in O(l_2^{2k})$$

if $l_1, l_2 > 0$.

Therefore, the time complexity of computing the edit-distance between two VPAs A_1 and A_2 is

$$(m_1m_2)^5 \cdot n_1n_2 \cdot \left(\sum_{i=0}^{2k} l_1^i\right)^5 \cdot \left(\sum_{i=0}^{2k} l_2^i\right)^5 \cdot l_1l_2 \in O((m_1m_2)^5 \cdot n_1n_2 \cdot (l_1l_2)^{10k}),$$

where k is the maximum of the length of the two shortest words from $L(A_1)$ and $L(A_2)$. □

Corollary 8. *Given two VCAs A_1, A_2 and a positive integer $k \in \mathbb{N}$ in unary such that $d(L(A_1), L(A_2)) \leq k$, we can compute the edit-distance between $L(A_1)$ and $L(A_2)$ in polynomial time.*

Proof. If $l_1 = l_2 = 1$,

$$\left(\sum_{i=0}^{2k} l_1^i\right) \in O(k) \text{ and } \left(\sum_{i=0}^{2k} l_2^i\right) \in O(k).$$

If we replace l_1^{2k} and l_2^{2k} by k from the time complexity, we obtain the time complexity $O((m_1 m_2)^5 \cdot n_1 n_2 \cdot k^{10})$ which is polynomial in the size of input.