Prefix Distance Between Regular Languages

Timothy Ng

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

Abstract. The prefix distance between two words x and y is defined as the number of symbol occurrences in the words that do not belong to the longest common prefix of x and y. We show how to model the prefix distance using weighted transducers. We use the weighted transducers to compute the prefix distance between two regular languages by a transducer-based approach originally used by Mohri for an algorithm to compute the edit distance. We also give an algorithm to compute the inner prefix distance of a regular language.

1 Introduction

Distance measures are used in a variety of applications to measure the similarity of data. For instance, the Hamming distance counts the number of positions in which two words of equal length differ. Another common measure is the Levenshtein distance, also called the edit distance, which counts the number of insertion, deletion, and substitution operations that are needed to transform one word to another. However, counting the number of edit operations to transform one word into another is not the only relevant way to measure the similarity between words. The prefix distance is defined in terms of the longest common prefix of two words. For the words x and y, their prefix distance is the number of symbols that do not belong to their longest common prefix. We can define the suffix and subword distances in a similar way in terms of the longest common suffix or subword of two words.

These distance measures can be extended in various ways to distances between sets of words, or languages. A common extension of a distance function for languages L_1 and L_2 takes the minimum distance between a word u in L_1 and a word v in L_2. An alternative extension is called the relative distance [4]. The relative distance from a language L_1 to a language L_2 is the supremum over all words w in L_1 of the smallest distance between w and L_2. Another notion of distance on languages is the inner distance of a language [11]. For a language L, the inner distance is the smallest distance between two words u and v in L.

Much of the work on computing distances on languages has been focused on the edit distance and its variants. Pighizzini [15] studied the hardness of computing the edit distance between a word and a language. Mohri [14] showed how to compute the edit distance and its variants between two regular languages in polynomial time. Benedikt et al. [1, 2] showed how to compute the relative edit distance between regular languages. Han et al. [8] gave a polynomial time algorithm for computing the edit distance between a regular language and context-free language. Konstantinidis [11] gave an algorithm for computing the inner edit distance of a regular language in quadratic time. Kari et al. [10] gave a quadratic time algorithm for computing the inner Hamming distance of a regular language. Konstantinidis and Silva [12] showed how to compute the inner distance for variants of the edit distance.

Naturally, the same extensions to languages can be applied to the prefix, suffix, and subword distances and some of these extensions have already been studied. Bruschi and Pighizzini [3] studied the hardness of computing the prefix distance between a word to a language in the context of intrusion detection. Choffrut and Pighizzini [4] showed that the relative prefix distance between two regular languages is computable. Kutrib et al. [13] considered a parameterized prefix distance between languages to measure fault tolerance of finite-state devices.

In this paper, we show how to compute the prefix distance between two regular languages. We show how to model prefix distance using edit systems and construct transducers which realize these models. We use these transducers to compute distances using a similar approach to Mohri's edit distance algorithm for weighted automata from [14]. We also show how to use the weighted transducer approach to compute the inner prefix distance of a given regular language. We also give polynomial time algorithms based on the transducer-based approach to compute the suffix distance and the subword distance between two regular languages.

2 Preliminaries

Here we briefly recall some definitions and notation used in the paper. For all unexplained notions on finite automata and regular languages the reader may consult the textbook by Shallit [16] or the survey by Yu [17]. More on weighted automata and transducers can be found in the textbook by Droste et al. [7]. A survey of distances is given by Deza and Deza [6].

In the following, Σ is always a finite alphabet, the set of all words over Σ is denoted Σ^*, and ε denotes the empty word. The reversal of a word $w \in \Sigma^*$ is denoted by w^R. The length of a word w is denoted by $|w|$. The cardinality of a finite set S is denoted $|S|$ and the power set of S is 2^S. A word $w \in \Sigma^*$ is a *subword* or *factor* of x if and only if there exist words $u, v \in \Sigma^*$ such that $x = uwv$. If $u = \varepsilon$, then w is a *prefix* of x. If $v = \varepsilon$, then w is a *suffix* of x.

A *nondeterministic finite automaton* (NFA) is a tuple $A = (Q, \Sigma, \delta, Q_0, F)$ where Q is a finite set of states, Σ is an alphabet, δ is a multi-valued transition function $\delta : Q \times \Sigma \to 2^Q$, $Q_0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of final states. We extend the transition function δ to $Q \times \Sigma^* \to 2^Q$ in the usual way. A word $w \in \Sigma^*$ is *accepted* by A if for some $q_0 \in Q_0$, $\delta(q_0, w) \cap F \neq \emptyset$ and the language recognized by A consists of all words accepted by A. An ε-NFA is the extension of an NFA where transitions can be labeled by the empty word

ε. If it is known that every ε-NFA has an equivalent NFA without ε-transitions with the same number of states. An NFA is a *deterministic finite automaton* (DFA) if $|Q_0| = 1$ and for all $q \in Q$ and $a \in \Sigma$, $\delta(q, a)$ either consists of one state or is undefined. The size of A, denoted $|A|$, is defined as the sum of the number of states and transitions of A, $|Q| + |\delta|$.

A *weighted finite-state transducer* [7] with weights in the $(\min, +)$-semiring $\mathbb{K}$ is a 6-tuple $T = (Q, \Sigma, \Delta, I, F, E)$ where Q is a finite set of states, Σ is the input alphabet, Δ is the output alphabet, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, $E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \times \mathbb{K} \times Q$ is a finite set of transitions with weights in $\mathbb{K}$. The size of T, denoted $|T|$, is defined as the sum of the number of states and transitions of T, $|Q| + |E|$.

A *path* or *computation* of T is a word π over the alphabet of transitions E

$$\pi = (p_1, u_1, v_1, w_1, q_1) \cdots (p_n, u_n, v_n, w_n, q_n)$$

with $q_i = p_{i+1}$ for $1 \leq i < n$. A path π from p to q is accepted if $p \in I$ and $q \in F$. Let $\omega : E^* \to \mathbb{K}$ be a weight function for paths defined by $\omega(\pi) = \sum_{i=1}^{n} w_i$, the sum of the weights of each transition in π. The label of a path π, denoted $\ell(\pi)$ is the pair of words (x, y) with $x = u_1 \cdots u_n$ and $y = v_1 \cdots v_n$. Let $w : \Sigma^* \times \Sigma^* \to \mathbb{K}$ be a weight function for labels (x, y) defined as the weight of the minimum weight accepted path labeled by (x, y),

$$w(x, y) = \min_{\pi \in E^*} \{\omega(\pi) \mid \ell(\pi) = (x, y)\}.$$

A function $d : \Sigma^* \times \Sigma^* \to \mathbb{N} \cup \{0\}$ is a *distance* if it satisfies for all $x, y \in \Sigma^*$

1. $d(x, y) = 0$ if and only if $x = y$,
2. $d(x, y) = d(y, x)$,
3. $d(x, z) \leq d(x, y) + d(y, z)$ for $z \in \Sigma^*$.

A distance between words can be extended to a distance between a word $w \in \Sigma^*$ and a language $L \subseteq \Sigma^*$ by

$$d(w, L) = \min\{d(w, w') \mid w' \in L\}.$$

We generalize this to a distance between two languages L_1 and L_2,

$$d(L_1, L_2) = \min\{d(w_1, w_2) \mid w_1 \in L_1, w_2 \in L_2\}.$$

The *inner distance* of a language L (also called the *self distance*) is the minimal distance between any two distinct words that both belong to L.

$$d(L) = \min\{d(w_1, w_2) \mid w_1, w_2 \in L, w_1 \neq w_2\}.$$

The *prefix distance* of x and y counts the number of symbols which do not belong to the longest common prefix of x and y. It is defined by

$$d_p(x, y) = |x| + |y| - 2 \cdot \max_{z \in \Sigma^*}\{|z| \mid x, y \in z\Sigma^*\}.$$

Similarly, the *suffix distance* of x and y counts the number of symbols which do not belong to the longest common suffix of x and y and is defined

$$d_s(x,y) = |x| + |y| - 2 \cdot \max_{z \in \Sigma^*} \{|z| \mid x, y \in \Sigma^* z\}.$$

The *subword distance* of x and y counts the number of symbols which do not belong to the longest common subword of x and y and is defined

$$d_f(x,y) = |x| + |y| - 2 \cdot \max_{z \in \Sigma^*} \{|z| \mid x, y \in \Sigma^* z \Sigma^*\}.$$

3 Edit Strings and Edit Systems

Edit systems, also called error systems, were first studied extensively by Kari and Konstantindis in [9] as a formalization for errors in terms of formal languages. Informally, an edit system is a formal language over the alphabet of edit operations and are used to model different types of errors. We present some basic definitions for edit systems and model the prefix, suffix, and subword distances using edit systems.

For an alphabet Σ, let $\mathcal{E}_\Sigma$ be the alphabet of *edit operations* over Σ,

$$\mathcal{E}_\Sigma = \{(a/b) \mid a, b \in \Sigma \cup \{\varepsilon\}, ab \neq \varepsilon\}.$$

We use $\mathcal{E}$ whenever Σ is obvious from the context. An *edit string* or *alignment* of two words is an element of $\mathcal{E}^*$. For an edit string $e = (a_1/b_1)(a_2/b_2)\cdots(a_n/b_n)$, we call $a_1a_2\cdots a_n$ the *input part* of e and $b_1b_2\cdots b_n$ the *output part.* We define the edit morphism to be the morphism $h : \mathcal{E}^* \to \Sigma^* \times \Sigma^*$ by $h(e) = (a_1\cdots a_n, b_1\cdots b_n)$.

We can define subsets of the alphabet of edit operations which correspond to the classical edit operations of substitution, insertion, and deletion and the identity operation by

- the set of substitution operations $\mathcal{S} = \{(a/b) \mid a \neq b, a, b \in \Sigma\}$,
- the set of insertion operations $\mathcal{I} = \{(\varepsilon/a) \mid a \in \Sigma\}$,
- the set of deletion operations $\mathcal{D} = \{(a/\varepsilon) \mid a \in \Sigma\}$,
- the set of identity operations $\mathcal{E}_0 = \{(a/a) \mid a \in \Sigma\}$.

We define a cost function $c : \mathcal{E} \to \mathbb{N}$ which assigns a cost to each element of the edit alphabet. Note that the standard definition of edit distance assigns the cost of every non-identity symbol $(a/b) \in \mathcal{E} \setminus \mathcal{E}_0$ to be 1. However, the prefix, suffix, and subword distances count each additional symbol in both words, so our cost function c is defined

- $c((a/a)) = 0$, for all $(a/a) \in \mathcal{E}_0$,
- $c((\varepsilon/a)) = 1$, for all $(\varepsilon, a) \in \mathcal{I}$,
- $c((a/\varepsilon)) = 1$, for all $(a/\varepsilon) \in \mathcal{D}$,
- $c((a,b)) = 2$, for all $(a/b) \in \mathcal{S}$.

The cost of an edit string $e = e_1 e_2 \cdots e_n$ is then the sum of the cost of each symbol

$$c(e) = \sum_{i=1}^{n} c(e_i).$$

A language defined over $\mathcal{E}$ is called an *edit system*. A regular edit system can be modeled by a finite automaton defined over $\mathcal{E}$. Such an edit system may also be realized as a finite-state transducer, where for each symbol $(a/b) \in \mathcal{E}$, the transitions on (a/b) are considered transition labels with a as the input part and b as the output part. Thus, a computation path of a finite state transducer over $\mathcal{E}$ corresponds with an edit string.

We now define the language of edit strings for the prefix distance L_p by

$$L_p = \mathcal{E}_0^* (\mathcal{E} \setminus \mathcal{E}_0)^*.$$

Informally, this is the set of edit strings with a prefix of identity operations followed by non-identity edit operations. We define the function $d'_p : \Sigma^* \times \Sigma^* \to \mathbb{N}$ on $x, y \in \Sigma^*$ by

$$d'_p(x, y) = \min_{e \in L_p} \{c(e) \mid h(e) = (x, y)\}.$$

In the following proposition, we show that $d'_p(x, y)$ is exactly the prefix distance of x and y.

Proposition 1. *Let $x, y \in \Sigma^*$ be two words. Then $d'_p(x, y) = d_p(x, y)$.*

Proof. Consider two words x and y with $x = px'$ and $y = py'$, where p is the longest common prefix of x and y. By definition, we have $d_p(x, y) = |x| + |y| - 2|p| = |x'| + |y'|$. Now consider an edit string $e \in L_p$ with $h(e) = (x, y)$. Since $e \in L_p$, we split e into two parts $e = e_0 e_1$, where $e_0 \in \mathcal{E}_0^*$ and $e_1 \in (\mathcal{E} \setminus \mathcal{E}_0)^*$. To minimize the cost $c(e)$, we require e_0 to be as long as possible and minimize the length of e_1, since $c(e_0) = 0$. Thus, e_0 corresponds to a string of identity operations for the longest common prefix p of x and y. This means that e_1 is the edit string such that $h(e_1) = (x', y')$. Thus, $c(e) = c(e_1)$ and since $e' \in (\mathcal{E} \setminus \mathcal{E}_0)^*$, we have $c(e') = |x'| + |y'|$. $\square$

We can similarly define the same notions for suffix and infix distances. Let $L_s = (\mathcal{E} \setminus \mathcal{E}_0)^* \mathcal{E}_0^*$ be the language of edit strings for suffix distance and let $L_f = (\mathcal{E} \setminus \mathcal{E}_0)^* \mathcal{E}_0^* (\mathcal{E} \setminus \mathcal{E}_0)^*$ be the language of edit strings for infix distance. We define the functions

$$d'_s(x, y) = \min_{e \in L_s} \{c(e) \mid h(e) = (x, y)\},$$
$$d'_f(x, y) = \min_{e \in L_f} \{c(e) \mid h(e) = (x, y)\}.$$

The following result is proved analogously as Proposition 1

Proposition 2. *Let $x, y \in \Sigma^*$ be two words. Then*

1. *$d_s(x, y) = d'_s(x, y)$, and*
2. *$d_f(x, y) = d'_f(x, y)$.*

4 Computing the Prefix Distance Between Regular Languages

We give a polynomial time algorithm to compute the prefix distance between two languages given by nondeterministic finite automata. Mohri [14] gave an algorithm for computing the edit distance between two regular languages by using weighted transducers. We use this approach by defining a weighted transducer with paths which correspond to edit strings in L_p.

We define the transducer $T_p = (Q, \Sigma, \Delta, I, F, E)$, by setting $Q = \{0, 1\}$, $\Delta = \Sigma$, $I = \{0\}$, $F = \{0, 1\}$, and the transition set E is given by

- $(0, a, a, 0, 0)$ for all $a \in \Sigma$,
- $(0, a, \varepsilon, 1, 1)$ for all $a \in \Sigma$,
- $(0, \varepsilon, a, 1, 1)$ for all $a \in \Sigma$,
- $(0, a, b, 2, 1)$, with $a \neq b$ for all $a, b \in \Sigma$,
- $(1, a, \varepsilon, 1, 1)$ for all $a \in \Sigma$,
- $(1, \varepsilon, a, 1, 1)$ for all $a \in \Sigma$,
- $(1, a, b, 2, 1)$, with $a \neq b$ for all $a, b \in \Sigma$.

The transducer is shown with $\Sigma = \{a, b\}$ in Figure 1. We claim that the transducer T_p takes as input some word w and outputs a word x such that any accepting computation path of T_p on w corresponds to an edit string in L_p which transforms w into x and that the weight of this path is the cost of the corresponding edit string. We prove this in the following lemma.

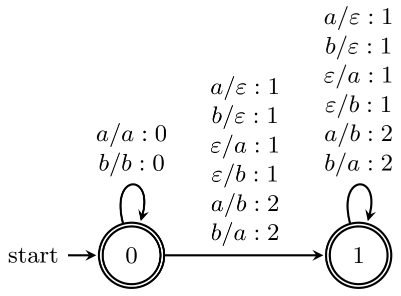

Fig. 1. The transducer T_p over the alphabet $\Sigma = \{a, b\}$.

Lemma 1. *The set of accepting paths of the transducer T_p over Σ corresponds to exactly the set of edit strings over Σ belonging to L_p. If π is an accepting path of T_p and e_π is the corresponding edit string, then the weight of π is $c(e_\pi)$.*

Proof. Let φ be a morphism $\varphi : E^* \to \mathcal{E}^*$ that maps a computation path of T_p to an edit string over $\mathcal{E}$ defined by $\varphi((p, a, b, i, q)) = (a/b)$. Consider an accepting path $\pi = \pi_1 \cdots \pi_n$ of T_p. Since both states 0 and 1 are final states, an accepting path may end in either state. If π ends in state 0, then π never leaves state 0 and π is of the form

$$\pi = (0, a_1, a_1, 0, 0) \cdots (0, a_n, a_n, 0, 0).$$

where $a_i \in \Sigma$ for all $1 \leq i \leq n$. Then $\varphi(\pi) = (a_1/a_1) \cdots (a_n/a_n) \in \mathcal{E}_0^*$. Note that every transition going from state 0 to itself has weight 0 and π therefore has weight 0. The cost of $\varphi(\pi)$ is also 0, as it consists only of identity operations, which have a cost of 0.

Now consider when π ends in state 1. Then π can be decomposed into $\pi = \pi_0\pi_1$ where for some $k < n$, we have

$$\begin{aligned}\pi_0 &= (0, a_1, a_1, 0, 0) \cdots (0, a_{k-1}, a_{k-1}, 0, 0)\\ \pi_1 &= (0, a_k, a'_k, i_k, 1)(1, a_{k+1}, a'_{k+1}, i_{k+1}, 1) \cdots (1, a_n, a'_n, i_n, 1)\end{aligned}$$

where $a_i \in \Sigma$ for $1 \leq i < k$ and $a_j, a'_j \in \Sigma \cup \{\varepsilon\}$ with $a_j \neq a'_j$ and $i_j \in \{1, 2\}$ for $k \leq j \leq n$. As in above, π_0 is a path which ends in state 0 and thus $\varphi(\pi_0)$ maps to a word over $\mathcal{E}_0$ with cost 0. The first transition in π_1 takes the machine to state 1. Since there are no transitions of the form $(1, a, a, 0, 1)$, the word $\varphi(\pi_1)$ contains no symbols from $\mathcal{E}_0$. In other words, $\varphi(\pi_1)$ is a word over the alphabet $\mathcal{E} \setminus \mathcal{E}_0$.

Now consider an edit string $e \in L_p$. We can decompose e into two parts $e = e_0e_1$ with $e_0 \in \mathcal{E}_0^*$ and $e_1 \in (\mathcal{E} \setminus \mathcal{E}_0)^*$. Then e_0 corresponds to a computation path that ends in state 0 and e_1 corresponds to a path which begins with a transition from state 0 to state 1 and ends on state 1. Thus any edit string in L_p corresponds to an accepting path in T_p.

It remains to be shown that the cost of $\varphi(\pi_1)$ is the same as the weight of the path π_1. By definition of T_p, each transition with a label a/ε or ε/a has weight 1 for all $a \in \Sigma$ and every transition with a label a/b with $a \neq b$ has weight 2. This corresponds to the costs assigned by the cost function c and thus the weight of π_1 is exactly the cost of $\varphi(\pi_1)$.

Thus, we have $\varphi(\pi) = \varphi(\pi_0)\varphi(\pi_1) \in \mathcal{E}_0^*(\mathcal{E} \setminus \mathcal{E}_0)^* = L_p$ and $w(\pi) = w(\pi_0) + w(\pi_1) = c(\varphi(\pi_0)) + c(\varphi(\pi_1)) = c(\varphi(\pi))$. □

Observe that if π is a minimum weight accepting path of T_p transforming a word w into a word x, then the weight of π is $d_p(w, x)$. This leads to the following result.

Proposition 3. *Let $x, y \in \Sigma^*$. Then the weight $w(x, y)$ of x and y in T_p is exactly $d_p(x, y)$.*

Proof. Recall that, by definition, the weight of a pair of words (x, y) in T_p is the minimum weight of all accepting paths of T_p with label (x, y). By Lemma 1, each path π in T_p corresponds to an edit string e_π in L_p and has weight equivalent to $c(e_\pi)$. Thus, we have

$$w(x, y) = \min_{e \in L_p} \{c(e) \mid h(e) = (x, y)\},$$

which is exactly $d_p(x, y)$ by Proposition 1. □

Now we move to the main result. We wish to compute the prefix distance of two given regular languages L_1 and L_2. To do this, we compute a transducer for which pairs of words (x, y) with $x \in L_1$ and $y \in L_2$ have weight equal to $d_p(x, y)$. Let A_1 and A_2 be finite automata recognizing regular languages L_1 and L_2, respectively. Recall that an unweighted finite automaton over Σ may be viewed as a weighted transducer with input and output alphabets Σ and in which each transition labeled by $a \in \Sigma$ is labeled by a/a and has weight 0.

The composition $T_1 \otimes T_2 = (Q, \Sigma, \Gamma, I, F, E)$ of two weighted transducers $T_1 = (Q_1, \Sigma, \Delta, I_1, F_1, E_1)$ and $T_2 = (Q_2, \Delta, \Gamma, I_2, F_2, E_2)$ is defined by $Q = Q_1 \times Q_2$, $I = I_1 \times I_2$, $F = Q \cap (F_1 \times F_2)$, and the transition set E consists of transitions of the form $((q_1, q_1'), a, c, w_1 + w_2, (q_2, q_2'))$ for each transition $(q_1, a, b, w_1, q_2) \in E_1$ and $(q_1', b, c, w_2, q_2') \in E_2$. The composition $T_1 \otimes T_2$ can be computed in $O(|T_1||T_2|)$ time.

Now consider the weighted transducer $T = A_1 \otimes T_p \otimes A_2$. We show in the following lemma that for $x \in L_1$ and $y \in L_2$, the weight of (x, y) in T is $d_p(x, y)$.

Theorem 1. *Let L_1 and L_2 be regular languages recognized by NFAs A_1 and A_2, respectively. If $x \in L_1$ and $y \in L_2$, then (x, y) is the label of an accepting path of $T = A_1 \otimes T_p \otimes A_2$ and the weight of (x, y) in T is $d_p(x, y)$.*

Proof. Consider two words $x \in L_1$ and $y \in L_2$. By definition of composition, for any accepting path of T, the input part must be recognized by A_1, the output part must be recognized by A_2, and the path must correspond to an edit string in the language L_p. Thus, there is an accepting path π of T with label $\ell(\pi) = (x, y)$ which corresponds to an edit string $e_\pi \in L_p$ with $h(e_\pi) = (x, y)$. By Proposition 3, the weight $w(x, y)$ of T must be $d_p(x, y)$. □

This result implies that the weight of the minimal weight path of $A_1 \otimes T_p \otimes A_2$ is the prefix distance between $L(A_1)$ and $L(A_2)$. This leads us to an efficient algorithm to compute the prefix distance between two regular languages.

Theorem 2. *For given NFAs A_1 and A_2 recognizing the languages L_1 and L_2, respectively, the value $d_p(L_1, L_2)$ can be computed in polynomial time.*

Proof. Recall that the prefix distance between L_1 and L_2 is defined

$$d_p(L_1, L_2) = \min\{d_p(x, y) \mid x \in L_1, y \in L_2\}.$$

By Theorem 1, for two words $x \in L_1$ and $y \in L_2$, the weight of (x, y) in the weighted transducer $T = A_1 \otimes T_p \otimes A_2$ is $d_p(x, y)$. By definition, this is the weight of the minimal weight path with label (x, y) accepted by T. Then the weight of a minimal weight accepting path in T from the initial state to a final state must be $d_p(L_1, L_2)$ by definition.

With T_p fixed, in the worst case, the composition of the weighted transducer $T = A_1 \otimes T_p \otimes A_2$ can be computed in time $O(|A_1||A_2|)$ and the size of T is $O(|A_1||A_2|)$ [7]. To compute $d_p(L_1, L_2)$, we compute T and find the shortest path from the initial state of T to a final state of T. Since there are no negative cycles, we use Dijkstra's single-source shortest path algorithm, which has running time

$O(|E| + |Q| \log |Q|)$, where E is the transition set of T and Q is the state set of T [5]. Thus, $d_p(L_1, L_2)$ can be computed in polynomial time. $\square$

In Proposition 2, we have characterized the suffix distance and the subword distance, respectively, in terms of the edit systems L_s and L_f. By using a weighted transducer based construction analogous to the one used for the prefix distance in Theorem 2, we can get a polynomial time algorithm for computing the suffix distance and subword distance between regular languages.

Theorem 3. *For given NFAs A_1 and A_2 recognizing the languages L_1 and L_2, respectively,*

1. *$d_s(L_1, L_2)$ can be computed in polynomial time, and*
2. *$d_f(L_1, L_2)$ can be computed in polynomial time.*

5 Computing the Inner Prefix Distance of a Regular Language

Kari et al. [10] give an algorithm for computing the inner Hamming distance of a regular language using a similar approach with NFAs over the edit alphabet. In the development of the algorithm, a crucial observation was the necessity of excluding all edit strings with cost 0, since $d(x, y) = 0$ if and only if $x = y$. Thus, for our algorithm, we need to modify the language L_p to exclude all edit strings with cost 0 and define a corresponding weighted transducer.

We define the language of edit strings for the prefix distance excluding all edit strings which result in identity,

$$L_p^{(1)} = \mathcal{E}_0^*(\mathcal{E} \setminus \mathcal{E}_0)^+.$$

The language $L_p^{(1)}$ is almost exactly the same as the language L_p with the exception that no edit strings $e \in \mathcal{E}_0^*$ are in $L_p^{(1)}$. That is, every edit string in $L_p^{(1)}$ must contain at least one symbol with nonzero cost.

Now, we define the transducer $T_p^{(1)} = (Q, \Sigma, \Delta, I, F, E)$ by choosing $Q = \{0, 1\}$, $\Delta = \Sigma$, $I = \{0\}$, $F = \{1\}$, and the transition set E is as in the definition of T_p. The transducer $T_p^{(1)}$ is the transducer T_p with the modification that state 1 is the sole final state. The transducer $T_p^{(1)}$ defined over the alphabet $\{a, b\}$ is shown in Figure 2. We show in the following lemma that $T_p^{(1)}$ realizes $L_p^{(1)}$.

Lemma 2. *The set of accepting paths of the transducer $T_p^{(1)}$ over Σ corresponds exactly to the language edit strings $L_p^{(1)}$. If π is an accepting path of of $T_p^{(1)}$ and e_π is the corresponding edit string, then the weight of π is $c(e_\pi)$.*

Proof. Consider an accepting path $\pi = \pi_1 \cdots \pi_n$ of $T_p^{(1)}$. Recall from the proof of Lemma 1 the definition of φ and observe that since 0 is not a final state of $T_p^{(1)}$, π must be of the form $\pi = \pi_0\pi_1$, with $\varphi(\pi_0) \in \mathcal{E}_0^*$ and $\varphi(\pi_1) \in (\mathcal{E} \setminus \mathcal{E}_0)^*$. Thus, $\varphi(\pi)$ must contain at least one non-identity operation and $c(\varphi(\pi)) > 0$.

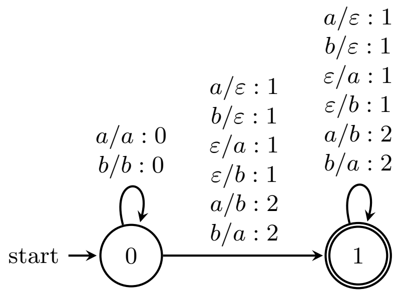

Fig. 2. The transducer $T_p^{(1)}$ over the alphabet $\Sigma = \{a, b\}$.

Now consider an edit string $e \in L_p^{(1)}$. We can decompose e into two parts $e = e_0 e_1$ with $e_0 \in \mathcal{E}_0^*$ and $e_1 \in (\mathcal{E} \setminus \mathcal{E}_0)^+$. Then e_0 corresponds to a computation path that ends in state 0 and e_1 corresponds to a path which begins with a transition from state 0 to state 1 and ends on state 1. Thus any edit string in L_p corresponds to an accepting path in T_p.

By the same argument from the proof of Lemma 1, the weight of an accepting path π of $T_p^{(1)}$ is exactly the cost of the edit string $\varphi(\pi)$. □

As was the case for T_p, for each pair of words (x, y) with an accepting path in $T_p^{(1)}$, the weight of (x, y) is exactly $d_p(x, y)$ by the same argument as in the proof of Proposition 3. This leads us to the analogue of Theorem 1 for $T_p^{(1)}$.

Theorem 4. *Let L be a regular language recognized by a finite automaton A. Then for $x, y \in L$ with $x \neq y$, (x, y) is an accepting path of $T = A \otimes T_p^{(1)} \otimes A$ and the weight of $(x, y) \in T$ is $d_p(x, y)$.*

Proof. Let $x, y \in L$ and consider the weighted transducer $T = A \otimes T_p^{(1)} \otimes A$. By definition of composition, for any accepting path π of T, the input and output labels must be words recognized by A and the path must correspond to an edit string in $L_p^{(1)}$. Thus, there is an accepting path π of T with label $\ell(\pi) = (x, y)$ which corresponds to an edit string $e_\pi \in L_p^{(1)}$ with $h(e_\pi) = (x, y)$. Furthermore, since $e_\pi \in L_p^{(1)}$, we have $x \neq y$. Thus, by Lemma 2, the weight $w(x, y)$ of T must be $d_p(x, y)$. □

From this it follows that can compute the inner prefix distance of a regular language by computing the appropriate weighted transducer and finding the minimal weight path from its initial state to one of its final states.

Theorem 5. *For a given NFA A recognizing the language L, the value $d_p(L)$ is computable in polynomial time.*

Proof. By Theorem 4, for $x, y \in L$, the weight of (x, y) in the weighted transducer $T = A \otimes T_p^{(1)} \otimes A$ is $d_p(x, y)$. Then the weight of a minimal weight accepting path in T must be $d_p(L)$ by definition.

Then, as in Theorem 2, the transducer T can be computed in time $O(|A|^2)$ in the worst case and the size of T is $O(|A|^2)$ [7]. Since there are no negative cycles, we can compute the minimal weight path of T in time $O(|E|+|Q|\log|Q|)$, where E is the transition set of T and Q is the state set of T, by using Dijkstra's algorithm [5]. Thus, $d_p(L)$ can be computed in polynomial time. □

We have shown how to compute the inner prefix distance of a regular language. We can make similar modifications to the edit systems L_s and L_f and construct transducers which model those edit systems. Such an edit system can be defined for the suffix distance by

$$L_s^{(1)} = (\mathcal{E} \setminus \mathcal{E}_0)^+ \mathcal{E}_0^*.$$

The case of subword distance is slightly more complicated, as we require at least one edit operation with nonzero weight. For an edit string e, such an operation can occur as either a prefix or a suffix but we cannot require that there is a symbol with nonzero weight in both the prefix and suffix. Thus, we can define the edit system $L_f^{(1)}$ by

$$L_f^{(1)} = ((\mathcal{E} \setminus \mathcal{E}_0)^* \mathcal{E}_0^* (\mathcal{E} \setminus \mathcal{E}_0)^+) \cup ((\mathcal{E} \setminus \mathcal{E}_0)^+ \mathcal{E}_0^* (\mathcal{E} \setminus \mathcal{E}_0)^*).$$

Then using an analogous approach as for the prefix distance, it is possible to compute the inner suffix and subword distances of a regular language in polynomial time.

Theorem 6. *For a given NFA A recognizing the language L,*

1. *$d_s(L)$ is computable in polynomial time, and*
2. *$d_f(L)$ is computable in polynomial time.*

6 Conclusion

We have shown how to compute the prefix distance of two regular languages in polynomial time by using weighted transducers. We have also used this algorithm to compute the inner prefix distance of a regular language in polynomial time. These algorithms can also be applied to compute the suffix and subword distances for regular languages.

One direction for future research is computing prefix, suffix, and subword distances between non-regular languages. It is known that computing distances between context-free languages is undecidable [4]. However, Han et al. [8] gave an algorithm for computing the edit distance between a regular language and a context-free language. For prefix, suffix, and subword distances, the problem of computing the distance between a regular language and a context-free language or subclasses of context-free languages remains open.

References

1. Benedikt, M., Puppis, G., Riveros, C.: Bounded repairability of word languages. Journal of Computer and System Sciences **79**(8) (2013) 1302–1321
2. Benedikt, M., Puppis, G., Riveros, C.: The per-character cost of repairing word languages. Theoretical Computer Science **539** (2014) 38–67
3. Bruschi, D., Pighizzini, G.: String Distances and Intrusion Detection: Bridging the Gap Between Formal Languages and Computer Security. RAIRO Informatique Théorique et Applications **40** (2006) 303–313
4. Choffrut, C., Pighizzini, G.: Distances between languages and reflexivity of relations. Theoretical Computer Science **286**(1) (2002) 117–138
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 2nd edn. MIT Press, Cambridge, Massachusetts (2001)
6. Deza, M.M., Deza, E.: Encyclopedia of Distances. Springer Berlin Heidelberg (2009)
7. Droste, M., Kuich, W., Vogler, H.: Handbook of Weighted Automata. Springer-Verlag Berline Heidelberg (2009)
8. Han, Y.S., Ko, S.K., Salomaa, K.: The Edit-Distance Between a Regular Language and a Context-Free Language. International Journal of Foundations of Computer Science **24**(07) (nov 2013) 1067–1082
9. Kari, L., Konstantinidis, S.: Descriptional complexity of error/edit systems. Journal of Automata, Languages and Combinatorics **9**(2/3) (2004) 293–309
10. Kari, L., Konstantinidis, S., Perron, S., Wozniak, G., Xu, J.: Computing the Hamming Distance of a Regular Language in Quadratic Time. WSEAS Transactions on Information Science & Applications **1**(1) (2004) 445–449
11. Konstantinidis, S.: Computing the edit distance of a regular language. Information and Computation **205**(9) (2007) 1307–1316
12. Konstantinidis, S., Silva, P.V.: Computing Maximal Error-detecting Capabilities and Distances of Regular Languages. Fundamenta Informaticae **101** (2010) 257–270
13. Kutrib, M., Meckel, K., Wendlandt, M.: Parameterized Prefix Distance between Regular Languages. In: SOFSEM 2014: Theory and Practice of Computer Science. (2014) 419–430
14. Mohri, M.: Edit-distance of weighted automata: General definitions and algorithms. International Journal of Foundations of Computer Science **14**(6) (2003) 957–982
15. Pighizzini, G.: How Hard Is Computing the Edit Distance? Information and Computation **165**(1) (2001) 1–13
16. Shallit, J.: A second course in formal languages and automata theory. Cambridge University Press, Cambridge, MA (2009)
17. Yu, S.: Regular languages. In Rozenberg, G., Salomaa, A., eds.: Handbook of Formal Languages. Springer-Verlag, Berlin, Heidelberg (1997) 41–110