An efficient algorithm for computing the edit distance of a regular language via input-altering transducers

Lila Kari[1], Stavros Konstantinidis[2], Steffen Kopecki[1,2], Meng Yang[2]

[1] Department of Computer Science, University of Western Ontario, London, Ontario, Canada, lila@csd.uwo.ca, steffen@csd.uwo.ca

[2] Department of Mathematics and Computing Science, Saint Mary's University, Halifax, Nova Scotia, Canada, s.konstantinidis@smu.ca, meyang.mike@gmail.com

Abstract. We revisit the problem of computing the edit distance of a regular language given via an NFA. This problem relates to the inherent maximal error-detecting capability of the language in question. We present an efficient algorithm for solving this problem which executes in time $O(r^2n^2d)$, where r is the cardinality of the alphabet involved, n is the number of transitions in the given NFA, and d is the computed edit distance. We have implemented the algorithm and present here performance tests. The correctness of the algorithm is based on the result (also presented here) that the particular error-detection property related to our problem can be defined via an input-altering transducer.

Keywords. algorithms, automata, complexity, edit distance, implementation, transducers, regular language

1 Introduction

The edit distance of a language L with at least two words—also referred to as inner edit distance of L—is the minimum edit distance between any two different words in L. In [14], the author considers the problem of computing the edit distance of a regular language, which is given via a nondeterministic finite automaton (NFA), or a deterministic finite automaton (DFA). For a given automaton $\mathbf{a}$ with n transitions and an alphabet of r symbols, the algorithm proposed in [14] has worst-case time complexity

$$O(r^2n^2q^2(q+r)), \tag{1}$$

where, in fact, q is either the number of states in $\mathbf{a}$ (if $\mathbf{a}$ is a DFA), or the square of the number of states in $\mathbf{a}$ (if $\mathbf{a}$ is an NFA). If the size of the alphabet is ignored and the automaton in question has only states that can be reached from the start state, then the number of states is $O(n)$ and the worst-case time complexity shown in (1) can be written as

$$O(n^5) \text{ for DFAs, and } O(n^8) \text{ for NFAs.} \tag{2}$$

In this paper, motivated by the question of whether certain error-detection properties can be defined via input-altering transducers, we obtain an efficient algorithm to compute the edit distance of a regular language given via

an NFA with n transitions—see theorem 12. The algorithm, which is called `DistBestInpAlter`, has worst-case time complexity

$$O(n^2 d), \tag{3}$$

where d is the computed distance, which is a significant improvement over the original algorithm in [14].

We note that an approach of computing the edit distance problem via the error-detection property is discussed briefly in [15]. A similar approach can be used for the edit distance problem via the error-correction property. The new algorithm `DistBestInpAlter` (see theorem 12) is based on (a) the new result that the error-detection property related to our problem is definable via an efficient input altering transducer—see theorem 11, and (b) the observation that the preliminary error-detection-based algorithm can be made significantly more efficient by cleverly utilizing the above new result. For clarity of presentation we present in detail not only the new algorithm, but also the intermediate versions, all of which have been implemented in Python using the well maintained library FAdo for automata [7]. We have also tested all versions experimentally, and we discuss in this paper the outcomes of the tests showing that, not only in theory, but also in practice algorithm `DistBestInpAlter` is clearly more performant.

We note that some related problems involving distances between words and languages can be found in [20,25] (edit distance between a word and a language), and in [3, 8, 9, 12, 18] (various distances between languages). The problem considered here is technically different, as the desired distance involves different words within the same language.

The paper is organized as follows. The next section contains basic notions on languages, finite-state machines and edit-strings, and a few preliminary lemmata. Section 3 describes the approach of computing the desired edit distance via the concepts of error-detection and -correction. Section 4 first presents the new result that the error-detection property in question is definable via an efficient input-altering transducer—see theorem 11—and then, the main result, algorithm `DistBestInpAlter` in theorem 12. Section 5 discusses the implementation and testing of the main algorithm and its intermediate versions. The last section contains a few concluding remarks and questions for future research.

2 Notation, background and preliminary results

Most of the basic notions presented here can be found in various texts such as [4, 21, 22, 26, 28].

2.1 Sets, words, languages, channels

If S is any set, the expression $|S|$ denotes the cardinality of S. When there is no risk of confusion we denote a singleton set $\{u\}$ simply as u. For example, $S \cup u$ is the union of S and $\{u\}$. We use standard basic notation and terminology for alphabets, words and languages—see [17], for instance. For example, Σ denotes

an alphabet, Σ^+ the set of nonempty words, λ the empty word, $|w|$ the length of the word w. We use the concepts of (formal) language and concatenation between words, or languages, in the usual way. We say that w is an *L-word* if $w \in L$ and L is a language.

A binary word *relation* ρ on Σ^* is any subset of $\Sigma^* \times \Sigma^*$. The *domain* of ρ is $\{u \mid (u, v) \in \rho \text{ for some } v \in \Sigma^*\}$. A *channel* γ is a binary relation on Σ^* that is domain-preserving (or input-preserving); that is, $\gamma \subseteq \Sigma^* \times \Sigma^*$ and $(w, w) \in \gamma$ for all words w in the domain of γ. When $(u, v) \in \gamma$ we say that u can be received as v via the channel γ, or v is a possible output of γ when u is used as input. If $v \neq u$ then we say that u can be received with errors (via γ). Here we only consider the channel $\mathrm{sid}(k)$, for some $k \in \mathbb{N}$, such that $(u, v) \in \mathrm{sid}(k)$ if and only if v can be obtained by applying at most k errors in u, where an error could be a deletion of a symbol in u, a substitution of a symbol in u with another symbol, or an insertion of a symbol in u—see further below for a more rigorous definition via edit-strings.

2.2 NFAs and transducers

A nondeterministic finite automaton with empty transitions, *λ-NFA* for short, or just *automaton*, is a quintuple $\mathbf{a} = (Q, \Sigma, T, s, F)$ such that Q is the set of states, Σ is the alphabet, $s \in Q$ is the start (or initial) state, $F \subseteq Q$ is the set of final states, and $T \subseteq Q \times (\Sigma \cup \lambda) \times Q$ is the finite set of transitions. Let (p, x, q) be a transition of $\mathbf{a}$. Then x is called the *label* of the transition, and we say that p has an *outgoing* transition (with label x). We also use the notation

$$p \xrightarrow{x} q$$

for a transition (p, x, q). The λ-NFA $\mathbf{a}$ is called an *NFA*, if no transition label is empty, that is, $T \subseteq Q \times \Sigma \times Q$. A deterministic finite automaton, *DFA* for short, is a special type of NFA where there is no state p having two outgoing transitions with different labels.

A *path* of $\mathbf{a}$ is a finite sequence of transitions of the form

$$(p_0, x_1, p_1), (p_1, x_2, p_2), \ldots, (p_{\ell-1}, x_\ell, p_\ell),$$

for some nonnegative integer ℓ. The word $x_1 \cdots x_\ell$ is called the *label* of the path. We write $p_0 \xrightarrow{x}{}^* p_\ell$ to indicate that there is a path with label x from p_0 to p_ℓ. A path as above is called *accepting* if p_0 is the start state and p_ℓ is a final state. The *language accepted* by $\mathbf{a}$, denoted as $L(\mathbf{a})$, is the set of labels of all the accepting paths of $\mathbf{a}$. The automaton $\mathbf{a}$ is called *trim*, if every state appears in some accepting path of $\mathbf{a}$.

A (finite) *transducer* [4, 28] is a sextuple $\mathbf{t} = (Q, \Sigma, \Gamma, T, s, F)$ such that Q, s, F are exactly the same as those in λ-NFAs, Σ is now called the input alphabet, Γ is the output alphabet, and $T \subseteq Q \times \Sigma^* \times \Gamma^* \times Q$ is the finite set of transitions. We write $(p, u/v, q)$, or $p \xrightarrow{u/v} q$ for a transition—the label here is (u/v), with u being the input and v being the output label. The concepts

of path, accepting path, and trim transducer are similar to those in λ-NFAs. However, the label of a transducer path $(p_0, x_1/y_1, p_1), \ldots, (p_{\ell-1}, x_\ell/y_\ell, p_\ell)$ is the pair $(x_1 \cdots x_\ell, y_1 \cdots y_\ell)$ of the two words consisting of the input and output labels in the path, respectively. The *relation realized* by the transducer $\mathbf{t}$, denoted as $R(\mathbf{t})$, is the set of labels in all the accepting paths of $\mathbf{t}$. We write $\mathbf{t}(x)$ for the set of possible outputs of $\mathbf{t}$ on input x, that is, $y \in \mathbf{t}(x)$ if and only if $(x, y) \in R(\mathbf{t})$. The transducer is called functional, if the relation $R(\mathbf{t})$ is a function, that is, $\mathbf{t}(x)$ consists of at most one word, for all inputs x. The transducer $\mathbf{t}$ is said to be in *standard form*, if each transition $(p, u/v, q)$ is such that $u \in (\Sigma \cup \lambda)$ and $v \in (\Gamma \cup \lambda)$. We note that every transducer is effectively equivalent to one (realizing the same relation, that is) in standard form.

If $\mathbf{m}$ is an automaton, or a transducer in standard form, then the *size* of $\mathbf{m}$, denoted by $|\mathbf{m}|$, is the number of states plus the number of transitions in $\mathbf{m}$.

2.3 Edit strings and edit distance.

The alphabet E_Σ of the *(basic) edit operations*, which depends on the alphabet Σ of ordinary symbols, consists of all symbols (x/y) such that $x, y \in \Sigma \cup \{\lambda\}$ and at least one of x and y is in Σ. If $(x/y) \in E_\Sigma$ and x is not equal to y then (x/y) is called an *error* [11]. The edit operations (a/b), (λ/a), (a/λ), where $a, b \in \Sigma - \{\lambda\}$ and $a \neq b$, are called *substitution, insertion, deletion*, respectively. We write (λ/λ) for the empty word over the alphabet E_Σ. We note that λ is used as a formal symbol in the elements of E_Σ. For example, if $a, b \in \Sigma$ then $(\lambda/a)(b/b) \neq (b/a)(\lambda/b)$. The elements of E_Σ^* are called *edit strings*. The *weight* of an edit string h, denoted as $\text{weight}(h)$, is the number of errors occurring in h. For example, for

$$g = (a/a)(a/\lambda)(b/b)(b/a)(b/b), \tag{4}$$

$\text{weight}(g) = 2$. The *input* and *output* parts of an edit string $h = (x_1/y_1) \cdots (x_n/y_n)$ are the words (over Σ) $x_1 \cdots x_n$ and $y_1 \cdots y_n$, respectively. We write $\text{inp}(h)$ for the input part and $\text{out}(h)$ for the output part of h. For example, for the g shown above, $\text{inp}(g) = aabbb$ and $\text{out}(g) = abab$. The *inverse of an edit string* h is the edit string resulting by inverting the order of the input and output parts in every edit operation in h. For example, the inverse of g shown above is

$$(a/a)(\lambda/a)(b/b)(a/b)(b/b).$$

The channel $\text{sid}(k)$ can be defined more rigorously via edit strings:

$$\text{sid}(k) = \{(u, v) \mid u = \text{inp}(h), v = \text{out}(h), \text{for some } h \in E_\Sigma^* \text{ with } \text{weight}(h) \leq k\}.$$

The *edit (or Levenshtein) distance* [16] between two words u and v, denoted by $\delta(u, v)$, is the smallest number of errors (substitutions, insertions and deletions) that can be used to transform u to v. More formally,

$$\delta(u, v) = \min\{\text{weight}(h) \mid h \in E_\Sigma^*, \text{ inp}(h) = u, \text{ out}(h) = v\}.$$

We say that an edit string h *realizes the edit distance between two words* u and v, if $\text{weight}(h) = \delta(u, v)$ and $\text{inp}(h) = u$ and $\text{out}(h) = v$. For example, for $\Sigma = \{a, b\}$, we have that $\delta(ababa, babbb) = 3$ and the edit string

$$h = (a/\lambda)(b/b)(a/a)(b/b)(a/b)(\lambda/b)$$

realizes $\delta(ababa, babbb)$. Note that several edit strings can realize the distance $\delta(u, v)$. If L is a language containing at least two words then the edit distance of L is

$$\delta(L) = \min\{\delta(u, v) \mid u, v \in L \text{ and } u \neq v\}.$$

Testing whether a given NFA accepts at least two words is not a concern in this paper, but we note that this can be done efficiently (in linear time via a breadth first search type algorithm) [27].

Definition 1. An edit string h of nonzero weight is called *reduced*, if (a) the first error in h is not an insertion, and (b) if the first error in h is a deletion of the form (a/λ), then the first non-deletion edit operation (x/y) that follows (a/λ) in h (if any) is such that $y \neq a$.

Lemma 2. *Let x, y, u, v be words. The following statements hold true.*

1. $\delta(xuy, xvy) = \delta(u, v)$.
2. *If $v <_p u$ then $\delta(u, v) = |u| - |v|$.*
3. *If $u \neq v$, then there is a reduced edit string h realizing $\delta(u, v)$.*

Proof. The first statement already appears in [16]. The second statement is rather folklore, but we provide a proof here for the sake of completeness. Let $u = \sigma_1 \cdots \sigma_n$ and $v = \sigma_1 \cdots \sigma_m$, where $m, n \in \mathbb{N}_0$ and $m < n$ and all σ_i's are in Σ. Then, the edit string

$$h = (\sigma_1/\sigma_1) \cdots (\sigma_m/\sigma_m)(\sigma_{m+1}/\lambda) \cdots (\sigma_n/\lambda)$$

has weight $n-m$ and $\text{inp}(h) = u$ and $\text{out}(h) = v$. We show that h realizes $\delta(u, v)$ by proving that, for any edit string g realizing $\delta(u, v)$, $\text{weight}(g) = n-m$. Indeed, first note that $\text{weight}(g) \leq \text{weight}(h) = n - m$. Let i and d be the number of insertions and deletions in g. Then $|v| = |u| + i - d$, which implies $n - m = d - i$. Now $\text{weight}(g) \geq d + i \geq d - i = n - m$, as required.

For the third statement, let g_0 be any edit string realizing $\delta(u, v)$. The following process can be used to obtain the required reduced edit string h.

1. If the first error in g_0 is a substitution, then $h = g_0$.
2. If the first error in g_0 is an insertion, then set g_0 to the inverse of g_0 and continue with the next step.
3. If the first error in g_0 is a deletion (a/λ), then g_0 is of the form

$$g_0 = (e_1 \cdots e_r)(a/\lambda)(a_1/\lambda) \cdots (a_d/\lambda)g_0',$$

where the e_i's are non-errors, $d \in \mathbb{N}_0$ and each (a_j/λ) is a deletion, and g_0' does not start with a deletion. If g_0' is empty or starts with an edit operation (x/y) in which $y \neq a$, then the required h is g_0. If g_0' starts with an edit operation (x/a), then it is of the form $g_0' = (x/a)g_1'$, and the edit string

$$g_1 = (e_1 \cdots e_r)(a/a)(a_1/\lambda) \cdots (a_d/\lambda)(x/\lambda)g_1',$$

realizes $\delta(u, v)$, as $weight(g_1) = weight(g_0)$. The process now continues from the first step using g_1 for g_0.

As the edit string g_0 is finite, the above process terminates with a reduced edit string h, as required. □

The bound $D_{\mathbf{a}}$ in the next lemma comes from [14]. It is always less than or equal to the number of states in the NFA $\mathbf{a}$. Moreover, there are NFAs for which this bound is tight—see Fig. 3 in Section 5.

Lemma 3. *For every NFA* $\mathbf{a}$ *accepting at least two words we have that*

$$\delta(L(\mathbf{a})) \leq D_{\mathbf{a}},$$

where $D_{\mathbf{a}}$ *is the number of states in the longest path in* $\mathbf{a}$ *from the start state having no repeated state.*

However, the bound $D_{\mathbf{a}}$ is of no use in our context, as the problem of determining the length of a longest path in a given automaton, or a graph in general, is NP-complete since an algorithm solving this problem can be used to decide the existence of a Hamiltonian path; see for example [23]. There are many ways to obtain an efficiently computable upper bound on the edit distance of $L(\mathbf{a})$ that is always at most equal to the number of states in $\mathbf{a}$. For example, that distance is always less than are equal to the distance of two shortest accepted words. We agree to use this as a working upper bound:

Lemma 4. *For every NFA* $\mathbf{a}$ *accepting at least two words we have that*

$$\delta(L(\mathbf{a})) \leq B_{\mathbf{a}},$$

where $B_{\mathbf{a}}$ *is the edit distance of two shortest words in* $L(\mathbf{a})$.

3 Edit distance via error-detection and -correction

In [15], the authors discuss a conceptual method for computing integral distances of regular languages—integral means that all distance values are positive integers—via the property of error-detection. In this section, we review that method and produce a *concrete* preliminary algorithm for computing the edit distance of a regular language. We also present here a similar method, via the property of error-correction, and the algorithm it entails. In fact this latter algorithm *estimates* the edit distance, as it returns two integers, differing by

1, one of which is the exact edit distance value. Both algorithms have been implemented as will be discussed in section 5.

A language L is *error-detecting for* a channel γ, if no L-word can be received as a different L-word via γ, that is, for any words u and v,

$$u, v \in L \text{ and } (u, v) \in \gamma \;\; \rightarrow \;\; u = v$$

Note: The definition of error-detection in [13] uses $L \cup \{\lambda\}$ instead of L in the above formula. This slight change makes the presentation here simpler and has no bearing on any existing results regarding error-detecting languages.

A language L is *error-correcting for* a channel γ, if no two different L-words can result into the same word via γ, that is,

$$u, v \in L \text{ and } (u, z), (v, z) \in \gamma \;\; \rightarrow \;\; u = v$$

This property of L ensures that any output z of the channel can be corrected to a unique L-word.

Remark 5. The error-detection method of [15], as well as the error-correction method, are based on the following observations, where $\mathbf{a}$ is an NFA and $\mathbf{t}$ is an input-preserving transducer.

1. A language L is error-detecting for $\text{sid}(m)$, if and only if $\delta(L) > m$.
2. A language L is error-correcting for $\text{sid}(k)$, if and only if $\delta(L) > 2k$, [16].
3. A language L is error-detecting for a channel γ if and only if the relation

$$\gamma \cap (L \times \Sigma^*) \cap (\Sigma^* \times L) \tag{5}$$

   is functional [13].
4. A language L is error-correcting for a channel γ if and only if the relation

$$\gamma^{-1} \cap (\Sigma^* \times L) \tag{6}$$

   is functional [13].
5. Suppose $\mathbf{a}$ accepts L and $\mathbf{t}$ realizes γ. A transducer, denoted as $(\mathbf{t} \downarrow \mathbf{a} \uparrow \mathbf{a})$, that realizes relation (5) can be constructed in time $O(|\mathbf{t}||\mathbf{a}|^2)$. Moreover, a transducer, denoted as $(\mathbf{t}^{-1} \uparrow \mathbf{a})$, that realizes relation (6) can be constructed in time $O(|\mathbf{t}||\mathbf{a}|)$ [13].
6. There is a quadratic time algorithm that decides whether a given transducer is functional [1, 2].

Using the above observations, we present first the error-detection-based algorithm for computing the desired edit distance, and further below the algorithm based on error-correction.

Algorithm `DistErrDetect`

0. Input: NFA $\mathbf{a}$
1. Let $B_{\mathbf{a}}$ be edit distance bound in Lemma 4
2. Let $\min \leftarrow 1$ and $\max \leftarrow B_{\mathbf{a}} - 1$
3. Perform binary search to find the largest k in $\{\min, \ldots, \max\}$ for which $L(\mathbf{a})$ is error-detecting for $\mathrm{sid}(k)$ as follows:
   while $(\min \leq \max)$
   a) Let $k \leftarrow \lfloor (\min + \max)/2 \rfloor$
   b) Construct transducer $\mathbf{t}$ realizing the channel $\mathrm{sid}(k)$—see Fig. 1
   c) Construct the transducer $\mathbf{t}' \leftarrow (\mathbf{t} \downarrow \mathbf{a} \uparrow \mathbf{a})$
   d) If ($\mathbf{t}'$ is functional) let $\min \leftarrow k + 1$
      Else let $\max \leftarrow k - 1$
4. **return** min

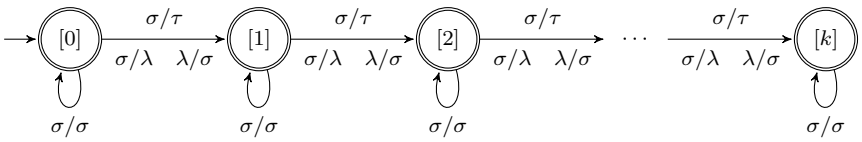

Figure 1: An input-preserving transducer realizing the channel $\mathrm{sid}(k)$. Each edge label σ/σ represents many transitions, one for each symbol σ of the alphabet, and similarly for σ/λ and λ/σ. Each edge label σ/τ represents many transitions, one for each pair of distinct symbols σ and τ from the alphabet. Thus, if the alphabet size is r, then the size of the transducer is $O(r^2 k)$, as $r, k \to \infty$, or simply $O(k)$ if r is fixed.

Corollary 6. *Algorithm* `DistErrDetect` *computes the edit distance of a language given via an NFA* $\mathbf{a}$ *in time*

$$O(|\mathbf{a}|^4 r^4 B_{\mathbf{a}}^2 \log B_{\mathbf{a}}),$$

where r *is the cardinality of the alphabet used in* $\mathbf{a}$.

Proof. For the correctness of the algorithm, first note that the loop in step 3 is set up such that $L(\mathbf{a})$ is always error-detecting for $\mathrm{sid}(\min -1)$. Also, based on the observations listed in the above remark, if $L(\mathbf{a})$ is error-detecting for $\mathrm{sid}(k)$ but not for $\mathrm{sid}(k+1)$, then the desired distance must be greater than k and at most $k+1$, hence equal to $k+1$.

For the time complexity, the while loop will perform $O(\log B_{\mathbf{a}})$ iterations. In each iteration, the value k is used to construct the transducer of size $O(r^2 k)$ shown in Fig. 1 with alphabet being the set of alphabet symbols appearing in the description of $\mathbf{a}$. Then, the transducer $\mathbf{t}'$ is constructed and its functionality is tested in time $O(|\mathbf{a}|^4 r^4 k^2)$. As $k < B_{\mathbf{a}}$, it follows that the total time complexity is as required. □

We note that, in the worst case, $B_{\mathbf{a}}$ is of order $O(|\mathbf{a}|)$ and, assuming a fixed alphabet, the above algorithm operates in time

$$O(|\mathbf{a}|^6 \log |\mathbf{a}|),$$

which is asymptotically better than the time complexity stated in [14] when the given automaton is an NFA.

Next we present the error-correction-based algorithm for estimating the desired edit distance.

Algorithm `DistErrCorrect`
0. Input: NFA $\mathbf{a}$
1. Let $B_{\mathbf{a}}$ be the bound in Lemma 4
2. Let min $\leftarrow 1$ and max $\leftarrow \lfloor (B_{\mathbf{a}} - 1)/2 \rfloor$
3. Perform binary search to find the largest k in $\{\min, \ldots, \max\}$ for which $L(\mathbf{a})$ is error-correcting for $\mathrm{sid}(k)$ as follows:
   while (min $\leq$ max)
   a) Let $k \leftarrow \lfloor (\min + \max)/2 \rfloor$
   b) Construct a transducer $\mathbf{t}$ realizing the channel $\mathrm{sid}(k)$
   c) Construct the transducer $\mathbf{t}' \leftarrow (\mathbf{t}^{-1} \uparrow \mathbf{a})$
   d) If ($\mathbf{t}'$ is functional) let min $\leftarrow k + 1$
      Else let max $\leftarrow k - 1$
4. **return** $\{2\min -1, 2\min\}$

Corollary 7. *Algorithm* `DistErrCorrect` *returns two values, differing by 1, one of which is the edit distance of the language given via* $\mathbf{a}$*, in time*

$$O(|\mathbf{a}|^2 r^4 (B_{\mathbf{a}}/2)^2 \log(B_{\mathbf{a}}/2)),$$

where r *is the cardinality of the alphabet used in* $\mathbf{a}$*.*

Proof. For the correctness of the algorithm, first note that the loop in step 3 is set up such that $L(\mathbf{a})$ is always error-correcting for $\mathrm{sid}(\min -1)$. Also, based on the observations listed in the above remark, if $L(\mathbf{a})$ is error-correcting for $\mathrm{sid}(k)$ but not for $\mathrm{sid}(k+1)$, then the desired distance must be greater than $2k$ and at most $2(k+1)$, hence equal to $2k+1$, or $2k+2$. Moreover, as $B_{\mathbf{a}} \geq 2k+1$, the initial value of max in step 2 is correct.

For the time complexity, the while loop will perform $O(\log B_{\mathbf{a}})$ iterations. In each iteration, the value k is used to construct the transducer of size $O(r^2 k)$ shown in Fig. 1 with alphabet being the set of alphabet symbols appearing in the description of $\mathbf{a}$. Then, the transducer $\mathbf{t}'$ is constructed and its functionality is tested in time $O(|\mathbf{a}|^2 r^4 k^2)$. As $k < B_{\mathbf{a}}/2$, it follows that the total time complexity is as required. □

As noted before, in the worst case, $B_{\mathbf{a}}$ is of order $O(|\mathbf{a}|)$ and, assuming a fixed alphabet, the above algorithm operates in time

$$O(|\mathbf{a}|^4 \log |\mathbf{a}|).$$

This time complexity is asymptotically better than the one in [14] even when the given automaton is a DFA.

4 An $O(n^2 d)$ algorithm for edit distance via input-altering transducers

In this section we present a new (exact) method for computing much faster the desired edit distance via input-altering transducers—see theorem 12 and the associated algorithm. A transducer $\mathbf{t}$ is called *input-altering*, if

$$w \notin \mathbf{t}(w), \quad \text{for all words } w,$$

that is, the output of $\mathbf{t}$ is never equal to the input used. The new method is based on the following two major observations.

(a) The new result (see theorem 11) that the property of error-detection for the channel $\mathrm{sid}(k)$ can be described via an input-altering transducer $\mathbf{t}_k$ of size $O(k)$.

(b) The new observation that, using an input-altering transducer in our algorithms, eliminates the need for a binary search loop that builds a new transducer in each iteration. Instead, this loop can be replaced with the incremental construction of an NFA $\mathbf{a}'_k$, which depends on $\mathbf{t}_k$, until a certain condition is satisfied, in which case the value of k is the desired edit distance.

The above observations are presented in two subsections.

4.1 An input-altering transducer for error-detection

We give first a quick summary of some concepts discussed in [6].

Remark 8. Let $\mathbf{t}$ be an input-altering transducer. The *property* $\mathcal{P}_{\mathbf{t}}$ *described by* $\mathbf{t}$ is the set of all languages L satisfying

$$\mathbf{t}(L) \cap L = \emptyset. \tag{7}$$

As explained in [6], this concept constitutes a formal method for specifying certain code properties defined via abstract binary relations [24], and allows one to decide efficiently the *property satisfaction problem* by testing condition (7). In particular, condition (7) can be tested in time

$$O(|\mathbf{a}|^2 |\mathbf{t}|), \tag{8}$$

where $\mathbf{a}$ is the NFA accepting the language L and $\mathbf{t}$ is the input-altering transducer describing the property for which L is to be tested. This approach has led to the development of an online language server, called I-LaSer [10].

We shall show (see theorem 11) that error-detection for $\mathrm{sid}(k)$ is definable via the input-altering transducer $\mathbf{t}_k$, which is shown in Fig. 2 and defined next. The value i in a state $[i]$ or $[i,a]$ is called the *error counter*, meaning that any path from $[0]$ to a state with error counter i has to be labeled u/v such that $\delta(u,v) \leq i$. More precisely, we will define the edges such that a state $[i,a]$ can be reached from $[0]$ via a path with label u/v if and only if $u = vax$ for some word x and $i = |ax|$, thus, v is a proper prefix of u and state $[i,a]$ remembers the left-most letter of u that occurs after its prefix v. A state $[i]$ with $i \geq 1$ can only be reached via a path labeled u/v from $[0]$ if $1 \leq \delta(u,v) \leq i$, thus, $u \neq v$. Furthermore, we make sure that for $u \neq v$ such that neither $u \leq_p v$ nor $v \leq_p u$ there is a path from $[0]$ to $[\delta(u,v)]$ which is labeled by u/v or v/u.

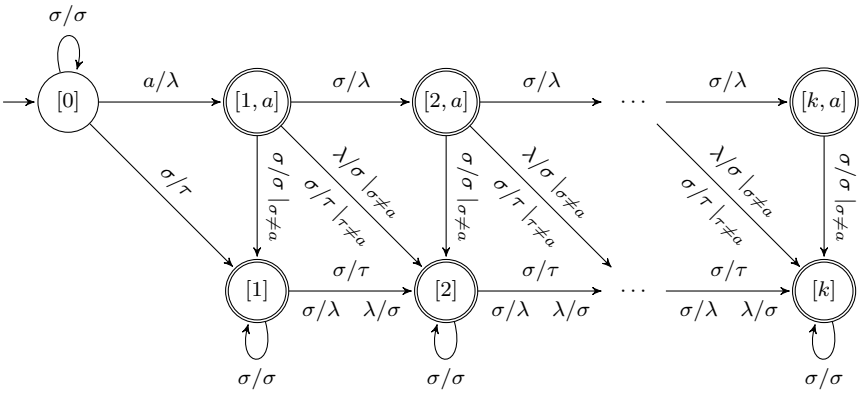

Figure 2: A segment of the input-altering transducer $\mathbf{t}_k$: for each $a \in \Sigma$ the complete transducer has k states of the form $[i,a]$. The labels σ and τ on an edge mean: one edge for each $\sigma, \tau \in \Sigma$ with $\sigma \neq \tau$; for some edge sets additional restrictions apply denoted, for example, by $|_{\sigma \neq a}$.

Definition 9. The transducer

$$\mathbf{t}_k = (Q, \Sigma, \Sigma, E, [0], F)$$

is defined as follows. The set of states is

$$Q = \{[i] \mid 0 \leq i \leq k\} \cup \{[i,a] \mid 1 \leq i \leq k, a \in \Sigma\}$$

with all but the initial state $[0]$ being final states:

$$F = Q \setminus \{[0]\}\,.$$

The edges in $\mathbf{t}_k$ can be divided into the four sets of edges $E = E_0 \cup E_s \cup E_i \cup E_d$. The edges from E_0 do not introduce any error, edges from the other sets model

one substitution (E_s), insertion (E_i), or deletion (E_d):

$$E_0 = \left\{ [i] \xrightarrow{\sigma/\sigma} [i] \;\middle|\; \sigma \in \Sigma, 0 \leq i \leq k \right\} \cup \tag{9}$$

$$\left\{ [i,a] \xrightarrow{\sigma/\sigma} [i] \;\middle|\; a, \sigma \in \Sigma, a \neq \sigma, 1 \leq i \leq k \right\} \tag{10}$$

$$E_s = \left\{ [i] \xrightarrow{\sigma/\tau} [i+1] \;\middle|\; \sigma, \tau \in \Sigma, \sigma \neq \tau, 0 \leq i < k \right\} \cup \tag{11}$$

$$\left\{ [i,a] \xrightarrow{\sigma/\tau} [i+1] \;\middle|\; a, \sigma, \tau \in \Sigma, \sigma \neq \tau, a \neq \tau, 1 \leq i < k \right\} \tag{12}$$

$$E_i = \left\{ [i] \xrightarrow{\lambda/\sigma} [i+1] \;\middle|\; \sigma \in \Sigma, 1 \leq i < k \right\} \cup \tag{13}$$

$$\left\{ [i,a] \xrightarrow{\lambda/\sigma} [i+1] \;\middle|\; a, \sigma \in \Sigma, a \neq \sigma, 1 \leq i < k \right\} \tag{14}$$

$$E_d = \left\{ [0] \xrightarrow{a/\lambda} [1,a] \;\middle|\; a \in \Sigma \right\} \cup \tag{15}$$

$$\left\{ [i] \xrightarrow{\sigma/\lambda} [i+1] \;\middle|\; \sigma \in \Sigma, 1 \leq i < k \right\} \cup \tag{16}$$

$$\left\{ [i,a] \xrightarrow{\sigma/\lambda} [i+1,a] \;\middle|\; a, \sigma \in \Sigma, 1 \leq i < k \right\} \tag{17}$$

Terminology. If $\mathbf{t} = (Q, \Sigma, \Sigma, E, q_0, F)$ is a transducer in standard form, then we write $\mathbf{t}^{\mathrm{e}}$ for the NFA

$$\mathbf{t}^{\mathrm{e}} = (Q, E_\Sigma, E, q_0, F)$$

over the edit alphabet E_Σ, where the labels of the transitions in $\mathbf{t}$ are viewed as elements of E_Σ. Note that, the label of a path P in $\mathbf{t}$ is a pair of words (u/v), whereas the label of the corresponding path in $\mathbf{t}^{\mathrm{e}}$, which we denote as P^{e}, is an edit string h such that $\mathrm{inp}(h) = u$ and $\mathrm{out}(h) = v$. This type of NFA is called an eNFA in [11].

Lemma 10. *Let $k \in \mathbb{N}$ and let u, v be words. The following statements hold true with respect to the transducer $\mathbf{t}_k$.*

i.) In $\mathbf{t}_k^{\mathrm{e}}$, every path from the start state $[0]$ to any state $[i]$ or $[i,a]$ has as label a reduced edit string whose weight is equal to i.

ii.) If $1 \leq \delta(u,v) \leq k$ and h is a reduced edit string realizing $\delta(u,v)$, then h is accepted by $\mathbf{t}_k^{\mathrm{e}}$.

iii.) If $v \in \mathbf{t}_k(u)$, then $1 \leq \delta(u,v) \leq k$.

iv.) If $\delta(u,v) \leq k$ and $va \leq_p u$, for some symbol a, then $[0] \xrightarrow{u/v}{}^{*} [\delta(u,v), a]$.

v.) If $i \in \mathbb{N}$ and $i + \delta(u,v) \leq k$, then $[i] \xrightarrow{u/v}{}^{*} [i + \delta(u,v)]$.

Proof. The first statement follows when we note that the definition of $\mathbf{t}_k$ and $\mathbf{t}_k^{\mathrm{e}}$ implies the following facts: (a) An edge exists between a state with error counter i to one with error counter $i+1$, if and only if the label of that edge is

an error; thus, in any path from $[0]$ to $[i]$ or $[i,a]$, the label of that path consists of exactly i errors. (b) Any edit string accepted by $\mathbf{t}_k^{\mathrm{e}}$ is indeed reduced.

For the second statement, consider any reduced edit string h realizing $\delta(u,v)$. If the first error in h is a deletion, then h is of the form

$$h = (e_1 \cdots e_r)(a/\lambda)(b_1/\lambda)\cdots(b_d/\lambda)h',$$

where each e_i is a non-error edit operation of the form (σ_i/σ_i), (a/λ) is a deletion error, $d \in \mathbb{N}_0$ and each (b_j/λ) is a deletion error, and h' is an edit string that is either empty or starts with a non-deletion edit operation (x/y) such that $y \neq a$. If h' is nonempty, then by definition of $\mathbf{t}_k^{\mathrm{e}}$ the following is a path

$$[0] \xrightarrow{(e_1\cdots e_r)}{}^{*} [0] \xrightarrow{(a/\lambda)(b_1/\lambda)\cdots(b_d/\lambda)}{}^{*} [1+d,a] \xrightarrow{h'}{}^{*} [1+d+\mathrm{weight}(h')]$$

accepting h. Similarly, a path accepting h exists in $\mathbf{t}_k^{\mathrm{e}}$, if h' is empty.

Finally, one verifies that if the first error in h is a substitution, then again h is accepted by $\mathbf{t}_k^{\mathrm{e}}$.

For the third statement, if $v \in \mathbf{t}_k(u)$, then (u/v) is the label of a path P from $[0]$ to a final state $[i]$ or $[i,a]$, with $0 < i \leq k$. As the label of the path P^{e} has exactly i errors, it follows that $\delta(u,v) \leq i \leq k$.

We also need to show that $\delta(u,v) \geq 1$, that is, $u \neq v$. First consider the case where the path P ends at $[i,a]$, with $1 \leq i \leq k$. Then, the label of P^{e} is an edit string of the form

$$h = (\sigma_1/\sigma_1)\cdots(\sigma_r/\sigma_r)(a/\lambda)(b_1/\lambda)\cdots(b_d/\lambda)$$

and $u = \mathrm{inp}(h) = \sigma_1\cdots\sigma_r a b_1 \cdots b_d$ and $v = \mathrm{out}(h) = \sigma_1\cdots\sigma_r$. Hence, $u \neq v$. Now consider the case where the path P ends at state $[i]$. There are three cases. (a) The states used in the path are $[0],[1],\ldots,[i]$. (b) The states used in P are $[0],[1,a],\ldots,[r,a],[r],\ldots,[i]$, for some appropriate $[r]$. (c) The states used in P are $[0],[1,a],\ldots,[r,a],[r+1],\ldots,[i]$, for some appropriate $[r]$. In all three cases, one verifies that $u \neq v$. For example, in case (b), u must be of the form $xa\sigma_1\cdots\sigma_{r-1}\sigma y$ and v of the form $x\sigma z$, where the σ_j's are symbols, x,y,z are words, and σ is a symbol other than a; hence, $u \neq v$.

For the fourth statement, let $u = a_1\cdots a_r a b_1 \cdots b_t$, with each a_i and b_j being a symbol, and $v = a_1\cdots a_r$. We use lemma 2. The edit string

$$h = (a_1/a_1)\cdots(a_r/a_r)(a/\lambda)(b_1/\lambda)\cdots(b_t/\lambda)$$

realizes $\delta(u,v)$. Moreover, this edit string is the label of a path in $\mathbf{t}_k^{\mathrm{e}}$ from $[0]$ to $[\delta(u,v),a]$. Hence, there is a path in $\mathbf{t}_k$ from $[0]$ to $[\delta(u,v),a]$ with label $(\mathrm{inp}(h)/\mathrm{out}(h)) = (u/v)$, as required.

For the fifth statement, let h be an edit string realizing $\delta(u,v)$. By definition of $\mathbf{t}_k^{\mathrm{e}}$, at each state of the form $[j]$ with $0 < j < k$, one can follow an edge whose label e can be of any of the four types of edit operations, and moreover, if e is an error, then the edge goes into $[j+1]$, that is, $[j] \xrightarrow{e} [j+1]$. As h contains exactly $\delta(u,v)$ errors, there is a path from $[i]$ to $[i+\delta(u,v)]$ whose label is made of the edit operations in h. Hence, there is a path in $\mathbf{t}_k$ from $[i]$ to $[i+\delta(u,v)]$ whose label is (u/v), as required. □

Theorem 11. *For each $k \in \mathbb{N}$, the transducer $\mathbf{t}_k$ is input-altering and of size $O(k)$, and describes the property of error-detection for the channel* $\mathrm{sid}(k)$.

Proof. By construction, it follows that $\mathbf{t}_k$ is trim and has a number of states and transitions that is linear with respect to k. Hence, it is indeed of size $O(k)$. The third statement of lemma 10 implies that the transducer is input-altering. For the error-detection part, using the first statement of remark 5, it is sufficient to show that, for every language L,

$$\mathbf{t}_k(L) \cap L = \emptyset \text{ if and only if } \delta(L) > k.$$

First, for the 'if' part, assume $\delta(L) > k$ and consider any words $u, v \in L$. We need to prove $v \notin \mathbf{t}_k(u)$. If $u = v$ then this holds as $\mathbf{t}_k$ is input-altering. Else, it follows from the third statement of lemma 10. Now for the 'only if' part, assume

$$\mathbf{t}_k(L) \cap L = \emptyset, \tag{18}$$

but, for the sake of contradiction, suppose there are different words $u, v \in L$ such that $1 \leq \delta(u, v) \leq k$. If v is a prefix of u, then $va \leq_p u$, for some $a \in \Sigma$, and the fourth statement of the above lemma implies $[0] \xrightarrow{u/v}{}^* [\delta(u, v), a]$ and, therefore, $v \in \mathbf{t}_k(u)$, which contradicts (18). By symmetry, a contradiction arises if u is a prefix of v.

Now consider the case where v is not a prefix of u, and u is not a prefix of v. Then, $u = xau'$ and $v = xbv'$ for some words x, u', v' and symbols $a, b \in \Sigma$ with

$$a \neq b.$$

We shall obtain a contradiction to (18) by showing the existence of a path $[0] \xrightarrow{u/v}{}^* \psi$, *or* $[0] \xrightarrow{v/u}{}^* \psi$, *where ψ is a final state of* $\mathbf{t}_k$. Let h be an edit string realizing $\delta(au', bv')$. Recall $\delta(u, v) = \delta(au', bv')$. As $a \neq b$, the first edit operation, say e, of h must be an error, that is, not of the form σ/σ. Let $h = eh'$. We consider three cases for e. First, if e is a substitution, then $e = (a/b)$ and $\delta(u, v) = 1 + \delta(u', v')$. By the fifth statement of the above lemma, $[1] \xrightarrow{u'/v'}{}^* [1 + \delta(u', v')]$. Then, the required path is

$$[0] \xrightarrow{x/x}{}^* [0] \xrightarrow{a/b}{}^* [1] \xrightarrow{u'/v'}{}^* [1 + \delta(u', v')].$$

Now consider the case where $e = (a/\lambda)$. Then, $\delta(u, v) = 1 + \delta(u', bv')$ and h' realizes $\delta(u', bv')$. Let d be the number of deletions (if any) at the beginning of h' so that any edit operation following these deletions is not a deletion. Thus, h' is of the form $h_1 h_2$ with $\mathrm{inp}(h_1) = u_1$ and $\mathrm{out}(h_1) = \lambda$, where u_1 is a word of length d, and $\mathrm{inp}(h_2) = u_2$ and $\mathrm{out}(h_2) = bv'$, for some word u_2, and $u' = u_1 u_2$, and $\delta(u', bv') = d + \delta(u_2, bv')$. As bv' is nonempty, also h_2 is nonempty, so let e' be the first edit operation of h_2, which cannot be a deletion. If $u_2 = \lambda$, then $e' = (\lambda/b)$ and h_2 consists of insertions, and the required path is

$$[0] \xrightarrow{x/x}{}^* [0] \xrightarrow{a/\lambda}{}^* [1, a] \xrightarrow{u_1/\lambda}{}^* [1 + d, a] \xrightarrow{e'}{}^* [1 + d + 1] \xrightarrow{\lambda/v'}{}^* [1 + d + 1 + \delta(\lambda, v')].$$

If $u_2 \neq \lambda$, then there is a symbol c such that $u_2 = cu_2'$. If $c = b$, then e' cannot be a substitution, so it must be the non-error (c/b) or the insertion (λ/b). Then, the required path is

$$[0] \xrightarrow{x/x}{}^* [0] \xrightarrow{a/\lambda}{}^* [1, a] \xrightarrow{u_1/\lambda}{}^* [1+d, a] \xrightarrow{e'}{}^* [1+d+t] \xrightarrow{z/v'}{}^* [1+d+t+\delta(z, v')],$$

where $z = u_2'$ and $t = 0$ (case of $e' = (c/b)$), or $z = cu_2'$ and $t = 1$ (case of $e' = (\lambda/b)$). If $c \neq b$, then e' must be the insertion (λ/b) or the substitution (c/b). Again, in either case, a path as required exists.

Finally, the case of $e = (\lambda/b)$, is symmetric to the previous one by simply switching the roles of u and v. □

4.2 The $O(n^2 d)$ algorithm for edit distance

Remark 8 and theorem 11 imply that the intermediate algorithm `DistFirstInpAlter` shown below correctly computes the desired edit distance. Moreover, by reasoning as in the proof of corollary 6, it follows that this algorithm executes in time $O(|\mathbf{a}|^2 r^2 B_{\mathbf{a}} \log B_{\mathbf{a}})$, where r is the cardinality of the alphabet used in $\mathbf{a}$.

Algorithm `DistFirstInpAlter`
0. Input: NFA $\mathbf{a}$
1. Let $B_{\mathbf{a}}$ be the bound in Lemma 4
2. Let min $\leftarrow 1$ and max $\leftarrow B_{\mathbf{a}} - 1$
3. Perform binary search to find the largest k in $\{\min, \ldots, \max\}$ for which $L(\mathbf{a})$ is error-detecting for sid(k) as follows:
   while (min $\leq$ max)
   a) Let $k \leftarrow \lfloor (\min + \max)/2 \rfloor$
   b) Construct the transducer $\mathbf{t}_k$
   c) Construct NFA $\mathbf{a}'$ accepting $\mathbf{t}_k(L(\mathbf{a})) \cap L(\mathbf{a})$
   d) If ($\mathbf{a}'$ accepts $\emptyset$) let min $\leftarrow k + 1$
      Else let max $\leftarrow k - 1$
4. **return** min

We note again that, in the worst case, $B_{\mathbf{a}}$ is of order $O(|\mathbf{a}|)$ and, assuming a fixed alphabet, the above algorithm operates in time

$$O(|\mathbf{a}|^3 \log |\mathbf{a}|),$$

which is asymptotically better than those of all other known algorithms. However, we now discuss in detail the second major observation stated in the beginning of section 4, which leads to the most efficient algorithm in theorem 12. In particular, for the sake of clarity, we present that algorithm in two steps. In the first place, we notice that the while loop in `DistFirstInpAlter` can be replaced with the construction of the automaton $\mathbf{t}_{B_{\mathbf{a}}-1}(\mathbf{a}) \cap \mathbf{a}$ and a search in that automaton for a path from the start state to a final one in which the error counter value is minimal (this value would be the required edit distance).

Algorithm `DistNextInpAlter`

0. Input: NFA $\mathbf{a}$
1. Let $B_{\mathbf{a}}$ be the bound in Lemma 4
2. Construct the transducer $\mathbf{t}_{B_{\mathbf{a}}-1}$
3. Construct NFA $\mathbf{a}'$ accepting $\mathbf{t}_{B_{\mathbf{a}}-1}(L(\mathbf{a})) \cap L(\mathbf{a})$
4. Starting at the start state of $\mathbf{a}'$, use breadth first search (BFS) to visit all states. In doing so, keep track of the smallest error counter min in the visited final states of $\mathbf{a}'$.
5. **return** min

As usual in product constructions, the states of $\mathbf{a}'$ are triples of the form (φ, q, q'), where φ is a state of $\mathbf{t}_{B_{\mathbf{a}}-1}$, and q, q' are states of $\mathbf{a}$. The start state of $\mathbf{a}'$ is $([0], q_0, q_0)$, where q_0 is the start state of $\mathbf{a}$, and the final states of $\mathbf{a}'$ are those triples consisting of final states in $\mathbf{t}_{B_{\mathbf{a}}-1}$ and $\mathbf{a}$. A transition

$$(\varphi, q, q') \xrightarrow{y} (\psi, r, r')$$

exists in $\mathbf{a}'$ if and only if the following transitions

$$\varphi \xrightarrow{x/y} \psi, \quad q \xrightarrow{x} r, \quad q' \xrightarrow{y} r'$$

exist in $\mathbf{t}_{B_{\mathbf{a}}-1}$, $\mathbf{a}^{\lambda}$ and $\mathbf{a}^{\lambda}$, respectively, for some label x, where $\mathbf{a}^{\lambda}$ results if we add to $\mathbf{a}$ empty loop transitions (q, λ, q) for all states q in $\mathbf{a}$. The correctness of the above algorithm follows from lemma 10 and the definition of $\mathbf{a}'$. The breadth first search process requires time linear with respect to the size of $\mathbf{a}'$, which is

$$O(|\mathbf{a}|^2 B_{\mathbf{a}}),$$

and this also is the time complexity of the above algorithm (when the alphabet is fixed).

The final improved algorithm results if we notice that the desired edit distance can be much smaller than $B_{\mathbf{a}}$ and that it can be computed using only an 'initial' part of $\mathbf{t}_{B_{\mathbf{a}}-1}$. In other words, one can first build $\mathbf{t}_1$ and $\mathbf{a}'_1$ accepting $\mathbf{t}_1(L(\mathbf{a})) \cap L(\mathbf{a})$, and test whether $\mathbf{a}'_1$ has any accepting path. If not, this process is repeated by extending $\mathbf{a}'_k$ to $\mathbf{a}'_{k+1}$ until some extended automaton, say $\mathbf{a}'_d$, has an accepting path, in which case the desired distance is equal to d.

Algorithm `DistBestInpAlter`

0. Input: NFA $\mathbf{a}$
1. Construct the transducer $\mathbf{t}_1$
2. Construct NFA $\mathbf{a}'$ accepting $\mathbf{t}_1(L(\mathbf{a})) \cap L(\mathbf{a})$
3. $k \leftarrow 1$
4. **while** ($\mathbf{a}'$ has no accepting path)
   a) $\mathbf{a}' \leftarrow$ `Extend`$(\mathbf{a}', k)$
   b) $k \leftarrow k + 1$
5. **return** k

The function `Extend` in the above algorithm works based on the structure of $\mathbf{t}_k$ in Fig. 2 and is *partially* shown below. For clarity, we emphasize the fact that, in each step k of this algorithm, the final states of $\mathbf{a}'$ are only triples of the form $([k,a],f,f')$ or $([k],f,f')$, that is, when $i<k$, no triples of the form $([i,a],f,f')$ or $([i],f,f')$ are final states in $\mathbf{a}'$.

Function `Extend`$(\mathbf{a}',k)$ (partial view)
let $\mathbf{b}$ be a copy of $\mathbf{a}'$
for each state of the form $([k,a],q,q')$ in $\mathbf{a}'$
  for each transitions $q \xrightarrow{\sigma} r$ and $q' \xrightarrow{\sigma'} r'$ in $\mathbf{a}$
    if $(a \neq \sigma'$ and $\sigma \neq \sigma')$
      add to $\mathbf{b}$ the transition $([k,a],q,q') \xrightarrow{\sigma/\sigma'} ([k+1],r,r')$
      if r and r' are final in $\mathbf{a}$ then $([k+1],r,r')$ is final in $\mathbf{b}$
    if $(a \neq \sigma'$ and $\sigma = \sigma')$
      add to $\mathbf{b}$ the transition $([k,a],q,q') \xrightarrow{\sigma/\sigma} ([k],r,r')$
      if r and r' are final in $\mathbf{a}$ then $([k+1],r,r')$ is final in $\mathbf{b}$
    .....
return the NFA $\mathbf{b}$

Based on the above discussion, the correctness of the following theorem has been established.

Theorem 12. *Algorithm* `DistBestInpAlter` *computes the edit distance of the language given via an NFA* $\mathbf{a}$ *in time* $O(|\mathbf{a}|^2 r^2 d)$*, where* r *is the cardinality of the alphabet used in* $\mathbf{a}$ *and* d *is the computed edit distance.*

5 Implementation and testing

We have implemented the main algorithm `DistBestInpAlter` of theorem 12, as well as the intermediate vesrions

`DistErrDetect, DistErrCorrect, DistFirstInpAlter,`

using the FAdo library for automata [7], which is well maintained and provides several useful tools for manipulating automata. Moreover, we have used some of the implementations of I-LaSer [10] involving product constructions between transducer and automaton objects of the FAdo library. We note that an implementation in C++ of the algorithm in [14] is discussed in [5], but the execution time is too slow to be used for any meaningful comparisons with the algorithms presented here. Our best algorithm can be executed online at [19]. The user can enter as input an NFA in Grail or FAdo format, select the algorithm to execute, and press the Submit button.

We have performed several tests[1] for the correctness of these algorithms, as well as two sets of tests for the time complexity, which confirm the theoretical

[1] All tests were performed on a machine with the following specification. Make: Acer, CPU: AMD Athlon(tm) II X2 215, Clock speed: 2.70 GHz, Memory (RAM): 4.00 GB, Operating System: Windows 7 64-bit.

result that `DistBestInpAlter` is indeed the fastest algorithm. We note that, for the other three algorithms, we have skipped the step of computing the upper bound $B_{\mathbf{a}}$ on the edit distance, as this step is the same for all these algorithms, thus resulting in faster execution without affecting in any essential way the performance comparisons.

The two sets of tests correspond to two sequences of automata $(\mathbf{a}_n)$ and $(\mathbf{b}_n)$ shown in the next two figures. The first test set is such that the desired distance is equal to n, for each NFA $\mathbf{a}_n$, that is, the distance grows with n and, in fact, it is a worst-case scenario where the distance is equal to the number of states of the NFA. The second test set is such that the desired distance is fixed, equal to 2, for all n.

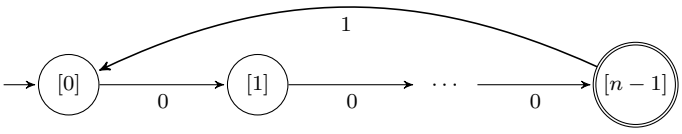

Figure 3: The automaton $\mathbf{a}_n$ accepting the language $0^{n-1}(10^{n-1})^*$.

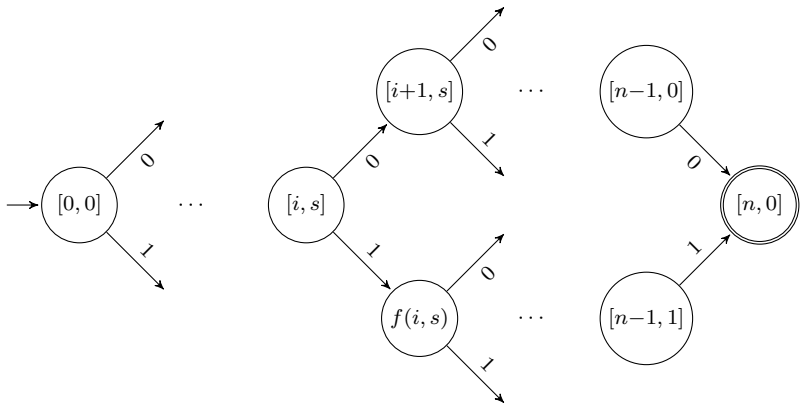

Figure 4: The automaton $\mathbf{b}_n$ with $n^2 + n + 1$ states, where $f(i, s) = [i+1, (s+i+1)\%(n+1)]$. The states are $[n, 0]$ and $[i, s]$, with $0 \leq i \leq n-1$ and $0 \leq s \leq n$. This automaton accepts the Levenshtein code consisting of all binary words $b_1 \cdots b_n$ of length n such that $(\sum_{i=1}^{n} i \cdot b_i)\%(n + 1) = 0$, where '%' is the integer division remainder operation. This code has edit distance equal to 2. On the other hand, its distance for insertion/deletion errors only is 3, so it is error-correcting for the 1-insertion/deletion per word channel.

The next table shows the actual running times of the four algorithms on the NFAs $\mathbf{a}_4, \ldots, \mathbf{a}_8, \mathbf{a}_{13}, \mathbf{a}_{21}, \mathbf{a}_{31}$. The number in parentheses next to each $\mathbf{a}_i$ indicates the number of states in $\mathbf{a}_i$.

NFA	ErrDetection	ErrCorrection	FirstInpAlter	BestInpAlter
$\mathbf{a}_5$ (5)	3.94s	0.35s	0.08s	0.008s
$\mathbf{a}_6$ (6)	19.20s	0.48s	0.11s	0.010s
$\mathbf{a}_7$ (7)	107.35s	2.54s	0.18s	0.013s
$\mathbf{a}_8$ (8)	442.01s	4.03s	0.33s	0.016s
$\mathbf{a}_{13}$ (13)	>5 hours	144.75s	1.31s	0.020s
$\mathbf{a}_{21}$ (21)	>5 hours	12475.27s	10.21s	0.029s
$\mathbf{a}_{31}$ (31)	>5 hours	>5 hours	46.28s	0.109s

The next table shows the actual running times of the four algorithms on the NFAs $\mathbf{b}_3, \ldots, \mathbf{b}_8$. The number in parentheses next to each $\mathbf{b}_i$ indicates the number of states in $\mathbf{b}_i$.

NFA	ErrDetection	ErrCorrection	FirstInpAlter	BestInpAlter
$\mathbf{b}_3$ (13)	0.889s	0.164s	0.098s	0.027s
$\mathbf{b}_4$ (21)	212.32s	7.06s	0.655s	0.039s
$\mathbf{b}_5$ (31)	>5 hours	72.25s	4.63s	0.097s
$\mathbf{b}_6$ (43)	>5 hours	3806.74s	40.79s	0.234s
$\mathbf{b}_7$ (57)	>5 hours	>5 hours	375.17s	0.735s
$\mathbf{b}_8$ (73)	>5 hours	>5 hours	2070.21s	1.919s

Let d be the computed edit distance in the above test sets. The best algorithm has about the same performance in both test cases, even though d is a parameter in its time complexity $O(n^2 d)$. A possible explanation is that the NFA $\mathbf{b}_i$ has more edges than those of $\mathbf{a}_j$, when both $\mathbf{b}_i$ and $\mathbf{a}_j$ have the same number of states. However, the other algorithms perform a lot better when d is small, probably because the NFAs which are generated in each step of the binary search become successively bigger for $\mathbf{a}_j$ and they become successively smaller for $\mathbf{b}_i$.

A further improvement to our algorithms, to be implemented, is that one can remove from Fig. 2 all the diagonal transitions from a state $[i, a]$ to a state $[i+1]$. This is because, for any edit string of the form

$$h = e_1 \cdots e_r \, (a/\lambda)(a_1/\lambda) \cdots (a_d/\lambda)(\sigma/\tau) \, h_1$$

accepted by $\mathbf{t}^{\mathrm{e}}_k$, where $\tau \notin \{a, \sigma\}$ and the e_j's are non-errors, the automaton $\mathbf{t}^{\mathrm{e}}_k$ also accepts

$$g = e_1 \cdots e_r \, (a/\tau)(a_1/\lambda) \cdots (a_d/\lambda)(\sigma/\lambda) \, g_1$$

such that $\mathrm{inp}(g) = \mathrm{inp}(h)$, $\mathrm{out}(g) = \mathrm{out}(h)$, and $\mathrm{weight}(g) = \mathrm{weight}(h)$. Moreover $\mathbf{t}^{\mathrm{e}}_k$ accepts g using none of the diagonal transitions that are to be removed as specified above. A similar observation holds if we replace in h the edit operation (σ/τ) shown in h with (λ/σ), where $\sigma \neq a$. Of course this improvement

does not affect in any significant way the performance comparisons among the four algorithms.

6 Conclusion

This paper represents a significant improvement in the time complexity of computing the edit distance of a given regular language. As discussed in [14], this problem is related to the inherent capability of a language to detect substitution, insertion, and deletion errors. The method based on using the input-altering transducer $\mathbf{t}_k$ seems to adapt to other types of errors as well. For example, one can construct a similar input-altering transducer for insertion/deletion only errors. It seems promising to investigate the problem when the errors have different costs (in the current setting, the cost of each error is 1).

The present contribution stemmed from the question of whether the error-detection property for the channel $\mathrm{sid}(k)$ can be described by an input-altering transducer. The more general question of whether an input-preserving transducer property of interest can be described by an input-altering transducer is important to investigate, as this would lead to more efficient algorithms for deciding the property satisfaction problem (whether, given a regular language and a transducer property, the language satisfies the property).

References

[1] C. Allauzen and M. Mohri. Efficient algorithms for testing the twins property. *Journal of Automata, Languages and Combinatorics*, 8(2):117–144, 2003.

[2] M. Béal, O. Carton, C. Prieur, and J. Sakarovitch. Squaring transducers: An efficient procedure for deciding functionality and sequentiality. *Theoretical Computer Science*, 292(1):45–63, 2003.

[3] M. Benedikt, G. Puppis, and C. Riveros. The cost of traveling between languages. In L. Aceto, M. Henziger, and J. Sgall, editors, *ICALP 2011, Part II. LNCS 6756*, pages 234–245, Heidelberg, 2011. Springer-Verlag.

[4] J. Berstel. *Transductions and Context-Free Languages*. B.G. Teubner, Stuttgart, 1979.

[5] A. Daka. Computing error-detecting capabilities of regular languages. Master's thesis, Dept. Mathematics and Computing Science, Saint Mary's University, Halifax, NS, Canada, 2011.

[6] K. Dudzinski and S. Konstantinidis. Formal descriptions of code properties: decidability, complexity, implementation. *Intern. J. Foundations of Computer Science*, 23:67–85, 2012.

[7] FAdo. Tools for formal languages manipulation. URL address: `http://fado.dcc.fc.up.pt/` Accessed in June, 2013.

[8] Y.-S. Han, S.-K. Ko, and K. Salomaa. Computing the edit-distance between a regular language and a context-free langauge. In H.-C. Yen and O. Ibarra, editors, *DLT 2012. LNCS 7410*, pages 85–96, Heidelberg, 2012. Springer.

[9] Y.-S. Han, S.-K. Ko, and K. Salomaa. Approximate matching between a context-free grammar and a finite-state automaton. In S. Konstantinidis, editor, *CIAA 2013. LNCS 7982*, pages 146–157, Heidelberg, 2013. Springer.

[10] I-LaSer. Independent LAnguage SERver. URL address: `http://laser.cs.smu.ca/independence/` Accessed in August, 2013.

[11] L. Kari and S. Konstantinidis. Descriptional complexity of error/edit systems. *Journal of Automata, Languages and Combinatorics*, 9:293–309, 2004. Full version of a paper in: Proc. Descriptional Complexity of Formal Systems, London, Ontario, 2002.

[12] L. Kari, S. Konstantinidis, S. Perron, G. Wozniak, and J. Xu. Finite-state error/edit-systems and difference-measures for languages and words. Report 2003-01, Mathematics and Computing Science, Saint Mary's University, Canada, 2003.

[13] S. Konstantinidis. Transducers and the properties of error-detection, error-correction and finite-delay decodability. *Journal Of Universal Computer Science*, 8:278–291, 2002.

[14] S. Konstantinidis. Computing the edit distance of a regular language. *Information and Computation*, 205(9):1307–1316, 2007. Full version of "Computing the Levenshtein distance of a regular language," in M.J. Dinneen (ed.): Proc. 2005 IEEE Information Theory Workshop (ITW 2005) on Coding and Complexity, pages 113–116.

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

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

[17] A. Mateescu and A. Salomaa. Regular languages. In Rozenberg and Salomaa [21], pages 1–39.

[18] M. Mohri. Edit-distance of weighted automata: general definitions and algorithms. *Intern. J. Foundations of Computer Science*, 14:957–982, 2003.

[19] O-LaSer. Other LAnguage SERver. URL address: `http://140.184.132.57/others/` Accessed in October, 2013.

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

[21] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages, Vol. I.* Springer-Verlag, Berlin, 1997.

[22] J. Sakarovitch. *Elements of Automata Theory.* Cambridge University Press, Berlin, 2009.

[23] A. Schrijver. *Combinatorial Optimization: polyhedra and efficiency.*

[24] H. Shyr and G. Thierrin. Codes and binary relations. In *Séminaire d'Algèbre Paul Dubreil, Paris 1975–1976 (29ème Année) , Lecture Notes in Mathematics*, pages 180–188, 1975.

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

[26] D. Wood. *Theory of Computation.* John Wiley & Sons, New York, 1987.

[27] M. Yang. Application and implementation of transducer tools in answering certain questions about regular languages. Master's thesis, Dept. Mathematics and Computing Science, Saint Mary's University, Halifax, NS, Canada, 2012.

[28] S. Yu. Regular languages. In Rozenberg and Salomaa [21], pages 41–110.