algorithms

MDPI

Article

Efficient Algorithms for Computing the Inner Edit Distance of a Regular Language via Transducers

Lila Kari [1], Stavros Konstantinidis [2,*], Steffen Kopecki [2,3] and Meng Yang [2]

[1] School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada; lila@uwaterloo.ca

[2] Department of Mathematics and Computing Science, Saint Mary's University, Halifax, NS B3H 3C3, Canada; steffen.kopecki@gmail.com (S.K.); meyang.mike@gmail.com (M.Y.)

[3] Department of Computer Science, University of Western Ontario, London, ON N6A 5B7, Canada

* Correspondence: s.konstantinidis@smu.ca; Tel.: +1-902-420-5117

Received: 6 October 2018 ; Accepted: 18 October 2018; Published: 23 October 2018

Abstract: The concept of edit distance and its variants has applications in many areas such as computational linguistics, bioinformatics, and synchronization error detection in data communications. Here, we revisit the problem of computing the inner edit distance of a regular language given via a Nondeterministic Finite Automaton (NFA). This problem relates to the inherent maximal error-detecting capability of the language in question. We present two efficient algorithms for solving this problem, both of which execute 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 one of the two algorithms and present here a set of performance tests. The correctness of the algorithms is based on the connection between word distances and error detection and the fact that nondeterministic transducers can be used to represent the errors (resp., edit operations) involved in error-detection (resp., in word distances).

Keywords: algorithms; automata; complexity; edit distance; implementation; transducers; regular language

1. Introduction

The concept of edit distance and its variants has applications in many areas such as computational linguistics [1], bioinformatics [2], and synchronization error detection in data communications [3]. 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 [4], 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 [4] has worst-case time complexity

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

where 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 Label (1) can be written as

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

In this paper, we present two efficient algorithms to compute the inner edit distance of a regular language given via an NFA with n transitions—see Theorems 1 and 3. Both algorithms, which are called `DistErrDetect` and `DistInpAlter`, have the same worst-case time complexity

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

where d is the computed distance, which is a significant improvement over the original algorithm in [4].

Our first algorithm, `DistErrDetect`, is based on the general method of [5] for computing distances via the error-detection property. Now, however, we have an efficient way of realizing algorithmically that general method using an incremental construction of a (nondeterministic) transducer and the test of [6] for partial identity for transducers. In our second algorithm, `DistInpAlter`, the idea is to model the edit operations of the desired distance using an efficient, in terms of size, input-altering transducer (a transducer whose output is always different from the input used). Please see subsequent sections for definitions of terms. For clarity of presentation, we give in detail not only the new algorithms, but also their preliminary versions `PrelimDistErrDetect` and `PrelimDistInpAlter` that could possibly be applied to other types of distances. We have implemented the preliminary and final versions of the second algorithm (`PrelimDistInpAlter` and `DistInpAlter`) in Python using the well maintained, open source package FAdo for automata [7]. We have also tested our implementation experimentally, and we present in this paper the outcomes of the tests.

We note that some related problems involving distances between words and languages can be found in [8,9] (edit distance between a word and a language), and in [10–14] (various distances between two languages). Also in [15], the newer concept of edit distance with moves is investigated. The problem considered here is technically different, however, as the desired distance involves *different words* within the *same* language. More specifically, if we used directly the tools of [10,11], for instance, to compute an edit string with minimal number of errors between the given language and itself, then that string would simply be an edit string of zero errors, as the edit distance between any word and itself is zero. We also note that the inner prefix distance of a regular language, which is quite different from the inner edit distance, is considered in [16] and computed in time $O(n^2 \log n)$.

The paper is organized as follows. The next section contains basic notions on languages, word relations, finite-state machines and edit-strings. Section 3 describes the approach of computing the desired edit distance via the concept of error-detection and presents the preliminary version `PrelimDistErrDetect` of the first algorithm. Then, Section 4 explains the improved and final version `DistErrDetect` of the first algorithm. In Section 5, it is shown that the edit distance is definable via an efficient input-altering transducer—see Theorem 2—and then the second algorithm `DistInpAlter` is presented. Section 6 discusses the implementation and testing of the second algorithm and its preliminary version. The last section contains a few concluding remarks. The appendix contains the proofs of two technical lemmata.

2. Notation, Background and Preliminary Results

This section contains basic terminology about formal languages, automata, transducers, and edit strings. Most of the basic notions presented here can be found in various texts such as [17–21].

2.1. Sets, Words, Languages, Channels

The set of positive integers is denoted by $\mathbb{N}$. Then, $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. If S is any set, the expression $|S|$ denotes the cardinality of S. We use standard basic notation and terminology for alphabets, words and languages—see [22], for instance. For example, Σ denotes an alphabet, Σ^+ the set of nonempty words, λ the empty word, $\Sigma^* = \Sigma^+ \cup \{\lambda\}$, $|w|$ the length of the word w. We write $u \leq_p w$ to indicate that the word u is a *prefix* of w, that is, $w = uv$ for some word v. Then, $u <_p w$ means that u is a *proper prefix*, that is, $u \leq_p w$ and $u \neq w$. 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 *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 (via γ) *with errors*. 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 finite 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 or edges. Let (p,x,q) be a transition of $\mathbf{a}$. Then, x is called the *label* of the transition, and we say that the transition *goes out* of p. 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 in which, for each state p, there are no two transitions with equal labels going out of p.

A *path* of $\mathbf{a}$ is a finite sequence of consecutive transitions:

$$(p_0, x_1, p_1)(p_1, x_2, p_2) \cdots (p_{\ell-1}, x_\ell, p_\ell), \tag{4}$$

for some nonnegative integer ℓ, where we use concatenation of these transitions to denote the path. Then, if P_1 and P_2 are two paths such that the last state of P_1 is equal to the first state of P_2,

P_1P_2 denotes the path resulting by concatenating the transitions of P_1 and P_2.

The word $x_1 \cdots x_\ell$ is called the *label* of the path in Label (4). 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 a *computation* of $\mathbf{a}$ if p_0 is the start state. It is called an *accepting* path/computation 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 nondeterministic) *transducer* [17,20] is a quintuple (In the literature, a transducer also has an output alphabet Γ, but we consider here that Γ is the same as the input alphabet Σ. Without further mention all transducers considered here are nondeterministic.) $\mathbf{t} = (Q, \Sigma, T, s, F)$ such that Q, s, F are exactly the same as those in λ-NFAs, Σ is the alphabet, and $T \subseteq Q \times (\Sigma \cup \{\lambda\}) \times (\Sigma \cup \{\lambda\}) \times Q$ is the finite set of transitions or edges. We write $(p, x/y, q)$, or $p \xrightarrow{x/y} q$ for a transition—the *label* here is (x/y), with x being the *input* and y being the *output* label of the transition. The concepts of path, computation, accepting path, and trim transducer are similar to those in λ-NFAs. However, the label of a transducer path $(p_0, x_1/y_1, p_1) \cdots (p_{\ell-1}, x_\ell/y_\ell, p_\ell)$ is the pair $(x_1 \cdots x_\ell, y_1 \cdots y_\ell)$ of the two words consisting of the concatenations of the input and output labels in the path, respectively. The *relation realized* by the transducer $\mathbf{t}$, denoted by $R(\mathbf{t})$, is the set of labels in all the accepting paths of $\mathbf{t}$. We write $\mathbf{t}(u)$ for the *set of possible outputs* of $\mathbf{t}$ on input u, that is,

$$v \in \mathbf{t}(u) \ \text{ if and only if } \ (u,v) \in R(\mathbf{t}).$$

The transducer $\mathbf{t}$ is called *functional*, if the relation $R(\mathbf{t})$ is a function, that is, $\mathbf{t}(u)$ consists of at most one word, for all input words u. We say that $\mathbf{t}$ *realizes a partial identity*, if $v \in \mathbf{t}(u)$ implies that $v = u$.

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

$$Q_{\mathbf{m}},\ T_{\mathbf{m}}\ \text{ for the sets of states and transitions of } \mathbf{m}\text{, respectively.}$$

If $\mathbf{m}$ is trim then $|Q_{\mathbf{m}}| \leq |T_{\mathbf{m}}| + 1$; thus,

$$\text{if } \mathbf{m} \text{ is trim then } |\mathbf{m}| = O(|T_{\mathbf{m}}|).$$

We recall that making an automaton or transducer $\mathbf{m}$ trim can be done in linear time $O(|\mathbf{m}|)$.

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* [23]. 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 by $\text{weight}(h)$, is the number of errors occurring in h. For example, for

$$g = (a/a)(a/\lambda)(b/b)(b/a)(b/b), \tag{5}$$

$\text{weight}(g) = 2$. 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:

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

The *edit (or Levenshtein) distance* [24] 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, either $\text{inp}(h) = u$ and $\text{out}(h) = v$, or $\text{inp}(h) = v$ and $\text{out}(h) = u$. 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) [25].

The next lemma comes from [4]. The bound $D_{\mathbf{a}}$ is always less than or equal to the number of states in the NFA **a**. Moreover, there are NFAs for which this bound is tight—see Section 6.

Lemma 1. *For every NFA* **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 **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 [26]. 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 **a**. For example, that distance is always less than or equal to the distance of the two shortest accepted words. We agree to use this as a working upper bound:

Lemma 2. *For every NFA* **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

In [5], the authors discuss a conceptual method for computing integral distances of regular languages—integral means that all distance values are nonnegative 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.

A language L is *error-detecting for* a channel γ, [27], if no L-word can be received as a different L-word via γ; that is (The definition of error-detection in [27] uses $L \cup \{\lambda\}$ instead of L in Formula 6. This slight change makes the presentation here simpler and has no bearing on any existing results regarding error-detecting languages.), for any words u and v,

$$u, v \in L \text{ and } (u, v) \in \gamma \;\rightarrow\; u = v. \tag{6}$$

Remark 1. *The error-detection method of [5] for computing inner distances of regular languages is based on the following observations, where* **a** *is an NFA and* **t** *is an input-preserving transducer.*

1. *A language L is error-detecting for $\mathrm{sid}(m)$, if and only if $\delta(L) > m$.*
2. *$\delta(L)$ is equal to the positive integer k such that L is error-detecting for $\mathrm{sid}(k-1)$ and L is not error-detecting for $\mathrm{sid}(k)$.*
3. *We have the following facts from [27]. A language L is error-detecting for a channel γ if and only if the following relation is a function*
$$\gamma \cap (L \times \Sigma^*) \cap (\Sigma^* \times L). \tag{7}$$
Moreover, if **a** *accepts L and* **t** *realizes γ, then a transducer $(\mathbf{t} \downarrow \mathbf{a})$ realizing $\gamma \cap (L \times \Sigma^*)$ can be constructed in time $O(|\mathbf{t}||\mathbf{a}|)$ and, analogously, a transducer $(\mathbf{t} \uparrow \mathbf{a})$ realizing $\gamma \cap (\Sigma^* \times L)$ can be constructed in time $O(|\mathbf{t}||\mathbf{a}|)$. Both constructions are cross-product constructions. In each case, the resulting transducer has $O(|Q_{\mathbf{a}}||Q_{\mathbf{t}}|)$ states and $O(|T_{\mathbf{a}}||T_{\mathbf{t}}|)$ transitions. Thus, the transducer*
$$(\mathbf{t} \downarrow \mathbf{a} \uparrow \mathbf{a})$$
realizes relation (7) and can be constructed in time $O(|\mathbf{t}||\mathbf{a}|^2)$.

4. *There is an $O(|T_{\mathbf{s}}|^2 + r|Q_{\mathbf{s}}|^2)$ time algorithm that decides whether a given transducer* **s** *is functional [6,28], where r is the size of the alphabet.*

Using the above observations, we present first a preliminary error-detection-based algorithm for computing the desired edit distance.

```
Algorithm PrelimDistErrDetect
0.  Input: NFA $\mathbf{a}$
1.  Let $B_{\mathbf{a}}$ be the edit distance bound in Lemma 2
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 $\le$ max)
    a)  Let $k \leftarrow \lfloor(\min + \max)/2\rfloor$
    b)  Construct transducer $\mathbf{sid}_k$ realizing the channel sid$(k)$—see Figure 1
    c)  Construct the transducer $\mathbf{t}'_k \leftarrow (\mathbf{sid}_k \downarrow \mathbf{a} \uparrow \mathbf{a})$
    d)  If $L(\mathbf{a})$ is error-detecting for sid$(k)$, let min $\leftarrow k + 1$
        Else let max $\leftarrow k - 1$
4.  return min
```

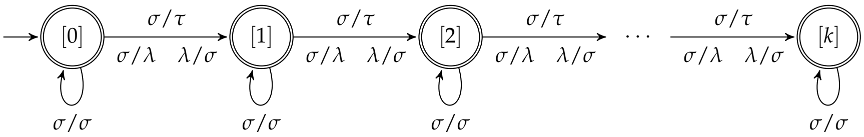

Figure 1. The input-preserving transducer $\mathbf{sid}_k$ realizing the channel sid(k). Each edge label σ/σ represents many transitions, one for each symbol σ of the alphabet, and similarly for σ/λ and λ/σ. Each edge label σ/τ represents many transitions, one for each pair of *distinct* symbols σ and τ from the alphabet. Thus, if the alphabet size is r, then the transducer has $O(k)$ states and $O(r^2k)$ transitions.

Remark 2. *Step (3d) of the above algorithm can be computed using the transducer functionality algorithm on* $\mathbf{t}'_k$*, which leads again to a polynomial but expensive algorithm. It turns out, however, using standard logical arguments, that*

$$\textit{Condition (6) is equivalent to whether } (\mathbf{t} \downarrow \mathbf{a} \uparrow \mathbf{a}) \textit{ realizes a partial identity,}$$

when **t** *realizes γ—in the above algorithm,* **t** *is* $\mathbf{sid}_k$. *Moreover, [6], there is an*

$$O(|T_{\mathbf{t}}| + r|Q_{\mathbf{t}}|)$$

time algorithm that tests whether a given transducer **t** *realizes a partial identity, where r is the size of the alphabet.*

Corollary 1. *Consider the algorithm* `PrelimDistErrDetect`. *Using the partial identity test for* $\mathbf{t}'_k$ *in step 3d, the algorithm computes the edit distance of a language given via a trim NFA* **a** *in time*

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

where r is the cardinality of the alphabet used in $T_{\mathbf{a}}$, and $n = |T_{\mathbf{a}}|$.

Proof. The correctness of the algorithm follows from Remarks 1 and 2. For the time complexity, the whole loop will perform $O(\log B_{\mathbf{a}})$ iterations. In each iteration, the value k is used to construct the transducer $\mathbf{sid}_k$ shown in Figure 1 with alphabet being the set of alphabet symbols appearing in the definition of $\mathbf{a}$. Then, the transducer $\mathbf{t}'_k$ is constructed having $O(k|Q_{\mathbf{a}}|^2)$ states and $O(kr^2|T^2_{\mathbf{a}}|)$ transitions. Then, the partial identity of $\mathbf{t}'_k$ is tested in time $O(|T_{\mathbf{a}}|^2kr^2)$. 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 $O(n)$ and, assuming a fixed alphabet, the above algorithm operates in time $O(n^3 \log n)$, which is asymptotically better than the time complexity of the algorithm in [4], even when the given automaton is a DFA.

4. An $O(n^2d)$ Algorithm for Edit Distance via Error-Detection

In this section, we observe that the algorithm of the previous section repeats a lot of computations, and we eliminate those repeated computations to arrive at an improved algorithm that computes the edit distance d of a trim NFA $\mathbf{a}$ in time $O(n^2dr^2)$, where r is the cardinality of the alphabet used in $T_{\mathbf{a}}$, and $n = |T_{\mathbf{a}}|$. The improved algorithm is based on the following two observations:

- The previous algorithm starts the binary search loop by constructing the transducer $\mathbf{t}'_{\lfloor B_{\mathbf{a}}/2 \rfloor}$, but the edit distance might be much smaller than $\lfloor B_{\mathbf{a}}/2 \rfloor$. It turns out that it is more efficient in the end to construct in turn $\mathbf{t}'_1, \mathbf{t}'_2, \ldots$ until the first $\mathbf{t}'_d$ that does not realize a partial identity.
- If $\mathbf{t}'_k$ is constructed and tested that does not realize a partial identity, then the transducer $\mathbf{t}'_{k+1}$ is constructed from scratch and the partial identity test is repeated for the part of $\mathbf{t}'_{k+1}$ that corresponds to $\mathbf{t}'_k$. We shall define the transducer $\mathbf{t}''_{k+1}$ to be the part that is added to $\mathbf{t}'_k$ in order to obtain $\mathbf{t}'_{k+1}$, plus some initial state. Moreover, we shall show that, if $\mathbf{t}'_k$ realizes a partial identity, then $\mathbf{t}'_{k+1}$ realizes a partial identity if and only if $\mathbf{t}''_{k+1}$ does so. Thus, the partial identity test in each step will apply only to the new part that is added to the transducer of the previous step.

We proceed with details based on the above observations.

Product construction of trim $\mathbf{t}' = \mathbf{t} \downarrow \mathbf{a} \uparrow \mathbf{a}$, given transducer $\mathbf{t}$ and NFA $\mathbf{a}$. As usual in cross product constructions, the states of $\mathbf{t}'$ are triples of the form (φ, q, q'), where φ is a state of $\mathbf{t}$, and q, q' are states of $\mathbf{a}$. The initial state of $\mathbf{t}'$ is (φ_0, q_0, q_0), where φ_0 is the initial state of $\mathbf{t}$ and q_0 is the initial state of $\mathbf{a}$. The construction is *incremental,* starting with the creation of (φ_0, q_0, q_0); then:

- If state (φ, q, q') has been created, and there are transitions $\varphi \xrightarrow{x/y} \psi$, $q \xrightarrow{x} r$, $q' \xrightarrow{y} r'$ of $\mathbf{t}$, $\mathbf{a}^\lambda$ and $\mathbf{a}^\lambda$, respectively, then the transition
$$(\varphi, q, q') \xrightarrow{x/y} (\psi, r, r')$$
is added in $\mathbf{t}'$. Here, $\mathbf{a}^\lambda$ is the λ-NFA that results if we add in $\mathbf{a}$ the loop transitions (q, λ, q) to all states q of $\mathbf{a}$.

The final states of $\mathbf{t}'$ are those constructed triples consisting of final states in $\mathbf{t}$ and $\mathbf{a}$. In the end, we also make $\mathbf{t}'$ trim.

Optimized construction of $\mathbf{t}''_{k+1}$ and $\mathbf{t}'_{k+1}$ from the trim $\mathbf{t}'_k$. Suppose that $\mathbf{t}'_k$ has been constructed, where initially $\mathbf{t}'_1 = \mathbf{sid}_1 \downarrow \mathbf{a} \uparrow \mathbf{a}$. Constructing $\mathbf{t}'_{k+1}$ using $\mathbf{t}'_k$ will be done again incrementally. The *first phase* of the incremental construction is to add the new transitions

$$([k], q, q') \xrightarrow{x/y} ([k+1], r, r'), \tag{8}$$

where x/y is an error and $q \xrightarrow{x} r$, $q' \xrightarrow{y} r'$ are transitions in $\mathbf{a}^\lambda$. There will be no new transitions of the form $([i], q, q') \xrightarrow{x/y} ([k+1], r, r')$ for $i < k$, because the transducer $\mathbf{sid}_{k+1}$ has no transitions from any state $[i]$ with $i < k$ to state $[k+1]$. Note that the numbers of new transitions and new states created as in (8) are $O(|T_{\mathbf{a}}|^2r^2)$ and $O(|Q_{\mathbf{a}}|^2)$, respectively.

After the first phase, the incremental construction proceeds from the new states $([k+1], r, r')$ in (8). Any new transition must be of the form

$$([k+1], q, q') \xrightarrow{\sigma/\sigma} ([k+1], r, r'), \tag{9}$$

where $\sigma \in \Sigma$. This is because the transducer $\mathbf{sid}_{k+1}$ has only transitions of the form $[k+1] \xrightarrow{\sigma/\sigma} [k+1]$ going out of the state $[k+1]$. The process ends when no new states are created. The transitions and final states of the transducer $\mathbf{t}'_{k+1}$ are those in $\mathbf{t}'_k$ plus the newly created ones, after removing any new states that cannot reach a final state (thus, also $\mathbf{t}'_{k+1}$ is trim). The transducer $\mathbf{t}''_{k+1}$ has as transitions and final states only the newly created ones, and has as initial state a new state $[-1]$ with transitions $[-1] \xrightarrow{\lambda/\lambda} ([k], q, q')$, for all states of the form $([k], q, q')$. □

Lemma 3. *Suppose the trim transducer $\mathbf{t}'_k$ realizes a partial identity.*

- *If C_1 is a computation of $\mathbf{t}'_k$ ending at a state of the form $([k], p, p')$, then the label of C_1 is of the form (w_1, w_1).*
- *$\mathbf{t}'_{k+1}$ realizes a partial identity if and only if $\mathbf{t}''_{k+1}$ does so.*

Proof. For the first statement, consider any computation C_1 of $\mathbf{t}'_k$ having some label (w_1, w'_1) and ending at a state of the form $([k], p, p')$. We show that $w_1 = w'_1$. If the state $([k], p, p')$ is final, then C_1 is an accepting computation, which implies $w_1 = w'_1$, as $\mathbf{t}'_k$ realizes a partial identity. If $([k], p, p')$ is not a final state, then, as $\mathbf{t}'_k$ is trim, there is a path C'_1 from $([k], p, p')$ to a final state of $\mathbf{t}'_k$, where all states of that path are of the form $([k], r, r')$ and all labels of that path are of the form σ/σ—this is because any transition of $\mathbf{sid}_k$ from state $[k]$ can only go to state $[k]$ and can only have a label of the form σ/σ. Thus, there is an accepting path of $\mathbf{t}'_k$ of the form $C_1C'_1$ with label $(w_1 z, w'_1 z)$ for some nonempty word z. Then, as $\mathbf{t}'_k$ realizes a partial identity, we have that $w_1 z = w'_1 z$, which implies $w_1 = w'_1$, as required.

For the 'only if' part of the second statement, assume that $\mathbf{t}'_{k+1}$ realizes a partial identity. Consider any accepting computation C_2 of $\mathbf{t}''_{k+1}$ with some label (w_2, w'_2). We show that $w_2 = w'_2$. Let $[-1] \xrightarrow{\lambda/\lambda} ([k], p, p')$ be the first transition of C_2. Let C'_2 be the path that results when we remove the first transition of C_2. By the construction of $\mathbf{t}'_{k+1}$, there is a computation C_1 of $\mathbf{t}'_k$ that ends at state $([k], p, p')$. Let (w_1, w_1) be the label of C_1. Then, $C_1C'_2$ is an accepting computation of $\mathbf{t}'_{k+1}$ with label $(w_1w_2, w_1w'_2)$. As $\mathbf{t}'_{k+1}$ realizes a partial identity, $w_1w_2 = w_1w'_2$, which implies $w_2 = w'_2$, as required.

For the 'if' part of the second statement, assume that $\mathbf{t}''_{k+1}$ realizes a partial identity. Consider any accepting computation C of $\mathbf{t}'_{k+1}$. We show that the label of C must be of the form (w, w). If C is already a computation of $\mathbf{t}'_k$, then this holds, as $\mathbf{t}'_k$ realizes a partial identity. Now suppose that $C = C_1C_2$ such that C_1 is a computation of $\mathbf{t}'_k$ and C_2 is a path in $\mathbf{t}'_{k+1}$ that starts with a transition as in (8) and then uses transitions as in (9). Let $([k], p, p')$ be the last state of C_1, which is also the first state of C_2. Then, C_1 has some label (w_1, w_1). In addition, the path

$$\left([-1] \xrightarrow{\lambda/\lambda} ([k], p, p')\right) C_2$$

is an accepting computation of $\mathbf{t}''_{k+1}$, which implies that it has some label (w_2, w_2). Hence, the label of C is (w_1w_2, w_1w_2) and, therefore, $\mathbf{t}'_k$ realizes a partial identity. □

The improved algorithm is shown next:

```
Algorithm DistErrDetect
0. Input: NFA $\mathbf{a}$
1. Construct the transducer $\mathbf{sid}_1$ realizing the channel $\mathrm{sid}(1)$—see Figure 1
2. Construct the trim transducer $\mathbf{t}'_1 = \mathbf{sid}_1 \downarrow \mathbf{a} \uparrow \mathbf{a}$
3. Let $k \leftarrow 1$
4. Let $\mathbf{s} \leftarrow \mathbf{t}'_1$
```

```
5. while (s realizes a partial identity)
    a) Construct $\mathbf{t}''_{k+1}$ and $\mathbf{t}'_{k+1}$ from $\mathbf{t}'_k$ using the optimized construction
    b) Let $\mathbf{s} \leftarrow \mathbf{t}''_{k+1}$
    c) Let $k \leftarrow k+1$
6. return $k$
```

Theorem 1. *Algorithm* `DistErrDetect` *computes the edit distance of a language given via a trim NFA* **a** *in time*

$$O(n^2dr^2),$$

where d is the computed edit distance, $n = |T_{\mathbf{a}}|$, and r is the cardinality of the alphabet used in $T_{\mathbf{a}}$.

Proof. The correctness of the algorithm follows from the optimized construction and the above lemma. For the time complexity of the algorithm, we note the following. First, $\mathbf{t}'_1$ is constructed in time $O(|\mathbf{a}|^2r^2)$. Then, $\mathbf{t}''_2, \ldots, \mathbf{t}''_d$ are constructed according to the optimized construction. Each of these is constructed in time $O(|\mathbf{a}|^2r^2)$ and has $O(|Q_{\mathbf{a}}|^2)$ states and $O(|T_{\mathbf{a}}|^2r^2)$ transitions. In addition, each $\mathbf{t}''_k$ is tested for partial identity in time $O\big(|T_{\mathbf{a}}|^2r^2 + |Q_{\mathbf{a}}|^2r\big)$, which is $O(|\mathbf{a}|^2r^2)$. □

5. An $O(n^2d)$ Algorithm for Edit Distance via Input-Altering Transducers

In this section, we present another algorithm for computing the desired edit distance via input-altering transducers—see Theorem 3 and the associated algorithm. A transducer **t** is called *input-altering*, if

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

that is, the output of **t** is never equal to the input used.

We explain now how input-altering transducers are related to edit-distance and error-detection. Let **t** be a transducer. A language L is **t**-*independent*, [29,30], if

$$u, v \in L \text{ and } u \in \mathbf{t}(v) \to u = v. \tag{10}$$

Of course, when $R(\mathbf{t})$ is input-preserving, then **t**-independence is the same as error-detection for the channel $R(\mathbf{t})$, and condition (10) can be tested as explained in Remark 2. On the other hand, if the transducer **t** is input-altering, then [30], condition (10) is equivalent to

$$\mathbf{t}(L) \cap L = \emptyset. \tag{11}$$

If L is accepted by some NFA **a**, then the above condition can be tested using two product constructions: first, construct an NFA **b** accepting $\mathbf{t}(L)$, then construct an NFA **c** by intersecting **b** with **a**, and then test whether there is a path from the start to a final state of **c**. Thus, condition (11) can be tested in time

$$O(|\mathbf{a}|^2|\mathbf{t}|). \tag{12}$$

Certain types of input-altering transducers are useful in constructing maximal **t**-independent languages [30]. In Theorem 2, we show how an input-altering transducer can be used to model the edit operations used in the definition of the edit distance.

5.1. An Input-Altering Transducer for Edit-Distance

We shall define the input-altering transducer $\mathbf{ia}_k$, which is partially shown in Figure 2. 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 $\delta(u,v) \le i$. 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.

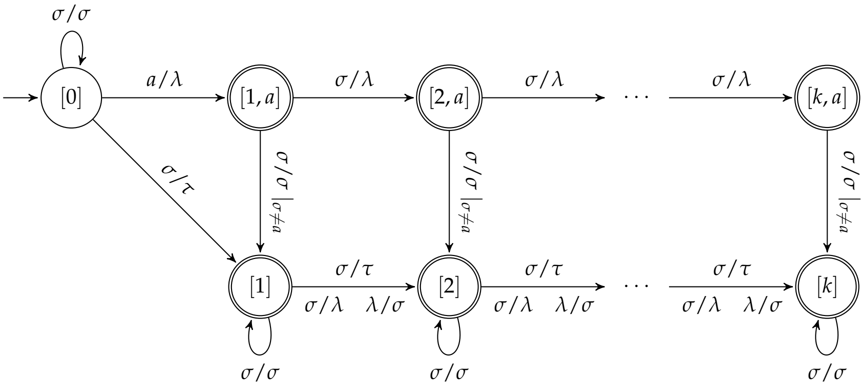

Figure 2. A segment of the input-altering transducer $\mathbf{ia}_k$: for each $a \in \Sigma$ the complete transducer has k states of the form $[i,a]$. The labels σ and τ on an edge mean: one edge for each $\sigma, \tau \in \Sigma$ with $\sigma \neq \tau$; for some edge sets, additional restrictions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

Definition 1. *The transducer*

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

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 transitions in $\mathbf{ia}_k$ *can be divided into the four sets of edges* $E = E_0 \cup E_s \cup E_i \cup E_d$*. The transitions 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*):*

$$\begin{aligned}
E_0 = & \Big\{ [i] \xrightarrow{\sigma/\sigma} [i] \;\Big|\; \sigma \in \Sigma, 0 \leq i \leq k \Big\} \cup, && (13)\\
& \Big\{ [i,a] \xrightarrow{\sigma/\sigma} [i] \;\Big|\; a, \sigma \in \Sigma, a \neq \sigma, 1 \leq i \leq k \Big\}, && (14)\\
E_s = & \Big\{ [i] \xrightarrow{\sigma/\tau} [i+1] \;\Big|\; \sigma, \tau \in \Sigma, \sigma \neq \tau, 0 \leq i < k \Big\}, && (15)\\
E_i = & \Big\{ [i] \xrightarrow{\lambda/\sigma} [i+1] \;\Big|\; \sigma \in \Sigma, 1 \leq i < k \Big\}, && (16)\\
E_d = & \Big\{ [0] \xrightarrow{a/\lambda} [1,a] \;\Big|\; a \in \Sigma \Big\} \cup, && (17)\\
& \Big\{ [i] \xrightarrow{\sigma/\lambda} [i+1] \;\Big|\; \sigma \in \Sigma, 1 \leq i < k \Big\} \cup, && (18)\\
& \Big\{ [i,a] \xrightarrow{\sigma/\lambda} [i+1,a] \;\Big|\; a, \sigma \in \Sigma, 1 \leq i < k \Big\}. && (19)
\end{aligned}$$

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

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

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 [23].

Definition 2. *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 that follows (a/λ) in h (if any) is of the form σ/σ with $\sigma \in \Sigma \setminus \{a\}$.*

Example. The edit string $(a/a)(a/b)(a/\lambda)(\lambda/a)$ is reduced as its first error is a substitution. The edit string $(a/a)(a/\lambda)(b/b)(b/a)$ is reduced as well. The edit string $(\lambda/a)(a/a)$ is not reduced as it starts with an insertion, and the edit string $(a/\lambda)(b/a)(b/b)$ is not reduced either.

The proofs of the next two lemmata are given in the appendix.

Lemma 4. *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 $\delta(u,v)$.*

Lemma 5. *Let $k \in \mathbb{N}$ and let u,v be words. The following statements hold true with respect to the transducer $\mathbf{ia}_k$.*

1. *In $\mathbf{ia}_k^{\mathrm{e}}$, 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.*
2. *If $1 \leq \delta(u,v) \leq k$ and h is a reduced edit string realizing $\delta(u,v)$, then h is accepted by $\mathbf{ia}_k^{\mathrm{e}}$.*
3. *If $v \in \mathbf{ia}_k(u)$, then $1 \leq \delta(u,v) \leq k$.*

Theorem 2. *For each $k \in \mathbb{N}$, the transducer $\mathbf{ia}_k$ is input-altering and of size $O(kr^2)$, where r is the cardinality of the alphabet, and satisfies the following condition, for any language L containing at least two words*

$$\mathbf{ia}_k(L) \cap L = \emptyset \ \textit{if and only if} \ \delta(L) > k. \tag{20}$$

Proof. By construction, it follows that $\mathbf{t}_k$ is trim and has $O(rk)$ states and $O(kr^2)$ transitions. Hence, it is indeed of size $O(kr^2)$. The third statement of Lemma 5 implies that the transducer is input-altering. Next, we show that (20) is true for all languages L containing at least two words.

First, for the 'if' part, assume $\delta(L) > k$ and consider any words $u, v \in L$. We need to prove $v \notin \mathbf{ia}_k(u)$. If $u = v$, then this holds as $\mathbf{ia}_k$ is input-altering. Else, it follows from the third statement of Lemma 5. Now, for the 'only if' part, assume

$$\mathbf{ia}_k(L) \cap L = \emptyset, \tag{21}$$

but, for the sake of contradiction, suppose there are different words $u, v \in L$ such that $1 \leq \delta(u,v) \leq k$. Let h be a reduced edit string realizing $\delta(u,v)$. By the second statement of Lemma 5, h is accepted by $\mathbf{ia}_k^{\mathrm{e}}$ via some path P^{e} and, therefore, either of (u,v) and (v,u) is the label of the path P of $\mathbf{ia}_k$, that is, we have $u \in \mathbf{ia}_k(v)$ or $v \in \mathbf{ia}_k(u)$, which contradicts (21). □

Corollary 2. *For each NFA $\mathbf{a}$ accepting at least two words and for each transducer $\mathbf{ia}_k$, with $k \in \mathbb{N}$, the following condition is satisfied:*

$$R\big(\mathbf{ia}_k \downarrow \mathbf{a} \uparrow \mathbf{a}\big) = \emptyset \ \textit{if and only if} \ \delta\big(L(\mathbf{a})\big) > k. \tag{22}$$

Proof. The statement follows from the above theorem and the fact (based on standard logic arguments) that $R\big(\mathbf{ia}_k \downarrow \mathbf{a} \uparrow \mathbf{a}\big) = \emptyset$ is equivalent to $\mathbf{ia}_k\big(L(\mathbf{a})\big) \cap L(\mathbf{a}) = \emptyset$. □

The reason why condition $R\big(\mathbf{ia}_k \downarrow \mathbf{a} \uparrow \mathbf{a}\big) = \emptyset$ is preferred to the equivalent one in Theorem 2 is explained further below in the remark that follows Theorem 3.

5.2. The Second $O(n^2d)$ Algorithm for Edit Distance

Here, we use the results of the previous subsection to arrive at the second algorithm for computing the desired edit distance. Corollary 2 implies that the *preliminary* algorithm `PrelimDistInpAlter` shown below correctly computes the desired edit distance. Moreover, by reasoning as in the proof of Corollary 1, it follows that this algorithm also executes in time $O(n^2r^2B_{\mathbf{a}} \log B_{\mathbf{a}})$, where r is the cardinality of the alphabet used in $T_{\mathbf{a}}$, and $n = |T_{\mathbf{a}}|$.

```
Algorithm PrelimDistInpAlter
0.  Input: NFA $\mathbf{a}$
1.  Let $B_{\mathbf{a}}$ be the bound in Lemma 2
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 $\le$ max)
    a)  Let $k \leftarrow \lfloor(\min + \max)/2\rfloor$
    b)  Construct the transducer $\mathbf{ia}_k$ (see Figure 2)
    c)  Construct the transducer $\mathbf{t}'_k \leftarrow \mathbf{ia}_k \downarrow \mathbf{a} \uparrow \mathbf{a}$
    d)  If $\big(R(\mathbf{t}'_k) = \emptyset\big)$ let min $\leftarrow k + 1$
        Else let max $\leftarrow k - 1$
4.  return min
```

We discuss now how to improve the above algorithm. The two observations we made at the beginning of Section 4 apply here as well if, instead of partial identity of $\mathbf{t}'_k$, we talk about the emptiness of $\mathbf{t}'_k$. Thus, we want the improved algorithm to construct in turn $\mathbf{t}'_1, \mathbf{t}'_2, \ldots$ until the first $\mathbf{t}'_d$ with $R(\mathbf{t}'_d) \neq \emptyset$. Moreover, when $\mathbf{t}'_k$ has been constructed and realizes $\emptyset$, we continue in the next step with new transitions added to $\mathbf{t}'_k$ in order to get $\mathbf{t}'_{k+1}$.

Optimized construction of $\mathbf{t}'_{k+1}$ from the trim $\mathbf{t}'_k$. Suppose that the trim $\mathbf{t}'_k$ has been constructed, where initially $\mathbf{t}'_1 \leftarrow \mathbf{ia}_1 \downarrow \mathbf{a} \uparrow \mathbf{a}$. Constructing $\mathbf{t}'_{k+1}$ using $\mathbf{t}'_k$ will be done again incrementally. The *first phase* of the incremental construction is to add two sets of new transitions: the new transitions

$$([k], q, q') \xrightarrow{x/y} ([k+1], r, r'), \tag{23}$$

where x/y is an error and $q \xrightarrow{x} r$, $q' \xrightarrow{y} r'$ are transitions in $\mathbf{a}^\lambda$; and the new transitions

$$([k, a], q, q') \xrightarrow{\sigma/\lambda} ([k+1, a], r, r'), \tag{24}$$

where $a, \sigma \in \Sigma$, and $q \xrightarrow{\sigma} r$, $q' \xrightarrow{\lambda} r'$ are transitions in $\mathbf{a}^\lambda$. Note that the total numbers of new transitions and states created in the first phase are $O(|T_{\mathbf{a}}|^2r^2)$ and $O(|Q_{\mathbf{a}}|^2r)$, respectively.

After the first phase, the incremental construction proceeds from the new states $([k+1], r, r')$ and $([k+1, a], r, r')$. Any new transition must be of the form

$$([k+1], r, r') \xrightarrow{\sigma/\sigma} ([k+1], t, t') \quad \text{or} \quad ([k+1, a], r, r') \xrightarrow{\sigma/\sigma} ([k+1], t, t'), \tag{25}$$

where $\sigma \in \Sigma$ and, in the second case above, $\sigma \neq a$. This is because the transducer $\mathbf{ia}_{k+1}$ has only transitions of the form $[k+1] \xrightarrow{\sigma/\sigma} [k+1]$ and $[k+1, a] \xrightarrow{\sigma/\sigma} [k+1]$, with $\sigma \neq a$, going out of the state $[k+1]$. The incremental process ends when no new states are created. The transitions and final states of the transducer $\mathbf{t}'_{k+1}$ are those in $\mathbf{t}'_k$ plus the newly created ones, after removing any new states that cannot reach a final state (thus, also $\mathbf{t}'_{k+1}$ is trim). □

Remark 3. *If the trim transducer $\mathbf{t}'_k$ has no final states, then $\mathbf{t}'_{k+1}$ has no final states if and only if none of the new created states in the optimized construction is a final state.*

```
Algorithm DistInpAlter
0.  Input: NFA $\mathbf{a}$
1.  Construct the transducer $\mathbf{ia}_1$—see Figure 2
2.  Construct the trim transducer $\mathbf{t}'_1 \leftarrow \mathbf{ia}_1 \downarrow \mathbf{a} \uparrow \mathbf{a}$
3.  Let $k \leftarrow 1$
4.  Let $\mathbf{s} \leftarrow$ the set of states of $\mathbf{t}'_1$
5.  while ($\mathbf{s}$ contains no final states)
    a)  Construct $\mathbf{t}'_{k+1}$ from $\mathbf{t}'_k$ using the optimized construction
    b)  Let $\mathbf{s}$ be the set of new states in the optimized construction
    c)  Let $k \leftarrow k+1$
6.  return $k$
```

Theorem 3. *Algorithm* `DistInpAlter` *computes the edit distance of the language given via a trim NFA* $\mathbf{a}$ *in time*

$$O(n^2 d r^2),$$

where d is the computed edit distance, $n = |T_{\mathbf{a}}|$, and r is the cardinality of the alphabet used in $T_{\mathbf{a}}$.

Proof. The correctness of the algorithm follows from the above optimized construction and Corollary 2. For the time complexity of the algorithm, we note the following: first, $\mathbf{t}'_1$ is constructed in time $O(|\mathbf{a}|^2 r^2)$. Then, $\mathbf{t}'_2, \ldots, \mathbf{t}'_d$ are constructed according to the optimized construction. Each of these is constructed in time $O(|\mathbf{a}|^2 r^2)$ and has $O(|Q_{\mathbf{a}}|^2 r)$ states and $O(|T_{\mathbf{a}}|^2 r^2)$ transitions. In addition, each $\mathbf{t}'_k$ is tested for final states in linear time. □

Remark 4. *When a final state f, say, of $\mathbf{t}'_k$ is created, then we know that there is an accepting path of $\mathbf{ia}_k \downarrow \mathbf{a} \uparrow \mathbf{a}$ ending at f. The label of that path is a word pair (u, v) such that $\delta(u, v) = k$. Thus, the above algorithm can be modified to return not only the edit distance of $L(\mathbf{a})$, but also a witness pair for that distance.*

6. Implementation and Testing

As both algorithms `DistErrDetect` and `DistInpAlter` have the same theoretical complexity, we chose to implement one of the two. We chose to implement `DistInpAlter` because it requires a simpler test for each constructed transducer (although $\mathbf{ia}_k$ is slightly more complex than $\mathbf{sid}_k$, the test for partial identity, [6], is more sophisticated than testing merely for existence of final states). We have also implemented the preliminary algorithm `PrelimDistInpAlter`.

Our implementation uses the FAdo package for automata, version 1.3.5.1, [7], which is well maintained and provides several useful tools for manipulating automata. We have performed several tests (All tests were performed on a laptop with the following specification. Make: Apple, Model: MacBook Pro, Processor 2.5 GHz Intel Core i7, Memory (RAM): 16.00 GB, Operating System: macOS High Sierra Version 10.13.6.) for the correctness of these algorithms, as well as two sets of tests for the time complexity, which confirm the theoretical result that `DistInpAlter` is indeed faster than `PrelimDistInpAlter`. The two sets of tests correspond to two lists of DFAs, $(\mathbf{a}_n)$ and $(\mathbf{b}_n)$, shown in Figures 3 and 4. The first test set is such that the desired distance is equal to n, for each DFA $\mathbf{a}_n$, 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 automaton. The second test set is such that the desired distance is fixed, equal to 2, for all n.

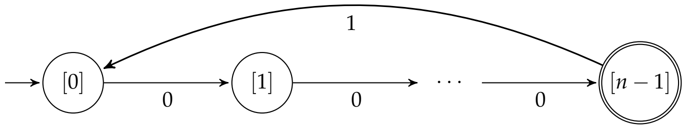

Figure 3. The automaton $\mathbf{a}_n$ accepting the language $0^{n-1}(10^{n-1})^*$.

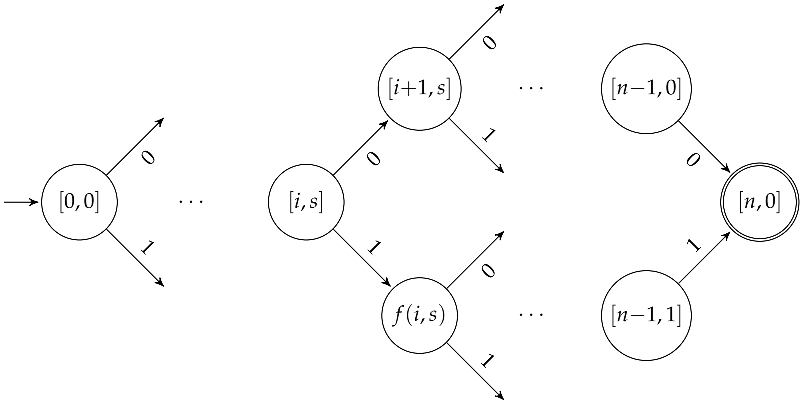

Figure 4. The automaton $\mathbf{b}_n$ accepting the Levenshtein code, which consists of all binary words $b_1 \cdots b_n$ of length n such that $(\sum_{i=1}^{n} i \cdot b_i)\%(n+1) = 0$, where '%' is the integer division remainder operation. This code has edit distance equal to 2. On the other hand, its distance for insertion/deletion errors only is 3. The automaton (before making it trim) has $n^2 + n + 1$ states: $[n, 0]$ and $[i, s]$, with $0 \le i \le n-1$ and $0 \le s \le n$. The meaning of state $[i, s]$ is that the automaton has read i bits $b_1 \cdots b_i$ and $s = 1 \cdot b_1 + \cdots + i \cdot b_i$. We have that $f(i, s) = [i+1, (s+i+1)\%(n+1)]$.

Table 1 shows the actual running times (in seconds) of the two algorithms on the DFAs $\mathbf{a}_{28}, \mathbf{a}_{41}, \mathbf{a}_{56}, \mathbf{a}_{76}, \mathbf{a}_{100}, \mathbf{a}_{124}, \mathbf{a}_{152}, \mathbf{a}_{184}$. The number in parentheses next to each $\mathbf{a}_n$ indicates the number of transitions in $\mathbf{a}_n$. The column d shows the computed edit distance, and the column $B_{\mathbf{a}_n}$ shows the computed upper bound on the edit distance (used in algorithm `PrelimDistInpAlter`).

Table 1. Outcomes of performance tests on the automata $(\mathbf{a}_n)$.

DFA	d	$B_{\mathbf{a}_n}$	PrelimDistInpAlter	DistInpAlter
$\mathbf{a}_{28}$ (28)	28	28	0.696s	0.078s
$\mathbf{a}_{41}$ (41)	41	41	3.977s	0.267s
$\mathbf{a}_{56}$ (56)	56	56	12.811s	0.691s
$\mathbf{a}_{76}$ (76)	76	76	52.086s	1.885s
$\mathbf{a}_{100}$ (100)	100	100	159.370s	4.841s
$\mathbf{a}_{124}$ (124)	124	124	354.306s	10.643s
$\mathbf{a}_{152}$ (152)	152	152	998.438s	21.991s
$\mathbf{a}_{184}$ (184)	184	184	2294.636s	43.484s

Table 2 shows the actual running times (in seconds) of the two algorithms on the DFAs $\mathbf{b}_6, \ldots, \mathbf{b}_{13}$. Again, the number in parentheses next to each $\mathbf{b}_n$ indicates the number of transitions in $\mathbf{b}_n$.

In both test sets, the empirical outcomes confirm the asymptotic outcome that the improved algorithm based on the optimized construction is faster than the preliminary one.

Table 2. Outcomes of performance tests on the automata $(\mathbf{b}_n)$.

DFA	d	$B_{\mathbf{b}_n}$	PrelimDistInpAlter	DistInpAlter
$\mathbf{b}_6$ (28)	2	3	0.112s	0.076s
$\mathbf{b}_7$ (41)	2	3	0.302s	0.196s
$\mathbf{b}_8$ (56)	2	4	0.731s	0.436s
$\mathbf{b}_9$ (76)	2	3	1.621s	0.927s
$\mathbf{b}_{10}$ (100)	2	3	3.223s	1.844s
$\mathbf{b}_{11}$ (124)	2	4	5.673s	3.238s
$\mathbf{b}_{12}$ (152)	2	5	61.416s	5.892s
$\mathbf{b}_{13}$ (184)	2	4	16.624s	9.272s

7. Conclusions

This paper represents a significant improvement in the time complexity of computing the inner edit distance of a given regular language. The performance tests of the implemented algorithm show that in practice the algorithm is reasonably fast for moderate size automata. As discussed in [4], this problem is related to the inherent capability of a language to detect substitution, insertion, and deletion errors.

The two preliminary algorithms can be applied to different distances as long as these distances can be related to appropriate transducers. For some of those distances, the idea in the optimized algorithms can also be used. For example, one can construct a transducer similar to $\mathbf{sid}_k$ for insertion/deletion only errors. A direction for future research is to investigate to what extent the methods used here can be extended to compute inner weighted distances or the inner edit distance with moves.

Author Contributions: All the authors contributed equally to this research. The outcome would not be possible without the contributions of all authors.

Funding: This research was supported by the Natural Science and Engineering Research Council of Canada (NSERC) Discovery Grants R2824A01 to Lila Kari and 220259 to Stavros Konstantinidis.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

In this appendix, we present the proofs of Lemmas 4 and 5.

Proof. *(Of Lemma 4)* The first statement already appears in [24]. 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 $\mathrm{inp}(h) = u$ and $\mathrm{out}(h) = v$. We show that h realizes $\delta(u, v)$ by proving that, for any edit string g realizing $\delta(u, v)$, $\mathrm{weight}(g) = n - m$. Indeed, first note that $\mathrm{weight}(g) \leq \mathrm{weight}(h) = n - m$. 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, $\mathrm{weight}(g) \geq d + i \geq d - i = n - m$, 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. We have the following subcases:

- If g_0' is empty or starts with an edit operation (σ/σ) in which $\sigma \in \Sigma \setminus \{a\}$, then the required h is g_0.
- If g_0' starts with an edit operation (x/τ) in which $\tau \in \Sigma \setminus \{a\}$ and $x \in \Sigma \cup \{\lambda\}$, then g_0' is of the form $g_0' = (x/\tau)g_1'$, and the required h is
$$h = (e_1 \cdots e_r)(a/\tau)(a_1/\lambda)\cdots(a_d/\lambda)(x/\lambda)g_1'.$$
- If g_0' starts with an edit operation (x/a) in which $x \in \Sigma \cup \{\lambda\}$, 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 $\mathrm{weight}(g_1) = \mathrm{weight}(g_0)$. 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. □

Proof. *(Of Lemma 5)* The first statement follows when we note that the definition of $\mathbf{ia}_k$ and $\mathbf{ia}_k^{\mathrm{e}}$ 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 $\mathbf{ia}_k^{\mathrm{e}}$ is indeed reduced.

For the second statement, consider any reduced edit string h realizing $\delta(u,v)$. We have two cases for the first error of h. 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-error (σ/σ) such that $\sigma \neq a$. Consider the following path of $\mathbf{ia}_k^{\mathrm{e}}$
$$P_1 = [0] \xrightarrow{(e_1\cdots e_r)}{}^{*} [0] \xrightarrow{(a/\lambda)(b_1/\lambda)\cdots(b_d/\lambda)}{}^{*} [1+d,a].$$

If h' is empty, then P_1 is an accepting path of $\mathbf{ia}_k^{\mathrm{e}}$. If h' is nonempty, then it is of the form $h' = (\sigma/\sigma)h''$, for some $\sigma \in \Sigma \setminus \{a\}$. Then, by definition of $\mathbf{ia}_k^{\mathrm{e}}$, the following is a path of $\mathbf{ia}_k^{\mathrm{e}}$
$$P_1\left([1+d,a] \xrightarrow{\sigma/\sigma} [1+d] \xrightarrow{h''}{}^{*} [1+d+\mathrm{weight}(h'')]\right)$$
accepting h. For the case where the first error in h is a substitution, one verifies that again h is accepted by $\mathbf{ia}_k^{\mathrm{e}}$.

For the third statement, if $v \in \mathbf{ia}_k(u)$, 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 two 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 $[r]$. In both 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, $u \neq v$. □

References

1. Sankoff, D.; Kruskal, J.B. (Eds.) *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*; CSLI Publications: Stanford, CA, USA, 1999.
2. Gusfield, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*; Cambridge University Press: New York, NY, USA, 1997.
3. Paluncic, F.; Abdel-Ghaffar, K.; Ferreira, H. Insertion/deletion detecting codes and the boundary problem. *IEEE Trans. Inf. Theory* **2013**, *59*, 5935–5943. [CrossRef]
4. Konstantinidis, S. Computing the edit distance of a regular language. *Inf. Comput.* **2007**, *205*, 1307–1316. [CrossRef]
5. Konstantinidis, S.; Silva, P. Computing maximal error-detecting capabilities and distances of regular languages. *Fundam. Inform.* **2010**, *101*, 257–270.
6. Allauzen, C.; Mohri, M. Efficient algorithms for testing the twins property. *J. Autom. Lang. Comb.* **2003**, *8*, 117–144.
7. FAdo. Tools for Formal Languages Manipulation. Available online: http://fado.dcc.fc.up.pt/ (accessed on 20 October 2018).
8. Wagner, R. Order-n correction for regular languages. *Commun. ACM* **1974**, *17*, 265–268. [CrossRef]
9. Pighizzini, G. How hard is computing the edit distance? *Inf. Comput.* **2001**, *165*, 1–13. [CrossRef]
10. Mohri, M. Edit-distance of weighted automata: General definitions and algorithms. *Intern. J. Found. Comput. Sci.* **2003**, *14*, 957–982. [CrossRef]
11. Kari, L.; Konstantinidis, S.; Perron, S.; Wozniak, G.; Xu, J. *Finite-State Error/Edit-Systems and Difference-Measures for Languages and Words*; Report 2003-01; Mathematics and Computing Science, Saint Mary's University: Halifax, NS, Canada, 2003.
12. Benedikt, M.; Puppis, G.; Riveros, C. The cost of traveling between languages. In *ICALP 2011, Part II. LNCS 6756*; Aceto, L., Henziger, M., Sgall, J., Eds.; Springer: Heidelberg, Germany, 2011; pp. 234–245.
13. Han, Y.S.; Ko, S.K.; Salomaa, K. Computing the edit-distance between a regular language and a context-free langauge. In *DLT 2012. LNCS 7410*; Yen, H.C., Ibarra, O., Eds.; Springer: Heidelberg, Germany, 2012; pp. 85–96.
14. Han, Y.S.; Ko, S.K.; Salomaa, K. Approximate matching between a context-free grammar and a finite-state automaton. In *CIAA 2013. LNCS 7982*; Konstantinidis, S., Ed.; Springer: Heidelberg, Germany, 2013; pp. 146–157.
15. Takabatake, Y.; Nakashima, K.; Kuboyama, T.; Tabei, Y.; Sakamoto, H. siEDM: An Efficient String Index and Search Algorithm for Edit Distance with Moves. *Algorithms* **2016**, *9*, 26. [CrossRef]
16. Ng, T. Prefix Distance Between Regular Languages. In Proceedings of the 21st CIAA, Seoul, South Korea, 19–22 July 2016; Volume 9705, pp. 224–235.
17. Berstel, J. *Transductions and Context-Free Languages*; B.G. Teubner: Stuttgart, Germany, 1979.
18. Wood, D. *Theory of Computation*; John Wiley & Sons: New York, NY, USA, 1987.
19. Rozenberg, G.; Salomaa, A. (Eds.) *Handbook of Formal Languages, Vol. I*; Springer: Berlin, Germany, 1997.
20. Yu, S. Regular Languages. *Handbook of Formal Languages, Vol. I*; Springer: Berlin, Germany, 1997; pp. 41–110.
21. Sakarovitch, J. *Elements of Automata Theory*; Cambridge University Press: Cambridge, UK, 2009.
22. Mateescu, A.; Salomaa, A. Formal Languages: an Introduction and a Synopsis. *Handbook of Formal Languages, Vol. I*; Springer: Berlin, Germany, 1997; pp. 1–39.
23. Kari, L.; Konstantinidis, S. Descriptional Complexity of Error/Edit Systems. *J. Autom. Lang. Comb.* **2004**, *9*, 293–309.
24. Levenshtein, V.I. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Sov. Phys. Dokl.* **1966**, *10*, 707–710.
25. Yang, M. Application and Implementation of Transducer Tools in Answering Certain Questions about Regular Languages. Master's Thesis, Department Mathematics and Computing Science, Saint Mary's University, Halifax, NS, Canada, 2012.

26. Schrijver, A. *Combinatorial Optimization: Polyhedra and Efficiency*; Springer Science & Business Media: Berlin, Germany, 2003.
27. Konstantinidis, S. Transducers and the Properties of Error-Detection, Error-Correction and Finite-Delay Decodability. *J. Univ. Comput. Sci.* **2002**, *8*, 278–291.
28. Béal, M.; Carton, O.; Prieur, C.; Sakarovitch, J. Squaring transducers: An efficient procedure for deciding functionality and sequentiality. *Theor. Comput. Sci.* **2003**, *292*, 45–63. [CrossRef]
29. Shyr, H.; Thierrin, G. Codes and Binary Relations. In *Séminaire d'Algèbre Paul Dubreil, Paris 1975–1976 (29ème Année)*; Lecture Notes in Mathematics; Springer: Berlin/Heidelberg, Germany, 1975; pp. 180–188.
30. Konstantinidis, S. Applications of Transducers in Independent Languages, Word Distances, Codes. In Proceedings of the DCFS 2017, Milano, Italy, 3–5 July 2017; Volume 10316, pp. 45–62.