# Finite-state error/edit-systems and difference-measures for languages and words*

Lila Kari[1], Stavros Konstantinidis[2,†], Steven Perron[2], Geoff Wozniak[1], Jing Xu[2]

[1]Dept. of Computer Science, University of Western Ontario,
London, Ontario, N6A 5B7 Canada,
lila@csd.uwo.ca, wozniak@csd.uwo.ca

[2]Dept. of Mathematics and Computing Science,
Saint Mary's University, Halifax, Nova Scotia, B3H 3C3 Canada,
s.konstantinidis@smu.ca, steven_perron@hotmail.com, j_xu@smu.ca

**Abstract**

We consider a special type of automaton, the weighted finite-state e-system (wfse-system), that allows us to describe formally the combinations of errors (edit operations) that are permitted in some information processing application. Given two regular languages and a wfse-system we can compute the set of all edit-strings that can transform a word of the first language to a word of the second one using only the edit operations permitted by the given wfse-system. Each wfse-system can be used to define a measure of the difference between words and languages. Our presentation provides a uniform treatment of certain algorithmic problems pertaining to the differences between such objects. In particular, we show how to find the Hamming distance of a given regular language in quadratic time and how to compute efficiently the general string to regular-language correction problem. Moreover, we discuss an implementation of our solution to this problem, which uses Grail to represent automata.

## 1 Introduction

The problem of measuring the difference between words (strings) and languages is important in various applications of information processing such as error control in data communications, bioinformatics, and spelling correction. Well-known measures of the difference between two words are the Hamming distance and the edit (or Levenshtein) distance, as well as the weighted edit distance. In general, the function describing the difference between two words need not be a distance function (a metric). Moreover, it is often the case that the errors (edit operations) that are used to transform one word to another can be combined only in certain ways. For example, in some applications of data communications [7] errors tend to occur in bursts, and in computer typesetting the likelihood of occurrences of certain word misspellings depend on the letters comprising the word – in particular, omitting the letter d is more probable when d follows e than when it follows g [2].

*Research partially supported by Grants R2824A01 and R220259 of the Natural Sciences and Engineering Research Council of Canada.

†Corresponding author.

Typical problems pertaining to differences between strings and languages are (i) computing the distance (also known as self-distance) of a given language, (ii) computing the edit-distance between two words, and (iii) correct a given word to a word of a given language using a minimum cost string of edit operations. The first problem can be used to find the maximum number of errors that a given code can detect. To our knowledge, it has not been addressed for the case of regular languages. The second problem can be solved using a dynamic programming algorithm – see [11]. The third problem is solved in [12] for regular languages and unrestricted edit operations using also dynamic programming. In [8] the author addresses this problem for more general classes of languages. Reference [1] discusses the interesting concept of $k$-reflexivity of a relation which generalizes the concept of distance between two languages. The definition of distance in [1], however, is totally different from ours.

In this work, we consider a special type of automaton, the weighted finite-state e-system (wfse-system), that allows us to describe formally the combinations of errors (edit operations) that are permitted in some information processing application. Each wfse-system can be used to define a "measure of the difference" between words and languages. Our presentation provides a uniform treatment of the above mentioned problems. The paper is organized as follows. In the next section we provide the basic notation about automata and e-strings, and we introduce the e-automaton accepting all the e-strings that can transform a word of one language to a word of another language without restrictions on the edit operations permitted. In Section 3, we define wfse-systems, show how they can be used to define measures of difference between words, and how to compute the automaton describing the differences between two languages according to the wfse-system in question. Section 4 shows a quadratic time algorithm for computing the Hamming distance of a given regular language. In Section 5 we show how to compute the general string to regular-language correction problem efficiently and discuss an implementation of our algorithm. Finally, Section 6 contains a few concluding remarks.

## 2 Basic Notation and the E-automaton

For a set $S$ we denote by $|S|$ the cardinality of $S$. An *alphabet* is a finite nonempty set of symbols. In the sequel we shall use a fixed alphabet $\Sigma$. A *word* or *string* (over $\Sigma$) is a finite sequence $a_1 \cdots a_n$ such that each $a_i$ is in $\Sigma$. The length of a word $w$ is denoted by $|w|$. The empty word, denoted $\lambda$, is the word of length zero. A (nondeterministic) finite automaton with $\lambda$-transitions, a *$\lambda$-NFA* for short, is a quintuple $A = (X, Q, s, F, T)$ such that $X$ is an alphabet, $Q$ is a finite nonempty set, the set of states, $s$ is the start state, $F$ is the set of final states, and $T$ is the set of transitions. Each transition in $T$ is of the form $q_1 x q_2$, where $q_1$ and $q_2$ are states and $x$ is either an alphabet symbol or $\lambda$ – we assume that the sets $Q$ and $X$ are disjoint. A *computation* of $A$ is an expression of the form $q_0 x_1 q_1 \cdots x_n q_n$ such that each $q_{i-1} x_i q_i$ is a transition in $T$. We say that such a computation is *accepting* if $q_0$ is the start state and $q_n$ is a final state and, in this case, $x_1 \cdots x_n$ is called the accepted word.

The automaton $A$ is called an *NFA* if $x$ is nonempty in every transition $q_1 x q_2$ of $A$. It is called deterministic, a *DFA* for short, if it is an NFA and for any two transitions of the form $q_1 x q_2$ and $q_1 x q_2'$ it is the case that $q_2 = q_2'$. We use $L(A)$ for the language *accepted by* $A$. The *size* $|A|$ of $A$ is the quantity $|Q| + |T|$. A $\lambda$-NFA is *trim* if every state is reachable from the start state and can reach a final state. Note that in every trim $\lambda$-NFA we have that $|Q| \leq |T| + 1$ and, therefore, the size of $A$ is $O(|T|)$. We assume that the reader is familiar with the basic concepts of automata and formal languages – see [10] for details.

We continue now with the concept of e-system as introduced in [5]. The alphabet $E_\Sigma$ of the *basic edit operations* is the set of all symbols $x/y$ such that $x, y \in \Sigma \cup \{\lambda\}$ and at least one of $x$ and $y$ is in $\Sigma$. If $x/y$ is in $E_\Sigma$ and $x$ is not equal to $y$ then we call $x/y$ an *error*. We write $\lambda/\lambda$ for the empty word over the alphabet $E_\Sigma$. We note that $\lambda$ is used as a formal symbol in the elements of $E_\Sigma$. For example, if $x$ and $y$ are in $\Sigma$ then $(x/\lambda)(x/y) \neq (x/y)(x/\lambda)$. The elements of $E_\Sigma^*$ are called *e-strings*. The *input* and *output parts* of an e-string $h = (x_1/y_1)\cdots(x_n/y_n)$ are the words (over $\Sigma$) $x_1 \cdots x_n$ and $y_1 \cdots y_n$, respectively. We write $\mathrm{inp}(h)$ for the input part and $\mathrm{out}(h)$ for the output part of the e-string $h$. An *e-system* is a subset of $E_\Sigma^*$ (or a language over $E_\Sigma$). The e-system is regular if it can be accepted by an NFA. In this case we shall call the NFA an *e-NFA*.

We close this section by introducing the e-NFA $A_1 \cap_E A_2$ of two given NFAs $A_1$ and $A_2$, where $E$ is a subset of the e-alphabet $E_\Sigma$. Recall – see for instance [13] – that for any two trim DFAs $A_1$ and $A_2$ one can use the standard product construction to define the trim DFA $A_1 \cap A_2$ of size $O(|A_1||A_2|)$ accepting the language $L(A_1) \cap L(A_2)$. The same construction remains valid even when the automata involved are NFAs. The construction of the e-NFA $A_1 \cap_E A_2$ can be viewed as a generalization of the standard product construction. We note that an interesting product construction between two copies of the same automaton is defined in [4] for the purpose of deciding the property of unique decodability for regular languages. Although the topic of [4] is not relevant to the present work, we wish to acknowledge that our product construction was inspired in part by the product construction in [4].

Given an NFA $A$ we denote by $A^\lambda$ the $\lambda$-NFA that results from $A$ if we add the transitions $q\lambda q$ for every state $q$ of $A$. It should be clear that the language accepted by $A^\lambda$ is equal to the language accepted by $A$.

**Construction 1.** Given two trim NFAs $A_1$ and $A_2$, and a subset $E$ of the e-alphabet $E_\Sigma$, the e-NFA $A_1 \cap_E A_2$ is defined to be the trim part of the e-NFA $C$ that is constructed as follows. The states of $C$ are all the pairs $(p, q)$, where $p$ and $q$ are states of $A_1$ and $A_2$ respectively. The start state of $C$ is the pair consisting of the start states of $A_1$ and $A_2$, and the set of final states of $C$ consists of all pairs $(p, q)$ such that $p$ and $q$ are final states of $A_1$ and $A_2$, respectively. The transitions of $C$ are of the form $(p_1, q_1)x/y(p_2, q_2)$ such that $x/y$ is in $E$, $p_1xp_2$ is a transition of $A_1^\lambda$, and $q_1yq_2$ is a transition of $A_2^\lambda$.

Next it is shown that the language accepted by the e-NFA $A_1 \cap_E A_2$ consists of all the e-strings $h$ that transform a word in $L(A_1)$ to a word in $L(A_2)$ using any combination of edit operations in $E$.

**Proposition 1** *The e-NFA $A_1 \cap_E A_2$ of two given NFAs $A_1$ and $A_2$ and a given set $E$ of edit operations is of size $O(|A_1||A_2|)$ and accepts the language*

$$L(A_1 \cap_E A_2) = \{h \mid h \in E^*,\ \mathrm{inp}(h) \in L(A_1),\ \mathrm{out}(h) \in L(A_2)\}.$$

*Proof.* First let $h = (x_1/y_1)\cdots(x_n/y_n)$ be an e-string accepted by some computation

$$(p_0, q_0)(x_1/y_1)(p_1, q_1)\cdots(x_n/y_n)(p_n, q_n)$$

of $A_1 \cap_E A_2$. By Construction 1, $h$ is in $E^*$ and the expressions $p_0x_1p_1 \cdots x_np_n$ and $q_0y_1q_1 \cdots y_nq_n$ are accepting computations of $A_1^\lambda$ and $A_2^\lambda$, respectively. Hence, $\mathrm{inp}(h)$ is in $L(A_1)$ and $\mathrm{out}(h)$ is in $L(A_2)$ as required.

Conversely, suppose $h = (x_1/y_1)\cdots(x_n/y_n)$ is an e-string in $E^*$ with $\mathrm{inp}(h)$ in $L(A_1)$ and $\mathrm{out}(h)$ in $L(A_2)$. Then one can define accepting computations of $A_1^\lambda$ and $A_2^\lambda$ of the form $p_0x_1p_1\cdots x_np_n$ and $q_0y_1q_1\cdots y_nq_n$, respectively. This implies that $(p_0,q_0)(x_1/y_1)(p_1,q_1)\cdots(x_n/y_n)(p_n,q_n)$ is an accepting computation of $A_1 \cap_E A_2$ and, therefore, $h$ must be in $L(A_1 \cap_E A_2)$. □

## 3 Weighted finite-state e-systems and difference-measures

In this section we define weighted finite-state e-systems, wfse-systems for short, that allow one to describe various combinations of edit operations and specify the cost of a sequence of edit operations. Each wfse-system can be used to define a measure of the difference between two words. Moreover, it is shown here how to compute a minimal such difference between two regular languages. Applications of this approach are discussed in the next sections.

**Definition 1** *A* weighted finite-state e-system, *or* wfse-system *for short, is a pair $\beta = (A_\beta, f_\beta)$ such that $A_\beta$ is a trim e-NFA and $f_\beta$ is a function, called the* weight function of $\beta$, *that assigns a nonnegative real number to every transition of $A_\beta$. The* alphabet of the e-system $L(A_\beta)$ *is a subset of the alphabet of edit operations and is denoted by $E_\beta$.*

*A wfse-system $\beta$ is called a* free e-system *if the e-NFA $A_\beta$ has exactly one state that is both the start and the final state (in this case there is no restriction in the way the edit operations in $E_\beta$ can be combined).*

Consider a wfse-system $\beta$ and a computation $q_0e_1q_1\cdots e_nq_n$ of the e-NFA $A_\beta$. The *cost of this computation* is equal to the sum of the weights of the transitions that appear in the computation: $\sum_{i=1}^n f_\beta(q_{i-1}e_iq_i)$. The *cost function $C_\beta$ defined by $\beta$* is the function that assigns to every e-string $h$ in $L(A_\beta)$ the minimum of the costs of the computations that accept $h$.

It should be clear that if $\beta$ is a free e-system then $L(A_\beta)$ is equal to $E_\beta^*$, and the cost function $C_\beta$ is a morphism of $L(A_\beta)$ into the monoid $(\mathbf{R}_+,+)$ of the nonnegative real numbers; that is, $C_\beta(\lambda/\lambda) = 0$ and $C_\beta(e_1\cdots e_n) = \sum_{i=1}^n C_\beta(e_i)$ for all e-strings $e_1\cdots e_n$ in $L(A_\beta)$. We agree that a free e-system $\beta$ is specified by the alphabet $E_\beta$ and the values $C_\beta(e)$, for all $e \in E_\beta$.

A wfse-system can be used to measure the difference between two words in $\Sigma^*$. More specifically, let $\beta$ be a wfse-system. For any two words $w_1$ and $w_2$ in $\Sigma^*$, the *$\beta$-difference* between $w_1$ and $w_2$ is the quantity

$$d_\beta(w_1,w_2) = \min\{C_\beta(h) \mid h \in L(A_\beta),\ \mathrm{inp}(h) = w_1,\ \mathrm{out}(h) = w_2\},$$

where we assume that $\min\emptyset = \infty$ (this means that it is impossible to transform $w_1$ to $w_2$ using the edit operations permitted by $\beta$). The concept of $\beta$-difference can be extended naturally in three ways as follows: Let $w$ be a word and let $L, L'$ be two languages, all over $\Sigma$. Then,

$$\begin{aligned} d_\beta(L) &= \min\{d_\beta(w_1,w_2) \mid w_1,w_2 \in L,\ w_1 \neq w_2\} \\ d_\beta(w,L) &= \min\{d_\beta(w,u) \mid u \in L\} \\ d_\beta(L,L') &= \min\{d_\beta(w,w') \mid w \in L,\ w' \in L'\}. \end{aligned}$$

**Definition 2** *Given a wfse-system $\beta$ and an e-NFA $D$, the wfse-system $\beta/D$ is defined such that $A_{\beta/D} = A_\beta \cap D$ and $f_{\beta/D}((p_1,q_1)e(p_2,q_2)) = f_\beta(p_1ep_2)$ for every transition $(p_1,q_1)e(p_2,q_2)$ of $A_{\beta/D}$ – note that $p_1ep_2$ must be a transition of the e-NFA $A_\beta$.*

**Proposition 2** *For every wfse-system $\beta$ and for every e-NFA $D$, we have that*

$$L(A_{\beta/D}) = L(A_\beta) \cap L(D) \quad \text{and} \quad C_{\beta/D}(h) = C_\beta(h),$$

*for all e-strings $h$ in $L(A_{\beta/D})$.*

*Proof.* The statement $L(A_{\beta/D}) = L(A_\beta) \cap L(D)$ follows immediately by the definition of $A_{\beta/D}$. Now consider an e-string $h = e_1 \cdots e_n$ in $L(A_{\beta/D})$, where each $e_i$ is an edit operation. By definition, $C_{\beta/D}(h)$ is the sum of the weights appearing in a minimum cost computation of $A_\beta \cap D$ that accepts $h$. Suppose

$$(p_0, q_0)e_1(p_1, q_1) \cdots e_n(p_n, q_n)$$

is such a computation. Then each weight $f_{\beta/D}((p_{i-1}, q_{i-1})e_i(p_i, q_i))$ is equal to $f_\beta(p_{i-1}e_ip_i)$, with each $p_{i-1}e_ip_i$ being a transition of $A_\beta$. Hence, $C_{\beta/D}(h) = \sum_{i=1}^{n} f_\beta(p_{i-1}e_ip_i) \geq C_\beta(h)$. If it were the case that $C_{\beta/D}(h) > C_\beta(h)$ then there would be a computation $p'_0e_1p'_1 \cdots e_np'_n$ of $A_\beta$ accepting $h$ with cost equal to $C_\beta(h)$. In this case, however,

$$(p'_0, q_0)e_1(p'_1, q_1) \cdots e_n(p'_n, q_n)$$

would be a computation of $A_{\beta/D}$ accepting $h$ with cost $C_\beta(h)$; a contradiction. □

**Definition 3** *Given a wfse-system $\beta$ and two NFAs $A_1$ and $A_2$, $A_1 \cap_\beta A_2$ is defined to be the e-NFA of the wfse-system $\beta/(A_1 \cap_{E_\beta} A_2)$; that is, $A_1 \cap_\beta A_2 = A_\beta \cap (A_1 \cap_{E_\beta} A_2)$.*

Next it is shown that the e-NFA $A_1 \cap_\beta A_2$ describes all the e-strings $h$ that can transform a word in $L(A_1)$ to a word in $L(A_2)$ using only the edit operations permitted by $\beta$.

**Proposition 3** *For every trim NFAs $A_1$ and $A_2$, and for every wfse-system $\beta$, the following statements hold true.*

$$\begin{aligned} L(A_1 \cap_\beta A_2) &= \{h \mid h \in L(A_\beta),\ \mathrm{inp}(h) \in L(A_1),\ \mathrm{out}(h) \in L(A_2)\} \\ d_\beta(L(A_1), L(A_2)) &= \min\{C_\beta(h) \mid h \in L(A_1 \cap_\beta A_2)\}. \end{aligned}$$

*Proof.* The first statement follows easily from Propositions 1 and 2. The second statement follows from the first one and Proposition 2. □

By Proposition 3, the $\beta$-difference between the two regular languages $L(A_1)$ and $L(A_2)$ is equal to the cost of the shortest weighted path from the start state to a final state in the graph $A_1 \cap_\beta A_2$. This value can be computed in time $O(n \log n)$, where $n$ is the size of the graph, using Dijkstra's algorithm.

**Corollary 1** *The following problem is computable in time $O(|A_1||A_2||A_\beta| \log(|A_1||A_2||A_\beta|))$.*
*Input: Two NFAs $A_1$ and $A_2$ and a wfse-system $\beta$.*
*Output: The $\beta$-difference $d_\beta(L(A_1), L(A_2))$.*

Note that Dijkstra's algorithm can be used to also return a shortest accepting path in $A_1 \cap_\beta A_2$. Moreover, the labels in the path would form an e-string $(x_1/y_1) \cdots (x_n/y_n)$ that defines a specific pair of words $x_1 \cdots x_n \in L(A_1)$ and $y_1 \cdots y_n \in L(A_2)$ with the property that these are words of the two languages whose $\beta$-difference is minimal.

In case the NFAs $A_1$ and $A_2$ are acyclic (accepting finite languages), the automaton $A_1 \cap_\beta A_2$ is also acyclic and, therefore, Dijkstra's algorithm would run in time linear with respect to the size of $A_1 \cap_\beta A_2$.

**Corollary 2** *The following problem is computable in time* $O(|A_1||A_2||A_\beta|)$.
*Input: Two acyclic DFAs* $A_1$ *and* $A_2$ *and a wfse-system* $\beta$.
*Output: The* $\beta$*-difference* $d_\beta(L(A_1), L(A_2))$.

# 4 Computing the Hamming distance of a regular language

The *Hamming e-system* is the free e-system $\sigma$ such that the set of permitted edit operations is $E_\sigma = \{x/y \mid x/y \in E_\Sigma,\ x \neq \lambda,\ y \neq \lambda\}$ and the cost of each edit operation $x/y \in E_\sigma$ is

$$C_\sigma(x/y) = \begin{cases} 1, & \text{if } x \neq y, \\ 0, & \text{if } x = y. \end{cases}$$

The *Hamming distance* between two words $w_1$ and $w_2$ is equal to $d_\sigma(w_1, w_2)$. The Hamming distance of a language $L$ is equal to $d_\sigma(L)$. In this section we show that the following proposition holds true.

**Proposition 4** *The following problem is computable in quadratic time.*
*Input: An NFA* $A$.
*Output: The Hamming distance of the language* $L(A)$.

Consider the wfse-system $\sigma'$ such that $E_{\sigma'} = E_\sigma$ and the e-NFA $A_{\sigma'}$ has two states $s$ and $g$, with $s$ being the start state and $g$ being the only final state, and transitions $sx/xs$, $sx/yg$, $gx/xg$, and $gx/yg$, for all symbols $x, y \in \Sigma$ with $x \neq y$. Moreover, $f_{\sigma'}(sx/xs) = f_{\sigma'}(gx/xg) = 0$ and $f_{\sigma'}(sx/yg) = f_{\sigma'}(gx/yg) = 1$, for all $x, y \in \Sigma$ with $x \neq y$. It is evident that $A_{\sigma'}$ accepts all e-strings in $E_\sigma^*$ containing at least one error. This implies that

$$d_\sigma(L(A)) = d_{\sigma'}(L(A), L(A)).$$

Therefore we can construct the e-NFA $A \cap_{\sigma'} A$ of size $O(|A|^2)$ and then, using Corollary 1, solve the above problem in time $O(|A|^2 \log |A|)$. Note that the factor $\log |A|$ in the time complexity is due to the fact that $A$ might contain cycles. However, using the fact that each weight in the graph $A \cap_{\sigma'} A$ is either 0 or 1, we show next that the shortest accepting path can be computed in time linear with respect to the size $O(|A|^2)$ of $A \cap_{\sigma'} A$, even when this graph contains cycles. This would also establish the validity of Proposition 4.

Consider a directed graph $G$ all the nodes of which are reachable from a start node $s$, and all the weights on the edges of the graph are 0 or 1. We can view $G$ as two graphs $G_0$ and $G_1$ such that $G_0$ results by removing from $G$ the edges of weight 1 and $G_1$ results by removing from $G$ the edges of weight 0. The main idea of the algorithm is as follows: Let $Q_0^{(0)}$ be equal to $s$. Define $Q_1^{(0)}$ to be $Q_0^{(0)}$ union the the set of all new nodes (i.e., nodes that have not been visited before) that are reachable from $Q_0^{(0)}$ via the graph $G_0$. The nodes in $Q_1^{(0)}$ are exactly those of distance 0 from $s$. Let $Q_0^{(1)}$ be the set of new nodes that are reachable from $Q_1^{(0)}$ using exactly one edge of $G_1$. Each node in $Q_0^{(1)}$ is of distance 1 from $s$. This process is repeated by defining $Q_1^{(i)}$ from $Q_0^{(i)}$, and $Q_0^{(i+1)}$ from $Q_1^{(i)}$, until all nodes in $G$ have been visited. It is evident that the nodes in $Q_1^{(i)}$ are exactly those of distance $i$ from $s$.

We turn the above idea to an algorithm by using two queues `Q0` and `Q1`, a counter `length` that plays the role of $i$, an array `Seen` to keep track of whether a node has been visited, and an array

`Distance` to store the distance of each node from the start node. The algorithm is presented below. As each node of the graph $G$ can be examined no more than two times, the algorithm runs in time proportional to the size of the graph.

**Algorithm.**

```
Define two empty queues Q0 and Q1
Initialize all entries of the boolean array Seen to false
Initialize all entries of the integer array Distance to 0
Q0.insert(startNode)
Seen[startNode] = true
length = 0;

while (Q0 is not empty)

  while (Q0 is not empty)
    a = Q0.front()
    for each edge (a,b) in G0 with not Seen[b]
      Q0.insert(b), Seen[b] = true;
      Distance[b] = length
    end for
    Q0.delete(),  Q1.insert(a)
  end while

  length = length + 1

  while (Q1 is not empty)
    a = Q1.front()
    for each edge (a,b) in G1 with not Seen[b]
      Q0.insert(b), Seen[b] = true;
      Distance[b] = length
    end for
    Q1.delete()
  end while

end while
```

## 5 The string to regular-language correction problem

In the general string to regular-language correction problem, we are given a string (word) $w$, an NFA $A$ and a wfse-system $\beta$ and we want to compute a minimum cost e-string $h$ in $L(A_\beta)$. More specifically, the language $L(A)$ is supposed to contain all the "syntactically correct words" and the e-string $h$ describes the edit operations permitted by $\beta$ that would transform $w$ to a syntactically correct word. The cost of $h$ is equal to $d_\beta(w, L(A))$. If we construct the $(|w|+1)$-state automaton $A_w$ accepting $\{w\}$ then we can use the e-NFA $A_w \cap_\beta A$ to solve the problem in time $O(|w||A||A_\beta|\log(|w||A||A_\beta|))$, or $O(|w||A||A_\beta|)$ if $A$ is acyclic.

TABLE 1

| Finite Machine | Word | Time for Algorithms: Old | New: initial, | improved |
|---|---|---|---|---|
| c20: accepts 113 unix commands ended with ; | dh_installxfonts; | 0m28.899s | 0m0.137s, | 0m0.009s |
| | The Dog Ran | 0m28.468s | 0m0.087s, | 0m0.019s |
| | dh_innnnnstallllllxfonts; | 0m29.597s | 0m0.215s, | 0m0.026s |
| c20plus: accepts any word consist-ing of one or more words in L(c20) | edusers;dig;evolution;scrollkeeper-install;rep;rep;python;python;python;ppmquantall;ppmquantall;plotfont;zsh4;bibtex;bibtex;bibtex;_java_classes; | 0m59.797s | 0m1.548s, | 0m0.021s |
| | edusersdigevolutionscrollkeeper-installrepreppythonpythonpythonppmquantallppmquantallplotfontzsh4bibtexbibtexbibtex_java_classes; | 0m56.909s | 0m1.319s, | 0m0.193s |
| | xxxxxxxxxxxxxxxxxxxx | 0m40.692s | 0m0.176s, | 0m0.040s |
| fm4: accepts the language (abc)*abc(abc)* | aduhqeoipaodijw | 0m0.005s | 0m0.005s, | 0m0.003s |
| | abcabcaabbcc | 0m0.005s | 0m0.005s, | 0m0.003s |
| dict: accepts 249083 dictionary words | qualificaton | | | 0m2.830s |
| | quamificaton | | | 0m2.950s |

The *Levenshtein e-system* is the free e-system $\tau$ that permits all possible edit operations, namely $E_\tau = E_\Sigma$, and the cost of each edit operation $x/y$ is $C_\tau(x/y) = 0$ if $x = y$, and $C_\tau(x/y) = 1$ if $x \neq y$. We consider $\tau$ to be fixed and, therefore, the above problem can be computed in time $O(|w||A| \log(|w||A|))$, or $O(|w||A|)$ if $A$ is acyclic. This problem has been solved also in [12] using a dynamic programming algorithm that operates in time $O(|w||A|^2)$ – see also [8] for cases where the language of valid words can be non-regular. We have implemented both algorithms in C++ on a Pentium III 1.4 GHz machine. The implementations are available in [9]. We used the class `fm` of Grail+ 2.5 [3] to represent automata (finite machines). This class stores the transitions of an automaton in an array and maintains a flag to indicate whether the array is sorted. To add a new transition, Grail first tests whether the transition already exists, either by performing binary search if the array is sorted, or by sorting the array and then performing binary search if the array is not sorted. With this approach, the cost of building a large automaton could be high and dominate the running time of the algorithms. For this reason in our tests, we have considered only cases where the transitions, which are contained in some text file, are already sorted; therefore, no sorting takes place when the transitions are read from the text file into an object of the class `fm`.

We note that, in our initial implementation of the new algorithm, we constructed explicitly the graph $A_w \cap_\tau A$ and then performed Dijkstra's algorithm on that graph. In the current, improved, implementation, given the graphs of $A_w$ and $A$, we first allocate space for the vertices $(p, q)$ of the graph $A_w \cap_\tau A$ and then we simply perform Dijkstra's algorithm by computing the edges of $A_w \cap_\tau A$ "on the fly" as needed. More specifically, if $(p_1, q_1)$ is a vertex of $A_w \cap_\tau A$ with minimum "path-length" then, to update the path-length of an unmarked vertex $(p_2, q_2)$ that is adjacent to $(p_1, q_1)$, we examine the transitions of $A_w^\lambda$ of the form $p_1 x p_2$ and the transitions of $A^\lambda$ of the form $q_1 y q_2$ and update the path-length of $(p_2, q_2)$ using the cost of the edit operation $x/y$ – see [6], for instance, for details on Dijkstra's algorithm. The actual running times of the algorithms on certain inputs are given in Table 1 – currently, we are working on the implementation of the algorithm for the general string to regular-language correction problem. We note that our implementation of the old algorithm [12] uses a lot of function calls to allocate and deallocate memory, as required by the dynamic programming approach, and this appears to slow the algorithm down considerably. It is not obvious, however, how to reduce the number of these function calls without introducing extra computation time.

## 6 Discussion

We have provided a method of representing error situations using basic tools from automaton theory. This allows us to address various algorithmic questions pertaining to the differences between words and languages. For the problem of computing the Hamming distance of a given regular language, we were able to give a fast algorithm by taking advantage of two facts specific to the Hamming e-system: (i) the weights involved are zero and one, and (ii) if an e-string of this system contains an error then the input and output parts of the string must be different. The second fact, however, is not true in the case of the Levenshtein e-system. More generally, the problem of computing, in polynomial time, the value of $d_\beta(L(A))$ for given NFA $A$ and wfse-system $\beta$ remains open.

For the general string to regular-language correction problem, our solution makes no assumptions about any specific properties of the objects involved. Of course, for certain types of regular languages and wfse-systems it might be possible to improve the algorithm or even follow a totally different approach. For example, the times for the automata `c20` and `dict` in Table 1 would be lower if one used the shortest path algorithm that is specific to acyclic graphs. We believe that our observations can be useful in addressing various algorithmic questions related to the topic of word and language comparisons.

## References


[1] C. Choffrut, G. Pighizzini, Distances between languages and reflexivity of relations. *Proceedings MFCS'97, Lecture Notes in Computer Science,* **1295** (1997), 199–208.

[2] K. W. Church, W. A. Gale, Probability scoring for spelling correction. *Statistics and Computing,* **1** (1991), 93–103.

[3] Grail+. Department of Computer Science, University of Western Ontario. URL address `www.csd.uwo.ca/research/grail/`

[4] T. Head, A. Weber, Deciding code related properties by means of finite transducers. *Proceedings of Sequences II, Methods in Communication, Security, and Computer Science, 1993,* Springer-Verlag, Berlin, 260–272.

[5] L. Kari, S. Konstantinidis, Descriptional complexity of error/edit systems. In: J. Dassow, M. Hoeberechts, H. Jürgensen, D. Wotschke (eds), *Pre-Proceedings of Descriptional Complexity of Formal Systems 2002,* London, Canada, 133–147.

[6] U. Manber, *Introduction to Algorithms, A Creative Approach.* Addison-Wesley Publishing Company, 1989.

[7] W. Peterson, E. Weldon Jr., *Error-Correcting Codes.* MIT Press 1972.

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

[9] Program for the general string to regular-language correction problem. Department of Mathematics and Computing Science, Saint Mary's University. URL address `http://cs.stmarys.ca/~s_perron/`

[10] G. Rozenberg, A. Salomaa (eds), *Handbook of Formal Languages, Vol. I.* Springer-Verlag, Berlin, 1997.

[11] D. Sankoff, J. Kruskal (eds), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* CSLI Publications, 1999.

[12] R. A. Wagner, Order-$n$ correction for regular languages. *Communications of the ACM* **17**(5) (1974), 265–268.

[13] S. Yu, Regular Languages. In [10], 41–110.