Incremental String Correction: Towards Correction of XML Documents

Ahmed Cheriat*, Agata Savary†, Béatrice Bouchou, and Mírian Halfeld Ferrari

Université François Rabelais de Tours - LI/Campus de Blois, France
3 place Jean Jaurès - 41000 Blois, France

ahmed.cheriat@etu.univ-tours.fr
{agata.savary, beatrice.bouchou, mirian}@univ-tours.fr

Abstract. We define a problem of an *incremental* string-to-string correction with respect to a regular grammar. A user is given a valid word which may be updated through one or more editing operations. If the resulting word is invalid we propose correction candidates that take not only the incorrect word but also the initial valid word into account. The method is based on the error distance matrix calculation as proposed by [9]. It has been developed in view of incremental XML document correction (as opposed to correction from scratch). Experimental results show a good performance of our algorithm despite its exponential theoretical complexity.

1 Introduction

We introduce an *incremental* string-to-string correction method with respect to a regular grammar. Given an initial correct (*valid*) word A (*i.e.* a word accepted by a regular grammar), a user can adapt this word to his needs by proposing one or more elementary operations (*updates*) on it under the condition that the resulting word B remains valid. If however B happens to be invalid (*e.g.* due to user's mistake when performing updates) the system should guess the user's intention and propose a set of plausible corrections. Thus, we are not willing to search for all nearest neighbors of B in the dictionary but only those that might result from A through a sequence of operations which are similar (but not identical) to the updates proposed by the user.

Our solution is to explore the finite-state automaton corresponding to the grammar in order to find valid words that are as close as possible to both A and B. Thus, we benefit from the achievements of the string-to-string correction domain ([11], [3]), as well as of their due to the finite-state representation of grammar or lexicon ([9]), while providing some new ideas focused on incrementality.

The motivation for the incremental string-to-string correction comes from the area of XML-document validation and correction. The validity of each node in such a document is described by one or more regular expressions. When a user wishes to

*Supported by Région Centre, France
†Partly supported by the IUT of Blois, France

modify a valid document but performs an invalid update on a node we may start with *locally* correcting this node's closest neighborhood using the incremental string-to-string approach. Thus, some good parts of the proposed correction tree may remain unchanged with respect to the initially valid XML-tree, which spares computation time and space.

As we place ourselves in a database context, updates are not treated one by one but grouped into sequences, or *transactions*. Thus, we are interested in the validity of the resulting word only at the end of each transaction. If the word turns out to be invalid we try to correct it with respect to the whole sequence of updates appearing in the transaction.

The paper is organized as follows. In Section 2, we resume some related work in the string-to-string correction domain. Then, in Section 3, we consider some particularities of our approach. Our incremental string correction method is described in Section 4. In Section 5 we discuss the complexity of our algorithm together with some experimental results. Finally, Section 6 concludes the paper, and gives some ideas of our future work.

2 String-to-String Correction with Respect to a Regular Grammar: State of the Art

The definition of the string-to-string correction problem aims at the formalization of the intuitive notion of similarity between two strings. As Wagner and Fischer ([11]) put it, the *edit distance* between two strings A and B is the minimum cost of all sequences of elementary edit operations (insertions, omissions and replacements) on letters which transform A into B. These operations may be written as rewriting rules of the form $a \to b$ where a and b are single letters and/or empty strings (ϵ) and $(a, b) \neq (\epsilon, \epsilon)$. Each edit operation $a \to b$ is assigned any non negative cost $\gamma(a \to b)$. We say that $a \to b$ takes A to B if $A = \sigma a \tau$ and $B = \sigma b \tau$.

An edit sequence S is a sequence $s_1, s_2, \ldots, s_m$, where s_i is an edit operation for each $0 \leq i \leq m$. Each edit operation s_{i+1} applies to the string resulting from the application of the preceding edit operation s_i. We say that S takes word A to word B if a sequence of strings $A_0, A_1, \ldots, A_m$ exists such that $A = A_0$, $B = A_m$ and s_i takes A_{i-1} to A_i for each $0 \leq i \leq m$. The cost $\gamma(S)$ of an edit sequence S is the sum of costs of all edit operations appearing in S.

Note that, with Wagner and Fischer, an edit sequence contains no reference to the word positions at which the edit operations operate. Due to this fact, the result of an edit sequence may be ambiguous. Moreover, a further edit operation may operate on a letter resulting from a former operation. For example, the application of the edit sequence $(a \to b, b \to c)$ to the word abb may result in any of the following words: cbb, bcb, bbc.

Furthermore, Wagner and Fischer ([11]) propose a useful model of a *trace* which is a visualization of a class of edit operation sequences as in the example on Figure 1. A line leading from position i of the source string A to position j of the target string B indicates that $A[i]$ should be replaced by $B[j]$ (if $A[i] \neq B[j]$) or that $A[i]$ should remain unchanged in B (if $A[i] = B[j]$). Characters of A untouched by any line are to be deleted and characters of B untouched by any line are to be inserted.

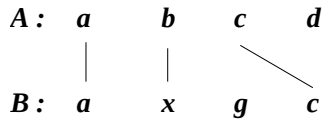

Figure 1: A trace between *abcd* and *axgc*

		ε	a	b	a	a	b	a
		0	1	2	3	4	5	6
ε	0	0	1	2	3	4	5	6
b	1	1	1	1	2	3	4	5
a	2	2	1	2	1	2	3	4
b	3	3	2	1	2	2	2	3

Figure 2: Edit distance matrix between *bab* and *abaaba*

Each trace T receives a non negative cost defined as follows:

$$\text{cost}(T) = \sum_{(i,j)\in T} (\gamma(A[i] \to B[j])) + \sum_{i\in I} (\gamma(A[i] \to \epsilon)) + \sum_{j\in J} (\gamma(\epsilon \to B[j]))$$

where I and J are the sets of positions in A and B, respectively, untouched by any line in T. For instance, if we assume that $\gamma(a \to b) = 1$ for $a \neq b$ then the trace on Figure 1 has cost 3.

It is further shown in [11] that a correspondence exists between edit sequences and traces:

- for every trace T from A to B, there is an edit sequence S taking A to B such that $\gamma(S) = \text{cost}(T)$
- for every edit sequence S taking A to B, there is a trace T from A to B such that $\text{cost}(T) \leq \gamma(S)$

Thus, looking for the minimum cost edit sequence taking A to B is equivalent to looking for the minimum cost trace from A to B. This minimum cost in both cases determines the edit distance between A and B. It can be obtained by a *dynamic programming* method which calculates an edit distance matrix H. For $0 \leq i \leq |A|$ and $0 \leq j \leq |B|$, element $H[i, j]$ contains the edit distance between the prefixes $A[1 : i]$ and $B[1 : j]$ of A and B (where $X[i : j]$ represents the subword $X_i, X_{i+1}, \ldots, X_{j-1}, X_j$). The matrix may be calculated column per column. Thus, each new element is deduced from its three top-left-hand neighbor elements which have been calculated previously. The bottom right-hand element of the whole matrix contains the edit distance between the two strings A and B. It is obtained with a time complexity of $O(|A| * |B|)$.

For example, the distance matrix on Figure 2 obtained by the above algorithm, with $\gamma(a \to b) = 1$ for $a \neq b$, indicates that the edit distance between strings *bab* and *abaaba* is 3.

Note that there may be more than one minimum cost trace (and thus more than one minimum cost edit sequence) between two words. In the above example, two such traces exist as shown on Figure 3.

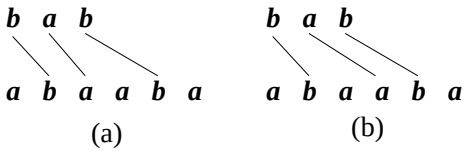

Figure 3: Two possible minimum cost traces between *bab* and *abaaba*

Lowrance and Wagner ([7]) extended the definition of the string-to-string correction problem to the case of four elementary editing operations on letters: the previous three operations were completed by a transposition of two adjacent letters. Thus, traces can contain crossing lines. However, the cost function was restricted to the case when all insertions, all deletions, all replacements, and all transpositions have the same costs W_I, W_D, W_C, W_S, respectively. An efficient solution ($O(|A| * |B|)$) for the edit distance calculation was proposed in case when $W_I + W_D \leq 2W_S$.

The addition of the transposition as the fourth elementary operation makes the mathematical model of the problem more complex. An elementary operation may still be represented as a rewriting rule of the type $\gamma(a \rightarrow b)$. However, the a and b symbols have now to be seen as sequences of letters rather than single letters. In the class of all possible sequences the choice of allowing only rules of type $xy \rightarrow yx$ seems very application-oriented. Note that with [7] the editing operations may still act on arbitrary positions in the source string, and in an arbitrary order (*e.g.* *ca* can be obtained from *abc* by two operations: deletion of b and transposition of a and c).

Du and Chang ([3]) modified this distance measure and renamed it to *error distance* by assigning cost 1 to each editing operation and by admitting that errors occur in linear order from left to right so that a later operation may not cancel the effect of an earlier operation. For example, two changes may not operate on the same word position while inversions occur only between letters that are adjacent in the original word and remain adjacent in the erroneous word (*e.g.* the error distance between *abc* and *ca* is 3). The linear order of editing operations in an edit sequence implies that each operation is assigned an integer corresponding to the current word position it operates on. For example, the edit sequence $(D(1), C(1, c), T(2))$ (*i.e.* deletion of letter at position 1, change of letter at position 1 to c, and transposition of letters at positions 2 and 3) applied to *abba* results in *cab*. Due to the equal cost of each editing operation, the error distance becomes a *metric*, *i.e.* a function satisfying four properties: non-negative values, reflexivity, symmetry, and triangular inequality.

The model simplification proposed by Du and Chang allows a substantial gain of efficiency to the algorithm of the error distance calculation. While in [7] the calculation of element $H[i, j]$ of the matrix needs, in the worst case, an access to each element of the previously calculated part of the matrix (to the left and above $H[i, j]$), with the linear error distance of Du and Chang this is no longer the case: $H[i, j]$ is calculated on the basis of its four neighbors only ($H[i-1, j]$, $H[i, j-1]$, $H[i-1, j-1]$, and $H[i-2, j-2]$). The matrix calculation has been further simplified due to some of its discovered properties.

Also in [3] the string-to-string correction is applied to the problem of finding, for a word, all its nearest neighbors in a dictionary. A distance threshold is one of the parameters of this problem. A nearest neighbor of X must stay within the error distance from X which is no bigger than the threshold. The dictionary is represented

in no particular form. A distance matrix has to be constructed from scratch for each new dictionary word with respect to the erroneous word. A cut-off criterion has been discovered which allows to stop the calculation of the matrix in its early stage as soon as it turns out that the error distance between two current strings exceeds the threshold. However, this calculation remains costly as it is roughly proportional to the number of words in the dictionary.

In the early applications of the approximate string matching ([4]), such as the automatized correction of computer programs, the vocabulary size was small (number of all key words and variable names in a program). Solutions as the one by [3] could then be applied with no problem of robustness.

As soon as the same string-to-string correction algorithms were to be used for spelling correction of natural language texts the vocabulary size often proved to be a bottleneck ([6]) which required additional dictionary reduction techniques. However, an extensive development of finite-state methods for natural language processing enabled a very time and space-efficient representation of large vocabularies. Furthermore, the dynamic programming method could be applied in the process of a finite-state dictionary access, thus providing a fast algorithm of searching for the nearest neighbors of a string in a dictionary. This technique was announced already by [10] for the 3-operation edit distance, but [9] was probably the first to extend it to the 4-operation error distance and test it extensively on large natural language vocabularies. In his algorithm, when a word is searched for in a finite-state lexicon, a part of the error matrix is calculated only once for all lexicon words that have the same common prefix. This optimization, in addition to the cut-off criterion of [3], provides an algorithm that rapidly finds, for a given word, all its t-distant neighbors in the dictionary.

More recent approaches to approximate string search in a finite-state dictionary, such as [8] which uses so-called Levenshtein automata and a "backward" dictionary, allow a further increase in speed of the string-to-string correction.

A string of symbols may be viewed as a trivial case of a tree whose depth is 1 and whose leaves are the elements of the string. Thus, the formalization of the string-to-string correction problem naturally inspired research on the tree-to-tree correction problem ([2]). Note that the diversity of the possible choices of elementary editing operations is even bigger in case of a tree as one can consider changes not only on the siblings' level but also on some ancestors' level. The most appropriate choice depends on the intuitive notion of tree proximity for the particular application. In our application, trees are XML documents which must be validated and corrected against their DTDs or XML schemata. However, compared to other tree correction approaches, our approach is to propose an *incremental* correction method as described in the following section.

3 Incremental String-to-String Correction with Respect to a Regular Grammar

The distance measure between two strings admitted in our approach is a simplified version of the edit distance by Wagner and Fischer ([11]) and of the error distance by Du and Chang ([3]). On the one hand, we allow only three elementary operations:

an insertion, a deletion, and a replacement of a single letter. On the other hand, we admit cost 1 for each of these operations.

The originality of our approach is due to three facts. Firstly, the definition of an edit operation (which we also call an *update*) and of an edit sequence (a *sequence of updates*) is particular. We attribute to each operation a word position it applies to as is the case with Du and Chang ([3]). However, all of these positions, numbered from 0 to the length of the word minus 1, concern the same initial word. For instance, the update sequence $(insert(a,0), replace(c,1), insert(d,3))$ takes the initial word abb to $aacbd$[1]. This definition of the word position is inspired by research on incremental XML validation by [1]. Note that this approach allows no later operation in a sequence to cancel the effect of an earlier operation, as is the case with [3].

Secondly, we place ourselves in a database context in which updates are not treated one by one but grouped into sequences, or *transactions*. That is because, given a sequence of n updates, a word may become incorrect after $i < n$ updates, but its validity may be re-established after all the n updates. For example, given the simple regular grammar $abcd + bced$, the initial valid word $abcd$, and the edit sequence $(delete(a,0), insert(e,3))$, the resulting word is valid ($bced$) and does not need any correction. If however we try to process the updates one by one we'll have to propose corrections for the intermediate invalid word bcd, which is useless for the user.

Thirdly, we wish to perform an *incremental* string-to-string correction in the context of a human-computer interaction. A user is given an initial correct word A (*i.e.* a word valid with respect to a regular grammar). He/she may adapt this word to his needs (or, in other words, construct a new word *incrementally*, or *evolutionarily*) by proposing one or more updates on this word under the condition that the resulting word B remains valid. If however B happens to be invalid (*e.g.* due to the user's ignorance with respect to the validity of words) the system should guess the user's intention and propose a set of plausible corrections. Thus, we are not willing to search for all nearest neighbors of B in the language described by the grammar but only those that might result from A through a sequence of operations which are similar (but not identical) to the updates proposed by the user. This approach, as opposed to a validation *from scratch* (where A is not taken into account), allows to possibly limit the computation time and space, as well as the number of correction candidates proposed to the user.

In our approach, the correction of words is done with respect to a regular grammar represented by a finite-state automaton. Thus, we can fully benefit from the optimizations offered by Oflazer's application ([9]) of Du and Chang's approach ([3]). Note that there is no need in [9] for the dictionary to be a finite set of words. It may as well be represented by a regular expression recognizing an infinite set of words.

Consider the following example :

- the dictionary is described by the regular expression $ab^*c + db^*$
- the initial valid word is $A = abc$
- the sequence of updates proposed by the user is $U = (insert(b,3), insert(b,3))$, *i.e.* two insertions of b at the end of the string
- the invalid word resulting from A by the application of U is $B = abcbb$

In the above case the nearest neighbors of B (of distance 2) are : $C_1 = abc$, $C_2 = abbc$, $C_3 = abbbc$ and $C_4 = dbbb$. However, C_2 and C_3 are more plausible correction

[1] Insertions are done *before* the letter on the corresponding position as is the case with [3].

candidates for B than C_1 and C_4 as they seem to better correspond to the user's intention. Proposing C_1 which is equal to the initial word A would ignore the user's wish of modification, while $C_4 = dbbb$ has very little in common with the initial word A that the user is supposed to adapt. Of course, even if C_1 and C_4 are judged less plausible they are never completely discarded as in some cases they may still best suit the user.

The motivation for the incremental string-to-string correction comes from the area of XML-document validation and correction. The validity of each node in such a document is described by a regular expression (in case of a DTD) or by a set of regular expressions (in case of an XML schema). For instance, with [1] the validation is done via a tree automaton whose translation rules are of the form $a, E \to q_a$ where E is a regular expression. Each transition rule indicates that a node having label a and whose children respect the schema rules established by E can be assigned to state q_a. Thus, given a node p labeled with a in the XML tree, a bottom-up automaton performs the validation by verifying whether the word composed by the concatenation of the states (previously) assigned to the children of p belongs to the language $L(E)$.

When a user wishes to modify a valid document but performs a set of invalid updates (*i.e.* leading to an invalid tree) we may start with *locally* validating and correcting the nodes concerned by the updates, together with their closest neighborhood: fathers, siblings, and sons. Since each set of siblings may locally be viewed as a string, we reduce a part of the tree correction to the string-to-string correction problem. Thus, we may often obtain our first valid correction candidates without even touching good parts of the whole tree (those that remain unchanged with respect to the initially valid XML tree) which allows to spare computation time and space and further motivates the notion of incrementality. Our intuition is that such a *shallow correction* approach will often offer the most plausible correction candidates because they vary from the initially valid tree only around the points which the user him/herself wished to modify. At the same time this approach does not exclude a *deep correction* ranging not only over the closest neighbors of the updated nodes but possibly over the whole tree.

The following section describes the computational solution of such incremental string-to-string correction which may be applied locally to an XML-tree on a single-node level.

4 Solution and Algorithms

Let us consider an initially correct word A, *i.e.* A appearing in the language $L(E)$ described by a regular expression E. A user can update A by inserting, deleting or replacing one or more symbols. If the resulting word B happens to be invalid, *i.e.* $B \notin L(E)$, we should propose a set of *valid candidate words*.

We have previously mentioned that in the context of incremental correction the proposed candidate should express the user's intentions as to the modifications of A: it should be obtainable from A by updates similar to those the user him/herself has performed. However, we find it non trivial to define an efficient similarity measure between sequences of updates, which consist of incomparable parameters - operation types, letters, and word positions - and which are non homogeneous (deletions carry no information about letters). Moreover, sequences of updates may show some degree

of redundancy (*e.g.* an operation is performed by one update and later canceled by another update). Therefore, it is not obvious if the user's intentions are best expressed by the updates he wished to perform or by the resulting (invalid) word he/she has produced.

Therefore, we propose an algorithm expressing the similarity of sequences of updates via the similarity of words resulting from these updates. Thus, a valid candidate word is the one that is as close as possible to both A and B, *i.e.* its distance from both A and B doesn't exceed a given threshold. We may calculate the set of such valid candidates applying the Oflazer's ([9]) dynamic programming method to two distance matrices in parallel: the one for the distance between A and C, and the other between B and C. When a particular correction candidate C has been chosen by the user we should instruct him/her on the right updates he/she should have done in order to generate C from A. This right sequence of updates may easily be deduced from the trace between A and C which on its turn may be generated on the basis of the A-C distance matrix.

4.1 Notations

Let E be a regular expression and let $M_E = \langle \Sigma,\ Q,\ \delta,\ q_0,\ F \rangle$ be a deterministic or a non deterministic finite state automaton over an alphabet Σ, a finite set of states Q, an initial state $q_0 \in Q$, a set of accepting states $F \subseteq Q$, and a transition relation $\delta \subseteq Q \times \Sigma \times Q$. Let W be a finite string (or word) of characters (or symbols): $W \in \Sigma^*$. W is valid (correct), iff $W \in L(E)$, where $L(E)$ is a language defined by E. In the following, we introduce some definitions of data types that will be used ahead in this work.

Definition 1. ***Type*** Tr_T ***(a trace)*** *is a list of pairs of integers* $(i,\ j)$ *such that for each* $Tr \in Tr\_T$ *if* $Tr = ((i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n))$ *then*

1. $i_p \neq i_r$ *and* $j_s \neq j_t$ *for* $1 \leq p, r, s, t \leq n$
2. $i_p < i_r$ *iff* $j_p < j_r$ *for* $1 \leq p, r \leq n$

The above definition reformulates the context-independent conditions of the trace definitions by Wagner and Fischer [11] (no character is touched by more than one line, and no two lines cross). In a particular context of two words A and C, a trace $Tr_{A,C} \in Tr\_T$ will always be such that $(Tr_{A,C},\ A,\ C)$ is a minimal cost trace in the sense of [11]. Thus, an extra context-dependent condition, ensuring that lines actually touch character positions of A and B, completes the above definition:

$$Tr_{A,C} = ((i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n)) \text{ where}$$
$$\forall_{1 \leq p \leq n}\ 0 \leq i_p \leq |A| - 1 \text{ and } 0 \leq j_p \leq |B| - 1$$

Recall that there may be several minimal cost traces between A and C.

Definition 2. ***Type*** STr_T ***(a set of traces)*** *describes a set of traces of type* Tr_T *each.*

A particular set of traces $STr_{A,C} \in STr\_T$ will be used in connection with a single pair of words A and C:

$$STr_{A,C} = \{Tr^1_{A,C},\ Tr^2_{A,C},\ \ldots,\ Tr^m_{A,C}\} \text{ where}$$
$$\forall_{i=1,\ldots,m} Tr^i_{A,C} \in Tr\_T \text{and } Tr^i_{A,C} \text{ is a minimal cost trace between } A \text{ and } C.$$

Definition 3. ***Type*** *$SCandid\_T$* ***(a set of candidates):*** *describes a list of elements of the form $(C, (ed_1, ed_2))$, where C is a word, and ed_1, ed_2 are two integers.*

A particular $SCandid_{A,B} \in SCandid\_T$ will be an *ordered* list used in connection with a single pair of words A (valid) and B (invalid), a regular expression E, and a threshold th. $SCandid_{A,B}$ will then describe a set of incremental correction candidates for B with respect to A (see the preceding section).

$SCandid_{A,B} =$
$((C_1, (ed_{A,C_1},\ ed_{B,C_1})), (C_2, (ed_{A,C_2},\ ed_{B,C_2})),\ \ldots, (C_k, (ed_{A,C_k},\ ed_{B,C_k})))$,
where for each $1 \leq i \leq k$, $C_i \in L(E)$, and ed_{A,C_i}, ed_{B,C_i} are the edit distances between A and C_i and between B and C_i respectively.

The ordering of the list is based on the edit distances of the candidates with respect to both A and B. The best correction candidates (found in the front of the $SCandid_{A,B}$ list) are those that are close to both A and B. However, a good candidate may not be equal to A (otherwise the user's intention to modify A would be neglected). For two candidates, if the sums of their distances from A and B are equal then we privilege the candidate that is closer to B (as it's B that best expresses the update intentions of the user). These rules of the ordering of candidates may be formally expressed as follows:

$$
\begin{array}{l}
(C_i, (ed_{A,C_i},\ ed_{B,C_i})) \prec (C_j, (ed_{A,C_j},\ ed_{B,C_j}))\ \textit{iff} \\
(ed_{A,C_j} = 0)\ \textit{or} \\
(ed_{A,C_i} + ed_{B,C_i} < ed_{A,C_j} + ed_{B,C_j})\ \textit{or} \\
(ed_{A,C_i} + ed_{B,C_i} = ed_{A,C_j} + ed_{B,C_j})\ \textit{and}\ (ed_{B,C_i} < ed_{B,C_j})
\end{array}
$$

Definition 4. ***Type*** *H_T* ***(edit distance matrix)*** *is a two dimensional matrix with indices starting from -2. A particular matrix $H_{A,B} \in H\_T$ will always be used in connection with two words A and B so that $H_{A,B}$ is defined as follows:*

$$
\begin{array}{lll}
H_{A,B}[i,j] = & H_{A,B}[i-1, j-1], & \textit{if } A[i] = B[j], \\
= 1 + & min\{H_{A,B}[i-1, j-1], & \\
& \quad H_{A,B}[i-1, j], & \\
& \quad H_{A,B}[i, j-1]\} & \textit{otherwise} \\
H_{A,B}[i,-1] = & i+1, & \textit{for } 0 \leq i \leq |A|-1 \\
H_{A,B}[-1,j] = & j+1, & \textit{for } 0 \leq j \leq |B|-1 \\
H_{A,B}[-2,j] = & H_{A,B}[i,-2] = +\infty, & \textit{(boundary definition)}
\end{array}
$$

Note that the above formula is very similar to those used by [9] and [11] for the edit distance calculation. However, there are some minor differences: we do not allow transpositions (contrary to [9]), the cost of each elementary operation is 1 (contrary to [11]), and the numbering of the edit distance matrix indices starts with -2, since the first symbol of a word is indexed by 0.

4.2 Algorithms

Our first algorithm computes all valid candidate words. It contains a recursive procedure, called Explore_rec, that generates new valid words starting with the prefix C,

and whose distance from the given words A and B does not exceed the threshold th. The automaton's state q is the one that has been reached while generating the correction prefix C. New candidates are attached to the list of those found previously ($SCandid$). In its first call, procedure Explore_rec receives, in particular, the initial state q_0, an empty set of candidates $SCandid$, and matrices $H_{A,C}$ and $H_{B,C}$ with their two first columns initialized according to Definition 4 with $C = \epsilon$.

```
1:  procedure Explore_rec (A, B, C, th, H_{A,C},H_{B,C}, M_E, q, SCandid)
2:  input
3:    A: word (a valid word)
4:    B: word (an invalid word resulting from updates of A)
5:    th: integer (error threshold)
6:    M_E: FSA (M_E = ⟨Q, Σ, δ, q_0, F⟩)
7:    q: state (q ∈ Q of M_E, the current state in the automaton)
8:  input/output
9:    C: word (a partial valid candidate word)
10:   H_{A,C}: H_T (edit distance matrix between A and C)
11:   H_{B,C}: H_T (edit distance matrix between B and C)
12:   SCandid: SCandid_T (set of valid candidate words)
13: begin
14:   if (q ∈ F and (H_{A,C}[|A| − 1, |C| − 1] ≤ th) and (H_{B,C}[|B| − 1, |C| − 1] ≤ th))
15:   /** A candidate is found. Candidates are sorted according to Def. 3 **/
16:     SCandid ← SortInsertion(SCandid, (C, (H_{A,C}[|A| − 1, |C| − 1],
17:                                          H_{B,C}[|B| − 1, |C| − 1])))
18:   end if
19:   for each (a, q') ∈ Σ × Q such that δ(q, a) = q'
20:     C ← concat(C, a)
21:     H_{A,C} ← AddNewColumn(H_{A,C}, A, a)
22:     H_{B,C} ← AddNewColumn(H_{B,C}, B, a)
23:     if ((cuted(A, C, H_{A,C}, th) ≤ th) and (cuted(B, C, H_{B,C}, th) ≤ th))
24:       Explore_rec(A, B, C, th, H_{A,C}, H_{B,C}, M_E, q', SCandid)
25:     end if
26:     H_{A,C} ← DeleteLastColumn(H_{A,C})
27:     H_{B,C} ← DeleteLastColumn(H_{B,C})
28:     C ← DelLastSymbol(C)
29:   end for each
30: end
```

The automaton M_E is explored in the depth-first order. Each time a transition is followed the current prefix C is extended (line 20) and new columns are added to both distance matrices (lines 21–22). That allows to check if C may still lead to a candidate remaining within the distance threshold from A and B (line 23). If it does the path is followed via a recursive call (line 24), otherwise the path gets cut off. In each case the transition is finally backed off (lines 26-28) and a new transition outgoing from the same state is tried out. If we arrive at a final state and the distance from C to both A and B does not exceed the threshold (line 14) then C is a valid candidate that gets inserted to the list of all candidates found so far (lines 16–17). The insertion is done according to Definition 3.

Note that the validation of the extended C with respect to the threshold (line 23) is done via the function *cuted* that computes the cut-off edit distance between A and

C, and between B and C, as defined by [9]. It corresponds to the minimum value of the current column in the edit distance matrix (*i.e.* the column corresponding to the last character in the extended C). It has been shown by [9] that if this value exceeds the threshold then there is no chance for further columns not to exceed the threshold. Thus, C may not be a prefix of a valid word whose distance from A and B is lower than the threshold.

Let's consider, for instance, a grammar $E = (aba+bab)^*$ and a valid word $A = bab$. If we apply the sequence of updates $S = (insert(a, 1), replace(a, 2))$ to A we obtain an invalid word $B = baaa$. For $th = 2$ the above function returns the following list of candidates: $SCandid = ((aba, (2, 2)),\ (bab, (0, 2)))$.

Given an ordered list of correction candidates the user may choose the one that best fits his/her needs. However, we also wish to show the user how to obtain C from A in order to let him/her avoid the same errors in future. The sequence of updates needed to take A to C can easily be deduced from a minimum cost trace between these two words. In the following we present a recursive function Trace_rec, that allows the construction of all minimal cost traces transforming A into C.

```
1:  function Traces_rec (A, C, H_{A,C}, i, j, Tr)
2:  input
3:    A: word (a valid word before updates)
4:    C: word (a valid candidate word)
5:    H_{A,C}: matrix (edit distance matrix between A and C)
6:    i, j: integers (indices of the current element of H_{A,C})
7:    Tr: Tr_T (a partial trace between A and C)
8:  result: STr_T (a set of traces between A and C)
9:  local variable
10:   STr: STr_T (a set of partial traces between A and C)
11: begin
12:   STr ← ∅      /* initialization */
13:   if ((i ≠ −1) or (j ≠ −1))
14:     if (H_{A,C}[i, j] = H_{A,C}[i − 1, j] + 1)      /* deletion */
15:       STr = STr ∪ Traces_rec(A,C,H_{A,C},i − 1,j,Tr) end if
16:     if (H_{A,C}[i, j] = H_{A,C}[i, j − 1] + 1)      /* insertion */
17:       STr = STr ∪ Traces_rec(A,C,H_{A,C},i,j − 1,Tr) end if
18:     if ((H_{A,C}[i, j] = H_{A,C}[i − 1, j − 1] + 1) and (A[i] ≠ C[j])) /*replacement*/
19:       STr = STr ∪ Traces_rec(A,C,H_{A,C},i − 1,j − 1,HeadInsert(Tr, (i, j))
20:       end if
21:     if ((H_{A,C}[i, j] = H_{A,C}[i − 1, j − 1]) and (A[i] = C[j])) /*no operation*/
22:       STr = STr ∪ Traces_rec(A, C, H_{A,C}, i − 1, j − 1, HeadInsert(Tr, (i, j))
23:       end if
24:   else
25:     STr = STr ∪ {Tr}
26:   end if
27:   return(STr)
28: end
```

The function runs over the error distance matrix from its bottom right-hand corner to its top left-hand corner. For the current matrix' element (i, j) the last parameter Tr holds all partial traces allowing to transform $A[i : |A| - 1]$ to $C[j : |A| - 1]$. In its first call the function receives an empty set of partial traces Tr, as well as $i = |A| - 1$

and $j = |C| - 1$, the indices of the bottom right-hand element of the matrix, *i.e.* the one that contains the edit distance between A and C.

In order to find a minimum cost trace between A and B it is sufficient to recall how the relevant elements of the error distance matrix $H_{A,C}$ have been calculated. The relevant elements are those that directly contribute to the computation of the final bottom left-hand element of $H_{A,C}$. Recall that each element $H_{A,C}[i,j]$ has been deduced in the procedure Explore_rec by the *AddNewColumn* function from one of its three top left-hand neighboring elements:

1. If $H_{A,C}[i,j]$ is equal to $H_{A,C}[i-1,j]+1$ it means that $C[0:j]$ can be obtained from $A[0:i]$ by the same edit operations as those needed for transforming $A[0:i-1]$ to $C[0:j]$, and by an additional deletion of $A[i]$ at position i. Thus, the trace between $A[0:i]$ and $C[0:j]$ is the same as the trace between $A[0:i-1]$ and $C[0:j]$ (line 15) since the letters to be deleted don't appear in the trace.
2. If $H_{A,C}[i,j]$ is equal to $H_{A,C}[i,j-1]+1$ it means that $C[0:j]$ can be obtained from $A[0:i]$ by the same edit operations as those needed for transforming $A[0:i]$ to $C[0:j-1]$, and by an additional insertion of $C[j]$ at position $i+1$ (line 17) as insertions occur before the given position. The trace between $A[0:i]$ and $C[0:j]$ is the same as between $A[0:i]$ and $C[0:j-1]$ since the letters to be inserted don't appear in the trace.
3. If $H_{A,C}[i,j]$ is equal to $H_{A,C}[i-1,j-1]+1$ and $A[i]$ is different from $C[j]$ it means that $C[0:j]$ can be obtained from $A[0:i]$ by the same edit operations as those needed for transforming $A[0:i-1]$ to $C[0:j-1]$, and by an additional replacement of $A[i]$ by $C[j]$ at position i. Thus, the trace between $A[0:i]$ and $C[0:j]$ is the same as the trace between $A[0:i-1]$ and $C[0:j-1]$ to which a replacement line of $A[i]$ by $C[j]$ has been added (line 19).
4. If $H_{A,C}[i,j]$ is equal to $H_{A,C}[i-1,j-1]$ and $A[i]$ is equal to $C[j]$ it means that $C[0:j]$ can be obtained from $A[0:i]$ by the same edit operations as those needed for transforming $A[0:i-1]$ to $C[0:j-1]$. Thus, the trace between $A[0:i]$ and $C[0:j]$ is the same as the trace between $A[0:i-1]$ and $C[0:j-1]$ to which an identity line between $A[i]$ and $C[j]$ has been added (line 21).

Let's consider the same example as on page 211. For candidate *aba* the above function returns the following set of minimum cost traces: $\{((0,1),(1,2)),\ ((1,0),(2,1))\}$.

5 Complexity and Experimental Results

Let $n = min(|A|,|B|)$ where B is the invalid word to be corrected, resulting from a valid word A. Let f_{max} be the maximum fan-out of our automaton M_E. Procedure Explore_rec has to perform, at worst, a depth-first exploration of M_E in which the depth of each path comes up to $n+th$ (because a word staying within the threshold th from both A and B may not be longer than $n+th$). Thus, the worst-case complexity of this procedure is $O(f_{max}^{n+th})$.

Function Traces_rec is called after procedure Explore_rec has determined the list of all candidates. At that moment the $H_{A,C}$ matrix for a candidate C chosen by the user does not exist any more and has to be recalculated which takes a time proportional to $|A| * |C|$. Function Traces_rec has to cross the error distance matrix from the bottom

Regular expression	Threshold	Number of updates	Number of candidates	Execution time(ms)
$E = (a\|b)c(d\|e)$	0	0	1	1
	1	2	1	1
	2	3	2	10
	3	4	1	1
	5	3	4	10
$E' = (a\|b)^*c(d\|e?)$	0	0	1	1
	1	2	3	1
	2	3	17	10
	3	4	10	1
	5	3	117	40

Table 1: Number of candidates and execution time obtained for the initial word *acd* when dealing with starred and non-starred regular expressions.

right-hand to the top left-hand corner in order to find all traces corresponding to the given candidate. In each position the path may only continue west, north or north-west. Since the matrix's size is no bigger than $n \times (n+th)$, the number of all possible recursive calls is less than $\sum_{i=1}^{n+th} 3^i = 3/2 * (3^{n+th-1} - 1)$. So the complexity of the trace calculation is $O(n^2 + 3^{n+th}) = O(3^{n+th})$.

Hence, the worst-case complexity of finding all candidates, and all traces for one chosen candidate is $O(f_{max}^{n+th})+ O(3^{n+th}) = O(c^{n+th})$ where $c = max(f_{max}, 3)$.

Although the complexity of our method seems to be discouraging, the worst cases rarely happen in practice. Our experimental results show that our algorithm is fast and gives good results in most cases. Our implementation was done in Java (JRE 1.4.1) running under Windows 2000. We use a 800 MHz Celeron Pentium system with 392 Mbytes of memory and a 40 GB hard disk with 5400 rpm.

We have performed 160 experiments by varying the regular expression, the threshold, the size of the initial word, and the number of updates. The statistical measures, chosen among those that are not disproportionately affected by extreme scores ([5]), give the following results: the *median* (the value separating the highest half from the lowest half of the results) is equal to 10 ms, the *mode* (the most frequent result) is 1 ms, and mean execution time of the 90% fastest runs is 44 ms.

We further examined the importance of different parameters on the number of candidates proposed by the program, and on its execution time. We noticed that the existence of starred sub-expressions, possibly embedded (*e.g.* $((ab)^*c)^*$) or ranging over a disjunction (*e.g.* $(a|b)^*c^*$), has a crucial importance for these two results.

Table 1 presents two test sets corresponding to regular expressions with and without Kleene-operators. In each test set, we varied two parameters: the error threshold and the number of updates. Columns 4 and 5 give the number of candidates generated by our method, together with the time needed for this computation.

We notice that for the same word a starred expression allows more correction candidates and their computation time may be several times higher than in the case of a non-starred expression. The reason is that the algorithm tries to compose different words containing repetitive characters within the range of the starred part of the

Label	Candidate	edit_distance(A, C_i)	edit_distance(B, C_i)
C_1	aaaaaacd	2	1
C_2	aaaaacd	1	2
C_3	aaaaabcd	2	2
C_4	aaaabacd	2	2
C_5	aaabaacd	2	2
C_6	aabaaacd	2	2
C_7	abaaaacd	2	2
C_8	baaaaacd	2	2

Table 2: Candidates for $E' = (a|b)^*c(d|e?)$, $A = aaaacd$ (valid), $B = aaaaacacd$ (invalid), and $th = 2$.

expression. For instance, given the regular expression $E' = (a|b)^*c(d|e?)$, the initial valid word $A = aaaacd$, the resulting invalid word $B = aaaaacacd$, and threshold 2, all correction candidates are obtained by modifying the subsequence recognizable by the subexpression $(a|b)^*$ while the suffix, recognizable by $c(d|e?)$, remains intact (see Table 2).

Our intuition is that word subsequences corresponding to starred sub-expressions, such as $(a|b)^*$, could be treated as blocks, so that their modification is not proposed if none of the user's updates falls within the range of the starred sub-expression. This heuristic might allow some optimizations of our method.

6 Conclusions and Future Work

We have introduced the problem of an incremental string-to-string correction: given a regular grammar E, a valid word A and a sequence S of updates (insertions, deletions, and replacements of letters) that transform A into an invalid word B, find all valid words C that may result from A by sequences of updates that are as similar as possible to S.

It seems non trivial to define an efficient similarity measure between sequences of updates. Therefore, we proposed an algorithm that addresses the above problem by expressing the similarity of sequences of updates via the similarity of words resulting from these updates. Thus, an incremental string correction may be implemented by the nearest-neighbor search in a finite-state automaton performed simultaneously for both A and B within a given threshold, according to algorithms proposed by [11], [3] and [9]. The reconstruction of a *trace* between the initial valid word and a correction candidate chosen by the user allows him/her to know the right update sequence needed to obtain this candidate.

Despite an exponential worst-case complexity (frequent in approaches based on an extensive finite-state automaton exploration), our algorithm gives good experimental results calculated over a large sample of tests with varying parameters. We think that some optimizations, concerning both the candidate's pertinence and the execution time, may be done if the internal structure of the regular expression is taken into account, particularly with respect to Kleene's operators. Moreover, it is also possible

to examine optimizations resulting from recent approaches to approximate search in a dictionary such as [8].

Another factor worth examination is the possibility of admitting two different threshold values for the two words A and B. That seems particularly relevant in the case of long sequences of updates: if the threshold is much lower than the number of updates the user wished to perform then there is a small chance for a candidate remaining within this threshold distance from A to reflect the user's intentions. For example, if the user has performed 10 updates he/she will probably not be satisfied with candidates the vary only by one or two operations from the initial word A. Admitting a higher threshold with respect to A and the lowest possible threshold with respect to B seems a good strategy that we wish to experiment on.

The definition of an incremental string-to-string correction problem is inspired by the domain of incremental XML-document correction, in which an initially valid XML-tree is taken into account in order to limit the correction space to contexts surrounding the points of updates. Thus, naturally, our main perspective is the extension of the presented method to deeper tree structures in which not only a node's siblings but possibly also its ancestors and descendants are taken into account.

References

[1] B. Bouchou and M. Halfeld Ferrari Alves. Updates and Incremental Validation of XML Documents. In *9th International Workshop on Data Base Programming Languages (DBPL), Potsdam, Germany*, 2003.

[2] G. Clarke D. T. Barnard and N. Duncan. Tree-to-tree Correction for Document Trees. Technical Report 95-372, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1995.

[3] M. W. Du. and S. C. Chang. A model and a fast algorithm for multiple errors spelling correction. *Acta Informatica*, 29:281–302, 1992.

[4] P. A. V. Hall and G. R. Dowling. Approximate String Matching. *Computing Surveys*, 12(4):381–402, 1980.

[5] David C. Howell. *Fundamental Statistics for the Behavioral Sciences*. Library of Congress Cataloging-in-Publication Data, 4th ed., 1999.

[6] K. Kukich. Techniques for Automatically Correcting Words in Text. *ACM Computing Surveys*, 24(4):377–439, 1992.

[7] R. Lowrance and R. A. Wagner. An Extension of the String-to-String Correction Problem. *Journal of the ACM*, 22(2):177–183, 1975.

[8] S. Mihov and K. U. Schulz. Fast approximate search in large dictionaries. *Computational Linguistics*, 30(4):451–477, 2004.

[9] K. Oflazer. Error-tolerant Finite-state Recognition with Applications to Morphological Analysis and Spelling Correction. *Computational Linguistics*, 22(1):73–89, 1996.

[10] R. A. Wagner. Order-n Correction for Regular Languages. *Communications of the ACM*, 17(5):265–268, 1974.

[11] R. A. Wagner and M. J. Fischer. The String-to-String Correction Problem. *Journal of the ACM*, 21(1):168–173, 1974.