

Correctors for XML data^{*}

Utsav Boobna¹ and Michel de Rougemont²

¹ I.I.T Kanpur, India utsav@iitk.ac.in

² LRI & Université Paris II, France mdr@lri.fr

Abstract. A corrector takes an invalid XML file F as input and produces a valid file F' which is not far from F when F is ϵ -close to its DTD, using the classical Tree Edit distance between a tree T and a language L defined by a DTD or a tree-automaton. We show how testers and correctors for regular trees can be used to estimate distances between a document and a set of DTDs, a useful operation to rank XML documents.

We describe the implementation of a linear time corrector using the Xerces parser and present test data for various DTDs comparing the parsing and correction time. We propose a generalization to homomorphic DTDs.

1 Introduction

For classification, search and querying semistructured data, it is important to detect approximate validity, i.e. if a file F approximately follows a structure M . In the case of XML data, the structure is a DTD or a schema and we want to decide if a file F approximately follows a DTD M . A search engine for structured data would specify a finite set M_1, \dots, M_k of structures and would like to efficiently rank all documents by estimating their distances to these DTDs. The documents were valid for their own DTD when they were created, but are most likely invalid for M_1, \dots, M_k because of linguistic differences in tags, small changes in the internal structure of DTDs and potential errors in the file. It is therefore important to efficiently estimate how close they are to a fixed set of DTDs.

A given file is well-formed if the tags follow a tree-like structure and is valid if the tree is accepted by the automaton associated with the DTD. The tool Tidy [13] takes an XML file as input and corrects it if it is not well-formed: it adds the missing tags and removes the incorrect ones to obtain a well-formed file close to the original one. We first describe the implementation of a similar tool for validity. It takes a well-formed XML file and a DTD as input and corrects it if it is invalid: it adds the missing leaves and subtrees, removes or modifies the incorrect ones to obtain a valid file close to the original one. Moreover it estimates in linear time the distance between a file and several DTDs which can be used to *rank documents* relative to these DTDs.

^{*} Work supported by *ACI Sécurité Informatique: VERA* of the French Ministry of Research.

The distance between two XML files is the classical Tree Edit Distance [14, 12, 2] applied to the DOM trees associated with the files. It measures the number of insertions and deletions of nodes and edges, and the number of label changes to a given node. It generalizes the classical Edit Distance on strings which measures the number of insertions, deletions and modifications of characters necessary to apply to a string s to obtain a string s' . These Edit Distances have many extensions when additional operators are used, such as Moves of substrings, Permutations and Cut/Paste operations. In this paper two distances are used on unranked ordered trees: the Tree Edit distance and the Tree-Edit distance with moves where we also allow to move an *entire subtree* in one step.

Correctors are related to the self-testers of [3,4] in the theory of program verification. The related notion of *property testing* was proposed in [11], its applications to graphs were studied in [8] and its applications to regular words in [1]. These last authors prove that regular properties of words have testers, i.e. we can decide by sampling a word in constant time, if it belongs to a regular language or if it is far from the language, i.e. at distance more than $\epsilon \cdot n$ if n is the length of the string. An XML tester for a *DTD* and some ϵ takes an XML file F and decides with high probability in constant time if F is valid or if F is ϵ -far from the *DTD*. In [9], we prove the existence of such a tester for regular trees and the Tree Edit Distance with moves.

A corrector for regular words transforms a word s close to a regular language, i.e. at a distance less than $\epsilon \cdot n$ into a word s' in the language such that s' is close to s . For regular tree properties, i.e. properties defined by tree automata, or by DTDs, such correctors have been introduced in [9] for the Tree-Edit distance with moves. A basic corrector was introduced in [7] for the classical Tree-Edit distance. It is a linear algorithm which takes a general invalid file F and a DTD M as input and produces a valid file F' close to F as output, when the distance between F and the DTD is constant, i.e. if there are not too many errors. It used the notion of a *global correction* which implied a time linear in n but exponential in the number of errors.

In this paper, we first emphasize a fundamental difference between the two Tree Edit distances. If we allow moves, we prove that the distance problem is NP-complete and non-approximable. Yet we can approximate it very efficiently in the sense introduced by testers and correctors: we can quickly decide if it close or far and we can approximate it when it is close. Ordered trees with moves behave as unordered trees and this fundamental difference becomes very important for XML schemas as efficient implementations may be easier for approximate validity than for exact validity.

We give a global presentation of correctors and describe the implementation of a *local corrector* for the classical Tree Edit distance (without moves), i.e. an algorithm which is both linear in n and the number of errors but may not yield the best correction. We describe its implementation using the Xerces parser and give test data for invalid files which follow three DTDs: the first represents deep trees, the second wide trees, and the third a combination of deep and wide trees.

The main result of the paper is to show that the correction time is always less than the parsing time and therefore scales up.

We discuss the generalization of a corrector to close DTDs. We consider a file (F, M_1) where M_1 is the DTD of F , and an external DTD M_2 close to M_1 , having defined the distance between two DTDs. We estimate the distance between (F, M_1) and M_2 , when F is close to M_1 , M_2 close to M_1 and therefore F is close to M_2 .

In section 2, we present the notations, the Tree Edit distances, property testing on trees, its connection with the correctors and prove that the distance problem for ordered trees with the Tree Edit distance with moves is NP-complete. In section 3, we describe the structure of correctors, the implementation of a local corrector and give an example. In section 4, we provide general figures on the relative parsing and correction time for three DTDs. In section 5, we introduce a distance on DTDs and generalize the approach to homomorphic DTDs.

2 Preliminaries

A well-formed XML file is composed of two parts: a *ranked ordered labelled tree* and a Document Type Definition DTD or Schema. We define these two objects, the Tree Edit Distance which gives a measure between ordered labelled trees and between a tree and a DTD, and the basic notions of testers and correctors.

2.1 Ranked ordered labelled trees

A *ranked labelled ordered tree* is a ranked ordered tree with labels, i.e. a structure $\mathcal{T}_{rl} = (D_n, Child_i, Label_j, root)_{i \leq m, j \leq p}$ where the domain $D_n = \{1, \dots, n\}$ is the set of nodes, the binary relation $Child_i(u, v)$ is satisfied if u is the i -th child of v for $i \leq m$ and a fixed m and $root$ is a distinguished element of D_n with no predecessors. The graph of the $Child_i(u, v)$ is a tree, $Label_j$ is a unary relation on D_n and the set $\{Label_j\}_{1 \leq j \leq p}$ is a partition of D_n .

Let $\mathcal{L} = \{l_1, l_2, \dots, l_j, \dots, l_p\}$ be the set of labels also called *tags*. A DTD defines one label as a *root* and declares a set of rules $l : (m)$ where l is a tag and m a regular expression on \mathcal{L} , also called the *content model* m . The content model ($\#PCDATA$) indicates a leaf. Each rule $l : (m)$ specifies a transition of an unranked tree-automaton. In this paper we consider bottom-up tree automata first on labelled trees but we may have taken equivalent models.

Our implementation uses the standard parsing tools, Sax and Xerces. After reading the file, we obtain a DOM tree, which is parsed bottom-up. We describe how to handle parsing errors, in order to modify a DOM tree and obtain a valid one. We interleave parsing steps with correction steps and obtain two outputs: a corrected file and an estimation of the Tree Edit Distance.

2.2 The Tree Edit Distance

The Edit Distance on strings has been introduced in [14] for comparing strings and generalized in [12] for trees. The survey [2] describes hardness results for

unordered trees and classical polynomial algorithms for ordered trees. Basic operations on ordered labelled trees include: *change* a label, *insertion* of a node on a given edge (transformed in two edges), *insertion* of an edge or *deletion* of an edge.

Definition 1 *The distance between T and T' is the minimum number of Tree Edit operations necessary to reach T' from T , noted $Dist(T, T')$. The distance between T and a language L , noted $Dist(T, L)$ is the minimum distance between $Dist(T, T')$ for $T' \in L$*

We say that two trees T and T' are k -close if their distance is less than k , and that T is ϵ -close to a DTD if the distance between T and the language L associated with the DTD is less than $\epsilon \cdot n$, if T has n nodes. T is ϵ -far from a DTD if it is not ϵ -close. It is a classical observation that the distance is P-computable for ordered trees and NP-complete on unorderes trees. In the context of XML, several papers [10, 5] estimate similarities between files using variations of the P-algorithm based on dynamic programming, an $O(n^2)$ algorithm. The methods we introduce can decide if the distance to a DTD is small or large in constant time (depending on ϵ) but independent of n if we use a tester and in linear time if we use a corrector which also produces a corrected file.

Distances with moves On strings, a *move* is simply the possibility to isolate an arbitrary substring t and a position i in a string s of size n and to *move* t in position i , shifting the word to the right in one step. The new Edit distance, called *Edit distance with moves* introduced in [6] fot strings has many applications in the database streaming model. In the case of trees, a *move* is the possibility to isolate a subtree t and a leaf i in a tree T of size n . We move t to position i in one step, as shown in the figure below.

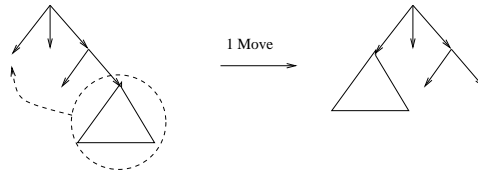


Fig. 1. Tree Edit Distance with moves.

This construction is indeed feasible in the DOM tree model by modifying two edges. The distance problem changes completely as the next result shows because ordered trees with moves behave as unordered trees.

Proposition 1 *The distance problem on ordered trees with the Edit distance with moves is NP-complete.*

Proof. We reduce an NP-complete problem 3SM (3-Set Matching or Exact Cover by 3-Sets) to the distance problem on unranked trees. We use unranked trees similar to the ones mentioned in [2] where it is shown that the distance problem on unordered trees is NP-complete. Consider a set $U = \{u_1, u_2, \dots, u_{3k}\}$ and n sets of 3 elements of U , i.e. $S_1 = \{u_{1_1}, u_{1_2}, u_{1_3}\}, \dots, S_i = \{u_{i_1}, u_{i_2}, u_{i_3}\}, \dots, S_n = \{u_{n_1}, u_{n_2}, u_{n_3}\}$. We have to decide if there are k subsets S_{i_1}, \dots, S_{i_k} which partition U , i.e. whose intersection is empty and their union is U .

Consider the two unranked trees below, T_1 and T_2 on an alphabet $\Sigma = \{u_1, u_2, \dots, u_{3k}, S_1, \dots, S_n, a, t\}$. The tree T_1 is built from the n sets $S_1 = \{u_{1_1}, u_{1_2}, u_{1_3}\}, \dots, S_i = \{u_{i_1}, u_{i_2}, u_{i_3}\}, \dots, S_n = \{u_{n_1}, u_{n_2}, u_{n_3}\}$ as in Figure 2 and the distance between the node S_i and the three leaves is $n + 5k$. The tree T_2 has u_1, \dots, u_{3k} as successors and a node labelled t with $3(n - k)$ leaves at distance $n + 5k$ labelled t .

If we can partition U with k such sets, we can use k delete of edges (a, S_i) followed by k moves to place the $u_{i_1}, u_{i_2}, u_{i_3}$ to the left. We reorder all the u_{i_j} and use up to $3k$ moves to obtain the left part of T_2 . We then use $n - k$ merges on the other sets to obtain the right part of T_2 and $3(n - k) + 1$ modifications of labels after we change all the labels to t . In the worst-case we make $2k + 3k + (n - k) + 3(n - k) + 1 = 4n + 2k + 1 = p$ operations.

If the instance of 3SM is positive, the distance between T_1 and T_2 is less than p . If the instance of 3SM is negative, let us show that the distance is greater than p . The transformation of the second part of the tree requires $3(n - k) + 1$ operations. If we select any subset of size k , there will be at least one duplicate u_i which requires $n + 5k$ operations to remove and in the best case, we apply $k + n + 5k$ delete. The distance is at least $4n + 3k + 1$. This proves that the distance problem on unranked trees with an infinite alphabet is NP-complete.

To reduce the problem to a finite alphabet by replacing nodes labelled by a letter of $\Sigma = \{u_1, u_2, \dots, u_{3k}\}$ by a binary tree of size $\log(3k)$ and suppressing the labels S_1, \dots, S_n . Construct larger trees T'_1 and T'_2 from T_1 and T_2 by replacing the nodes labelled $u_{i_1}, u_{i_2}, u_{i_3}$ by such finite trees. We keep the same construction but instead of $3(n - k) + 1$ modifications of labels, we first need to apply $3(n - k) \log(3k)$ deletions of edges to remove the finite binary trees, and then apply the $3(n - k) + 1$ modifications of labels. We just take $p' = 2k + 3k + (n - k) + (3(n - k) + 1) \log(3k) = (3(n - k) + 1) \log(3k) + n + 4k$. This concludes that the distance problem on unranked trees is NP-complete. If we code the trees T'_1 and T'_2 by binary trees T''_1 and T''_2 using a standard encoding, we notice that if $(T'_1, T'_2) = k$ then $(T''_1, T''_2) = 2k$: we reduce the distance problem on unranked trees to the distance problem on ranked trees. \square

The argument can be also used to show that the distance problem is not approximable as we can maintain an arbitrarily large gap between positive and negative instances. Although this distance is hard to compute, it becomes easy to approximate in the sense introduced by testers.

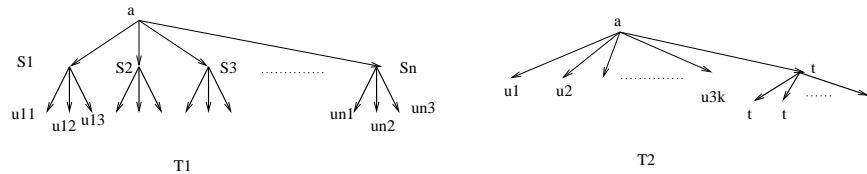


Fig. 2. Trees T_1 and T_2 .

2.3 Testers and Correctors

In the late eighties, the theory of *program checking* and *self-testing/correcting* was initiated in [3, 4]. Correctors in this context take a program, incorrect on a few instances, as an oracle and produce a correct program with high probability. Many interesting correctors for numerical computations and linear algebra are presented in [3].

The related notion of *property testing* was first proposed in [11]: it is an approximate randomized test which separates with high probability inputs which satisfy a property from those which are far from the property, using the Hamming distance to compare structures. For graphs, [8] investigated property testing for several properties such as ϵ -colorability which can be approximately tested in constant time. In [1], the authors prove that any regular property of strings can be tested and this result is generalized for the Edit Distance in [9]. For trees with the Tree Edit distance with moves, [9] provides a randomized algorithm A which given a tree T of size n and a regular tree language L defined by a DTD is such that: A accepts if $T \in L$ and A rejects with high probability if T is ϵ -far from L , i.e. the distance between T and L is greater than $\epsilon \cdot n$. The time of the algorithm is independent of n and depends on ϵ only. Such an algorithm samples the DOM tree by selecting a random node and a finite local subtree of size $1/\epsilon$, and finally checks a local property.

There is a corrector which takes a tree T ϵ -close to L , i.e. at distance less than $\epsilon \cdot n$ as input and produces a tree $T' \in L$, which is ϵ' -close to T , as output.

Notice that the Tree Edit distance with moves is hard to compute but we can estimate it if it is small. First use the tester to detect if it is large. If it is small, use a corrector for an estimate.

3 Correctors for XML

A Corrector takes an XML file (which may not be valid) and the corresponding DTD as input and produces a valid XML file close to the original one and the approximate Edit distance, as output. For the Tree Edit distance without moves, we don't know correctors which would correct up to distances $\epsilon \cdot n$ but we know correctors which correct up to a constant distance. We now present two classes of correctors and an implementation for this distance.

3.1 Corrector for the Tree Edit distance

There are two distinct steps proposed in [7], after having read a file.

1. **Inductive *-Marking of the DOM tree.** Follow a bottom-up run on the DOM tree and mark with a * the nodes where a parsing error occurs. The root of a subtree which contains *-nodes is a * node if all substitutions of * with feasible tags lead to a parsing error.

2. **Recursive correction of the *-subtrees.** Proceed top-down and propose local modifications in the neighborhood of the * node of maximum height. Compute the global distance obtained when the leaves are reached and choose the modification of minimum distance.

This algorithm guarantees that the obtained DOM tree is valid and at a predictable distance of the original tree if the original file was at distance k to the DTD. Notice that the number of * nodes is a first estimate of the distance. It is $O(n)$ if n is the size of the tree, but exponential in the distance k . We therefore consider Local Correctors where a local correction is made at each * node, in order to remove this exponential constant factor.

3.2 Local Correctors

A d -local Corrector makes corrections at each * node, looking at a finite neighborhood at distance d , below the node. Clearly local corrections may not be as good as a global correction, but there are more efficient as they remove the exponential constant factor 2^k . We propose a 1-local Corrector, or Local Corrector, i.e. look at the direct successors of a * node or neighborhood of size 1 below the node.

A local correction at a * node proceeds as follows. We propose a new label t for a * node which guarantees that the tree T will be valid. We consider the string s of labels of the successors of the * node, select a rule of the DTD which minimizes the Tree Edit Distance and apply a local correction.

Local Corrector

Input: a file F and a DTD.

Output: a valid file F' close to F .

1. **Inductive *-Marking of a tree.** Follow a bottom-up run on the DOM tree and mark with a * the nodes where a parsing error occurs.

2. Iterate Top-down at each * node. Propose a valid tag, i.e. a tag which leads to a valid tree, to a * node, apply a local correction at each node.

Local Correction

Input: a * node with a new label t , the string s of labels of successor nodes and a DTD.

Output: the closest string s' of the DTD, the Edit distance and a sequence of local corrections at the * node.

1. Consider all rules of type $t : m$ where m is a content model, i.e. a regular expression in the language of \mathcal{L} .

2. Compute the Edit-distance between m and s . Select the rule with the minimum distance and closest string s' . Make the local correction, i.e. the sequence of Deletion, Insertion and Modification to s to obtain s' .

The key function is to compute the distance between a string and a regular expression and there are efficient algorithms for this problem as in [15]. We then obtain a distance and a corrected string s' where nodes are added, removed or modified. In order to solve very efficiently the distance to a regular expression problem, we assume the DTD uses only the $*$, $+$ operators to a *single tag* and we say that such a DTD is in *Unary Normal Form*.

3.3 Java Implementation Details

An implementation of the corrector using the notion of local correction at every error node is available on the site <http://www.lri.fr/~mdr/xml>. It uses the Sax and Xerces parsers and its key features are:

Phase 1: Parsing As we read the file, we scan the DTD with the Sax Parser, stores the declaration of each element in a class and obtain the DOM structure of the document by parsing it with the Xerces-j parser. We travel the DOM structure Bottom-up, and assign *Height* and *SubtreeSize* to each node as attribute. *Height* will be used to set the preference for correction whereas *SubtreeSize* will be used to calculate the approximate edit distance. We isolate the error nodes (* nodes) along with the error statement as we consider several parsing possibilities. We store these nodes in an array. If there is any fatal error (i.e., the given XML document is not well formed) then exit, else go to the Correction phase.

Phase 2: Correction For each * node, we derive possible tags for that node such that the root of the tree accepts. For each possible tag t , we access the DNF of its Content Model, the minimum height of its subtrees (to compute the Edit distance while inserting or removing nodes) and the string s of tags of the successors. For each content model m , compute the Edit distance between s and m . Choose the model m with the minimum Edit distance. We obtain the string s' closest to s and a sequence of basic operations among M (Match), D (Delete), I (Insert), C (Change) which have to be realized for each letter of s . Modify the tree following these modifications. Check the attributes of each child node. If some attribute is missing then add it, if there is an undeclared attribute then remove it and if the attribute is not correctly defined then modify it.

Compute the total Edit distance, for all the corrections done. Remove the attributes added as part of this program. Serialize this DOM structure to an XML Document.

3.4 Example of a corrected file

Consider the following DTD which defines a class of right-branch trees.

```
<?xml version="1.0"?>
<!DOCTYPE a [<!ELEMENT a (1,r)><!ELEMENT r ((1,r)|q) >
<!ELEMENT 1 (#PCDATA) ><!ELEMENT q (#PCDATA) >>
```

Consider a file F whose DOM tree is represented in (a) of the following figure. At parsing time we generate two $*$ nodes. Correcting top-down, the first $*$ node is labelled r and the string s below at distance 1 is r . There are two possibilities for s' : $s'_1 = l, r$ or $s'_2 = q$ from the DTD. In the first case, we correct by introducing a left branch l and the Tree Edit Distance is 1. In the second case we drop the subtree labelled r and replaces it by a branch labelled with q . The Tree Edit Distance is the size of the subtree, i.e. 11 and this choice is not taken. We proceed until the next $*$ node labelled r and $s = l, r, q$. In this case $s' = l, r$ and we drop the q branch. The total Edit Distance is 2.

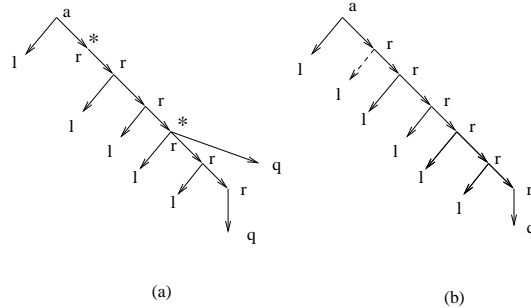


Fig. 3. Correction of a DOM tree.

4 Comparative Study

We consider three different DTDs and analyze the parsing and the correction time of files up to 800 nodes. The first DTD defines deep binary trees, the second DTD defines wide trees and the third DTD defines a mixture of deep and wide trees.

4.1 Deep Tree example

The first DTD defines the class of right-branch trees with the tags a, l, r, q , as in the example of section 3.5. Given a valid binary tree, we randomly add extra branches to a node r , remove some left branches and change some of the r tags. Assume a distance of 10, i.e. we add 10 errors to various files from 50 nodes to 800 nodes.

The Figure 5 shows the approximate time in milliseconds taken for the parsing as well as the correction time of the XML document. It clearly indicates a correction time negligible compared to the parsing time. In this case, the possible expanded words w at each stage are very short and the Edit distance computations are very efficient.

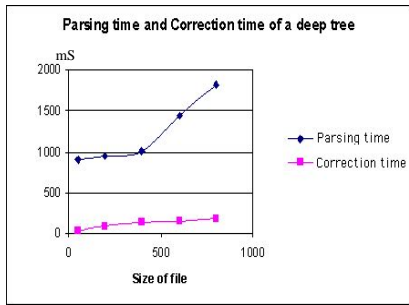


Fig. 4. Analysis of Deep trees.

4.2 Wide Tree Example

The second DTD is closer to classical examples in databases where the branching degree is large. We represent a document with a title and many lines, each one composed of many chars. We introduce 10 random errors in wide valid trees of size 50 up to 800 nodes.

```
<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE d [
<!ELEMENT d (title,line*)><!ELEMENT line (b,char*,c)>
<!ELEMENT title (#PCDATA)><!ELEMENT char (#PCDATA)>
<!ELEMENT b (#PCDATA)><!ELEMENT c (#PCDATA)>]>
```

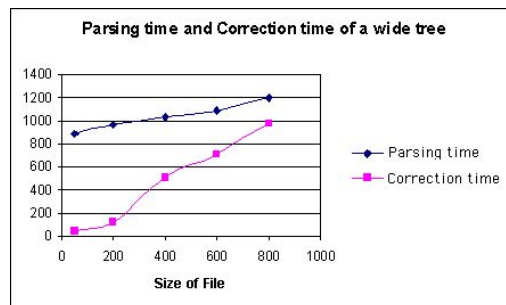


Fig. 5. Analysis of Wide trees.

In this case, the discrepancy in the parsing and correction time is due to the large number of expansions of the regular expressions to be considered. This number increases drastically and we need to compute the edit distance for many probable corrections and then sort out the smallest one. Notice that the correction time is still less than the parsing time.

4.3 Deep Tree having some wide branches

We now define a mixture of the previous DTDs. We have many *bs* nodes below which we attach a right-branch tree, as defined by the first DTD. Starting from valid trees, we generate invalid trees with 10 random errors of size 50 up to 800 nodes.

```
<?xml version="1.0"?><!DOCTYPE b [<!ELEMENT b ((a|b),r)>
<!ELEMENT a (l,r)><!ELEMENT r ((l,r)|q ) >
<!ELEMENT l (#PCDATA) ><!ELEMENT q (#PCDATA) >]>
```

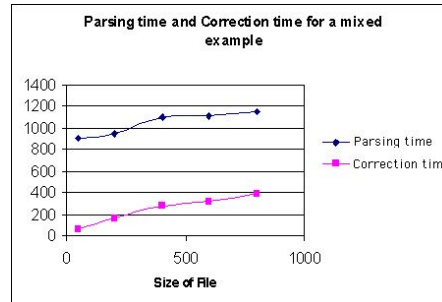


Fig. 6. Wide branches of deep trees.

In this last case, the correction time is the average of the two previous example and significantly less than the parsing time.

If we had considered the global correction, the situation would have been quite different. We would need to check all the permutations of all possible corrections at each error node and then select the best correction. For deep trees, it would be very inefficient whereas for wide trees, it would be acceptable. The global method may have produced a better correction, but very inefficient for deep trees.

4.4 The Book example

Consider the following standard realistic DTD describing a *book* document. On the site <http://www.lri.fr/~mdr/xml>, there are several examples of invalid files with few errors and the corrected valid file is obtained on-line. In general, the user can upload his own file, provided the DTD is included and in Unary Normal Form. If the distance is not too large, he will obtain a corrected valid file on-line.

```
<?xml version='1.0' ?>
<!ELEMENT book (chapter*,title,author)>
<!ELEMENT chapter (title,para*)><!ELEMENT title (#PCDATA)>
<!ELEMENT para (#PCDATA)><!ELEMENT author (#PCDATA)>
```

A more general problem concerns the correction of a file F , using a DTD M_1 with respect to a close DTD M_2 . It is essential to rank Web documents relative to a given DTD. We have to consider not only the given DTD but also close ones, in particular to adjust with the different languages used on the Web. The following *DTD* is a close french version of the previous one. If we wish to rank documents of type *Book* with their Edit-distance, we need to also consider french books, chinese books and so on.

```
<?xml version='1.0' ?>
<!ELEMENT livre (chapitre*,editeur,auteur,titre)>
<!ELEMENT chapitre (titre,para*)>
<!ELEMENT titre (#PCDATA)><!ELEMENT para (#PCDATA)>
<!ELEMENT auteur (#PCDATA)><!ELEMENT editeur (#PCDATA)>
```

We show that we can generalize the corrector to handle DTDs which are close to a given DTD.

5 A corrector for homomorphic Data Type Definitions

The correction algorithm can be generalized to DTDs which may not be the original DTD of the file, provided they are close. If a document F is a french book with a DTD close to the english one, we wish to find a mapping between french and english tags and an approximate distance between the french document and the english DTD. We generalize the distance between a tree and a DTD to a distance between two DTDs by defining a function of n , as the maximum of the distances between pairs of trees in M_1, M_2 of size n , making the definition symmetric. When they are close, i.e. the distance is $O(1)$, we propose to adapt the previous corrector.

5.1 Distance on *DTDs*

Consider two regular expressions or two DTDs. We first consider the case when they use the same language of tags \mathcal{L}_1 and then different languages \mathcal{L}_1 and \mathcal{L}_2 .

Definition 2 *The distance between two regular expressions r and t on the same language \mathcal{L}_1 is the function $Dist(n, r, t) = Max\{ Dist1(n, r, t), Dist1(n, t, r) \}$ where*

$$Dist1(n, r, t) = Max_{w \in r, |w|=n} \{ dist(w, t) \}$$

Similarly, the distance between two DTDs M_1 and M_2 on the same language \mathcal{L}_1 is the function $Dist(n, M_1, M_2) = Max\{ Dist1(n, M_1, M_2), Dist1(n, M_2, M_1) \}$ where

$$Dist1(n, M_1, M_2) = Max_{F \in M_1, |F|=n} \{ dist(F, M_2) \}$$

We say that two DTDs are at distance $O(1)$ or at constant distance if $Dist(n, r, t) = O(1)$. We are interested in short distances, say up to $\log n$, as we wish to capture close DTDs. In the sequel we only consider DTDs at constant distances.

Example. Let $\mathcal{L}_1 = \{a, b, c, d, u, v\}$. If $r = au * bv * cd$ and $t_1 = au * av * c$ then $Dist(n, r, t_1) = O(1)$. If $t_2 = ab * v * c$ then $Dist(n, r, t_2) = O(n)$.

If regular expressions or DTDs use different languages \mathcal{L}_1 and \mathcal{L}_2 , we wish to generalize this definition. We consider mappings π from $\mathcal{L}_1 \cup \{\perp\}$ to $\mathcal{L}_2 \cup \{\perp\}$ such that $\pi(\perp) = \perp$. We then extend the previous definition to two regular expressions r and t by taking the Minimum over all π of the distance between $\pi(r)$ and t . We interpret \perp as the empty symbol, i.e. $a.\perp = \perp.a = a$

Definition 3 *Two regular expressions r and t are isomorphic if there exists a mapping π between the symbols (tags) of r in \mathcal{L}_1 and the symbols (tags) of t in \mathcal{L}_2 such that $\pi(r) = t$. Two regular expressions r and t in Unary Normal Form such that $|r| \geq |t|$ are homomorphic if there exists a mapping π between $\mathcal{L}_1 \cup \{\perp\}$ and $\mathcal{L}_2 \cup \{\perp\}$ such that the distance between $\pi(r)$ and t is $O(1)$.*

In case $|r| \leq |t|$, consider mappings from \mathcal{L}_2 into \mathcal{L}_1 and check $\pi(t)$ with r .
Example. Let $\mathcal{L}_1 = \{a, b, c, d, u, v\}$ and $\mathcal{L}_2 = \{a, b, c, d, u, v\}$ If $r = au * bv * cd$ and $t_1 = cx * y * e$ then $Dist(n, r, t_1) = O(1)$ with π such that $\pi(a) = c, \pi(u) = x, \pi(b) = \perp, \pi(v) = y, \pi(c) = e$ and $\pi(d) = c$. Hence $au * bv * cd$ and $cx * y * e$ are homomorphic.

If $t_2 = cx * y * e*$ then $Dist(n, r, t_2) = O(n)$ for any mapping π and these regular expressions are not homomorphic. We generalize now the previous definition to two

DTDs on different languages, assuming they are reduced, i.e. each rule influences the tree language associated with the DTD. Consider that the DTD provides the *DNF* form for each tag. For a tag a , let $DNF(a)$ be the regular expression which defines a . Suppose without loss of generality that $|\mathcal{L}_1| \geq |\mathcal{L}_2|$. Call a tag a *recursive* if either $a*$ occurs in some content model or if a occurs in the *DNF* form of a or if a is on a loop in the dependency graph whose nodes are tags and edges link a tag b with all the tags in its *DNF*.

Definition 4 *Two DTDs M_1 and M_2 with roots r_1 and r_2 are homomorphic if there exists a mapping π between $\mathcal{L}_1 \cup \{\perp\}$ and $\mathcal{L}_2 \cup \{\perp\}$ such that:*

- if a is recursive in M_1 then $\pi(a)$ is recursive in M_2 and $\pi(DNF(a))$ is isomorphic to $DNF(\pi(a))$
- if b is non recursive in M_1 then $\pi(DNF(b))$ is homomorphic to $DNF(\pi(b))$.
- $\pi(r_1) = \pi(r_2)$.

If $|\mathcal{L}_1| \leq |\mathcal{L}_2|$, we require π to map $\mathcal{L}_2 \cup \{\perp\}$ into $\mathcal{L}_1 \cup \{\perp\}$ as before.

Example. Let $\mathcal{L}_1 = \{book, chapter, title, author, para\}$ for the french DTD and $\mathcal{L}_2 = \{livre, chapitre, titre, auteur, para, editeur\}$ for the english DTD. In this case *chapitre, para, titre* are recursive in the french DTD M_2 .

Let π such that $\pi(livre) = book, \pi(chapitre) = chapter, \pi(titre) = title, \pi(auteur) = author, \pi(para) = para, \pi(editeur) = \perp$.

In this case the recursive tags are mapped to isomorphic *DNFs* and the non recursive tag *livre* is mapped to a homomorphic *DNF* as $\pi(chapitre*, editeur, auteur, titre) = (chapter*, title, author)$ is at a constant distance from $(chapter*, author, title)$.

Proposition 2 *Two homomorphic regular expressions r, t are at distance $O(1)$. Two homomorphic DTDs are at distance $O(1)$.*

Given a file F with a DTD M_1 , and another external DTD M_2 , we wish to estimate the distance between F and M_2 . We can look for all possible π which are homomorphisms between M_1 and M_2 , and look at the distance between $\pi(F)$ and M_2 with the previous corrector. This procedure would be exponential in the number of tags and inefficient in practice. We can however generalize the approach of a corrector: we build bottom-up a possible π and correct top-down as before.

5.2 Generalized Local Corrector

A (d, l) -local Generalized Corrector looks bottom-up at finite neighborhoods at distance d above a node and constructs the π which minimizes the edit distance. As we proceed, bottom-up, we only extend a given π and quit if the distance is too large. We apply the corrector top-down as before to estimate the distance between F and an external DTD M_2 .

The local construction of a π is not as good as the exhaustive search for the best π but is very efficient. We propose a $(2, 1)$ -local Generalized Corrector: we look at depth 2 above as we proceed bottom-up and at depth 1 top-down as before. It has the advantage of catching quickly the expressions a^* which have to be matched to a b^* in the other DTD. We call GLC for Generalized Local Corrector such a $(2, 1)$ -local Generalized Corrector.

GLC

Input: a file (F, M_1) , an external DTD M_2 and parameter k .

Output: the approximate distance between F and M_2 , when it is less than k and a corrected file F' .

1. Follow a bottom-up run on the DOM tree and inductively construct the best π . Mark with a $*$ the nodes where a parsing error for M_2 occurs.
2. Proceed Top-down as in the classical Corrector. Propose a valid tag, i.e. a tag which leads to a valid tree, to a $*$ node, apply a local correction at each node.

We build a potential mapping π , bottom-up and estimate the Edit Distance. If it is greater than k , we stop and consider another mapping. If we reach the root with a potential mapping π , we correct $\pi(F)$. The advantage of this construction is that very few possible π are considered, especially when started with a random subtree of depth 2 from the leaves. We construct an approximate π bottom-up and obtain with the Corrector an approximate distance Top-down.

6 Conclusion

A corrector for XML documents can estimate in linear time the distance between a document and a DTD when this distance is small. It is a useful operation to rank documents relative to several DTDs. The proposed implementation of a local corrector uses the Xerces parser and shows that in general the correction

time is less than the parsing time and therefore scales up. It only corrects documents for a fixed constant distance but we intend to generalize the corrector for the Tree Edit distance with moves. In this case we would correct for a distance up to $\epsilon \cdot n$, as predicted by the theory.

We proposed the generalization of a corrector to external DTDs, i.e. to handle a file (F, M_1) whose DTD is M_1 with an external DTD M_2 , provided the distance between M_1 and M_2 is constant. At the heart of the problem lies the approximate equivalence of regular expressions.

References

1. N. Alon, M. Krivelevich, I. Newman, and M. Szegedy. Regular languages are testable with a constant number of queries. *IEEE Symposium on Foundations of Computer Science*, 1999.
2. A. Apostolico and Z. Galil. Pattern matching algorithms, chapter 14: Approximate tree pattern matching. *Oxford University Press*, 1997.
3. M. Blum and S. Kannan. Designing programs that test their work. *ACM Symposium on Theory of Computing*, pages 86–97, 1989.
4. M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *ACM Symposium on Theory of Computing*, pages 73–83, 1990.
5. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD*, pages 493–504, 1996.
6. G. Cormode. Sequence distance embeddings. *Ph.D. thesis, University of Warwick*, 2003.
7. M. de Rougemont. A corrector for XML. *ISIP: Franco-Japanese Workshop on Information Search, Integration and Personalization, Hokkaido University*, <http://ca.meme.hokudai.ac.jp/project/fj2003/>, 2003.
8. O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. In *IEEE Symposium on Foundations of Computer Science*, pages 339–348, 1996.
9. F. Magniez and M. Rougemont. Property testing of regular tree languages. *ICALP*, 2004.
10. A. Nierman and H. V. Jagadish. Evaluating structural similarity in XML documents. In *Proceedings of the fifth International Workshop on the Web and Databases*, pages 61–66, 2002.
11. R. Rubinfeld and M. Sudan. Robust characterizations of polynomials with applications to program testing. *SIAM Journal on Computing*, 25:23–32, 1996.
12. K. C. Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery*, 26:422–433, 1979.
13. Tidy. *HTML Tidy Library Project*, <http://tidy.sourceforge.net>. 2000.
14. R. Wagner and M. Fisher. The string-to-string correction problem. *Journal of the Association for Computing Machinery*, 21:168–173, 1974.
15. S. Wu, U. Manber, and E. Myers. A subquadratic algorithm for approximate regular expression matching. *Journal of algorithms*, 19:346–360, 1995.