Bridging the gap between tracking and detecting changes on XML

Paolo Ciancarini[1], Angelo Di Iorio[1]*, Carlo Marchetti[2], Michele Schirinzi[3], and Fabio Vitali[1]

[1]*Department of Computer Science and Engineering, University of Bologna, Bologna, Italy*
[2]*University of 'La Sapienza' and Senato della Repubblica Italiana, Rome, Italy*
[3]*CINI, Consorzio Interuniversitario Nazionale per l'Informatica, Rome, Italy*

SUMMARY

There are two main approaches to manage changes in XML documents: change-tracking and diff. Change-tracking tools, which record edit actions while they are performed on the source document, are able to capture the exact editing process. That is much more difficult for diff algorithms, which have to reconstruct it by comparing two different versions. Interestingly, these algorithms process both text-centric and data-centric XML documents the same way.
In this paper we show that more accurate, clear, and human-readable results can be achieved on text-centric resources, by employing specific models and algorithms. We describe and discuss a specialised diff algorithm for such a class of documents. We also compare a Java implementation of the algorithm — named JNDiff — with other general-purpose or data-oriented diff tools, focusing on the quality of their output.

Received . . .

KEY WORDS: Change detection; Change-tracking; diff algorithms; XML diff; edit operations; high-

1. INTRODUCTION

Diff algorithms have been introduced to determine the differences between documents. On the other side, for years, word processing suites such as MS Word and Open Office have been able to track and represent visually the differences between modified versions of a document.

Even if these two approaches are strongly related, there is a key difference: diff algorithms work *backward* from two documents to deduce the modifications occurred from the older to the newer version, while change-tracking records such modifications *while they are happening*.

It is easier to use change-tracking for capturing *exactly* what happened during an editing session. On the other hand, this solution is not always viable. Users often need to compare documents created by other people, with a plethora of tools and without activating change-tracking capabilities. Moreover, change-tracking strongly depends on the editing tool used, and more so on the specifics of its internal data format. Diff algorithms are more versatile and may work knowing either nothing about the data format or about the meta-syntax used (e.g. XML).

There is a lively discussion in the research community about diff algorithms and change-tracking, with a series of events organised to '*discuss these topics from different perspectives and to understand which are the most common issues and which are the peculiarities of each domain and each approach*'[1][2].

*Correspondence to: Angelo Di Iorio (angelo.diiorio@unibo.it), Department of Computer Science and Engineering, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy

These issues are particularly relevant in the world of XML. Several specialised diff algorithms are available that take two XML documents as input and produce a *delta* showing their differences[3][4][5]. Also, several collaborative editing systems are now able to track and serialise changes on XML documents in multiple formats–like OOXML, ODF, DITA and DocBook– and support complex workflows on such documents[6].

However, the quality of XML diff algorithms is not the same of that of change-tracking tools. This does not mean that XML diff algorithms produce incorrect outputs, rather that they detect different set of changes compared to those that a change-tracking application would detect. In fact, deltas are hardly ever unique, since multiple different sequences of operations can be thought of, that are all capable of generating the newer document from the older one. XML diff algorithms focus on minimality and efficiency, with less emphasis on the way changes are modelled, summarised, and displayed to the final users.

There are also historical reasons that explain the difference. The research on XML diff algorithms has been carried on primarily by the database community, that has to deal with huge quantities of data and to reduce space and time consumption.

This leads us to a further consideration: comparing textual documents in XML is intrinsically different from comparing XML data structures, because the nature of changes and their application are different. Other works support our hypothesis on the specificity of diff-ing text-centric documents. For instance, Ronnau et al. [7] [8] conducted extensive experiments on editing patterns for text-centric documents and studied how users' edit actions are reflected into changes to the underlying XML tree. Thus, diff algorithms cannot be designed and evaluated by only taking into account delta size, performance and computational complexity, that are instead characteristics of a data-centric approach. A new generation of diff algorithms specialised for text-centric XML is needed, which is able to capture the most common XML editing operations.

Particularly relevant to this work is the ongoing discussion on 'common and natural' XML editing actions. In the W3C Change Tracking Markup Community Group, in particular, a standard format for tracking changes on XML documents is under discussion[9]. The proposal, which has not yet been formally standardized, includes two levels of conformity.

Level 1 provides basic operations (changes to attributes, insertion/deletion of elements, insertion/deletion of text fragments). Level 2 handles higher-level operations: for instance, the insertion of a return in the middle of a paragraph (so that two paragraphs are created, each containing one part of the text node, and neither node individually contains exactly the same text as before) or the formatting of a text fragment in bold (so that a single text node is turned into three nodes: a text node before, a bold element in between, and a text node afterwards). These changes are of great relevance to the final users and should be detected by XML diff algorithms, with the same precision as they are recorded by change-tracking tools.

In this article we show how to bring diff algorithms closer to change-tracking approaches, so that, although maintaining independence from any editing tool and, in general, the data format, they can be used to capture and show differences in clearer and more understandable way. We present an algorithm that is able to produce high-quality output on text-centric XML. We go into the details of the algorithm, which is called JNDiff and is natively designed to handle such documents and simulates users' behaviour in comparing them. Ample space is given to the comparison of the JNDiff Java implementation with other XML diff tools, in terms of readability and clarity for the final users, through a novel semi-automatic evaluation process.

The rest of the paper is then structured as follows. Section 2 describes the available diff algorithms for XML. Section 3 discusses the main issues in comparing XML text-centric documents. We present our algorithm in Section 4 and evaluate a Java implementation on a set of heterogeneous documents in Sections 5, before concluding in Section 6.

2. XML DIFF ALGORITHMS

A variety of algorithms that calculate differences among XML documents exist. Since XML documents are at the core text documents, any diff algorithm operating on generic text content (such as the *Unix diff* [10] or faster file comparison tools[11]) could be used for XML. Text-based algorithms are powerful and fast, but they can be imprecise when working on specific syntaxes and tree-based structures such as XML.

Thus, we focus on tree-based approaches only. Each shows peculiar strengths: some solutions are particularly efficient, others use a limited amount of memory, others are specialized for specific domains (and data formats), others are very general. However, one aspect is still mostly unevaluated or under-evaluated in their design and implementation: the quality of the output in terms of readability, clarity and accuracy *for human users*.

The algorithms are clustered in three groups: (1) *general-purpose*, (2) *suitable for data-centric content* and (3) *suitable for text-centric documents*.

2.1. Tree-based general-purpose algorithms

The problem of comparing trees was introduced well before the introduction of XML or even SGML, as the *tree-to-tree* editing problem [12] trying to minimize the *edit distance*, i.e., the number of individual operations needed to transform one tree into another [13]. The fastest algorithm to find the minimum cost editing distance between two ordered labeled trees was proposed in [14].

The most relevant approach for our purposes was proposed by Chawathe et al.[4]. The authors extended the idea of ‘minimum edit distance’ by introducing the ‘weighted’ edit distance, which is the distance computed by a specific algorithm. It may not be optimal but produces better results in terms of quality as it introduces new operations such as *move* and *copy*. These operations are in fact present in the set of changes discussed in this paper too.

Lee et al.[15] proposed a diff algorithm able to detect the same meaningful changes on SGML and XML documents, but in a more efficient way. The computational complexity is $O(l_1 \times l_2 + (m_1/l_1 + m_2/l_2) \times l_2)$, where m_1,m_2 denote the number of interior nodes and l_1,l_2 the number of leaves in the input documents.

Diffxml[16] and XMLdiff[17] are two other XML-aware implementations of the original algorithm proposed by Chawathe et al.[4]. The first one is a Java tool that detects changes and outputs the edit script in a custom XML vocabulary (called DUL, Delta Update Language). The set of recognized edit operations is very basic (*insert*, *delete*, *update* and *move*) and does not include some operations recognized by Chawathe (such as *copy* and *glue*). XMLdiff is a Python-based implementation of the same algorithm: the basic model and the set of detected changes does not change but performance is very low especially for medium/large input files.

The idea of comparing trees and making differences understandable for the final users have also been investigated in software versioning. In [18], the authors introduced a single-pane interface for displaying unmodified content and differences. Though focused on generic files comparison, the work is also relevant here since it adopts a fine classification of changes (including some specialized modifications, besides additions and deletions) and a fine level of granularity, in order to improve the readability of changes for the final users. In [19] the authors proposed an efficient algorithm to compare programs by comparing their tree representations; the implementation was also sided by a tool for pretty-printing those differences and helping developers to filter and capture them.

2.2. Tree-based algorithms for data-centric documents

The database community has proposed some of the fastest and most sophisticated algorithms for XML diff-ing. On the other hand, these algorithms often improve the efficiency to the detriment of the output quality and minimization.

Cobéna et al. [3] [20] created XyDiff, the fastest algorithm we have found, as shown by Ronnau et al.[7] and confirmed by the experiments presented in Section 5. The computational complexity is very low: $O(n \times log(n))$, where n indicates the number of nodes. The algorithm was in fact designed to manage massive volumes, but the same authors recognized such efficiency has a trade off with regard to quality.

Faxma[21] is another very fast algorithm, whose performance is comparable to XyDiff. In fact, they both share the same overall diff strategy. The authors even claim that ”operations of higher complexity were left out since [they] considered the potential gain in output quality to be very limited compared to the accompanying increase in complexity"[21]. The output format of Faxma is another interesting aspect: the algorithm does not generate a sequence of edit operations but an XML format (XMLR) that uses XPath-like expressions to refer to the unchanged fragments of the original document and, among them, interposes the newly inserted elements/attributes. It is therefore impossible to identify the deleted content from the delta. On the other hand, such a solution guarantees a very limited use of memory and resources.

Xandy[22] uses an alternative approach to compute differences: XML documents are converted into relational tuples and changes are detected by executing SQL queries. Such a solution, as well as many others in this context, work very well on data-centric content but have significant limitations on text-centric documents, whose structure is hardly regular.

In general, these data-centric algorithms detect and store changes of a few types (in most cases just insertions, deletions and updates) but provide sophisticated storage architectures and numbering schemes to archive and rebuild versions efficiently[23][24].

2.3. *Tree-based algorithms for text-centric documents*

There are very few XML diff algorithms explicitly tailored towards text-centric documents. One of the most relevant - and the closest to our work - is DocTreeDiff[5]. The authors analyzed patterns in editing office documents and extracted rules for their identification. In particular, they studied the mapping between high-level changes to the document and consequences on the underlying XML tree[7][8].

DocTreeDiff computes the longest common subsequence (LCS)[25] of all the text included in leaves (by exploiting the Myer's algorithm [26]) and uses a dynamic programming approach to compare parent nodes, in order to detect all matching text fragments. Updates, insertions and deletions are detected in the second phase by analyzing non-matching nodes.

The approach of CX-Diff[27] is also very relevant in this context. CX-Diff detects changes to the textual content of XML documents, without taking into account structural modifications. The basic idea is to compute differences only between the parts of the document that are directly visible to the final users. Changes to single or contiguous words are detected, even though they span multiple nodes in the XML structure. CX-Diff uses a two-phases approach that extracts text units and matches the best common subtrees by analyzing duplicates. Some strong assumptions were possible for the CX-Diff domain: the fact that structural changes are less important for the final users and the fact that pages are relatively static. Those assumptions makes CX-Diff rather difficult to be used in a broader domain, though the idea of customizing the *diff* to the users' perception of the documents looks very interesting.

3. DIFFING AND TRACKING CHANGES ON TEXT-CENTRIC XML

It is very hard for XML diff algorithms to produce edit scripts that match *exactly* the editing process. The problem is that deltas are not unique, since multiple different sequences of operations exist that are all capable of producing the same final result, and these algorithms, as discusses in the previous section, tend to produce deltas that can be found faster and that contain a smaller amount of detected edits, giving less importance to other qualitative aspects.

Things are radically different with XML change-tracking tools. The diffusion of XML vocabularies as internal format for word processors, like OpenXML for Microsoft Word and ODT for OpenOffice, or as serialisation formats, like DocBook and JATS, made these tools widespread and well-integrated in many editors. Their accuracy is very high: they record modifications directly while they are happening and, doing so, manage to capture the edit history in a very precise way. XML diff algorithms do not guarantee the same quality: they produce correct deltas but they detect a different set of changes compared to those that a native change-tracking application would detect.

Let us now discuss these differences with some examples. Consider a very frequent situation: the formatting of text, where a bold style is added to a small fragment of text by wrapping it into a new element. A basic example in XHTML is shown in Figure 1.

ORIGINAL	**MODIFIED**	**DELTA**
`<p>Some bold text.</p>`	`<p>Some <b>bold</b> text.</p>`	`<p>Some <del>bold</del> <ins><b>bold</b></ins> text.</p>`

Figure 1. Formatting a text-fragment in bold.

On the right side, we show how such a change is recognised by most tools: as the deletion of the text fragment and the insertion of the new element that contains a new text node. This output is correct but it does not reproduce exactly the edit action performed by the author. In fact these two operations are logically interconnected and represent an atomic change in the text, i.e. adding an in-line container around the text-fragment. This is the operation that change-tracking tools detect in similar cases. That is correctly tracked, for instance, by OpenOffice (in ODF format) and MS Office (in OOXML format). OpenOffice uses *milestones* to delimit the boundaries of the wrapped text and a new inline container to associate properties to that fragment, while MS Word uses *fragmentation* to split the content into pieces, some of which are wrapped by a bold element. These techniques for overlapping markup are required to track multiple edits by multiple authors, even conflicting and overlapping. They are out of scope for this paper (the interested reader can find an analysis of how these approaches are used within widespread editors in [28]), like so a review of the syntax and namespaces of each format. What is relevant here is that both these solutions capture a *wrapping* operation around some content, instead of a pair of insertions and deletions.

Figure 2 shows a clear way of expressing the same change, compliant to a standard format for tracking changes on XML documents submitted by the authors of DeltaXML[29] to the W3C Change Tracking Markup Community Group[9].

```
<text:p>Some
 <text:span
      delta:insertion-type="insert-around-content"
      delta:insertion-change-idref="ct1234"
      text:style-name="bold-style">bold</text:span> text.
 </text:span>
</text:p>
```

Figure 2. Tracking the formatting of a text fragment in bold, as proposed to the W3C Change Tracking Markup Community Group.

The proposal is still under discussion and includes two levels of conformity: Level 1 for basic operations (changes to attributes, text and elements insertions/deletions) and Level 2 for higher-level changes that are clearer for the final users and can be obtained by combining low-level operations. Apart form syntactic details, the proposal is relevant for this work: it stresses on the need of capturing 'natural edit actions' and describing them with high accuracy.

Wrap operations are also relevant around XML elements and groups of elements. Consider a situation where two paragraphs in a document are moved within a new section. In the best cases, the algorithms described in Section 2 detect two independent changes: the deletion of the two paragraphs, and, separately, the insertion of a new section containing exactly the same paragraphs. The delta would clearly be more effective if it detected the re-structuring of the content and the fact that the section wraps the two paragraphs, that remain unchanged.

As expected, detection the opposite *unwrap* operation is useful too. It occurs when an intermediate node is removed within a hierarchy, or around a piece of text. Even unwraps are very common when editing text documents, since authors are used to change, polish and re-organize the structures of their document.

Two other operations are widely supported by word processors and editors but not fully detected by XML diff algorithms: *cut&paste* (or *move*) an *copy&paste*.

Moved elements are quite well detected —for instance, by XyDiff[3] or DocTreeDiff[5]— though this affects performance heavily; in fact, many tools allow users to disable moves detection. A common and solid solution to detect moves consists of using a threshold which indicates the 'range' within which a node must be considered moved. Such an approach identifies those elements that are unchanged but in a different position and labels them as moved only if that position is 'close enough' to the original one. The situation is much more complex for moved text fragments, since existing algorithms do not handle the pieces of XML text nodes as independent units but only capture insertions and deletions of entire XML nodes.

Similar issues are evident for the *copy* operation, in which the same subtree or text fragment is duplicated with no associated deletions (as for moves). There is no XML diff algorithm we know of that detects copied text fragments. Copied subtrees have been partially supported by some diff tools, even before XML, for instance by Chawathe et al.[4] and Lee et al.[15], who introduced some heuristic-based approaches to identify such situations. Note also that the detection of *copy* is slightly different form the detection of *move*: copied nodes can be detected by identifying new nodes that match exactly with some nodes in the original document.

There is an important aspect to remark in conclusion. All these operations are hardly frequent in a data-centric XML (database records, data tuples, dumps, etc.) but they are rather common in text-centric documents and whenever we find mixed content elements, containers and sub-containers, and semi-structured content. In fact, database records can be moved or updated, but their subcomponents are hardly pushed down or up in the hierarchy; also, it is quite uncommon that text fragments in database fields are marked-up and wrapped by further elements. Diff algorithms do not take such differences into account. Rather, they primarily focus on optimising delta size, performance and computational complexity, that are instead characteristics of a data-centric approach. More relevance should be given to the set of detected changes and their interpretability and clarity, in order to get these algorithms closer to their change-tracking counterparts.

4. JNDIFF: DETECTING CHANGES ON TEXT-CENTRIC XML

In this section we present an algorithm natively designed to detect changes on XML text-centric documents. The current implementation is in Java, thus we call it JNDiff. The algorithm takes two XML trees as input and returns a set of relationships between their nodes, that are eventually serialized in the final delta. The syntactical details of the delta and the serialisation format depend on the implementation and are not relevant at this stage.

Figure 3 shows the overall pseudo-code of JNDiff. It adopts an *iterative* approach: it tries to understand which are the relationships between the parts of the two input documents and refines these relationships progressively. It is divided in two logical blocks: JNDiff Core (lines 1–3) and JNDiff Refinement (lines 4–16). The first one is mandatory and consists of identifying the 'longest matching parts' between the two documents and detecting low-level changes—namely insertions and deletions— on the remaining parts. The second block consists of 'interpreting and refining' these changes.

The core of JNDiff (lines 1–3) consists of (i) building internal data structures by serialising input trees as sequences with information about each node and subtree, and (ii) identifying the unchanged parts between the trees by detecting the equal parts between those sequences. The strategies used to identify these subsequences (line 3) play a key role in JNDiff and strongly impacts on its applicability to text-centric documents and on the quality of its output. We discuss these strategies in detail in the following section. As first approximation, the unmatched parts in B are considered 'inserted' and the unmatched parts in A are considered 'deleted' (lines 4–6).

These changes are then analysed and refined in order to detect higher-level changes. For instance, the current implementation of JNDiff detects as 'updates' those parts which have been slightly modified but do not change their position (lines 7–9) or as 'moved' those parts which do not change but are in a different position (lines 10–12). Note that each pair of insertions/deletions that is translated into an higher-level change is contextually removed from the lists of low-level insertions and deletions (lines 8–9 or 11–12).

Note also that each of these refinements is optional and can be executed in different order (thus we could invert lines 7–9 and 10–12 in the pseudo-code). This allows users to customize

```
Input: A, B                                                              ▷ Takes two trees as input
Output: $\Delta_{AB}$                                ▷ Returns a set of relationships between their nodes

 1: $VT_A \leftarrow$ BUILDVTREE$(A)$                                       ▷ Builds sequences from trees
 2: $VT_B \leftarrow$ BUILDVTREE$(B)$

 3: MATCH$_{AB} \leftarrow$ OPTIMIZEDFINDNLCSS$(VT_A, VT_B)$     ▷ Identifies the longest matching
    subsequences

 4: INS$_{AB} \leftarrow$ REMOVEFROM$(VT_B,$ MATCH$_{AB})$      ▷ Unmatched sequences in B are *insertions*
 5: DEL$_{AB} \leftarrow$ REMOVEFROM$(VT_A,$ MATCH$_{AB})$      ▷ Unmatched sequences in A are *deletions*
 6: $\Delta_{AB} \leftarrow$ INS$_{AB} \cup$ DEL$_{AB}$

 7: UPD$_{AB} \leftarrow$ DETECTTEXTUPDATES(INS$_{AB}$, DEL$_{AB}$) ▷ Refine pairs of *insert/delete* as *updates*
 8: INS$_{AB} \leftarrow$ REMOVEFROM(INS$_{AB}$, UPD$_{AB}$)
 9: DEL$_{AB} \leftarrow$ REMOVEFROM(DEL$_{AB}$, UPD$_{AB}$)

10: MOVE$_{AB} \leftarrow$ DETECTMOVES(INS$_{AB}$, DEL$_{AB}$)      ▷ Refine pairs of *insert/delete* as *moves*
11: INS$_{AB} \leftarrow$ REMOVEFROM(INS$_{AB}$, MOVE$_{AB}$)
12: DEL$_{AB} \leftarrow$ REMOVEFROM(DEL$_{AB}$, MOVE$_{AB}$)

13: FALSEMATCHES$_{AB} \leftarrow$ MATCHESEXPANSION(INS$_{AB}$, DEL$_{AB}$)      ▷ Refine matches and
    *wrap/unwrap*
14: INS$_{AB} \leftarrow$ REMOVEFROM(INS$_{AB}$, FALSEMATCHES$_{AB}$)
15: DEL$_{AB} \leftarrow$ REMOVEFROM(DEL$_{AB}$, FALSEMATCHES$_{AB}$)

16: $\Delta_{AB} \leftarrow$ UPD$_{AB} \cup$ MOVE$_{AB} \cup$ INS$_{AB} \cup$ DEL$_{AB}$      ▷ High-level changes and residual
    *insert/delete*
```

Figure 3. The JNDiff core structure and its optional refinement phases.

the algorithm according to their preferences and needs. Although we tailored JNDiff for text-centric documents, the overall structure could be specialized for different applications domains. For instance, a more efficient configuration for data-centric structures could be set up by deactivating modules for moves and wraps/unwraps detection, since these operations are very uncommon in that context. Furthermore, new modules could be implemented and easily plugged in the same structure, in order to detect further operations. The configurable order of these phases is another aspect worth remarking. Very different results are produced according to their relative positions since the sooner a phase is executed, the more its corresponding edit action is given importance.

Let us now go into details of each phase of JNDiff. The discussion is organized in two parts, following the overall structure: *JNDiff Core* and *JNDiff Refinement*.

4.1. JNDiff Core: VTrees and NLCSS

The core of JNDiff lies in two preliminary phases, respectively called *VTree Linearization* and *Partitioning*, that are mandatory and independent from the set of changes we need to detect. They build data structures and clusters that will be used afterward to detect changes.

4.1.1. VTree Linearization For each input document, JNDiff firstly creates a data structure, called *VTree*, which makes it easy and fast to identify and compare documents' elements and subtrees. Basically a *VTree* is an array of records built with a *pre-order depth-first* visit of a document[†]. Each record represents a node and contains: a hash-value which identifies that node (and its attributes), a

[†]The name in fact stands for *Visited Tree*.

hash-value which identifies the whole subtree rooted in that node (derived from the hash-values of its children and itself) and a pointer to that node in the document, and other application-dependent data not described here, due to space limits.

Hash fingerprints proved to be useful to improve the efficiency of diff algorithms. XyDiff[3] computes hash values of nodes by combining the node's content and the children signatures. These values are used to identify uniquely the entire subtree rooted at a given node and to find matches by simply comparing integers. DocTreeDiff uses fingerprints extensively[30]. The basic idea is to also compact information about the context of an edit action (i.e. the position in the tree and the 'close' nodes involved in that change) by using hash functions and to use those values to detect conflict when merging multiple versions. When hashing a node DocTreeDiff only uses the name and attributes of that node, differently from XyDiff that also takes information from (the signatures of) its children. JNDiff uses the same recursive approach of XyDiff while the core hash functions are based on CanonicalXML[31] and MD5[32] as in DocTreeDiff.

The *VTree* structures play a crucial role for JNDiff. The algorithm in fact works on these arrays in order to detect relationships among nodes that are later translated into edit actions on the tree. Note also that a subtree corresponds to a continuous interval of a VTree, because of the pre-order depth-first visit. That makes it possible to match whole subtrees by only matching two integer fingerprints (in *constant time*) and to skip several comparisons while processing the trees. Finally note that the construction of a VTree is linear on the number of nodes, since JNDiff basically visits twice a tree and properly generates meaningful hash-values.

4.1.2. Partitioning The *Partitioning* phase consists of finding unchanged parts of the two documents, and is the core of the algorithm. The search of matching subtrees is implemented as a variant of the longest common substring (LCSS) problem on the corresponding VTrees. Since contiguous items in a Vtree correspond to subtrees in the input documents, in fact, relations between trees/subtrees can be directly translated into relations between strings/substrings, and vice versa.

The LCSS problem consists of finding the longest *substring* common to two input strings. The LCSS problem is a special case of the longest common subsequence (LCS) problem, that consists of finding the longest *subsequence* between two strings[33]. There is an important difference between substrings and subsequences: a substring is a *contiguous* part of the input string, while a subsequence is not. Algorithms for LCS have been used by XML diff tools to detect matching subtrees, for instance by DocTreeDiff[5]. DocTreeDiff uses the Myer's solution that works in $O(N \times D)$, where N is the sum of the lengths of the two input sequences and D is the size of the minimum edit script between them [26].

JNDiff uses a different approach that is less efficient but produces a more precise output for the final users. Roughly speaking, it consists of searching the **largest pieces of content** shared by two input trees and taking them **in the order they appear**. The search is implemented on top of LCSS. Notice that a LCS-based approach identifies a longer subsequence of smaller pieces of content, while LCSS finds larger common fragments. These fragments are useful to detect common edit operations especially on text-centric documents, that are usually modified by leaving large unchanged parts.

We call our approach NLCSS[‡]. To give a formal definition of NLCSS, we adopt an incremental approach.

Definition 1
Given two strings S_1 and S_2, we indicate with $O\text{-}LCSS(S_1, S_2)$ the set of the longest common substrings between S_1 and S_2, in decreasing order according to their length. Formally, denoting with OL_i the i-*th* element of the set, it holds:

$$\forall\, i: \; length(OL_i) \geq length(OL_{i+1})$$

[‡]It is a variant of LCSS and the starting 'N' stands for 'Natural' as in JNDiff.

The set $O\text{-}LCSS$ is useful to select the substrings (VTrees) corresponding to the largest unmodified parts of the documents. When selecting the longest substrings, in fact, we want to consider only those that do not overlap as they represent disjoint parts of the trees.

Let us define some auxiliary functions to model such condition. Given a substring s of the string S, the function $spos(S, s)$ returns the position in S of the left edge of s, while the function $epos(S, s)$ returns the position in S of the right edge of s. The function $intersect()$ exploits $spos()$ and $epos()$ to check if two substrings of the same string overlaps.

Definition 2
Given two substrings s_1 and s_2 of the string S the function insersect() is defined as follows:

$$intersect(s_1, s_2, S) = \begin{cases} true & if\ (spos(s_1, S) < spos(s_2, S) \wedge epos(s_1, S) \geq spos(s_2, S)) \vee \\ & (spos(s_2, S) < spos(s_1, S) \wedge epos(s_2, S) \geq spos(s_1, S)) \vee \\ & spos(s_1, S) = spos(s_2, S) \\ & \\ false & otherwise \end{cases}$$

This function allows to identify the longest common disjoint subtrees, by exploiting the fact that the elements in $O\text{-}LCSS$ are in decreasing order.

Definition 3
We indicate with $D\text{-}LCSS(S_1, S_2)$ the subset of $O\text{-}LCSS(S_1, S_2)$, defined as follows:

$$OL_i \in D\text{-}LCSS(S_1, S_2) \Leftrightarrow \forall\, i, j : j < i \rightarrow\ \neg intersect(OL_i, OL_j, S_1) \wedge \neg intersect(OL_i, OL_j, S_2)$$

where OL_i denotes the i-th element in $O\text{-}LCSS(S_1, S_2)$

The definition needs to be further refined in order to filter the pairs of subtrees that are in the same relative order in both the input documents. Moved subtrees, in fact, do not have to be included in NLCSS. We introduce a second function that takes as input two non-overlapping substrings and verifies if the first one occurs before the second one.

Definition 4
Given two non-overlapping substrings s_1 and s_2 of the string S the function precedes() is defined as follows:

$$precedes(s_1, s_2, S) = \begin{cases} true & if\ epos(s_1, S) < spos(s_2, S) \\ & \\ false & otherwise \end{cases}$$

The function applied to the VTree structures checks the relative order of common subtrees. The definition of $N\text{-}LCSS(S_1, S_2)$ is eventually refined.

Definition 5
We indicate with $N\text{-}LCSS(S_1, S_2)$ the subset of $D\text{-}LCSS(S_1, S_2)$, defined as follows.

$$DL_i \in N\text{-}LCSS(S_1, S_2) \Leftrightarrow \forall\, i, j : i \neq j \rightarrow\ precedes(DL_i, DL_j, S_1) = precedes(DL_i, DL_j, S_2)$$

where DL_i denotes the i-th element in $D\text{-}LCSS(S_1, S_2)$

Let us discuss a simple example to explain NLCSS. Consider strings S_1 and S_2 :

$S_1 = NEW\underline{\mathbf{DATA}}\mathbf{SET}$
$S_2 = \mathbf{DA}XX\mathbf{TA}R\underline{DATA}AND\mathbf{SET}Q$

Their computed LCS (Myer's algorithm), $O\text{-}LCSS$, $D\text{-}LCSS$ and $N\text{-}LCSS$ are shown below.

$LCS(S_1, S_2) =$ **DA, TA, SET**

$O\text{-}LCSS(S_1, S_2) =$ **DATA, DAT, ATA, SET, DA, AT, TA, SE, . . . , N**

$D\text{-}LCSS(S_1, S_2) =$ **DATA, SET, N**
$N\text{-}LCSS(S_1, S_2) =$ **DATA, SET**

The Myer's algorithm for LCS detects the initial substring 'DA' as a common part between the two strings, together with the strings 'TA' and 'SET'. This is correct but the detection of the longer string 'DATA' (underlined in the example) would have been more clear and intuitive. The LCS algorithm misses it since it obtains the longest subsequence (7 characters) with a greedy approach. NLCSS gives more relevance to longer strings, keeping their relative order. Notice in fact that the string 'N' is not included in the final result since its position relative to 'DATA' and 'SET' changed. Similarly, applying the NLCSS approach to VTrees implies detecting clearer and more intuitive edit actions.

In a sense, our approach is orthogonal to DocTreeDiff that is based on LCS. While JNDiff matches the largest common parts of the trees and then refines the result, DocTreeDiff matches much more 'fragmented' elements and then glues them together.

The implementation of an algorithm to find NLCSS without any optimization is trivial and executed in quadratic time. The algorithm can adopt a recursive approach. It takes as input two sequences A and B and finds the longest $M = LCSS(A, B)$. Then computes the subsequences L_A and R_A that respectively precedes and follows M in A, and the subsequences L_B and R_B that respectively precedes and follows M in B. The algorithm is then applied to compare L_A with L_B and R_A with R_B, and goes on recursively.

An important aspect of this approach is that the larger the $LCSS$s sequences are, the less recursive invocations are required. This means that **similar sequences** are compared with good performance. For this reason, this solution is particularly suitable for diff-ing text-centric documents. Two versions of such documents, in fact, usually have a lot of common content.

```
function OptimizedFindNLCSS($A, B$)
    $max \leftarrow 0$;
    for $i \leftarrow 0, length(A)$ do
        $id_{class} \leftarrow$ getIdClass($A[i]$)            ▷ Identify the current element cluster

        $C_B \leftarrow$ getElementsFromClass($id_{class}, B$)

        for $j \leftarrow 0, |C_B|$ do                        ▷ Search only the relevant cluster
            $explorer \leftarrow 0$;
            while $A[i + explorer] = B[position(C_B[j]) + explorer]$ do

                $explorer \leftarrow explorer +$ nodesInTree($A[i]$)

                if $explorer > max$ then
                    $max \leftarrow explorer$
                    $E_A \leftarrow [i, i + explorer]$
                    $E_B \leftarrow [j, j + explorer]$
                end if
            end while
        end for
    end for
    return $E_A$
end function
```

Figure 4. An optimized algorithm to find the LCSS between two sequences A and B

JNDiff includes an optimized version named OptimizedFindNLCSS(A,B) and shown in Figure 4. The notation $A[i]$ indicates the i-th element of A, while $A[i, j]$ indicates the subsequence of A from the element in position i to the element in position j. The function NodesInTree($A[i]$)

returns the number of elements in the subtree rooted in $A[i]$. Note also that the function returns E_A but it could have alternatively returned E_B as they are equal.

Basically, the algorithm scans all elements in A (line 3) and searches matching elements in B. Instead of comparing each element of A against each element of B, the algorithm skips some comparisons by exploiting the VTree structures. Two strategies are implemented:

- Dividing elements in classes, where each class contains those elements with similar prefixes and local names. The elements belonging to the same class are candidates to match each other, while they surely do not match those in other classes. Note that the information about each element – that is used to assess their similarity and to build classes – is available from their corresponding hash signatures, created in the previous VTree Linearization phase.
- Skipping comparisons between elements that belong to matching subtrees (subtrees with the same signature). In a VTree, the fact that the hash signatures of two elements match implies that all elements in their subtrees match. Formally: $S_1[i] = S_2[j] \rightarrow S_1[i+k] = S_2[j+k]$, with $K = 1 \ldots N$ where N is the number of nodes rooted in the subtree associated to i. Thus, the algorithm can skip a lot of comparisons on contiguous elements, if some of their ancestors already matched.

When searching a match for a given element in A, the algorithm only compares that element with the ones in B that belong to the same class (lines 4, 6, 8). The algorithm tries to identify the longest match (line 14 and 15) skipping comparisons between contiguous already-matched elements (line 12).

The identification of the classes is very important. In fact, the lesser elements belong to each class, the faster the algorithm is (as the number of iterations in line 8 is reduced). It is not difficult to substitute the current strategy of JNDiff with a different one, that could be tailored for specific contexts and elements' vocabularies. Finally note that equal sequences are compared in constant time, thanks to the VTree structures.

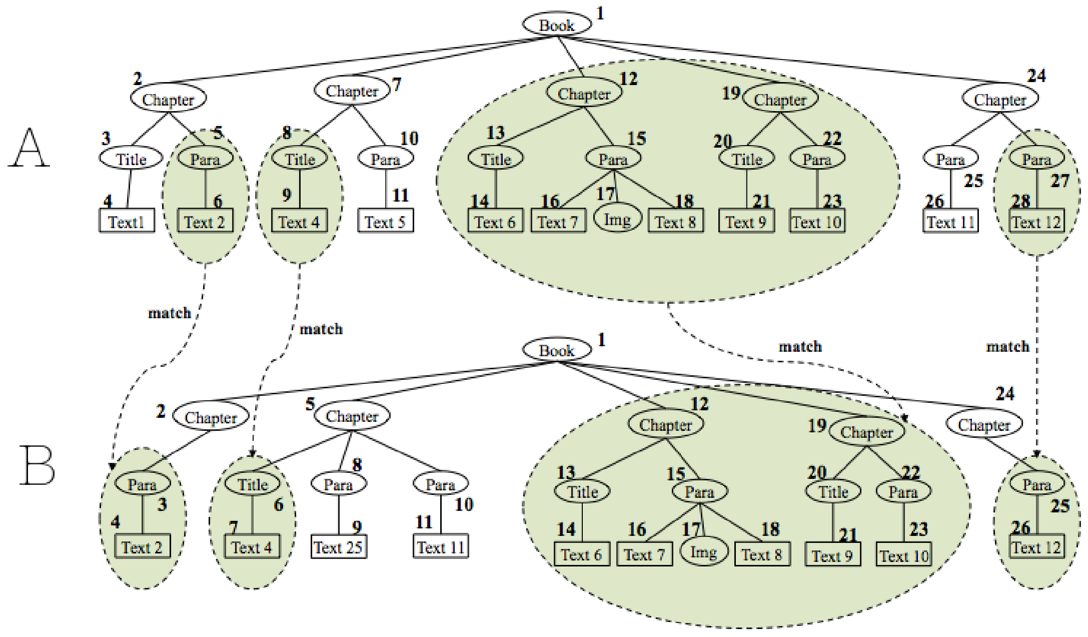


Figure 5. Phase 1 matches the largest common subtrees through the NLCSS approach. The matching parts between the two input documents A and B are highlighted in green and connected by dashed lines.

At the end of the partitioning phase, unchanged parts are identified and connected by a *matching* relation. The *edit script* is only composed by a set of Insertions/Deletions of the residual nodes: nodes are then considered *inserted*, if they only are in the second document (and VTree), or *deleted*, if they only are the first one. Note also that the list of insertions/deletions contains both subtrees and nodes scattered along paths to the root. Elements' wraps and unwraps, as those described in Section 3, are then detected at this stage; these are eventually refined in the 'match expansion phase' described later.

Figure 5 shows a possible output of this phase when diff-ing two XML-encoded books divided in chapters, titles and paragraphs. The two input documents are named A (the original one) and B (the modified version). The matching (unchanged) parts are connected by dashed lines and highlighted in green. Note that we use a DOM representation to be clearer, though comparisons and associations are actually performed on linearized structures. Finally, note also that nodes are numbered: these labels will be useful to explain the following phases.

4.2. *JNDiff Refinement*

The algorithm finally includes some optional phases, collectively named JNDiff Refinement, which refine insertions and deletions as higher-level changes. The current implementation passes through: *Text Updates Detection*, *Move Detection* and *Matches Expansion*.

4.2.1. Text Updates Detection The text update detection is the first optional step of JNDiff. Figure 6 shows the output of this phase on the sample documents A and B. After partitioning, the text change on node 11 (from 'Text 5' to 'Text 25') had not been detected yet. This step establishes a new relation between the subtrees rooted in nodes A10 (node 10 in document A) and B8 (node 8 in document B). In the picture, these subtrees are connected and highlighted in purple. The *edit script* is in fact refined by substituting the corresponding pair INS/DEL (of text nodes) with an update action.

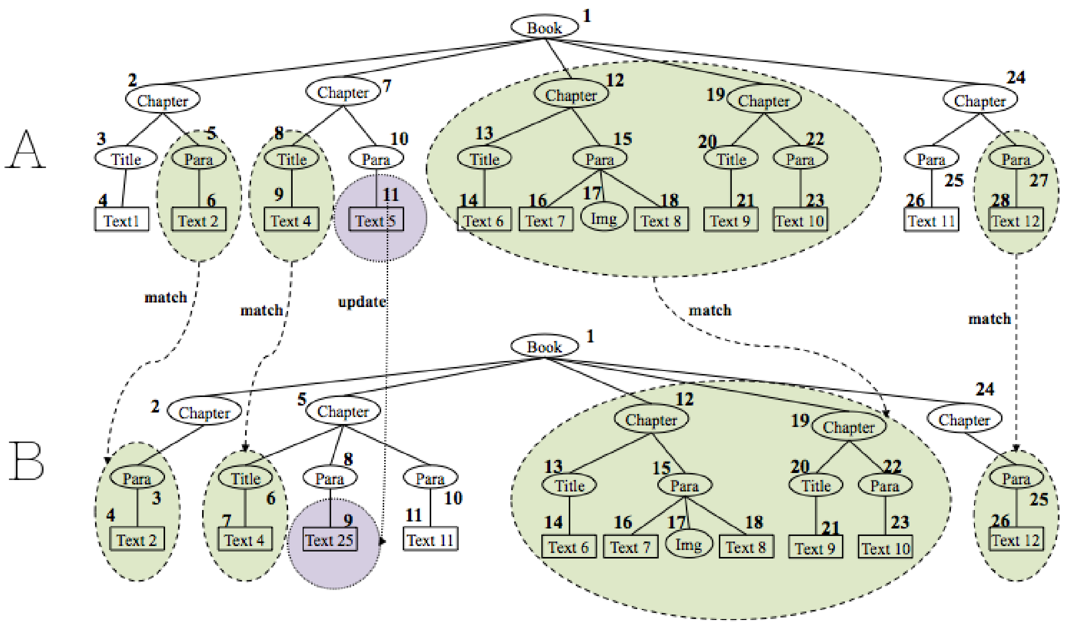


Figure 6. Phase 2 detects updated nodes within corresponding unmatched partitions. These nodes are now connected and highlighted in purple.

Two steps are performed. Potential updated texts are first searched according to the *principle of locality*. It states that two text nodes should be considered 'updated' it they lie between two subtrees matched earlier (actually, if their signatures lay between two previously-matched fragments in both the VTrees). That encodes the property that an updated text fragment is probably in the same part of the document.

The candidates are later checked against a threshold to verify their *similarity*. The formula currently used by JNDiff is the following.

$$f(T_1, T_2) = K \times (MAX_{(T_1,T_2)} + min_{(T_1,T_2)})$$

$$MAX_{(T_1,T_2)} = MAX(\frac{length(T_c)}{length(T_1)}, \frac{length(T_c)}{length(T_2)})$$

$$min_{(T_1,T_2)} = MIN(\frac{length(T_c)}{length(T_1)}, \frac{length(T_c)}{length(T_2)})$$

The value K is a percentage value, $length(x)$ indicates the number of characters in a string and T_c indicates the NLCSS (as in partitioning) between T_1 and T_2. The idea is to measure the amount of common text between the two strings, following the assumption that updated nodes do not change their position and do slightly change their content. The parametrization of the similarity threshold is another key aspect to be remarked. The current implementation uses the value $K = 50$. This means that text nodes are considered updated if they share more than half of their total content. This value has been set by best-practice from a preliminary study on a large collection of text-centric documents (see Section 5 for more details).

Finally, note that such an analysis increases the precision of the algorithm on text fragment. For instance, makes it possible to identify with great accuracy insertions and deletions of small pieces of content, or text wrap operations as those described in Section 3. The elements that contain text, in fact, are not considered as other subtrees. Their textual content is treated in a special way being scanned and compared locally. That makes JNDiff more effective on text-centric resources and closer to change-tracking tools and word processors.

4.2.2. Moves Detection The detection of moves is a further optional step of JNDiff. Figure 7 shows its execution on the sample documents A and B.

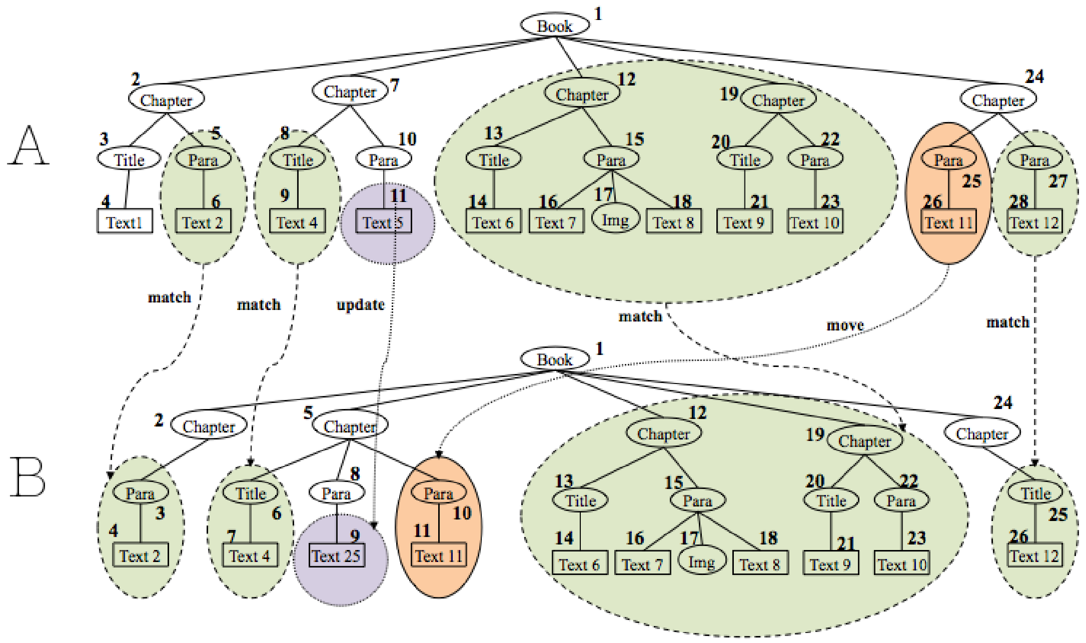


Figure 7. Phase 3 detects moves by scanning nodes with the same signatures in different partitions. Moved nodes are now connected and highlighted in orange.

At this stage, subtrees rooted in A25 and B10 are not matched, because they belong to different partitions, and do not belong to the LCSS found in the partitioning phase. They are then considered as unrelated deletion/insertion of subtrees. JNDiff matches these subtrees and establishes a new type of relation (i.e. *moves*) between them. In the picture, they are now connected and highlighted in orange. The *edit script* is further refined by substituting the corresponding pair Insertion/Deletions with a move action.

JNDiff measures the distance between the original position of a subtree and the position of any candidate to be marked as 'moved' (i.e. any identical subtree but in a different position). Then it classifies as 'moved' only those subtrees whose distance is under a given threshold, according to the assumption that in most cases authors move content within a limited space. The distance is measured by counting the previously-matched partitions and nodes between them, and the threshold is an integer, normalized to the size of the trees.

4.2.3. Matches Expansion The 'matches expansion' phase propagates changes *bottom-up*, in order to refine the output and to add operations on attributes. The approach is similar to the XyDiff propagation[3] but it produces different results as the input structures are very different, due to the NLCSS-based partitioning phase. This step is again optional but highly recommended to have better results.

Intuitively, JNDiff 'goes up' from leaves to the root by scanning intermediate elements in order to refine the interpretation of some insertions and deletions. It may happen in fact that two VTree elements have different signatures and are recognized as inserted/deleted even if they are actually unchanged. The reason is that their VTree hash-values have changed because something have changed in their descendants (for instance, the *book* or *chapter* elements of the sample) rather than in the elements themselves. Remind that VTree signatures are computed by also taking as input the signatures of the children nodes. JNDiff removes those false positives among insertions/deletions and polishes the edit script.

Alternatively the two VTree elements may have different signatures because of changes in their attributes. In that case the signatures of the children are all equal but these attributes have been added/removed/updated. The output is then refined to capture these actions too. The final result for the sample documents A and B, in fact, will consider the *para* elements A10 and B8 as matching, will mark as unchanged the root and all elements *chapter*, and will eventually leave the subtree rooted in A3 as deleted.

4.3. Computational Complexity

Defining the complexity of JNDiff is not straightforward, as it is strongly dependent on the number and order of the phases that are executed. For this reason we will discuss the costs of each phase separately. In particular, we will express complexity in terms of four parameters: n (number of nodes of document A), m (number of nodes of document B), p (number of leaves of document A), q (number of leaves of document B).

The VTree linearization (Section 4.1.1) is realized by pre-visiting the DOM trees, so it costs $O(n+m)$. The partitioning-phase (Section 4.1.2) consists of finding the matching parts in the linearized VTrees. JNDiff uses the NLCSS-based approach in order to capture the largest subtrees, especially for text-centric documents. It finds a NLCSS in $O(n \times m)$ but it has a $\Omega(1)$ lower bound. The internal structure of VTrees, in fact, makes it easy and fast to compare identical subtrees.

Thus, JNDiff works very well on documents with few differences. We have found in fact that in text-centric documents changes tend to concentrate in a few specific parts of it, so that we find more unmodified than modified subtrees.

Similar considerations can be applied to the optional phases of JNDiff. For instance, the complexity of text-updates-detection(Section 4.2.1) is $O(p \times q) \times f$, where f is the cost of the function that calculates the similarity between text nodes. Given two text leaves t_1 and t_2, the function $f(t_1, t_2)$ costs $O(length(t_1) \times length(t_2))$ because it uses our NLCSS algorithm although it can be sped up by using Myer's.

Similary, a move-detection phase (Section 4.2.2) costs $O(n \times m)$ when all nodes need to be scanned. In the same way, the complexity of the match-expansion phase (Section 4.2.3) is $O(min(p, q) \times log(n))$ since JNDiff has to scan the tree for $log(n)$ nodes.

The total complexity of JNDiff, in the worst case, is then $O(n \times m)$, which is high in comparison with other more efficient solutions, as discussed in Section 2, such as, for instance, XyDiff (whose complexity is $O(n \times log(n))$) and faxma ($O(n)$ in the average case and $O(n^2)$ in the worst one). However these algorithms were explicitly designed to address efficiency possibly to the detriment of quality. Notice also that JNDiff has a quadratic complexity in the worst case, as well as the very fast faxma algorithm. In the average cases its complexity ends up being much lower.

In particular, JNDiff achieves good results on text-centric documents. The nature of such documents - or better, the nature of changes on such documents - make computational costs acceptable in practical applications. Considering that each JNDiff phase works only on those nodes that are not previously matched and that most nodes are matched after partitioning, the computational costs decrease progressively.

5. EXPERIMENTAL EVALUATION OF JNDIFF

JNDiff is currently implemented in Java. The tool is open-source and freely available at `http://sourceforge.net/projects/jndiff/`. It takes two XML files as input and produces a delta, in XML format too. The syntactical details of the delta, that could be easily changed and customised, are not relevant for this paper.

The delta generated by JNDiff lists all changes to be applied back to obtain the original file from the newer one. In fact, JNDiff is coupled with JNMerge, a further Java open-source tool that embeds namespace-specific elements and attributes in the output document in order to highlight changes. These markup items can be easily stripped off in order to obtain the original document or can be formatted to show differences to the final users.

The quality of the output for human readers is the most relevant aspect in the design of JNDiff. Thus, our evaluation starts from this parameter and then analyses quantitative measures on size and performance, that have been widely used in the literature.

An important aspect of JNDiff is worth highlighting at this point. The tool relies on a DOM parser, that is invoked in the *VTree linearisation* phase to build the internal data structures. As a consequence, the execution is limited by the memory of the machine running JNDiff (i.e. the capability of loading the whole DOM). We performed some preliminary experiments to assess the applicability of JNDiff, concluding that the current prototype cannot be used on huge files ($\sim$500Mb) but still works well on large files ($\sim$50 Mb). On the other hand, we think this is an acceptable limitation considering that the tool is designed for text-centric documents and these documents are not very large, since they do not contain binaries but only plain text. Furthermore, very large XML are mostly generated by automatic tools –for instance, from database dumps– while we focus on documents created and edited by the users. These documents are expected to loaded in editors, which often have the same limitations and slow performance on very large files.

5.1. *Quality of the output*

Comparing the output of JNDiff to that of other XML diff algorithms is difficult. Not only because each algorithm uses its own format to express differences but because each of them detects its own set of changes. Even more important, these algorithms do not support natively and uniformly the set of changes we are primarily interested in, as outlined in Section 3.

Thus, our need is to find a quality measure that enables the comparison of heterogeneous algorithms and that takes high-level changes into account, even if not natively supported by the algorithms being compared.

The quality has been often associated to the idea of ‘minimization of the edit script’. As summarized by Cobena[3], in the majority of the algorithms “*quality is described by some minimality criteria. [...] Minimality is important because it captures to some extent the semantics that a human would give when presented with the two versions*”. However, existing algorithms are usually evaluated by measuring the size of the delta or, with more accuracy, the number of edit operations. A smaller number of edits might be a good indicator of quality but the internal structure of changes should also be taken into account.

Another issue is that the evaluation is performed on the delta as a whole. Looking at changes as separate units gives us a clearer picture of the accuracy of the deltas and of the ability of each algorithm to detect effective changes. This cannot be done in a fully automatic way, as human interpretation is needed to identify the expected changes, but an hybrid approach is feasible.

In fact we used a semi-automatic approach. The basic idea is to examine how close was the delta generated by each algorithm to the *ideal* delta, i.e. to the delta expressed in terms of high-level changes meaningful for the final users. For our purposes, we assume that humans would basically agree on the interpretation of an edit history, so that differences in human interpretation of the edit history of a document are minimal.

The overall process consists of three steps:

1. creating the *gold standard* by comparing visually and structurally pairs of document versions. This operation is manual; it produces ad edit script, in terms of high-level changes as discussed in Section 3.
2. identifying *clusters of changes* in the delta produced by each algorithm that correspond to each change in the gold standard; this step is manual as well: we need to inspect the deltas since the structure of the output is different for all algorithms;
3. assigning a *score* to each cluster and compare corresponding clusters of different algorithms.

Note that the manual creation of the gold standard and the manual inspection of the output of each algorithm guarantees a more accurate comparison, since it goes into the details of each detected change, represented by each cluster. The question is now on how to score the quality of each cluster.

Besides the number of edits, we believe it is interesting to study how much space is used within each cluster. In fact, we propose to also count the number of nodes reported within each of them. The idea is to give a higher score to clusters with fewer nodes: that would give penalties to vaguer edit scripts and would award those scripts with less false positives. As discussed in Section 3, high-level changes tend to contain the modified elements and a few more, with a low degree of redundancy. If the same change is expressed as combination of insertions and deletions, on the other hand, many more nodes are involved. That is the case, for instance, of the wrap/unwrap or move and split operations. Note also that this quantity gets rid of the danger of giving a high score to the trivial delta (the one that is composed of exactly two operations, a DEL of all the old content of the document and an INS of all the new content), since a large number of nodes would be involved in this case that in reality were never ever touched. The mere count of the number of edits, on the other hand, would (wrongly) award this type of delta.

The precision of the edits—especially on text fragments—is a further dimension to consider. As previously discussed, some operations are unable to identify exactly the boundaries of the text fragments that are involved, and thus widen them to include, for instance, whole text nodes. For instance, an algorithm identifying exactly the insertion/deletion of some words in a paragraph must get a higher score than an algorithm that detects the deletion of the whole paragraph and the insertion of a new one with much of its text unexplainably identical to text of the old version. Thus, the minimality of the length of text nodes should also affect the overall score.

In fact we award *minimality* on three dimensions: *number of edits*, *number of affected nodes*, *number of characters in the affected text nodes*. The score of the *i*-th cluster in a delta Δ is the inverse of the weighted sum of the three parameters:

$$f(\Delta, i) = (w_e \times m_e \times EDITS_i + w_n \times m_n \times NODES_i + w_c \times m_c \times CHARS_i)^{-1}$$

Given two deltas Δ_A and Δ_B, we consider the quality of Δ_A on the cluster *i*-th higher than that of Δ_B if:

$$f(\Delta_A, i) > f(\Delta_B, i)$$

The weights w_e, w_n and w_c are needed to normalize the scores on a]0,1] range, assigning score 1 to clusters identical to the corresponding operation in the gold standard and score near to 0 to the trivial algorithm that describes any operation as the DEL of the existing structure and the INS of the new structure. After running some training tests, we calibrated these weights as follows: $w_e = 0.2$, $w_n = 0.1$ and $w_c = 0.7$.

The coefficients m_e, m_n and m_c are also needed to provide a balance among the weights. In fact, the number of characters in text nodes is on average in textual documents much higher than the number of edit actions or of affected nodes, and needs to be normalized. The contribution of each parameter, therefore, have to be calculated for the documents under evaluation. This is expected to be done in a pre-processing step by measuring the ratio between text lengths, number of nodes (elements and texts) in the input documents and in the deltas.

5.1.1. Experimental results We compared JNDiff to XyDiff[3] and faxma[21], two of the most used XML diff algorithms. Originally we also included diffxml[16], an implementation of the algorithm for *meaningful changes* proposed by Chawathe et al.[4], but after a few tests we decided to drop it since it produced incredibly verbose output from which it was almost impossible to identify the clusters.

The test set is organized as follows:

- Documents 1 and 2 are taken from the evaluation of DocTreeDiff[7]: the first one is a one-page letter while the second is a bibliography of about 15 pages. They are both in the XML format used by Open Office 2.x and available on the web[§].
- Documents 3 and 4 are XML-encoded legislative acts and bills. They are highly structured in articles, clauses and paragraphs and follow precise rules to encode textual content.
- Document 5 is an XHTML specification of a web protocol, used for a schoolwork project. It is structured in sections and contains a lot of internal references and code snippets.

In most cases the gold standard was available since the modifications had been recorded during the editing process itself. When not available, changes were identified by manually comparing the versions of the documents.

Table I shows some statistics about each version of each document. In particular, it reports the length in bytes, the number of nodes (elements, text nodes and attributes) and the number of characters in the text nodes.

Table I. Statistics on the data set used for evaluating quality. Each document is described by one row. The row contains a pair of columns for each parameter, corresponding to the original and modified version.

		# BYTES		# NODES		# ELEMS		# TEXTS		# ATTRS		# CHARS (TEXTS)	
1	*Letter*	3342	4133	71	103	27	36	6	10	38	57	372	320
2	*Biblio*	67235	67764	1893	1924	630	638	189	201	1074	1085	16027	16071
3	*DL1184*	6412	6922	101	105	49	54	34	34	18	17	5378	5901
4	*DL2221*	8910	9101	192	194	98	98	56	58	38	38	6004	6067
5	*Protocol*	24154	29020	637	707	234	270	335	355	68	82	17897	20135

We evaluated the output of the algorithms on each document separately, following the previously-described process and instantiating the score formula f as follows:

$$f(\Delta, i) = (0.2 \times \mathit{EDITS}_i + 0.1 \times \mathit{NODES}_i + 0.007 \times \mathit{CHARS}_i)^{-1}$$

The final coefficients were obtained by measuring the average number of elements, texts and nodes on all documents (shown in Table I) and deltas.

Table II shows as example the full evaluation data for Document 4 (DL2221). The gold standard contained 14 changes. Each row corresponds to one of those changes and shows the score of the three corresponding clusters, one for each algorithm. The table shows also the input values

[§]The original files are at http://www.icsi.berkeley.edu/~jan/projects/OOVersionControl/. We extracted their XML main content.

calculated for each cluster: number of edits, number of affected nodes, number of characters in the affected text nodes. The last row reports the total score, as the sum of the scores of each cluster.

Table II. Detailed evaluation of the algorithms on document DL 2221. Each row corresponds to a change in the gold standard (i.e. a cluster). Columns are grouped per algorithm and indicate the score on that cluster and the three values used to calculate it.

	JNDiff				Faxma				XyDiff			
	EDITS	NODES	CHARS	**SCORE**	EDITS	NODES	CHARS	**SCORE**	EDITS	NODES	CHARS	**SCORE**
Change 1	2	1	12	**1,712**	1	1	166	**0,794**	1	137	1	**0,684**
Change 2	2	1	12	**1,712**	1	1	245	**0,496**	1	1	164	**0,691**
Change 3	1	1	38	**1,767**	1	1	447	**0,292**	1	1	115	**0,905**
Change 4	1	1	12	**2,604**	1	1	198	**0,593**	1	1	198	**0,593**
Change 5	2	1	64	**1,055**	1	1	658	**0,204**	1	1	261	**0,47**
Change 6	4	1	21	**0,955**	1	1	507	**0,26**	2	2	506	**0,241**
Change 7	4	1	53	**0,787**	1	1	475	**0,276**	2	2	474	**0,255**
Change 8	1	1	6	**2,924**	1	1	116	**0,899**	1	1	103	**0,979**
Change 9	1	2	128	**0,772**	1	1	130	**0,826**	3	3	130	**0,552**
Change 10	6	3	26	**0,595**	3	3	103	**0,617**	1	5	103	**0,704**
Change 11	2	2	0	**1,667**	1	1	0	**3,333**	1	0	0	**5**
Change 12	1	1	4	**3,049**	2	2	206	**0,49**	3	3	209	**0,423**
Change 13	1	5	138	**0,6**	3	3	141	**0,53**	4	9	285	**0,271**
Change 14	1	0	0	**5**	1	1	0	**3,333**	3	0	0	**1,667**
TOTAL				**25,199**				**12,833**				**13,545**

Note that we identified the clusters manually and calculated the score of each of them automatically. Faxma made comparisons a bit more complex. The algorithm, in fact, does not produce a sequence of edit actions as other algorithms but generates a set of pointers to the original document interspersed with the nodes to be inserted (see Section 2 for more details). Furthermore, the output does not contain the deleted content, as it corresponds to the lack of specification of a pointer. Deleted fragments, therefore, were handled by processing the original document too.

JNDiff scored higher in most of the clusters. The reason is that most changes were text modifications or wrap/unrap of elements within mixed content-models. While JDNiff is specialized for working on text-centric documents, XyDiff and faxma show some limitations in that context. In fact, XyDiff identifies a lot of changes as insertions/deletions of large text fragments and elements, while faxma rebuilds changes as a combination of a lot of nested elements and small text fragments. The quality of all algorithms was comparable on a few clusters, in presence of combined changes. Finally, JNDiff obtained a much lower score when detecting the creation of a new comma within an article, whose other commas had been modified (cluster 11). That happened since JNDiff pivoted around some characters that were found in both versions. In that case, the text should have been given less importance considering the structural modification of the whole article.

We performed a similar analysis for all other documents. Due to space limits, we do not go into the details of each test, rather we show an overview of the algorithms' scores. The full experimental results and input documents are available online[¶]. Figure 8 summarises the overall results. For each document (in the X axis) the plot shows three bars. Each of them indicates the sum of the scores of the clusters produced by each algorithm. Note also that the scores on different documents are not comparable, since they depend on the length of the documents and the number of clusters.

JNDiff obtained the highest score in all tests. For the Document 1 (Letter) scores were rather similar for all algorithms. The deltas in fact contained a lot of structural changes like insertions/deletions of entire subtrees that all tools detect in very similar ways. In Document 2 (Biblio) the results of JNDiff were very good because of the large number of mixed content models.

[¶]http://diff.cs.unibo.it/jndiff/tests/

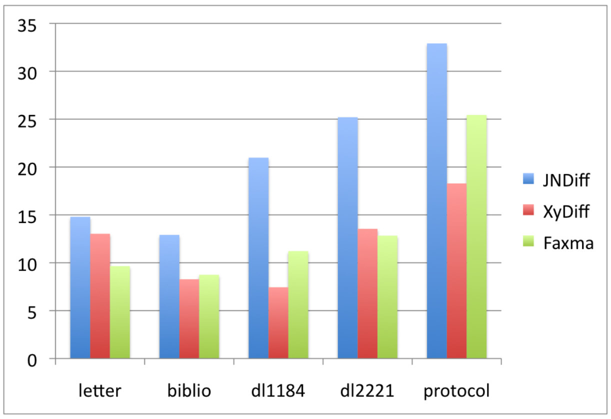


Figure 8. Comparing the quality of JNDiff and other algorithms on all documents. For each document (in the X axis) the plot shows three bars. Each of them indicates the sum of the scores of the clusters produced by each algorithm. In case of Document 4 (DL2221), for instance, they correspond to the values in Table II. An higher scores indicates an higher overall quality on that document.

In particular, it detected exactly the deletion of small text fragments in nested in-lines where other algorithms detected more tangled insertions/deletions of elements and text fragments.

Even in Document 3 (DL1184) the presence of a lot of text modifications made JNDiff get the highest score. No algorithm proved to be absolutely better than the others when detecting combined changes (for instance, the re-structuring of an article in multiple clauses or the re-numbering of other clauses). The reason is that each algorithm uses its own strategies to approximate those changes and makes some mistakes that end up balancing each other.

Finally, in Document 5 (Protocol) the difference between JNDiff and the other algorithms was less evident. JNDiff obtained an higher because of its higher precision on text fragments but the overall quality was not very high. The nature of the input files heavily affected these results: they are XHTML pages rendered from a source partially written in XHTML and partially in the wiki syntax. That generated a lot of irregular structures and, as a consequence, made changes very difficult to detect. Just to mention one issue: some text blocks are wrapped by `p` elements while others are not. When a modification involves a fragment within those blocks that are not correctly wrapped, the algorithms detect complex re-structuring which should not be detected.

Nonetheless, the overall behavior of JNDiff compared to other algorithms proved to be highly satisfactory. Most limitations derive from well-known editing patterns and classes of changes we are currently working on, such as refactoring of mixed content models, concurrent moves&updates of nodes, modifications of multiple attributes of the same node.

5.2. *Delta size and number of edits*

The amount of detected edit actions is a dimension widely used to compare diff algorithms. To complete our evaluation we conducted some experiments on a synthetic data set in order to measure such parameter, along with performance.

5.2.1. Synthetic dataset The synthetic data set has been generated through a random change simulator that produces new versions of XML documents by modifying elements, subtrees, attributes and text nodes. The change simulator is parametric with respect to the overall percentage of nodes to be changed and the probability of each edit operation to occur. Change simulators are useful to evaluate diff algorithms as they allow designers to control the amount of changes and to perform batch experiments with different settings. The same approach has been used, for instance, to evaluate XyDiff[3], X-diff[34], XANDY[22] and CX-Diff[27]. We implemented our own change simulator customized for text-centric documents, that will be briefly described below.

We have built the data set starting from 60 XML files, encoded in multiple formats: from DocBook to AkomaNTOSO, from IDML to XHTML, from XSL-FO to OpenXML[‖]. In order to give the reader a clearer picture we summarized some statistics about these files in Table III.

Table III. Statistics on the input data set, used to generate the synthetic dataset for the qualitative evaluation. Rows indicate *minimum*, *average* and *maximum* of each dimension over the initial 60 files.

	BYTES	NODES	ELEMS	UNIQUE ELEMS	TEXTS	ATTRS	MAX-PATH	MIN-PATH	AVG-PATH	MAX-WIDTH	CHARS (TEXTS)	TERMS	UNIQUE TERMS
min	126	4	2	2	0	0	3	2	3	1	0	0	0
avg	79858.8	1745.0	598.4	24.1	468.7	677.9	9.4	3.3	6.3	218.4	42219.6	6275.1	1504.3
max	223269	7672	2188	98	2360	7159	24	5	11	680	177495	26530	6426

Notice that there is a great variability in the size of the files and in the amount of XML nodes, elements, attributes and text fragments. As expected, there is a large difference between `elems` and `unique_elems` too. Each language in fact provides a set of objects that authors use many times in the same document, for instance to encode paragraphs or lists.

This initial set of files was processed by the change-simulator in order to generate the actual resources used for our experiments. In particular, we changed different percentages of elements (from 10% to 70%) and produced seven new versions for each input files, for a total amount of 480 files, all available in the experiments web page. Notice that each version is generated directly from the original file. That means, for instance, that there is no connection between the version with 60% and 70% of changed elements. Table IV shows some statistics on the full dataset, including original files and changed ones.

Table IV. Statistics on the synthetic dataset used for the qualitative evaluation. Rows indicate *minimum*, *average* and *maximum* of each dimension over all 480 files.

	BYTES	NODES	ELEMS	UNIQUE ELEMS	TEXTS	ATTRS	MAX-PATH	MIN-PATH	AVG-PATH	MAX-WIDTH	CHARS (TEXTS)	TERMS	UNIQUE TERMS
min	108	2	1	1	0	0	2	2	2	1	0	0	0
avg	71483.9	1513.5	533.7	44.5	433.8	546.0	10.0	3.0	6.2	178.7	38878.8	5708.9	1515.7
max	223275	7672	2208	161	2360	7159	25	6	12.76	680	183343	26530	6426

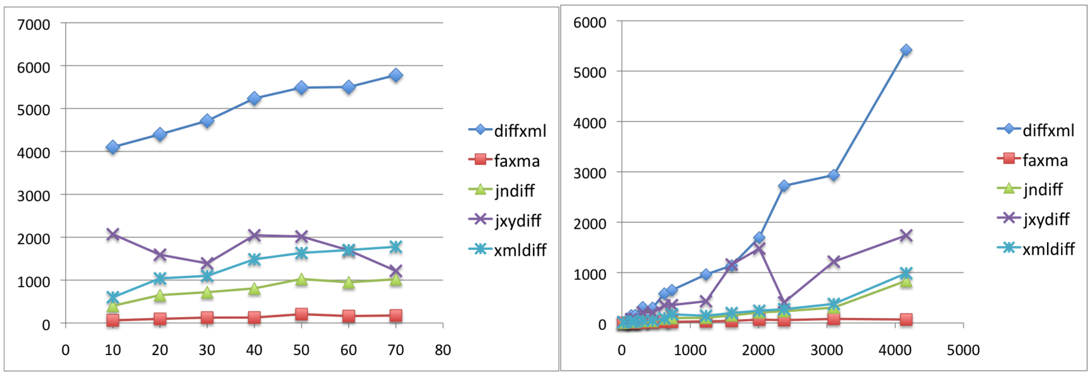
The table helps us to point out some features of the change-simulator, useful to interpret the following experimental data. The first aspect is that changed documents do not seem to be much different from the original ones. Such a behavior depends on two main factors. First of all, the fact that changes are applied randomly but each operation has a different probability to occur. In particular, text deletions and insertions are more frequent (30% against 10% of other operations) since text modifications. Notice also that element deletions might remove large subtrees and reduce drastically the size of the files. The second aspect is that some synthetized edit actions may cancel others, involving the same nodes. Just think about the insertion of an element and the following deletion of the subtree containing that element. As a consequence, the eventual number of changes is less than the expected percentage of input elements and, above all, unmodified nodes are much more than changed ones. Our goal was to simulate editing patterns on text-centric documents: when creating new versions of such documents, in fact, users tend to modify limited parts of them and to work primarily on textual content (as also shown by the experiments of [7]).

5.2.2. Experimental results We run multiple experiments on this synthetized data set executing JNDiff and other algorithms to compare each version (of each file) to the original one. In particular we analyzed XyDiff[3], faxma[21], diffxml[16] and xmldiff[17]. In total, more than two thousand comparisons were performed.

[‖]Full details on the languages, as well as detailed statistics about all the input files are available at http://diff.cs.unibo.it/jndiff/tests/

The first set of experiments compared the amount of edit actions detected by each algorithm. Since each algorithm uses its own syntax - and set of operations - we had to implement some ad-hoc functions to extract and normalize such data. A clarification is necessary about faxma[21] at this stage: as discussed in Section 2 the algorithm does not produce a sequence of edit actions but a set of pointers to the original document and, among them, includes new and changed nodes. Rather than interpreting the faxma internal format and merging the original file and the delta, we decided to only count the number of element and text insertions detected by the algorithm. That is a partial evaluation but gives us a quite clear picture of the relation between faxma and other algorithms.

Apart from faxma, JNDiff produced the minimum edit scripts. Figure 9a shows how the length of the edit script generated by each algorithm changes for documents of about 200K when increasing the change ratio from 10% to 70%. Since different changes may produce very different results, we processed ten files and calculated the average script length. This means, for instance, that the value associated to the change-ratio 10% is the average value of ten edit scripts, each generated by diffing one file to the version labeled 10%. The ten original files have all size of about 200K.



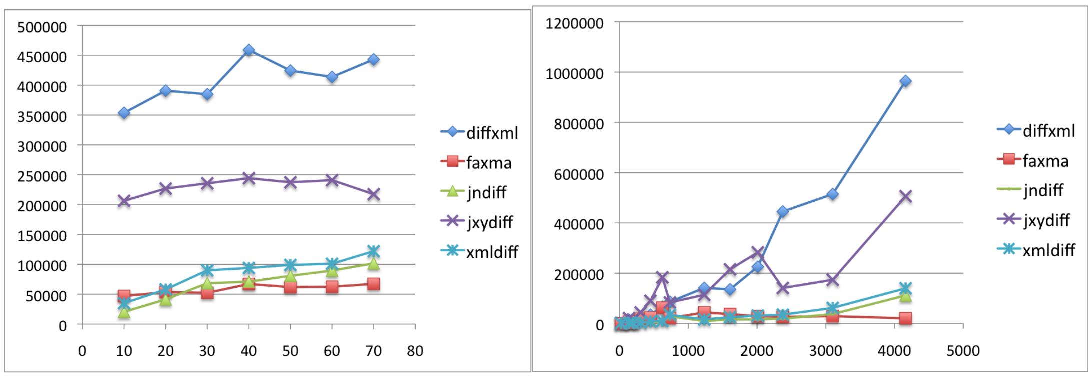
(a) size: 200K — changes: 10-70%   (b) changes: 20% — nodes: 20-4200

Figure 9. Measuring the number of detected operations (shown in the Y axis) when varying the percentage of changes (figure a) and the number of input nodes (figure b). Lower values indicate a better result.

The length of the JNDiff edit script increases regularly when increasing the change-ratio. As expected the edit script produced by XyDiff is not minimized. The algorithm in fact uses a *greedy approach* to increase performance but produces 'completed' deltas that contain redundant information[20]. The irregularity of XyDiff is explained by the internal format of the delta, that stores in different ways the edit operations: the script length depends on the class of applied changes. DiffXML is the most verbose format, while the results of XMLDiff are very good, even if its performance is very low as shown in the following section. Notice that the overall growth of the script length is not very high. That depends on the behaviour of the simulator, that allows edit actions to override and gives more relevance to (small) text modifications.

Figure 9b shows an orthogonal experiment: measuring the variation of the number of detected changes when increasing the size of the input documents and keeping constant the change ratio (20%). We considered the length of the edit scripts generated by diffing each file in the input dataset to the version labeled 20%. In order to increase the significance of the experiment we clustered files in groups (around the number of nodes) and identified each group with the average number of nodes of its members. The trend is in line with the previous one: both JNDiff and XMLDiff produced optimized deltas that increase linearly wrt the number of input nodes, while the script length of XyDiff and diffxml was much higher and increased faster. Faxma results are again very compact.

We also measured the variation of the delta size against the variation of the change ratio and/or the number of input nodes. Figure 10a shows how the change ratio impacts on the delta size for 100K documents. The experiment is similar to the one in Figure 9a: we collected ten files of about 100K, compared them to all new versions and calculated the average script length for each percentage. The



(a) size: 100K — changes: 10-70% (b) changes: 10% — nodes: 20-4200

Figure 10. Measuring the delta size (shown in the Y axis, in bytes) when varying the percentage of changes (figure a) and the number of input nodes (figure b). Lower values indicate a better result.

overall trend does not change. JNDiff, xmldiff and faxma proved to produce compact deltas, while other algorithms generated much larger output (note again the irregular pace of XyDiff).

A slightly different experiment is summarized in Figure 10b: measuring the variation of the delta size when processing files with an increasing number of nodes and a fixed change ratio of 10%. The experiment is similar to the one in Figure 9b: we considered the deltas generated by diffing each file in the input dataset to the version labeled 10%, and clustered them to calculate average values. Similarly to the behavior shown in Figure 9b, all algorithms produce slightly larger deltas when increasing the number of input nodes. While XyDiff and diffxml show an exponential growth, JNDiff and xmldiff deltas increases linearly. The results of faxma are quite similar but they have to be read again considering the internal format of the delta.

It is interesting to note that the trends in Figure 10 are very similar to those observed in Figure 9, even if these experiments were performed on two disjoint set of files and different set of random changes. In fact, both the number of edit actions and the delta size capture the same information: the verbosity of the output format produced by each algorithm. Repeating experiments of different set of files and on different percentages we observed very similar trends**.

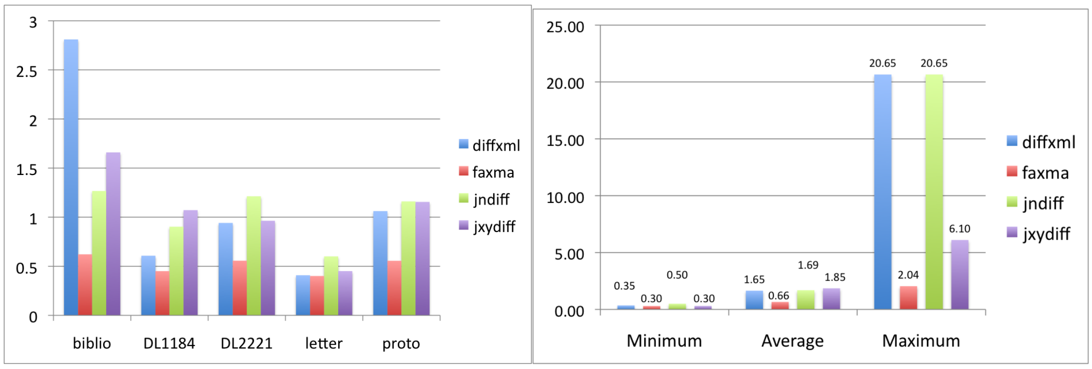
5.3. *Performance*

We finally compared the execution time of JNDiff with all other algorithms for all the experiments discussed so far. We excluded xmldiff[17] after some preliminary tests. These tests highlighted that its performance is very bad on real files, even if the output quality is rather good.

The performance analysis have been carried out on a MacBook Pro with 2.53GHz Intel Core 2 Duo processor and 2GB of RAM. Note also that all tested algorithms are implemented in Java and can be compared directly. There are no issues related to different memory management techniques, garbage collection strategies and other similar details, that would have been very relevant if we also included implementations in C, Python or other languages.

Figure 11a shows the execution time of all algorithms on the small dataset used in Section 5.1. Notice that such documents were modified by human users and not by a change-simulator. The significance of the performance analysis on these files is then very high. Faxma proved to be the fastest solution. JNDiff performance was anyway acceptable, though the execution time was almost double, with an average of 1.1 seconds for documents with an average size of 30K, 644 nodes and 240 elements. The behavior of XyDiff could appear unusual: the algorithm is very fast but had bad performance in this case. The reason is again connected to the nature of the documents: data-centric algorithms, like XyDiff, behave in a very different way on text-centric resources.

**For the sake of clarity we did not include all data in the paper, but they are available online.



(a) real sample files (b) synthetized files

Figure 11. Measuring the execution time (shown in the Y axis, in seconds) when diffing files in the evaluation datasets. The results for each of the documents discussed in Section 5.1 are shown on the left; the right plot shows the minimum, average and maximum execution time on all documents in the synthetic dataset used in Section 5.2.2. Lower values indicate better performance.

Figure 11b summarizes execution times on the whole dataset used in Section 5.2.2. The most relevant parameter is the average execution time (the central part of the plot), showing that all algorithms run in less than 2 seconds. Faxma proved to be again the fastest one, while JNDiff was three times slower but still acceptable. The behavior of XyDiff on text-centric resources was confirmed by these experiments too. Notice also that algorithms proved to be comparable on very similar files (the minimum execution time) but show significant differences in the worst case (maximum execution time). JNDiff and diffxml, in fact, were much slower than faxma and XyDiff. Such a result is not good but, since it occurred a few times we consider it acceptable.

Many other experiments confirmed the same trend. They are fully available online and show how the execution time changes on input files with a fixed size and variable change ratios and, orthogonally, with a fixed change ratio and a variable amount of nodes. In conclusion, we are aware that the performance and computational costs of JNDiff are not comparable to other (very fast) diff algorithms but we believe such a loss is balanced by the output quality in most real cases.

ACKNOWLEDGEMENTS

This work was partially supported by CINI (Consorzio Interuniversitario Nazionale per l'Informatica). The authors would like to thank CINI for their interest and financial support.

6. CONCLUSIONS

JNDiff sacrifices some efficiency in order to obtain a clearer and more readable output. The NLCSS approach and the other strategies presented in this paper make the algorithm particularly effective on text-centric documents. In fact, our research is rooted in the idea of comparing these documents in a different way from data-centric ones, in order to reduce the existing gap between XML change-tracking and diff tools. We are refining multiple aspects of our prototype: testing optimization techniques - such as parallelizing the creation of the VTree data structures or using caching - to improve the efficiency, adding new modules to detect a larger set of changes, experimenting new thresholds and parameters to customise the algorithm's behaviour.

The next major step of our research is to explore *domain-oriented* diff tools, which are able to capture and express changes according to the terminology of each domain (for instance, on legislative documents or medical reports or scientific papers and so on). The basic idea is that each domain has its own set of typical changes, though these changes lie at a higher level of abstraction in comparison to those discussed in this paper. The faithful representation and detection of the underlying structural changes on XML is then an essential step in that direction, on top of which novel tools, services and interfaces can be delivered to the final users.

REFERENCES

1. Barabucci G, Borghoff UM, Di Iorio A, Maier S. Document Changes: Modeling; Detection; Storing and Visualization (DChanges). *Proceedings of the 2013 ACM Symposium on Document Engineering*, DocEng '13, ACM: New York, NY, USA, 2013; 281–282.
2. Barabucci G, Borghoff UM, Di Iorio A, Maier S, Munson E. Document Changes: Modeling; Detection; Storing and Visualization (DChanges). *To appear in the Proceedings of the 2014 ACM Symposium on Document Engineering*, DocEng '14, ACM: New York, NY, USA, 2014.
3. Marian A, Abiteboul S, Cobena G. Detecting changes in XML documents. *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, IEEE: Washington, DC, USA, 2002; 41.
4. Chawathe SS, Rajaraman A, Garcia-Molina H, Widom J. Change detection in hierarchically structured information. *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, ACM: New York, NY, USA, 1996; 493–504.
5. Ronnau S, Philipp G, Borghoff UM. Efficient change control of XML documents. *DocEng '09: Proceedings of the 9th ACM Symposium on Document Engineering*, ACM: New York, NY, USA, 2009; 3–12.
6. Brooke PJ, Paige RF, Power C. Document-centric XML workflows with fragment digital signatures. *Software: Practice and Experience* 2010; **40**(8):655–672.
7. Ronnau S, Scheffczyk J, Borghoff UM. Towards XML version control of office documents. *DocEng '05: Proceedings of the 2005 ACM Symposium on Document Engineering*, ACM: New York, NY, USA, 2005; 10–19.
8. Ronnau S, Borghoff UM. Versioning XML-based office documents. *Multimedia Tools Applications* 2009; **43**(3):253–274.
9. La Fontaine R, Mitchell T. XML Change Tracking Representing Change Tracking in any XML Document. http://lists.w3.org/Archives/Public/public-change/2013Mar/0021.html 2013.
10. Hunt JW, McIlroy MD. An Algorithm for Differential File Comparison. *Technical Report 41*, Bell Laboratories, Murray Hill, NJ 1976.
11. Miller W, Myers EW. A file comparison program. *Software: Practice and Experience* 1985; **15**(11):1025–1040.
12. Selkow S. The tree-to-tree editing problem. *Information Processing Letters* December 1977; **6**(6):184–186.
13. Tai K. The tree-to-tree correction problem. *Journal of the ACM* July 1979; **26**(3):422–433.
14. Zhang K, Shasha D. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing* 1989; **18**(6):1245–1262.
15. Lee MKH, Choy MYC, Cho MSB. An efficient algorithm to compute differences between structured documents. *IEEE Transactions on Knowledge and Data Engineering* 2004; **16**(8):965–979.
16. Mouat A. Diffxml and patchxml: Tools for comparing and patching XML files. http://diffxml.sourceforge.net/ 2009.
17. Thenault S, Mascio AD, Fayolle A. xmldiff: a python tool to diff XML resources. http://www.logilab.org/project/xmldiff 2009.
18. Lanna M, Amyot D. Spotting the difference. *Software: Practice and Experience* 2011; **41**(6):607–626.
19. Yang W. Identifying syntactic differences between two programs. *Software: Practice and Experience* 1991; **21**(7):739–755.
20. Marian A, Abiteboul S, Cobena G, Mignet L. Change-centric management of versions in an XML warehouse. *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2001; 581–590.
21. Lindholm T, Kangasharju J, Tarkoma S. Fast and simple XML tree differencing by sequence alignment. *DocEng '06: Proceedings of the 2006 ACM Symposium on Document Engineering*, ACM: New York, NY, USA, 2006; 75–84.
22. Leonardi E, Bhowmick SS. XANDY: a scalable change detection technique for ordered XML documents using relational databases. *Data Knowl. Eng.* 2006; **59**(2):476–507.
23. Buneman P, Khanna S, Tajima K, Tan WC. Archiving scientific data. *ACM Trans. Database Syst.* Mar 2004; **29**(1):2–42.
24. Chien SY, Tsotras VJ, Zaniolo C, Zhang D. Supporting complex queries on multiversion XML documents. *ACM Trans. Internet Technol.* Feb 2006; **6**(1):53–84.
25. Wagner RA, Fischer MJ. The string-to-string correction problem. *J. ACM* 1974; **21**(1):168–173.
26. Myers EW. An O(ND) difference algorithm and its variations. *Algorithmica* 1986; **1**(2):251–266.
27. Jacob J, Sachde A, Chakravarthy S. CX-DIFF: a change detection algorithm for XML content and change visualization for WebVigiL. *Data Knowl. Eng.* 2005; **52**(2):209–230.
28. Di Iorio A, Peroni S, Vitali F. A semantic web approach to everyday overlapping markup. *J. Am. Soc. Inf. Sci. Technol.* Sep 2011; **62**(9):1696–1716.
29. Fontaine RL. Merging XML files: A new approach providing intelligent merge of XML data sets. *In Proceedings of XML Europe 2002*, IDEAlliance: Barcelona, Spain, 2002.
30. Ronnau S, Pauli C, Borghoff UM. Merging changes in XML documents using reliable context fingerprints. *DocEng '08: Proceedings of the eighth ACM Symposium on Document Engineering*, ACM: New York, NY, USA, 2008; 52–61.
31. Boyer J. Canonical XML. http://www.w3.org/TR/xml-c14n 2001.
32. Rivest R. The MD5 Message-Digest Algorithm. http://tools.ietf.org/html/rfc1321 1992.
33. Bergroth L, Hakonen H, Raita T. A survey of longest common subsequence algorithms. *Proc. of the seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, IEEE: Washington, DC, USA, 2000; 39–.
34. Y Wang JC D DeWitt. X-diff: an effective change detection algorithm for XML documents. *ICDE '03: Proc. of the 18th International Conference on Data Engineering*, IEEE: Washington, DC, USA, 2003; 519–530.