Semantics to the rescue of document-based XML diff: A JATS case study

Milos Cuculovic, Frederic Fondement, Maxime Devanne, Jonathan Weber, Michel Hassenforder

Received: 20 July 2021 | Revised: 17 January 2022 | Accepted: 18 January 2022

DOI: 10.1002/spe.3074

RESEARCH ARTICLE

WILEY

Semantics to the rescue of document-based XML diff: A JATS case study

Milos Cuculovic[1,2] | **Frederic Fondement**[1] | **Maxime Devanne**[1] | **Jonathan Weber**[1] | **Michel Hassenforder**[1]

[1]IRIMAS, University of Haute-Alsace, Mulhouse, France

[2]R&D, MDPI, Basel, Switzerland

Correspondence
Milos Cuculovic, IRIMAS, University of Haute-Alsace, Mulhouse, 68100, France
Email: milos.cuculovic@uha.fr

Funding information
MDPI, Grant/Award Number: cuculovic-thesis

ABSTRACT

The writing of digital text documents has become a longer process that usually goes through revision rounds. Document comparison is important for the human reader interested in changes made by the authors. These documents contain structural data using text-centric XML as one of their main storage systems. Current XML diff algorithms are able to represent differences with a limited number of edit operations: insert, delete, move and update. This approach does not fit the scope of digital text document comparison where the human reader needs to understand actual modifications made by the author. With JATS being a text-centric XML vocabulary, we propose within this paper a new XML diff algorithm called jats-diff, able to support bijection between higher-level modifications made by the authors, such as structural changes and restyling, and the changes detected between XML documents. In addition, jats-diff provides similarity information between different nodes in order to measure the impact of the text changes on the XML tree.

KEYWORDS

academic publishing, change semantics, diff algorithms, document comparison, high-level changes, JATS, text-centric XML, XML diff

1 | INTRODUCTION

With the appearance of digital textual documents, many researchers have expressed their interest in extracting, analysing and understanding textual differences. Text diff algorithms[1,2] have been studied and described. Some of them are still in use, such as Hunt–McIlroy's[3] algorithm (currently used in the GNU Diff utility) and Myers'[4] algorithm. Most of these algorithms are line-based and rely on two edit operations: Insert and Delete. In order to detect the differences, each line of the original is compared with the corresponding line of the modified text document. The range of applications for text diff algorithms is wide; they are present in version control systems (Git, Apache Sub-version) and many other tools such as IDEs (Eclipse compare) or text editors (Notepad++ compare), where tracking textual differences is important.

Starting from the late 1990s, with the appearance of semistructured documents, some research groups had their focus on a specific type of text document—XML.[5] This document type was widely adopted in a short period of time in the domain of high technology. Compared with plain text documents, the particularity of XML resides in its hierarchical

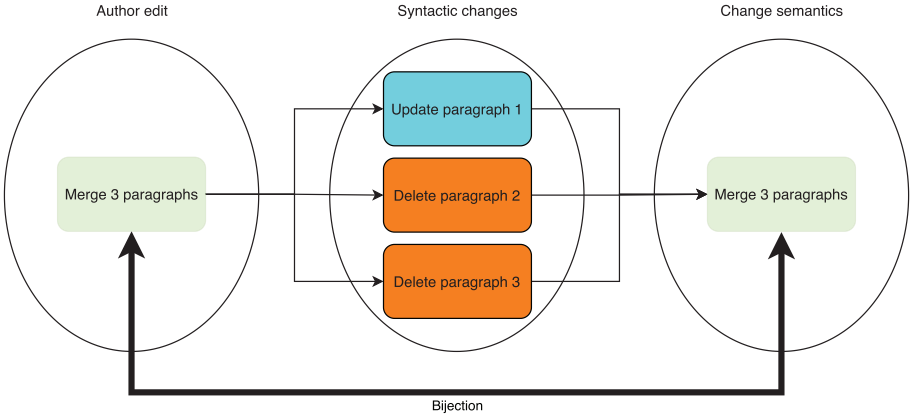

FIGURE 1 Example of bijection used while detecting a paragraph merge. Three paragraphs were merged together by the author and detected as three syntactic changes: two delete and one update. Using change semantics, the merge edit is detected in the same vocabulary of the user edition

structure, also called tree structure, and nodes can also have attributes to carry specific information. The tree-to-tree editing problem defined by Selkow[6] makes existing text diff algorithms unsuitable for comparing XML documents. This was further demonstrated by several research groups.[7-9] Several other projects[10] also developed HTML-based diff algorithm implementations providing content and styling differences. As we are interested in content changes only, this approach is not further considered.

Looking into the historical usage and development of XML documents, we can determine that they initially played a role in data archiving and interchange, called data-centric XML. With the appearance of digital textual documents, some text processors started using XML to store textual data, and the era of text-centric (or document-centric) XML appeared, as described in References 11 and 12 (see chapter *Text-centric vs. data-centric XML retrieval*). Several research groups[11,13,14] demonstrated that XML diff algorithms designed to be data-centric are not suitable for text-centric XML documents. This is mainly due to the fact that data-centric XML diff algorithms were mostly evaluated on their delta size and computing performance. Those are characteristics of a data-centric approach which is not suitable for text-centric documents where the quality of the delta output (readability and accuracy) is the most important.

Among the existing XML diff algorithms, 12 of them were tested,[15] and none were entirely suitable for text-centric XML comparison made by humans. The highest-scoring algorithms[9,11,16] were the ones built for text-centric XML documents; however, those algorithms produce delta outputs made to be machine readable and mainly used for merging (versioning) purposes. This makes their use inconvenient for human readers who are not able to determine a strong relationship between edits made by authors, on one side, and the changes represented in their deltas. Since text-centric XML documents are used as central storage systems for digital textual documents, we believe it is important for the human readers to understand the real changes made by authors while comparing them. Making a bijection between author modifications on one side and the detected differences on the other (see Figure 1) would help the human reader to achieve that goal.

There are several text-centric XML document types today which are largely used, such as JATS[*], DocBook[†], TEI[‡] and DITA[§]. Each of them has its own area of application. For example, JATS (Journal Article Tag Suite) developed by NISO (National Information Standards Organization) is used as the main document type for the archiving and interchange of scientific literature. In the rest of this article, we will use JATS as the subject of the testing carried out to validate our methods.

In the scientific literature, there are several main document types largely used by authors while writing their articles; among them are well-known and established formats such as Tex[¶], docx[#]and odt[‖]. Academic publishers convert the

[*]https://jats.nlm.nih.gov.
[†]https://docbook.org/.
[‡]https://tei-c.org/.
[§]http://docs.oasis-open.org/dita/dita/v1.3/dita-v1.3-part3-all-inclusive.html.
[¶]https://foldoc.org/TeX.
[#]https://loc.gov/preservation/digital/formats/fdd/fdd000397.shtml.
[‖]https://loc.gov/preservation/digital/formats/fdd/fdd000428.shtml.

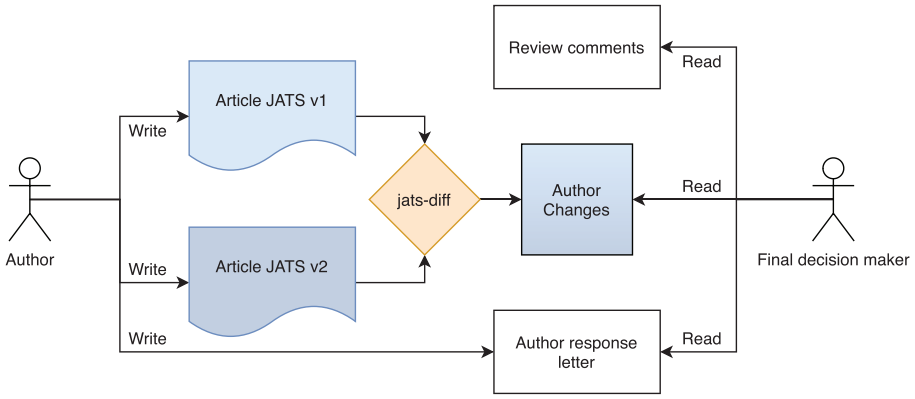

FIGURE 2 Article final decision-making process where the decision maker has to read review comments, author response letter and compare different versions of the article in order to evaluate whether the author made all the changes requested by the reviewer

articles from their initial document types to JATS, being *de facto* standard for the XML representation of academic articles and is used by major indexing companies, including PubMed Central[**] and SciELO[††]. JATS has the advantage of being machine readable and independent of text processors; it also has no layout information and carries only the article data and structure. The layout, depending on the publisher preferences, is applied at a later stage, while the JATS is converted to HTML, LaTeX and other formats. The article information is divided into three main elements: front, body and back. The front contains sub-tree elements about the metadata information: the journal, title, authors, affiliations and so forth. The body contains the article content organised in sections, sub-sections and paragraphs. It is the largest part of the document, dominated by text blocks represented as paragraphs in a similar way to HTML <p> tags. In addition to the text, paragraphs are also composed of <xref> elements used for citing objects contained in the back part of the JATS (references, figures, tables) and styling elements such as <b>, <i>, <sub>, <sup> and so forth.

We propose within this article a new text-centric XML diff algorithm that can help to better understand and represent human edits on a text processor by analysing their impact on XML representation. Among others, one of the practical application areas where this new algorithm can be used is the article comparison done by the final decision maker during the peer-review process in the academic publishing. In order to take the final decision (see Figure 2), senior scientists have to read reviewer comments, compare different versions of the article and read author response letter with the purpose to evaluate whether the authors made the requested changes during the revision rounds. The article comparison together with the change description reading tasks are time consuming. Moreover, the change detection is written by the authors and may not always reflect all, nor real changes made during the revisions. It is then of great importance to automate this task as much as possible in order to obtain a higher quality and faster final decision making. Some text processors (Open Office, Libre Office and MS Office) provide two interesting functions: document compare and change tracking. Although useful, both of those functions represent changes only visually, and change sets cannot be extracted for further processing. Those also rely on text processors and, in addition, the change tracking function lies in the author's hands and can be disabled at any time.

From the decision maker perspective, having direct and easy access not only to basic comparison between the two articles, but to real modifications made by the authors is more convenient than reading the change description document and comparing different versions of the article in order to mindfully evaluate the changes.

Regarding the organisation of this article, we start in Section 2 with the focus on three existing text-centric XML diff algorithms that scored the best regarding their capacity to compare JATS documents. In Section 3, we present different author modification patterns made in text processors and their impact on JATS. In Section 4, we propose seven new and specific edit actions, proper for text-centric XMLs: structural upgrade, downgrade, split and merge, inline style edit, text move and citable node edit. An important factor also considered is the flexibility to detect all those changes when minor text edits interfere, this being very specific to author edits on typesetter tools. In Section 5, we present a similarity index calculation between two JATS documents, its benefits and propagation towards the XML tree. In Section 6, we compare jats-diff with the three other state-of-the-art XML diff algorithms we presented in Section 2. Finally, in Section 7, we

[**]https://www.ncbi.nlm.nih.gov/pmc/.
[††]https://scielo.org/en/.

discuss additional text change semantics that could be beneficial for distinguishing between simple sentence rephrase and sentence meaning changes. We also mention the additional improvements that could be done in regards to the delta representation and the execution time and memory usage. Moreover, we also discuss different practical usages of the new jats-diff algorithm.

2 | RELATED WORK

As mentioned previously, XML documents are divided into two types, text- and data-centric. Historically, data-centric XML diff algorithms inherited the existing text comparison capacities and additionally added so-called "Level 1" or "basic" edit operations[17] on tree elements: insert, delete and attribute change. Those basic edit operations are very important for the XML tree structure, as without attribute change detection, modifying the highest parent attribute would result in deleting the entire XML tree and inserting the same tree with its attribute change. Data-centric XML diff delta output was rather large using only those basic edit actions, which was not an issue as the change information consumers were usually machines looking for precise data modifications. The number of individual text nodes being high but small in size, modifying a text node was naturally represented as a simple delete and insert combination. Another reason for the restrictive list of changes was their detection performance. Higher-level changes require more computation power and time to execute; thus, keeping the list of changes simple would allow better performance.

Recent XML diff algorithms are mostly text-centric and are proposing a specific higher level or "Level 2" changes that are not found in data-centric XML diff algorithms. "Level 2" changes are composed of a combination of "Level 1" basic edit operations. Each time a specific insert–delete sequence is observed, it then gets converted to a unique "Level 2" change. The individual text nodes are usually larger in size but smaller in number; thus, it makes sense to think about higher-level changes that replace a combination of basic insert and delete operations. Moreover, as humans are usually the main producers of those text-centric XML documents, there is a need to use a strong relationship between the modifications made and the changes detected. Among the "Level 2" tree edits, existing algorithms are able to detect and represent tree move and wrap/unwrap (observed in Reference 11). Wrap is used to detect edit patterns where a specific portion of text within a given node was wrapped by another node, unwrap being the opposite of wrap. Adding/deleting styling nodes around a text portion is represented as wrap/unwrap. In the literature, there is a discussion about an additional number of "Level 2" changes. Some research groups propose element merge and split[17] and also text move.[11] Unfortunately, none of the existing algorithms propose a solution regarding how to detect and represent such changes.

In order to cope with performance issues while comparing large textual nodes, most text-centric XML diff algorithms[9,11,16] use the two-pass logic where the first pass is used to assign fingerprint values to the XML tree structure on both versions of the document. In the second pass, differences are detected by comparing node fingerprints, instead of comparing their content.

We shall now briefly describe three existing text-centric XML diff algorithms and their capacity in detecting and representing changes:

- **JNDiff**[11] was published in 2016 with the purpose of reducing the differences between the existing change tracking functions available in text processors and the change detection between two XML documents. The algorithm implementation was written in JAVA and is available at Sourceforge. Instead of concentrating on high computing performance and delta size, JNDiff focuses on the delta output quality (human readability, accuracy and clearness). Its performance is, however, on a good level due to the use of hash fingerprints. The algorithm supports four "Level 2" edit actions: Update, Move, Wrap and Unwrap. The research paper also mentions the need for additional edit operations to detect paragraph merge and split; those are, however, not further developed. JNDiff scored the highest regarding JATS comparison,[15] mostly due to its capacity to detect and represent edits within text nodes containing other nodes as styling and citing elements. This is very common in JATS, and most of the existing XML diff algorithms are not able to cope with such cases. The way of detecting text updates is also very useful. JNDiff has a similarity threshold that can be defined as follows: if the similarity of the two compared text nodes is higher than the threshold, the change is considered as an update, whereas if lower, the change is considered as a delete–insert combination.
- **XyDiff**[9] was published in 2002 by Gregory Cobena within an PhD project called Xyleme. The algorithm was implemented in C++ and is available at Github. The purpose of the Xyleme project was to index and analyse the changes on the web where different parts of websites were stored as XML documents. XyDiff also uses the two-pass approach and

assigns so-called "XID" to each node. The algorithm supported two "Level 2" edit actions: update and move. It is one of the fastest XML diff algorithms as this research group had to deal with a large number of documents to compare. In order to detect text updates, XyDiff uses LCS (longest common sub-string) with the goal to minimise the edit distance. This approach is good for machine readable delta but not for human readers, while it is difficult to understand the actual changes an author made by analysing the character-based LCS representation. The capacity of the algorithm is very limited, while edit detection is done on text nodes containing other nodes.

- **XCC**[16] was published in 2012 with the purpose of comparing office documents where the content is saved in XML format—more precisely, the OpenDocument odt format. The algorithm, implemented in Java and available at Launchpad, supports two "Level 2" edit actions: update and move. XCC introduces context fingerprints were the position in the tree and the neighbourhood involved in the change is part of that fingerprint. One of the main goals of XCC is to produce a human-readable delta. Similar to XyDiff, XCC comes with a low performance regarding the edit detection on text nodes containing other nodes which is, again, very often seen in text-centric XML documents.

In addition to detecting existing "Level 1" and "Level 2" changes, there is also a need for additional "Level 2" change detection such as structural upgrade, downgrade, split and merge, inline style edit, text move and citable node edit. For this to happen, we need to apply similar bijection methods as seen on Figure 1 where change semantics are used to recognise author edits as a specific sequence of lexical and syntactic changes. Several research groups have started working on XML diff algorithms for semantic change extraction.[18-21] The main idea shared by those research groups was to track the evolution of an XML document in time by extracting change semantics between elements that do not necessarily share the same structure and the same identifiers. There are algorithm proposals[18] able to support an XML versioning system where the XML structural changes are ignored and change detection is made by first detecting identifiers for elements that are common across the versions and then using these identifiers to associate elements among the versions. XKeyMatch algorithm proposed in Reference 19 follows the same goal as[18] and uses XML keys to match elements that refer to the same entity among the versions. They use this extension for pre-processing the structural analysis phase that ordinary XML diff algorithms do in order to match similar elements but having structural changes between the versions. Oliveira et al.[20] identified syntactic change patterns in order to deduct semantic changes. In their example, a combination of employees with a salary increase and employees with title change could be used to extract employees' promotion semantics. In Reference 21, the authors described the Phoenix algorithm used for the same goal to match the same elements among different document versions that had their identifiers changed. Phoenix uses similarity metrics for this purpose.

3 | IMPACT OF AUTHOR EDITS ON XML: A JATS EXAMPLE

The existing text processor track change tool is very efficient for generating a human-readable description of the edits applied by the author on a given digital textual documents. Unfortunately, this tool lies in the author's hands and depends on the text processors. On the other hand, the current XML diff algorithms generate delta outputs with a limited number of edit patterns where one author edit action is interpreted as a sequence of different lower-level edit actions (insert/delete), which makes the delta hard to read and be understood by humans. In this section, we analyse some common edit actions observed on digital textual documents (academic article being taken as example) that authors regularly produce and correlate those with their impact on XML.

Academic articles usually follow a standard structure in terms of how they are written regarding the sections they embed and in which order those appear. An article usually starts on the first page with the journal, title, authors, affiliation, abstract and keywords information. What follows is the largest part with several sections, each of which can contain sub-sections, paragraphs, citations, figures, tables, maths formulas, and so forth. At the end, we usually find the acknowledgments and reference list. As the largest part of an article is text blocks known as paragraphs, most of the changes made by authors are mainly observed there. Figure 3 shows some common author edit actions and their impact on JATS XML—see Figure 4. The current "Level 2" edit actions are unfortunately not able to represent all those edits in a human-readable way. Instead, those are mostly presented with a combination of "Level 1" (insert and delete) edits. In the following text, we use abbreviations I, D, A, U and M for insert, delete, attribute edit, update and move representations of XML edits, respectively.

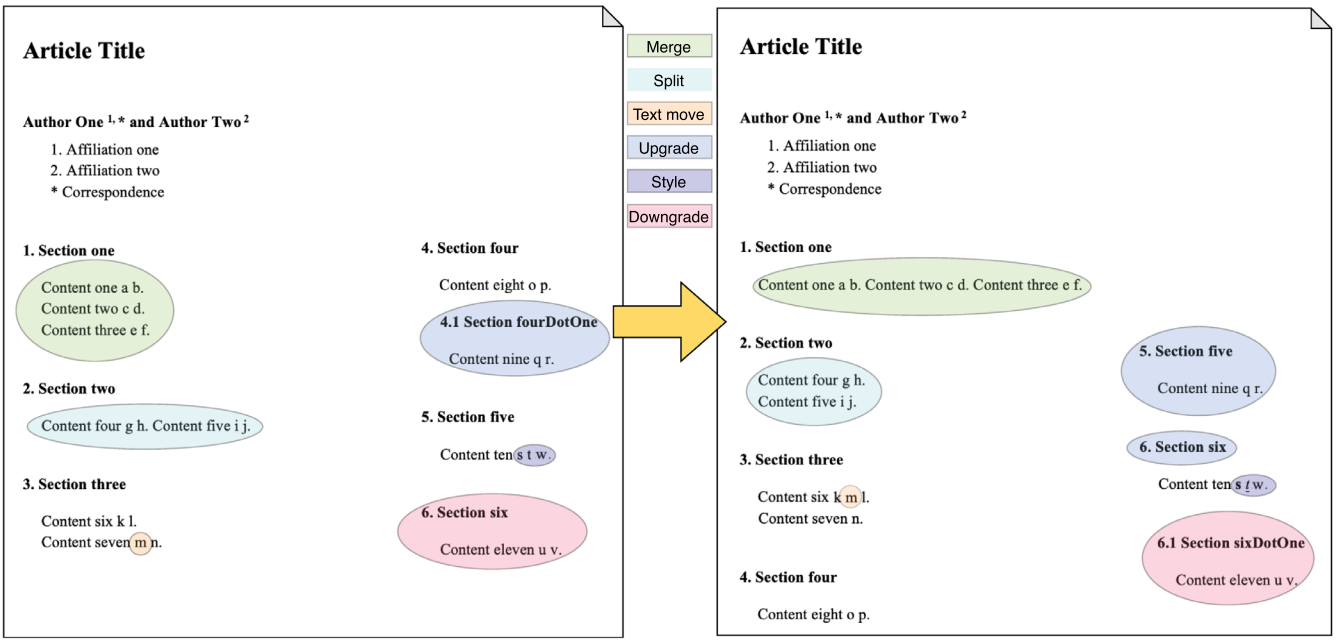

FIGURE 3 Two article versions side by side written in a text processor, highlighting the following author changes: Three paragraphs from section 1 merged into one; one paragraph from section 2 split into two; text portion moved from the second into the first paragraph in section 3; subsection 4.1 upgraded as section 5, implying the auto-increment of the previous section 5 to section 6; styling edits on the paragraph in section 5; initial section 6 downgraded as a new subsection 6.1

3.1 | Paragraph merge and split

Two of the first paragraph edit actions we observed are merge and split. Authors usually split large paragraphs into smaller ones or merge several small paragraphs into a bigger one. This action is relatively easy in text processors and consists of adding or removing line breaks between text blocks.

Figure 3 (section 1) shows how a paragraph merge action is done by the author on the text processor and its impact on XML (see Figure 4, sect. 1), with a higher complexity than simply removing line breaks. We can observe a combination of U+(n−1)D, n being the initial number of the paragraphs to merge. Looking at the delta output of existing XML diff algorithms, the merge modification in our example will appear as three or more edit actions: U+2D in the best case scenario or I+3D, while U is seen as a combination of D+I. The split edit action is the opposite of merge as can be seen in Figure 3 (section 2) where content "i j" lands in a new paragraph annotated 4' in Figure 4 (sect. 2), representing the split edit impact on JATS. The delta representation for this change is symmetrical and of similar complexity as merge.

From the human reader perspective, reading a delta output representing a structural split or merge edit action as a combination of three or four basic edit actions is not convenient and requires some higher-level interpretation.

3.2 | Text move

Another common edit action we observed is the text move where authors move sentences or part of text from one paragraph to another within the document. Figure 3 (section 3) shows how part of a text labelled "m" was moved from the second to the first paragraph. The impact of this edit on XML will appear as 2U in the best case scenario—see Figure 4, sect. 3—or as 2(D+I).

Again, from the human reader perspective, representing a text move as two node updates is not self intuitive and cannot be easily interpreted as a text move.

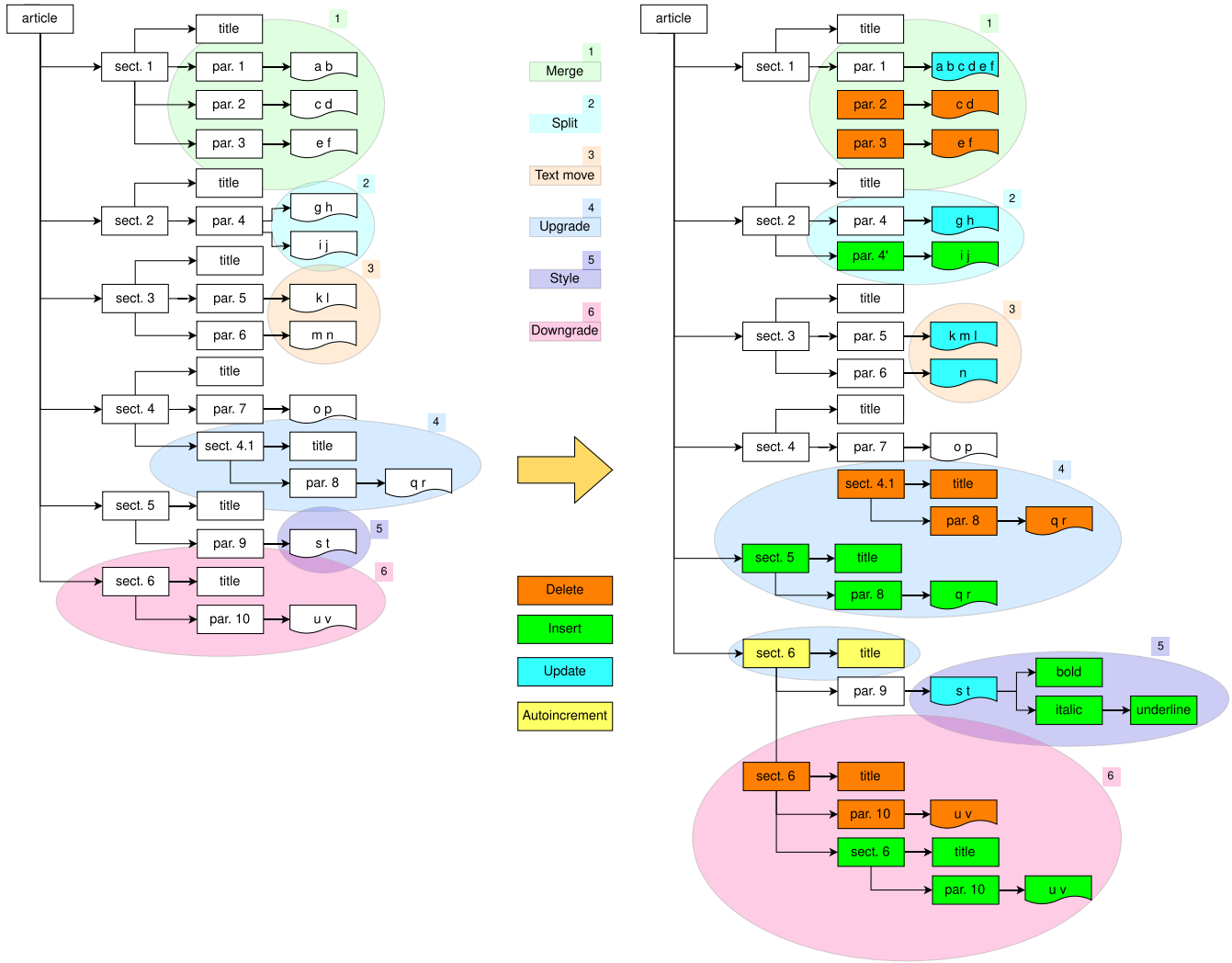

FIGURE 4 Two JATS XML versions side by side, highlighting the following author changes: Three paragraphs from sect. 1 merged into one; one paragraph from sect. 2 split into two; text portion moved from the second into the first paragraph in sect. 3; sect. 4.1 upgraded as sect. 5, implying the auto-increment of the previous sect. 5 to sect. 6; styling edits on the paragraph in sect. 5; initial sect. 6 downgraded as a new sect. 6.1

3.3 | Sub-section upgrade/section downgrade

Sub-section upgrade is yet another edit action we observed and happens when authors upgrade a sub-section to a section. Each section/sub-section being composed of one label, one title and the section body, upgrading a sub-section in the text processor is usually seen as a combination of font increase, indent decrease and label change—see Figure 3 (section 4). As JATS XML does not hold any layout information, the changes observed in the text processor cannot be directly detected, which increase the detection complexity. We can observe in Figure 4 (sect. 4.1 → sect. 5) that multiple nodes are affected by the change. The entire sect. 4.1 is removed and inserted as sect. 5. Moreover, an important inducted change is observed in sections following the upgraded subsection as their label, and the ID will be automatically changed according to the numbering plan. In our case, the initial sect. 5 will be renumbered as sect. 6. Depending on the XML diff algorithm, comparing the two JATS files will result in a delta showing a full D of sect. 4.1 and a full I of sect. 5, followed by an attribute and label change on the initial sect. 5 becoming sect. 6. Eventually, the full delete–insert combination is also seen on some XML diff algorithms as a simpler sect. 4.1 parent and title node D, followed with the new sect. 5 parent and title node I and a M of sect. 4.1 child elements in the newly created sect. 5, resulting from a combination of 2(D+I)+M.

The downgrade edit action is the opposite of upgrade. Downgrading a section in the text processor is usually seen as a combination of font decrease, indent increase and label change—see Figure 3 (section 6) for author edits in a text processor. Its impact on XML—see Figure 4 (sect. 6.1)—is exactly the same as for an upgrade, but in the opposite direction.

Both of the mentioned author edit patterns are not recognised in the existing XML diff algorithms and make the delta difficult to interpret from a human reader perspective.

3.4 | Style edit

Although JATS XML documents do not hold any layout information, textual styling is still part of it and represents important information in the document. Most of the paragraphs within the document hold text styling information, where we can observe bold, italic, underline, subscript, superscript and others. Styling is purely a visual change in the text processor—see Figure 3 (section 5 → section 6) where one part of the text was styled bold and another one italic and underlined. On the other hand, styling information is represented as node elements on XML, and their insertion/removal directly impacts the XML tree—see Figure 4 (sect. 5 → sect. 6). This adds another layer of complexity for the XML diff algorithms as inserting a bold style around a text portion consists of inserting a new node that will wrap this text content. Most XML diff algorithms will represent this change as a new node I containing the edited text portion and an U of the edited paragraph text node. This kind of delta output is, again. not easy for human readers and is hard to interpret as a style edit action made by the author.

3.5 | Citable object reference edit

Another common element we can observe in paragraphs are citable object references. Authors usually cite bibliographies, figures, tables and other sections within the article. Those references appear as <xref> nodes in XML and are visible in most of the paragraphs. In order to cite an object, its label and ID are used. Those auto-incremental values are assigned to each citable object and are dependent on its appearance in the citable objects list. A reference citation is seen as <xref ref-type="bibr" rid="B2-molecules-25-00430">2</xref>, where "B2-molecules-25-00430" represents the ID, and the number 2 represents the label. If the paragraph is changed from citing the bibliography B2-molecules-25-00430 to B3-molecules-25-00430, both the ID and the label will be changed within the <xref> node. Those two properties make it difficult to track author edits on the citable objects list. Let us assume a scenario where there are five references that the author is using as the bibliography. If a new reference were to be added at the position 2, the IDs and labels of all the following references would be incremented, that is, the "B2-molecules-25-00430" would switch to "B3-molecules-25-00430" and so on. The same applies for their labels. This change would then also impact all the references those objects have within the article, where the <xref> from our previous example would change to <xref ref-type="bibr" rid="B3-molecules-25-00430">3</xref>, and the same for the following references. Having those inducted changes while adding or removing citable objects would result in them being represented in most of the XML diff tools as, instead of a simple citable object I, I+n(A+U) in the best case scenario, or I+n(D+I) if the A and U are seen as full <xref> D+I, n being the number of auto-incremented bibliographies.

This type of simple author edits creates a lot of noise in the delta output that needs to be detected and eliminated by the XML diff algorithm. Without any filtering, one reference I or D can reproduce hundreds of other edits within XML due to the inducted changes it generates.

4 | "LEVEL 2" EDIT ACTIONS

As seen in Section 3, we can observe several edit patterns made by the authors in their text processors that have a higher-level impact on XML. Nowadays, those edit patterns are not recognised by any of the existing XML diff tools and are usually represented as a combination of insert and delete edit actions. From a human reader perspective, having a bijection with the author edits on one side and the XML diff output on the other—see Figure 1—is valuable and can help us to understand what modifications are really made by the author on a given document. The following edit patterns that the new jats-diff algorithm is able to interpret and correctly represent are described within this section: structural upgrade, downgrade, split and merge, inline style edit, text move and citable node edit.

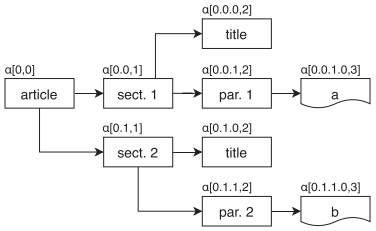

FIGURE 5 Example of XML tree on document A using the conventional labelled ordered tree model $\alpha[m, n]$ and $\beta[p, q]$, where m and p represent the node identifier and n and q the depth

4.1 | XML diff base—JNDiff

Due to the high scoring of the JNDiff algorithm[15] in comparing text-centric XML documents, we decided to ground our new edit pattern recognition using the core functionalities of JNDiff for "Level 1" XML edit detection. JNDiff has good performance and is highly reliable in respect of detecting differences between two XML documents. The logic of the algorithm is as follows: It first builds one virtual tree object per document; then, it detects inserts and deletes as basic/"Level 1" edit actions; finally, it tries to refine the detected "Level 1" differences and convert them to "Level 2". Each time a specific insert–delete sequence is turned to "Level 2" change, it gets removed and replaced in the change list by its "Level 2" representation. Our new edit pattern detection will be added to the JNDiff refinement logic in order to recognise and represent them in the delta.

4.2 | XML tree annotation

We use the conventional labelled ordered tree model in order to represent the XML trees in the following edit pattern detection on JATS XML. As we always compare two XML documents A and B—two XML trees—each node belonging to the document A is labelled α and the document B β. We assign to each node an identifier m for document A and p for document B. In addition to the identifier, the node depth regarding the tree represented by n for document A and q for document B is also added, resulting with the annotation $\alpha[m, n]$ for document A and $\beta[p, q]$ for document B.

As we can see in Figure 5 representing the XML tree of the document A, each node has a specific annotation. $\alpha[0, 0]$ represents the article node, being the root node. $\alpha[0.0, 1]$ and $\alpha[0.1, 1]$ represent sect. 1 and sect. 2, respectively, sect. 1 being the node number 0.0 at depth 1 and sect. 2 the node number 0.1 at depth 1. Section titles and paragraphs are within depth 2 and have node numbers 0.0.0 to 0.1.1. Finally, text nodes are at depth 3 and have node numbers 0.0.1.0 and 0.1.1.0. On the left, we can observe shallow nodes, and on the right, deeper nodes.

4.3 | Text similarity versus text equality

All existing XML diff algorithms we tested use text equality while trying to match "Level 2" among the "Level 1" changes. Taking the example of structural merge, see Figure 6 where the P1 and P2 nodes, representing paragraph 1 and 2, are merged into P3. The merge is detected only if the text in P3 is exactly the same as the sum of the texts in P1 and P2. This behaviour is suitable for data-centric XML documents where high precision is required. On the other hand, text-centric XML documents are prone to small textual edits where grammar and sentence rephrasing are common. It is then important to replace text equality checks with text similarity while evaluating "Level 2" changes. This way, the algorithm is more flexible and can detect "Level 2" changes even with small textual change interference represented in Figure 6 by the addition of "but the". All the following "Level 2" change patterns we present use text similarity with a threshold that we experimentally defined at 95%; however, this value can be changed for further fine tuning. Once the "Level 2" change pattern has been detected, and if the text node has a similarity different from 100%, the text updates have to be detected in the usual way regardless of the "Level 2" change.

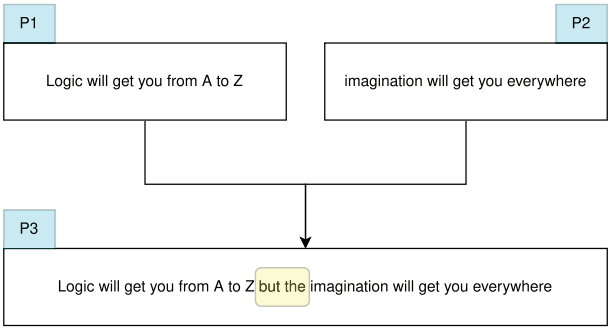

FIGURE 6 Nodes P1 and P2 merged into P3 while a text change common to text-centric XML document was introduced, showing the benefits of text similarity over text equality usage for "Level 2" change detection

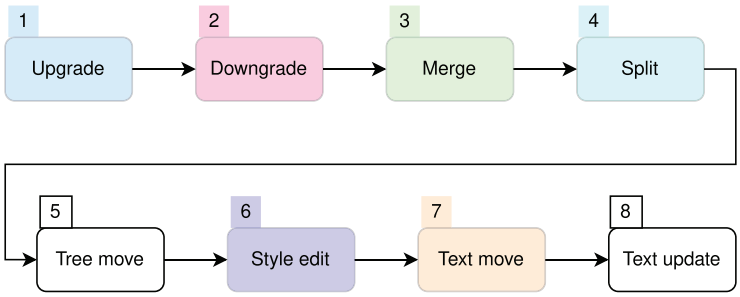

FIGURE 7 Edit pattern detection order used as higher-level edits is usually composed of other "Level 1" or "Level 2" edits, which makes their detection order important

4.4 | Order of the edit detection

The order in which we recognise the following edit patterns remains important as some "Level 2" edits are composed of a combination of "Level 1" or even other "Level 2" edits. In the example of two structural merge, seen as one node delete ("Level 1") and one node update ("Level 2"), it is important to run the structural merge pattern detection before the node delete and text update detection. Regarding the structural upgrade, this edit pattern is composed of several tree moves and text updates, and it is thus important to run this pattern detection before tree move and text update. Following an empirical evaluation, we decided to choose the following "Level 2" edit pattern detection order (Figure 7).

4.5 | Structural upgrade/downgrade

As seen in Figure 4 (sect. 4 → sect. 5), upgrading one child node to the same depth as its parent represented by upgrading one sub-section with x nodes into a section results in x(D+I) "Level 1" edits. Moreover, as the remaining section numbering will also change due to their auto-incremental nature, we observe additional y(A+D+I)) "Level 1" edits, y being the number of remaining sections following the upgrade: A for id; D+I for the title. In total, a simple section upgrade will result in x(D+I) + y(A+D+I) "Level 1" edits. Within our example, see Figure 8, the structural changes that consist of upgrading (sect 1.1) node into (sect. 2) result in depth lowering on each of the upgraded nodes followed by attribute id and title change.

By applying the following mathematical formula on the lists of node changes detected between documents A and B, respectively annotated cα and cβ, we can evaluate if the specific change pair fulfils the structural upgrade condition:

$$(\forall c\beta)(\forall c\alpha)\beta[p,q].content \simeq \alpha[m,n].content \wedge q < n. \quad (1)$$

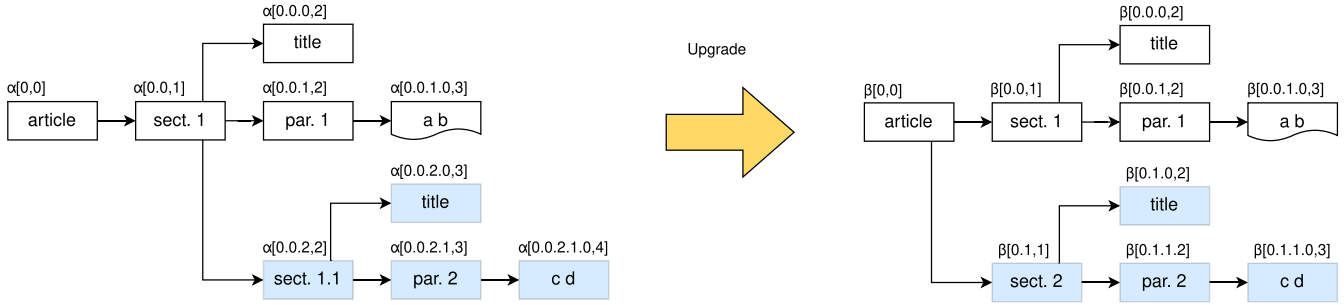

FIGURE 8 Impact of structural upgrade on XML tree: we observe that the depth of the upgraded (sect. 1.1) node is lowered by one, becoming, on the depth level, the same as their previous parent (sect. 1)

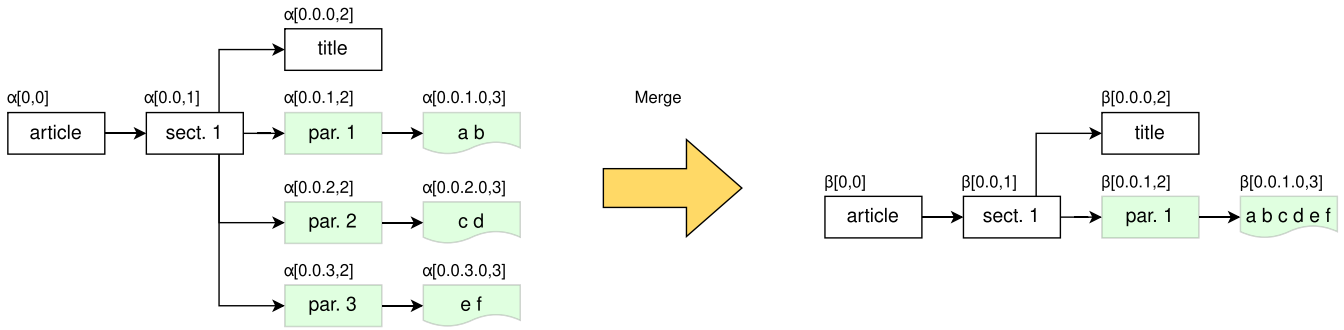

FIGURE 9 Impact of structural merge on XML tree: we observe that the text content "cd" and "ef" from nodes (par. 2) and (par. 3) is merged with the text content "ab" of the (par. 1) node

The formula verifies if the contents of (sect. 1.1) and (sect. 2) nodes are similar, and in addition, if the depth of (sect. 2) is lower than the depth of (sect. 1.1) node. Having this condition satisfied will result in a structural upgrade detection. By running the formula on our example, the following scenario occurs: for each change detected in document B ($\beta[p, q]$), evaluate if the text content of $\beta[0.1, 1]$ "title"+"cd" is similar to the text content of $\alpha[0.0.2, 2]$ "title"+"cd". In addition, evaluate if the depth of the modified $\beta[0.1, 1]$ element is lower than the depth of the $\alpha[0.0.2, 2]$ element. As both conditions are satisfied, the structural upgrade pattern is recognised.

The delta output of such pattern detection is represented as one "Level 2" change named "upgrade", having two information elements: "upgrade_from" and "upgrade_to".

Structural downgrade is the opposite of upgrade, where it is enough to invert the depth comparison in order to adapt the upgrade formula to detect downgrade patters. Once a structural downgrade pattern has been detected, the delta output will contain one "Level 2" change named "downgrade", having two information elements: "downgrade_from" and "downgrade_to"

4.6 | Structural merge/split

As seen in Figure 4 (sect. 1), merging x nodes, represented by paragraphs, into a unique one will be seen as x+1 "Level 1" edits, composed of xD+I. Using the update "Level 2" edit, the number of edits observed is lowered to x, with (x-1)D+U. We propose here a new way of detecting structural merge. Within our example, see Figure 9, nodes $\alpha[0.0.2, 2]$ and $\alpha[0.0.3, 2]$ are merged with node $\alpha[0.0.1, 2]$. Represented with "Level 1" changes, this edit pattern detection results in 3D ($\alpha[0.0.1, 2]$ with content "ab", $\alpha[0.0.2, 2]$ with content "cd" and $\alpha[0.0.3, 2]$ with content "ef"), followed by I of $\beta[0.0.1, 2]$ with content "abcdef".

By applying the following mathematical formula on the lists of node changes detected between documents A and B, respectively annotated cα and cβ, we can evaluate if specific node pair fulfils the structural merge condition, the merge being valid only if two or more nodes from cβ fulfils the condition:

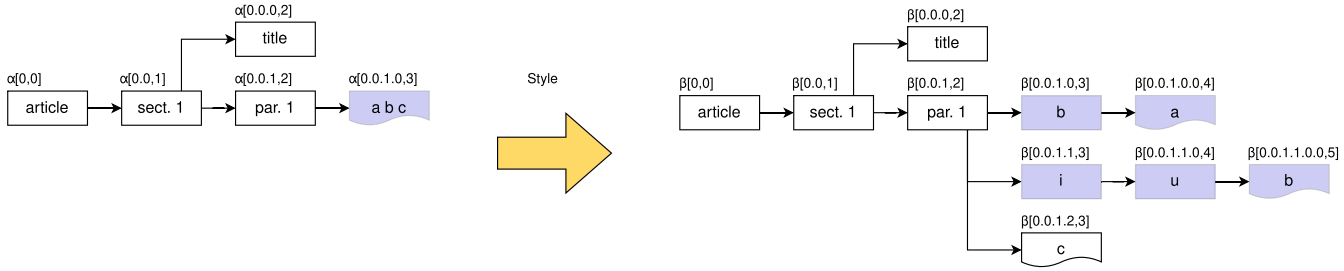

FIGURE 10 Impact of inline style change on XML tree: we observe that adding bold to one part of the text and italic + underline to another changes the XML tree structure, as each inline style addition is seen as a new element added to the existing XML tree

$$(\forall c\beta)(\forall c\alpha)\alpha[m,n].content \subset \beta[p,q].content \wedge q = n. \tag{2}$$

This mathematical formula verifies for every node content on β if there are nodes on α whose content is subset β node content and where both depths on β and α are identical. By running the formula on our example from Figure 9, the following scenario occurs: for each change detected in document B $\beta[p,q]$, test if the mathematical condition is verified for a given node in document A ($\alpha[m,n]$); if so, the examined node is a merge candidate. The algorithm will test for a given node in document B $\beta[0.0.1,2]$ all nodes within the same depth in document A $\alpha[0.0.1,2]$, $\alpha[0.0.2,2]$ and $\alpha[0.0.3,2]$, with their respective text content "ab", "cd" and "ef". As their text contents are all contained in $\beta[0.0.1,2]$ "abcdef" and they all have the same depth 2, they will be added to the merge candidate pool. At the end, if there is more than one merge candidate in the pool, a "Level 2" tree merge edit is detected. The resulting delta using the structural merge pattern detection while merging n nodes into a unique node will be seen as one "Level 2" edit, containing n−1 merge_from and one merge_to information elements.

Structural split is the opposite of merge, where instead of evaluating if the text content of changed $\alpha[m,n]$ nodes are contained by $\beta[p,q]$, we evaluate if the text content $\beta[p,q]$ nodes are contained by $\alpha[m,n]$. While using structural split edit pattern detection, splitting one into n nodes will be represented as one "Level 2" edit containing one split_from and n split_to information.

4.7 | Inline style edit

As seen in Figure 4 (sect. 5) → (sect. 6), styling information is seen as XML nodes. The most often observed styling elements are for bold, <i> for italic, <sub> for subscript, <sup> for superscript and <u> for underline. Style edits have no narrative impact, and the node textual structure remains the same. On the other hand, the XML structure is heavily impacted by those styling nodes, which makes their change detection complex. Most of the existing XML diff algorithms have difficulties representing text changes in nodes containing styling elements. Having no impact on the narrative structure, one of the possible solutions we propose to deal with inline style edits is to separate styling and textual change detection on XML. This is possible by converting style nodes to simple text using encryption (XML tags to text conversion). This way, the bold "hello" text is encrypted from initially hello to a pure text variant, for example _|b|_hello_|/b|_. This simplifies a lot the detection of the inline style changes as there is no need to operate with complex tree changes—everything is seen and treated as simple text.

In Figure 10, we can see an example where some parts of the (par. 1) node content is styled. Text "a" is made bold, "b" is made italic and underlined and "c" remains unchanged. By analysing the impact of this modification on XML, we can observe that the node $\alpha[0.0.1,2]$ changes from having one child text node "abc" $\alpha[0.0.1.0,3]$ to six nodes: $\beta[0.0.1.0,3]$, $\beta[0.0.1.0.0,4]$, $\beta[0.0.1.1,3]$, $\beta[0.0.1.1.0,4]$, $\beta[0.0.1.1.0.0,5]$ and $\beta[0.0.1.2,3]$. By encrypting all those newly added styling elements to simple text, we retrieve only one text node for (par. 1), which facilitates the change detection.

Once we have simple text nodes on both sides, we split them into two lists, *listOfTextA* and *listOfTextB*, by using the styling encrypted tags as separators. The two lists are then compared using the JAVA DiffUtils[‡‡] library that returns the *diffList* containing two parameters: difference content and type. With the type having one of three values (INSERT,

[‡‡]https://github.com/java-diff-utils/java-diff-utils.

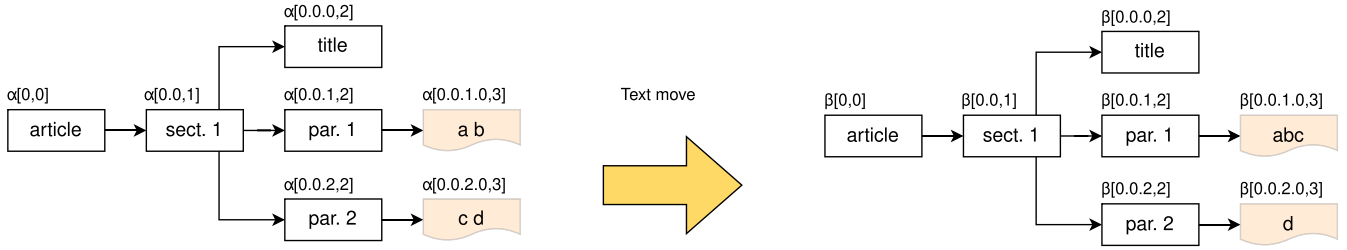

FIGURE 11 Impact of text move on we observe that moving some text content from the (par. 2) to the (par. 1) node has an impact on both nodes

DELETE or CHANGE), we are able to find inline style insertions, deletions and updates. In our example, DiffUtils will return three style inserts: bold, italic and underline. Deletions are observed when styling is removed and edits when styling type or styled text portion change. An example can be demonstrated in Albert Einstein's quote Logic will get you from A to Z; imagination will get you everywhere that is changed to Logic will get you from A to Z; imagination will get you everywhere . Note the bold part change from A to Z to from A to Z. DiffUtils will return two differences in this example: the bold part content change with "from" added and the text part changed with "from" deleted. We interpret this change as an inline style edit where the styled portion of text changed.

By using the described approach, jats-diff is able to detect three different inline style changes: "text-style-insert", "text-style-delete" and "text-style-update". The "text-style-update" is used for both style type changes and style content changes.

Using the inline styling "Level 2" pattern recognition in our previous example allows us to change the delta output from D+6I to three text-style-insert. The change consumer can understand this way that there is only inline style and no content changes applied by the author.

4.8 | Text move

As seen in Figure 4 (sect. 3), moving text portions from one node to another will result in four "Level 1" edit actions, 2(D+I). Making text moves within the document will be represented in a similar way to making real content changes which does not represent real modification made by the author. Within our example (see Figure 11), text "c" from node $\alpha[0.0.2.0, 3]$ has been moved to node $\alpha[0.0.1.0, 3]$. There, we can observe two change pairs: $\alpha[0.0.1.0, 3]$ - $\beta[0.0.1.0, 3]$ and $\alpha[0.0.2.0, 3]$ - $\beta[0.0.2.0, 3]$.

By applying the following mathematical formula for each of the detected change pairs between documents A and B, respectively annotated cα and cβ, we can evaluate if two specific change pairs fulfil the text move condition:

$$(\forall c\beta)(\forall c\alpha)c\beta[m,n].content - c\alpha[m,n].content \simeq c\alpha'[m,n].content - c\beta'[m,n].content. \quad (3)$$

The formula evaluates if each of the change pairs differences have common text between them. If true, then the text move pattern is detected. By running the formula on our example, the following scenario occurs: for each change pair between document A and B, $\alpha[m,n]$ and $\beta[m,n]$, evaluate if there is another change pair, $\alpha'[m,n]$ and $\beta'[m,n]$, where the content difference between both change pairs is similar. This results in verifying whether $\beta[0.0.1.0, 3]$.content - $\alpha[0.0.1.0, 3]$.content is similar or equals to $\alpha'[m,n]$.content - $\beta'[m,n]$ content. As both content differences will return "c", the text move condition is satisfied.

The delta output of such pattern detection is represented as one "Level 2" change named "text-move", having two information elements: "text-move_from" and "text-move_to".

4.9 | Citable node

As seen in Section 3.5, in addition to styling nodes, text nodes are also composed of references used to cite other nodes within the document. The most common citable nodes on JATS XML are bibliographies, but we also observe figures,

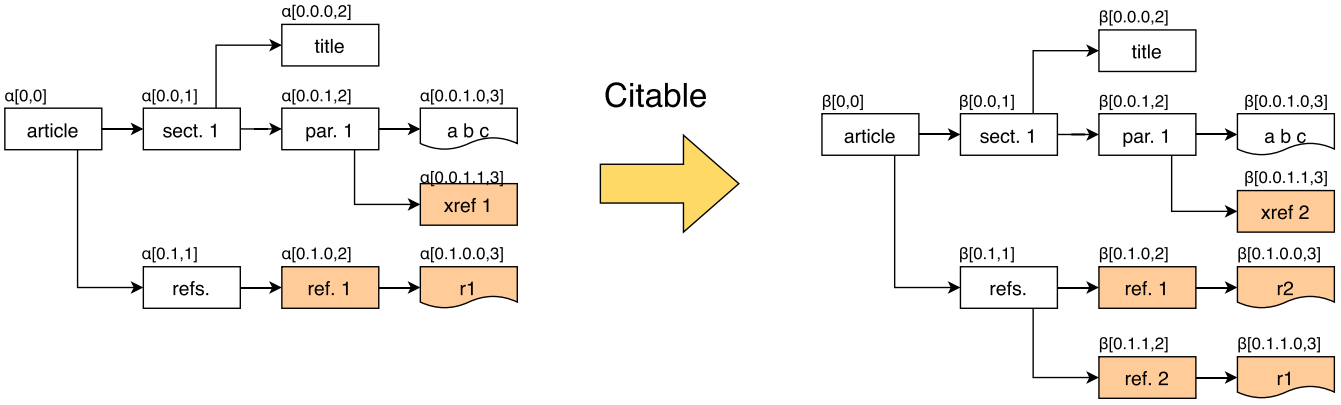

FIGURE 12 Impact of citable node insert on XML tree: we observe that adding a new citable node (bibliography reference) in the references list will change the auto-incremental values (label and id) of the following reference nodes, which directly impacts all citing (xref) nodes within the document

tables and sections. References are inserted as <xref> nodes containing the "ref-type" and "id" as attributes and the citing reference label as text. The "id" and the label are auto-incremental values dependent on the cited node appearance order; thus, inserting or removing a cited node automatically changes the remaining citable nodes' auto-incremental values. Those inducted changes are not interesting for the human reader and should be ignored as they are not directly made by the author.

Figure 12 shows the impact of adding one additional bibliography node $\beta[0.1.0, 2]$ at position one in the bibliography list. This change will move the initial position one bibliography node $\alpha[0.1.0, 2]$ to position two $\beta[0.1.1, 2]$, which implies that its label and attribute id auto-incremental numbering values will change from 1 to 2. This then has a direct impact on the citing xref node $\alpha[0.0.1.1, 3]$ that will change to $\beta[0.0.1.1, 3]$ with a different attribute "id" and a different label, citing the previous (ref. 1) that became (ref. 2) node. This kind of inducted changes is not interesting for the human reader, for whom the only relevant information is the insertion of the new bibliography. We propose here a solution on how to ignore those non-relevant changes and only keep the relevant changes made by the author. The main idea is to first scan the citable nodes list, detect insertions, deletions and the impact of those edits on their positions within the list. Citable node insertion will auto-increment and deletion will auto-decrement all following citable node id's and labels, which will then impact all citing references within the text nodes. A precise list containing the original and new citable node numbering values is then used to scan all citing references within the paragraphs and ignore the changes detected where the original numbering value is changed to the new value as an inducted change. This way, only real citing reference changes are kept in the delta output and the inducted ones are ignored.

5 | SIMILARITY INDEX BETWEEN THE TWO DOCUMENTS

Text nodes are the most important part of the text-centric XML documents. Having a similarity index between the two documents is beneficial for the final decision maker that can evaluate the impact that the modifications had on the textual content of the article. Due to the XML tree structure, using ordinary text diff algorithms is not possible, which is why we developed a simple and efficient algorithm that can calculate text similarity between modified text nodes and propagate upwards in the XML tree.

5.1 | Text similarity index propagation

After evaluating different text diff algorithms, we decided to embed the Jaccard index[22] and term frequency (TF)[23] that is calculated for every change node pair between document A and document B, regardless of whether the change is of

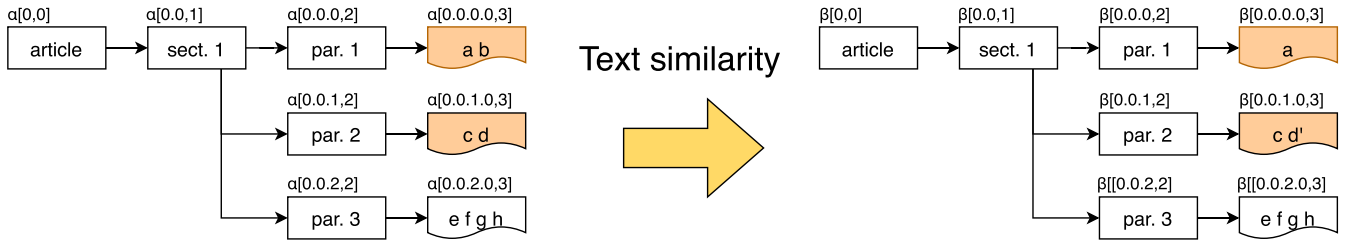

FIGURE 13 Text node modifications between two XML trees. Paragraphs (par. 1) and (par. 2) represent 25% each of the (sect. 1) content, while (par. 3) represents the remaining 50%. We can observe that text element pairs "ab", "cd" and "efgh", respectively, on text nodes $\alpha[0.0.1.0, 3]$, $\alpha[0.0.2.0, 3]$ and $\alpha[0.0.3.0, 3]$ represent 100% each of the text node content and 12.5% each regarding the (sect. 1) node content

"Level 1" or "Level 2". Once the similarity index is calculated for every change in the delta, those are propagated upwards in the XML tree by applying the following equation:

$$ParentSym = \sum_{n=0}^{n=N-1} Sn * In$$

using N as the total number of child nodes for a given parent, n as the child node number, S as their text similarity index and I as their text content ratio within a specific parent. A similar result can be achieved without using propagation but is more expensive in calculation power where, instead of completing one similarity calculation per change and propagating it upwards in the XML tree, one similarity calculation has to be done per node pair, increasing the number of similarity calculation operations.

Figure 13 presents two JATS XML versions where node $\alpha[0.0.0.0, 3]$ lost "b", representing half of its initial content, and node $\alpha[0.0.1.0, 3]$ had 50% textual content changes on "d" that represent 50% of the entire text node content. The delta output will show in this example one text update per modified node $\alpha[0.0.0.0, 3]$ and $\alpha[0.0.1.0, 3]$.

Using our similarity calculation algorithm, we could deduce that the text node $\alpha[0.0.0.0, 3]$ has a similarity of 50% compared to its document B version node $\beta[0.0.1.0, 3]$. Calculating the same for the $\alpha[0.0.1.0, 3]$ and $\beta[0.0.1.0, 3]$ nodes, we can deduce that the two text nodes have a similarity of 75% ("d" representing 50% of the entire text node, and modified by 50%). Once both similarities have been calculated, we can now propagate those upwards on the tree in order to measure the similarity between the two section trees $\alpha[0.0, 1]$ and $\beta[0.0, 1]$, both containing three paragraph nodes each. Here, you can find details on applying the previous formula to the Figure 13 example:

- N = 3 as sect. 1 has three child nodes;
- n0 represents $\alpha[0.0.0, 2]$; n1 represents $\alpha[0.0.1, 2]$; n2 represents $\alpha[0.0.2, 2]$;
- $S_{n0} = 0.5$; $S_{n1} = 0.75$; $S_{n2} = 1$;
- $I_{n0} = 0.25$; $I_{n1} = 0.25$; $I_{n2} = 0.5$.

$$ParentSym(\alpha[0.0, 1]) = 0.5 * 0.25 + 0.75 * 0.25 + 1 * 0.5 = 0.8125 \sim 81\%.$$

The example previously provided in Figure 13 is rather simple for comprehension purposes, but the same mathematical formula can be applied to more complex cases where, for example, we can observe text moves, node moves, structural upgrades, downgrades, and so forth as soon as we convert an xml sub-tree to simple text by concatenating their individual text nodes to a single text block.

5.2 | Element lists and special objects similarity

We previously saw how to calculate text similarity and propagate this similarity upwards on the XML tree. In JATS, having the text similarity makes sense for paragraphs, sub-sections and sections; it is, however, rather useless for other types of

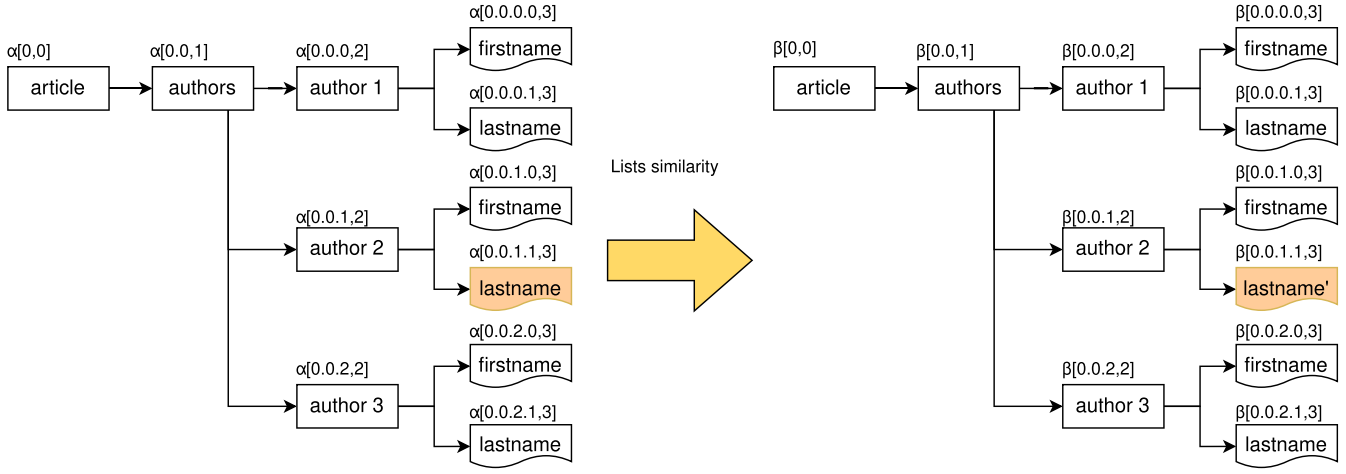

FIGURE 14 Text modifications on element lists. The (lastname) text node $\alpha[0.0.1.1, 3]$ was modified for the (author 2) node $\alpha[0.0.1, 2]$. Measuring the impact of this textual change on the parent (authors) node $\alpha[0.0, 1]$ using similarity propagation is not well reflecting the real changes made by the author. It is much better to use child element counters and represent their parent element similarity that way

sub-trees that are presented as lists (authors, references, tables and figures). For those, it makes more sense to express the similarity in number of changed/unchanged elements (4/5 authors, or 28/30 references).

Figure 14 shows a modification on the author 2 last name. If we use the similarity propagation to calculate the similarity between the "authors" parent nodes $\alpha[0.0, 1]$ and $\beta[0.0, 1]$, we would observe a similarity percentage that is highly influenced by the length of the modified last name. For one word last names, which are the most common, our two algorithms would return a similarity of 0% although only one character has changed in the last name. To accentuate this problem even more, let us assume authors 1 and 3 have very short first and last names, and author 2 has a short first and a very long last name. The author two last name could, for example, be composed of 10 characters, while the other first and last names are composed of only two characters, meaning that regarding its size, the author 2 last name is of the same size as all the other author text nodes.

We can conclude that text similarity calculation for those special types of JATS XML sub-trees can be inappropriate as this is purely based on text content. In such cases, it is much better to use child element counters and represent their parent element similarity that way. For this concrete example, we would say that authors have a similarity of 2/3, as two authors are exactly the same, and one was modified. In order to have even a higher precision, we propose to use the following semantic information for such lists:

- **Initial**: number of child elements on document A;
- **Final**: number of child elements on document B;
- **Modified**: number of modified child elements;
- **Deleted**: number of deleted child elements;
- **Inserted**: number of inserted child elements.

This way, the human reader can judge the changes applied by the author on such special sub-trees in an easy and convenient way. For additional information, consulting the delta output remains always available.

As the new edit actions detection and the similarity index have been described, we compare in the next section the new jats-diff algorithm with the other text-centric state-of-the-art XML diff algorithms.

6 | PERFORMANCE ANALYSES

The performance analyses of jats-diff[§§] are divided into two sub-sections, one on the information extraction capacity and the other on execution performance. This being a state-of-the-art algorithm, our main effort was set on the capacity to detect new edit patterns and change semantics extraction, rather than its implementation performance.

[§§] github.com/milos-cuculovic/jats-diff.

TABLE 1 Level 1/2 edit detection and similarity index calculation capacities for jats-diff, JNDiff, XYDiff and XCC

			Success				Nb. of actions			
Lev.	**Edit type**	**Human edits**	**jats-diff**	**JNDiff**	**XyDiff**	**XCC**	**jats-diff**	**JNDiff**	**XyDiff**	**XCC**
1	Text del.	Del. title part	✓	✓	✓	✓	1	1	1	1
1	Text ins.	Ins. title part	✓	✓		✓	1	1	2	1
1	Tree del.	Del. author	✓	✓	✓		1	1	1	5
1	Tree ins.	Ins. author	✓	✓	✓	✓	1	1	1	1
1	Tree attr.	Corr. author	✓	✓	✓	✓	1	1	1	1
2	Text upd.	Update title	✓	✓	✓	✓	2	2	1	1
2	Tree move	Move author	✓	✓	✓		2	2	2	8
2	Style ins.	Ins. bold	✓				1	3	4	2
2	Style del.	Del. bold	✓				1	3	4	2
2	Style type	Bold->italic	✓				2	2	4	2
2	Style cont.	Extend bold	✓				2	2	3	2
2	Upgrade	sect. 2.3->sect. 6	✓				2	6	4	2
2	Downgrade	sect. 5->sect. 2.4	✓				2	20	4	2
2	Split	Split one p	✓				3	2	3	2
2	Merge	Merge two p	✓				3	2	3	2
2	Text move	Move text	✓				2	4	4	2
2	Citable	Del. reference	✓				1	8	9	17
	Sim. index	Real-life edits	✓							

6.1 | Information extraction capacity

The initial evaluation phase consists of comparing the "Level 1" and "Level 2" information extraction capacities of the new jats-diff algorithm with JNDiff, XyDiff and XCC. During this performance analysis, we have created different XML file pairs, one original and one modified version of the same document. The modified version is composed of one of the human edits that is described in the "Human edit description" column. The output of each of the compared algorithms is then verified for its ability to detect the given edit type.

As seen in Table 1, jats-diff is able to detect all of the "Level 1" and "Level 2" edits. In addition, there is a similarity index calculated and propagated upwards of the XML for each change detected. This is followed by JNDiff with a perfect score for "Level 1" and a low score for "Level 2" edits. JNDiff can also detect "wrap" and "unwrap" edit patterns that are similar to style edits. This is followed by XyDiff with similar results in addition to text insert detection, where XyDiff mostly uses text updates to represent text inserts. This is because XyDiff calculates the longest common sub-string (LCS) and minimises the edit distance, which increases the complexity for a human reader to interpret the results. XCC follows XyDiff but with additional issues in detecting tree delete and tree move edits compared to JNDiff.

Concerning the delta output, Table 1 shows that jats-diff uses the minimal number of edit actions for almost all edit pattern detection. For a few of them where JNDiff, XyDiff or XCC output a lower number of edit actions, they are usually represented as a simple delete–insert combination which does not reflect the real changes made by humans at all, which we observe in the next section where we evaluate the delta file size for each of the jats-diff. If we push the theory of minimising the number of edit actions, one could think of using the delete–insert combination on the complete document, which will minimise the number of edit actions but maximise the delta file size.

6.2 | Execution performance

Although not critical in our working environment, algorithm performance for both execution time and memory usage stays important. Compared to other XML diff algorithms that are made with the purpose of comparing hundreds of

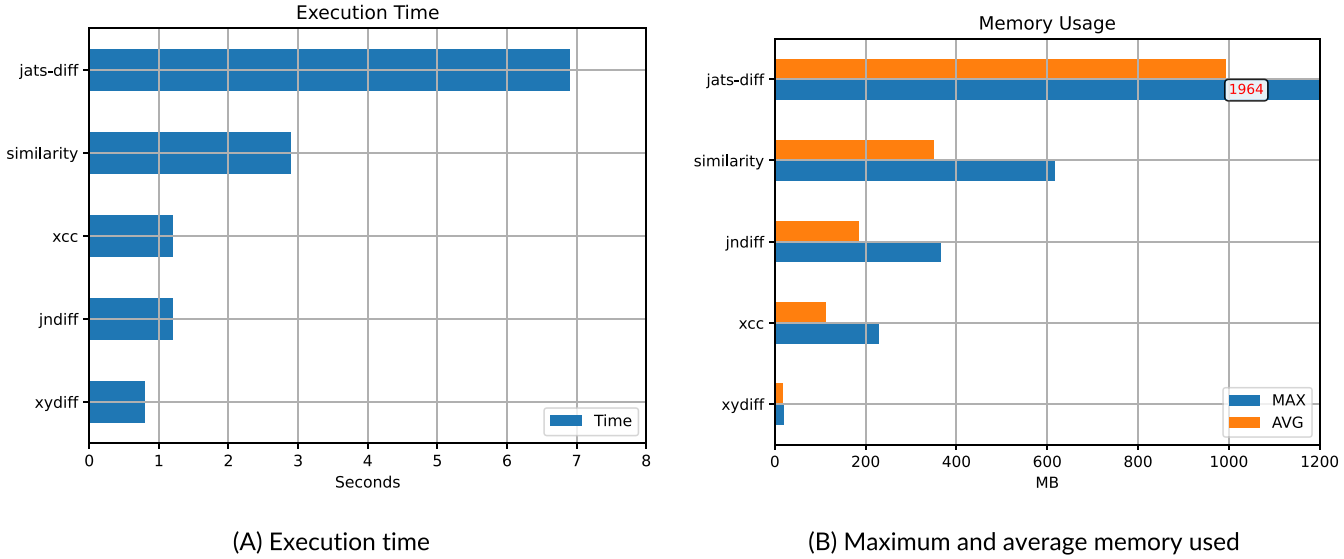

FIGURE 15 Execution time and memory usage for comparing two real-life author changes in JATS documents

thousands of XML documents (for example, XyDiff and webpage difference extraction), jats-diff will have to compare academic articles in JATS XML format during the publication process. Their number is counted in hundreds per day, which is far from the number of documents to be compared that XyDiff usually faces.

Using the JNDiff core functions for "Level 1" change detection, the execution time and memory usage of jats-diff is minimum at the level of JNDiff. Adding new "Level 2" pattern detection requires additional analysis of the detected "Level 1" differences; thus, it requires more time and memory for the algorithm to be executed. The similarity calculation and propagation is done separately by analysing the delta output and requires additional execution time.

We divide jats-diff in two parts: first, the "Level 1" and "Level 2" pattern detection, and second, the similarity calculation and propagation. JATS articles are large XML files that may vary from 100 KB to 400 KB. The tests[¶¶] were run on a JATS document pair A and B representing real-life author changes during a revision round, affecting every aspect of the article: the title, authors, affiliations, paragraphs, figures, tables and references.

Figure 15 shows both the execution time (Figure 15A) and the maximum and average memory used (Figure 15B) during a comparison of two real-life author changes in JATS documents. As expected, both parts of the new tools take more time and memory to perform the diff and semantics extraction; however, those are acceptable within our environment as the information consumers are humans with the aim to compare the original and revised version of academic articles, which does not need to be done in real time while the authors submit their revised version.

7 | DISCUSSION

The previous section illustrates how new "Level 2" edit patterns can be detected and how important is for the human reader to have a bijection of the changes made by the author and the modifications detected by analysing the two XML documents. In addition, the similarity index is also useful in order to get a broader picture of the impact that the differences made by the author have had.

The current algorithm still has possibilities for further improvements, one of which is when the document is heavily rewritten by the author in regard to sentence rephrasing and grammar changes. Taking JATS XML as an example, this can be requested by the reviewers in order to improve the article writing and organisation while keeping the sentence meanings. In order to cope with this, additional text change similarity indexes could be included in the algorithm that will allow the reader to distinguish between simple sentence rephrasing and sentence meaning. Those indexes could be calculated using topic model,[24] word2vec[25,26] and BERT.[27]

[¶¶] The evaluation was done on an Apple MacBook Pro (16-inch, 2019); Processor: 2.4 GHz 8-Core Intel Core i9; Memory: 32 GB 2667 MHz DDR4; SSD.

On another note, the current output of jats-diff still needs to be improved; information is visualised as XML delta outputs for "Level 2" and tree structure text output for the similarity index. Having a better and more understandable representation of this information could help the human reader even more. An idea would be to convert the XML document pair to HTML in order to have a readable representation of the document, in a similar way as done in versioning control systems (Git/sub-version, etc.). On top of this, we could use our change pattern detection and similarity index in order to visually annotate the changes made by the author, and also the textual impact those changes had on specific key elements of the document—for JATS as example, on specific sections.

Regarding execution time and memory use efficiency, our algorithm is less efficient than the state-of-the-art algorithms available. The main reason for this is the fact that we mostly focused on new edit pattern detection being the main algorithm efficiency indicator, rather than the execution time and memory use that we planned to improve during the industrialisation phase. "Level 2" changes are composed of a combination of "Level 1" basic edit operations. Each time a "Level 1" edit operation is observed, it gets analysed as per "Level 2" changes until the right sequence is found, or after going through all "Level 2" types and not finding a match. For each of those "Level 2" edit patterns, there is a specific independent part of code that needs to be executed, which increases time and memory usage. By sharing information between the "Level 2" detection steps, we could avoid constant re-parsing of the XML tree and reduce the execution time and improve the memory use efficiency.

As another practical use case of the presented jats-diff algorithm, we plan to further use it in assisting the final decision maker by correlating the effective changes provided by jats-diff and the expected changes extracted from the reviewer comments. In Figure 2, we can see that the jats-diff helps with comparing different article versions; however, the final decision maker should still read the reviewer comments and match them with the changes made by the authors. Having a bijection of the modifications made by the authors and the changes detected by comparing the JATS versions and, in addition, calculating the similarity index could be used by the final decision maker to understand the effective changes made by the author. In order to correlate those with the reviewer comments, Named Entity Recognition (NER) could be used in order to extract information from the reviewer comments on how, where and what should be changed in the article, representing expected changes. Once both pieces of information are available, we could correlate them and provide valuable insight to the final decision maker that will assess if the author made the changes requested by the reviewer. Potential candidates for the Named Entities regarding reviewer comments could be *Location*, *Action* and *Content*. *Location* could be matched with the change location within the article, *Action* with the modification pattern and *Content* with the change semantics. Additional author change information could also be used in order to further improve the software, one example being the author response letter that is usually uploaded together with the revised version of the article where the authors claim their changes and modifications as an answer to the reviewer comments. This information could be used to validate and enrich the modifications detected by the algorithm.

One last idea about a similar topic regarding document comparison semantics and their benefits would be its usage in version control systems (Git, sub-version, etc.) If those version control systems were to be aware of the programming language grammar they are detecting changes for, using change semantics to interpret some edit actions could be beneficial: for example, one class name change and its impact on other files where this initial class name is called could replace dozens or hundreds of detected modifications with one single edit which initiated all the other changes.

8 | CONCLUSION

In this article, we assessed the current text-centric state-of-the-art XML diff algorithms and their capacity to detect higher-level changes made by authors on XML documents, using JATS as a text-centric XML document type for testing purposes. Those algorithms all support "Level 1" edit pattern detection; however, their capacity to detect higher "Level 2" edit patterns is very limited.

We proposed a new XML diff algorithm called jats-diff, based on the existing JNDiff core functions, able to detect "Level 2" edit patterns which are closely related to text document edits made by the authors in their typesetter tools. This allows us to have a bijection of the author modifications and changes detected between two text-centric XML documents. In order to assess the need for the new "Level 2" edit pattern recognition, we started by evaluating different edit actions authors make during the document revision. We then assessed the impact that those edits have on XML and realised that there is a need for new edit pattern recognition: structural upgrade, downgrade, split and merge, inline style edit, text move and citable node edit. Afterwards, we proposed solutions on how to use change semantics on different combinations of existing "Level 1" and "Level 2" edit actions made by authors and how to recognise the new "Level 2" edit patterns.

We also proposed a way on how to calculate the XML node text similarity index and propagate it through the XML tree. Finally, we conducted a performance analysis comparing jats-diff with three other state-of-the-art XML diff algorithms: JNdiff, XyDiff and XCC. First, we evaluated the "Level 2" edit capacities where we could clearly observe that jats-diff is able to detect and represent all existing and new edit patterns described within this article. Afterwards, we evaluated the execution performance, where we measured the impact of the new "Level 2" edit detection and text similarity index computation on the time and memory used to compare two real-life author change documents.

Compared to existing XML diff algorithms that represent differences between two documents with a limited set of edit pattern recognition, jats-diff proposes an extended set of author modifications and changes detected by comparing the two XML versions. Among different use case scenarios, one of them is to help the Editor-in-Chief in the final decision-making process by automating the manual comparison of different article versions. This is achieved by offering enough details to the final decision maker to assess whether author changes follow the reviewer requirements. The similarity index computed on different parts of the document also provides a clearer picture to the final decision maker in order to understand which parts of the articles are the most impacted by the change.

As for the future of jats-diff, there is still work to be done on a better visualisation, execution and memory use performance, and additional information that can be added in order to enrich change detection.

ACKNOWLEDGEMENTS

The authors would like to thank Dr. Shu-Kun Lin and MDPI for their financial and technical support regarding our study. The authors would also like to thank Mr. Sasa Simic, IT Engineer from MDPI, for his work on the "Level 2" change implementation and Ms. Chams Azouz, student from the University of Haute-Alsace, for her work on the similarity index calculation implementation.

AUTHOR CONTRIBUTIONS

Sasa Simic: Level 2 edit detection implementation in JAVA. Chams Azouz: Level 3 edit detection implementation (similarity index) in JAVA.

DATA AVAILABILITY STATEMENT

The algorithm code and data are available on GitHub: https://github.com/milos-cuculovic/jats-diff.

ORCID


Milos Cuculovic https://orcid.org/0000-0003-2154-9652

REFERENCES

1. Tichy WF. The string-to-string correction problem with block moves. *ACM Trans Comput Syst (TOCS)*. 1984;2(4):309-321.
2. Miller W, Myers EW. A file comparison program. *Softw Pract Exper*. 1985;15(11):1025-1040.
3. Hunt JW, MacIlroy MD. *An Algorithm for Differential File Comparison*. Bell Laboratories; 1976.
4. Myers EW. An O (ND) difference algorithm and its variations. *Algorithmica*. 1986;1(1-4):251-266.
5. W3C Extensible Markup Language (XML); 2016. https://www.w3.org/XML
6. Selkow SM. The tree-to-tree editing problem. *Inf Process Lett*. 1977;6(6):184-186.
7. Chawathe SS, Rajaraman A, Garcia-Molina H, Widom J. Change detection in hierarchically structured information. *ACM SIGMOD Rec*. 1996;25(2):493-504.
8. Chawathe SS, Garcia-Molina H. Meaningful change detection in structured data. *ACM SIGMOD Rec*. 1997;26(2):26-37.
9. Cobena G, Abiteboul S, Marian A. Detecting changes in XML documents. Proceedings 18th International Conference on Data Engineering; 2002:41-52; IEEE, San Jose, CA.
10. W3C HtmlDiff; 2018. https://www.w3.org/wiki/HtmlDiff
11. Ciancarini P, Iorio AD, Marchetti C, Schirinzi M, Vitali F. Bridging the gap between tracking and detecting changes in XML. *Softw Pract Exper*. 2016;46(2):227-250.
12. Manning CD, Raghavan P, Schütze H. *Introduction to Information Retrieval*. Cambridge University Press; 2008.
13. Rönnau S, Scheffczyk J, Borghoff UM. Towards XML version control of office documents. Proceedings of the 2005 ACM Symposium on Document Engineering DocEng '05; 2005:10-19; Association for Computing Machinery, New York, NY. 10.1145/1096601.1096606
14. Rönnau S, Borghoff UM. Versioning XML-based office documents. *Multimed Tools Appl*. 2009;43(3):253-274.
15. Cuculovic M, Fondement F, Devanne M, Weber J, Hassenforder M. Change detection on JATS academic articles: an XML diff comparison study. Proceedings of the ACM Symposium on Document Engineering; 2020:1-10; ACM, New York, NY.
16. Rönnau S, Borghoff UM. XCC: change control of XML documents. *Comput Sci Res Develop*. 2012;27(2):95-111.

17. La Fontaine R. Standard change tracking for XML. Proceedings of the Balisage: The Markup Conference; 2014:5-8; Balisage Series on Markup Technologies.
18. Zhang S, Dyreson C, Snodgrass RT. Schema-less, semantics-based change detection for XML documents. Proceedings of the International Conference on Web Information Systems Engineering; 2004:279-290; Springer, New York, NY.
19. Dos Santos RC, Hara C. A semantical change detection algorithm for XML. SEKE 2007; 2007:438.
20. Oliveira A, Murta L, Braganholo V. Towards semantic diff of XML documents. Proceedings of the 29th Annual ACM Symposium on Applied Computing; 2014:833-838; ACM, New York, NY.
21. Oliveira A, Tessarolli G, Ghiotto G, et al. An efficient similarity-based approach for comparing XML documents. *Inf Syst*. 2018;78:40-57.
22. Jaccard P. Distribution of the alpine flora in the dranse's basin and some neighbouring regions. *Bull Soc Vaud Sci Nat*. 1901;37(1):241-272.
23. Luhn HP. A statistical approach to mechanized encoding and searching of literary information. *IBM J Res Dev*. 1957;1(4):309-317.
24. Du J, Jiang J, Song D, Liao L. Topic modeling with document relative similarities. Proceedings of the 24th International Joint Conference on Artificial Intelligence; 2015.
25. Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space; 2013. arXiv preprint arXiv:13013781.
26. Jatnika D, Bijaksana MA, Suryani AA. Word2Vec model analysis for semantic similarities in English words. *Proc Comput Sci*. 2019;157:160-167. Proceedings of the 4th International Conference on Computer Science and Computational Intelligence (ICCSCI 2019): Enabling Collaboration to Escalate Impact of Research Results for Society. https://www.sciencedirect.com/science/article/pii/S1877050919310713
27. Devlin J, Chang MW, Lee K, Toutanova K. Bert: pre-training of deep bidirectional transformers for language understanding; 2018. arXiv preprint arXiv:181004805.

How to cite this article: Cuculovic M, Fondement F, Devanne M, Weber J, Hassenforder M. Semantics to the rescue of document-based XML diff: A JATS case study. *Softw Pract Exper*. 2022;1-21. doi: 10.1002/spe.3074