A *diff* procedure for XML music score files

Computation and visualization of the differences between two xml music score files

Francesco Foscarin
CÉDRIC lab
CNAM Paris
Paris, France
francesco.foscarin@cnam.fr

Raphaël Fournier-S'niehotta
CÉDRIC lab
CNAM Paris
Paris, France
fournier@cnam.fr

Florent Jacquemard
INRIA
Paris, France
florent.jacquemard@inria.fr

ABSTRACT

Comparing XML music scores is an important task for many activities such as collaborative score editing, version control and evaluation of optical music recognition (OMR) or music transcription. Following the Unix *diff* model for text files, we propose an original procedure for computing the differences between two XML score files. It performs a comparison of scores at the notation (graphical) level, based on a new intermediate tree representation of the music notation content of a score and a combination of sequence- and tree-edit distances. We also propose a tool to visualize the differences between two scores side-by-side, using the music notation engraving library Verovio, and we employ it to test the procedure on an OMR dataset.

ACM Reference Format:
Francesco Foscarin, Raphaël Fournier-S'niehotta, and Florent Jacquemard. 2019. A *diff* procedure for XML music score files: Computation and visualization of the differences between two xml music score files. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Algorithms and tools for comparing text files have been around for more than forty years (Unix diff [13],[12]). Their purpose is to identify differences between two text files relatively similar (typically two versions of the same file), at the granularity level of lines with the *Longest Common Subsequence algorithm* (LCS), or at the granularity level of characters, with edit-distances. They output, in a normalized format, a list of differences between the two files that corresponds to our intuitive notion of difference. This list is called a *patch* or an *edit script* and, combined with either one of the two files, enables the reconstruction of the other file. Another application is to merge two files containing independent changes into a single file. Those tools are widely applied to text based documents nowadays, for software development, collaborative edition and version control systems.

Similarly to text files, comparing music scores is relevant for several applications. At a global level, it helps to define metrics capable of grasping the *distance* between two scores, for example to evaluate the quality of a transcription process (see [7, 17] for recent proposals). At a detailed level, it is very valuable for musicologist and developers of version control systems to get precise clues on the *locations* of the differences between scores (*e.g.,* between two editions of the same score). One difficulty that immediately arises for defining a *diff* tool for music scores is that, due to the nature/complexity of the music language, a music score contains multiple levels[9] that can be compared.

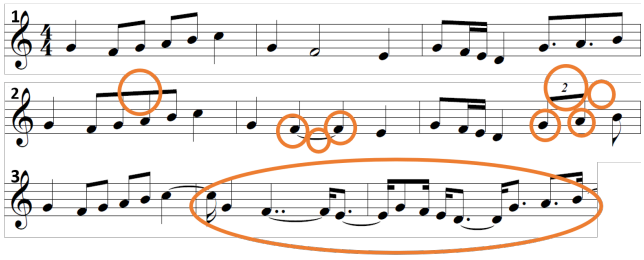

Figure 1: Three staves highlighting the differences between a music content comparison and a music notation comparison. The staves (1,2) have the same music content but differ in their music notation. The staves (1,3) differ only in one note (a longer C at the end of the first bar) from a music content point of view, but they are very different from a music notation point of view.

Most of the literature so far focuses on the *musical content* level, *i.e.* the sequence of events in the score that describe the intended production of sounds, independently from any encoding or rendering concern. The comparison is made with some extended *edit-distance* [2, 7, 16–18], or employing tree models [4, 21].

In the present paper, instead, we focus on the *musical notation* level, *i.e.* how the music is intended to be displayed in the score. This involves aspects such as note figures, beams, tuplets, ties/dots, pitches spelling, and ca be found in XML scores along with musical content (Figure 1). The authors of [3] , following the work of [15], observe that it does not make sense to apply directly the standard Unix `diff` utility to XML score files. The hierarchical structure of note beaming and tuplet grouping motivates the need to compare scores in terms of hierarchical structure (rather than lines or characters), by using on XML files a *tree-edit distance* based on tree nodes operations, as proposed by [25] or [6].

In this paper we propose a complete methodology for score comparison at music notation level, articulated in four main contributions: (i) the definition of an original tree-based representation of

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the music notation content of a measure, (ii) a new edit distance algorithm yielding a list of differences between two scores, (iii) a graphical tool to represent two scores side-by-side with coloured annotations and (iv) a visual evaluation on a OMR dataset.

The tree-based representation of the music notation content of a measure (Section 3.1), acts as a canonical model for disambiguation of the rhythmic content for XML score formats. Indeed, one score can have many different XML presentations that lead to the same graphical output, *e.g.*, by using or not a nested structure for beaming, interchanging the order of elements, *etc.* Moreover, using an intermediate representation allows to decouple our approach from the exact file format (in general MusicXml or MEI); anyway for practical reason explained in Section 4 we chose to work mainly with scores in MEI format [22].

The new edit distance algorithm is based on our original tree-structured representation and yield a list of differences between two scores first at measure level (Section 3.2) and then at a finer granularity level (Section 3.3), similarly to a text diff tool that first finds the different lines and then explores the differences between the words of each line. The distance is the size of the smallest list of modifications required to transform one score into the other, and it reflects visual differences between two scores.

The graphical tool (Section 4) represent two scores side-by-side and shows coloured annotations, highlighting the notation differences between the two scores, in a similar fashion as visual diff tools for text. While it is currently based on our algorithm only, it may be extended to present other metrics and should also be easy to integrate into other programs.

The entire workflow is apply on a dataset of scores produced by OMR and allows to easily spot the mistakes made by the transcription algorithm comparing to the manually annotated version (Figure 2).

Related Work. Our aim is to create for music notation a tool similar to the `diff` program by Douglas McIlroy [14], which compares texts. Besides its practical usage, the latter has motivated theoretical studies which led to efficient algorithms for edit-distances computation (*e.g.*, [19, 23]). We draw from this work the process of computing the differences with dynamic programming. Comparing music scores has been the object of previous work with different objectives: evaluation of OMR and automatic music transcription (AMT), collaborative score editing, *etc.*, leading to different approaches to the problem.

Knopke and Byrd [15] pioneered with their "work towards a musicdiff program", focusing on comparing several OMR outputs. They show that a traditional XML `diff` tool cannot be used *as is* on MusicXML, and categorize the comparison difficulties. They also propose a rudimentary visualisation tool. Our workflow is similar to [15] although our objective differ: we build a tool for users, and put more focus on the second step, with a dedicated representation and tree-edit distance computation procedure.

Recently, the subject has been studied in the MIR context of transcription evaluation. Cogliati [7] present an edit-distance similar to the Levenshtein distance, aiming at exhaustive research incorporating the characteristics of music notation into their metric. They set up a very sound evaluation process involving human experts, which highlights how algorithms are biased towards one category of differences or another. Our objectives are different since we are mostly interested in computing and displaying the whole list differences between scores, and not only evaluating an edit distance. Mcleod [17] improve slightly their work towards the goal of a joint metric for AMT performance, but they still do not take into account typesetting differences.

Aiming at improving collaborative editing of scores, [3] introduce the hierarchic paradigm that we followed, and worked with the Zhang-Shasha tree-edit distance [3]. They bring advances from theoretical computer science (*e.g.*, [5]) into the music notation community. They present one example on an MEI-encoded file, showing that writing a diff tool for XML music scores should be liable to the problems identified by [15] (*i.e.* XML coding style differences). We introduce an original tree-based representation to go beyond the problem.

Our last contribution, the graphical tool to visualize differences between scores, is inspired by similar tools for texts. They are now ubiquitous, either being standalone dedicated programs like Meld[1] (cross-platform) or FileMerge/opendiff (MacOS), or integrated into IDEs (Eclipse, IntelliJ). To the best of our knowledge, our work is the first proposition in the context of music scores.

2 MUSIC CONTENT DIFF

In this first section we consider a simple comparison between two monophonic scores at music content level. Intuitively, its purpose is to be able to compare scores according to *the way they sound*. We use a intermediate score representation similar to a piano-roll, with no overlapping notes.

This approach is not original, but it gives a good baseline to introduce our music notation comparison in Section 3.

2.1 Lossy Linear Score Representation

We assume given an XML score, composed of a single monophonic *part*. From the score we extract a sequence of triples $\langle pitch, onset, duration \rangle$, where the *pitch* is a MIDI value in $[0, 127]$ and *onset* and *duration* are expressed in fraction of beats.

This representation in sequences captures the semantic information of a music score, but loses other structural elements of music notation important to the musicians, such as metric cues indicated by grouping events with beams, ties, dots or chords, and information about pitch spelling.

2.2 String Edit-Distance

We compare two parts from two different scores by applying the Levenshtein edit distance to extracted sequences. It is based on the three following edit operations [24] on the above triplets (n represent a triple as above, and ε denotes the empty sequence):

$$\begin{array}{lcll} \varepsilon & \rightarrow & n & \text{insertion of a triplet,} \\ n & \rightarrow & \varepsilon & \text{deletion of a triplet,} \\ n & \rightarrow & n' & \text{substitution of a triplet by another.} \end{array}$$

Every such operation $\alpha \rightarrow \alpha'$ is associated a cost value $\delta(\alpha, \alpha')$. We assume that $\delta(\alpha, \alpha) = 0$ for all α, and triangle inequality. The cost of an edition sequence is the sum of the costs of all operations involved in the sequence. The *edit distance* $D(s, s')$ between

[1] http://meldmerge.org/

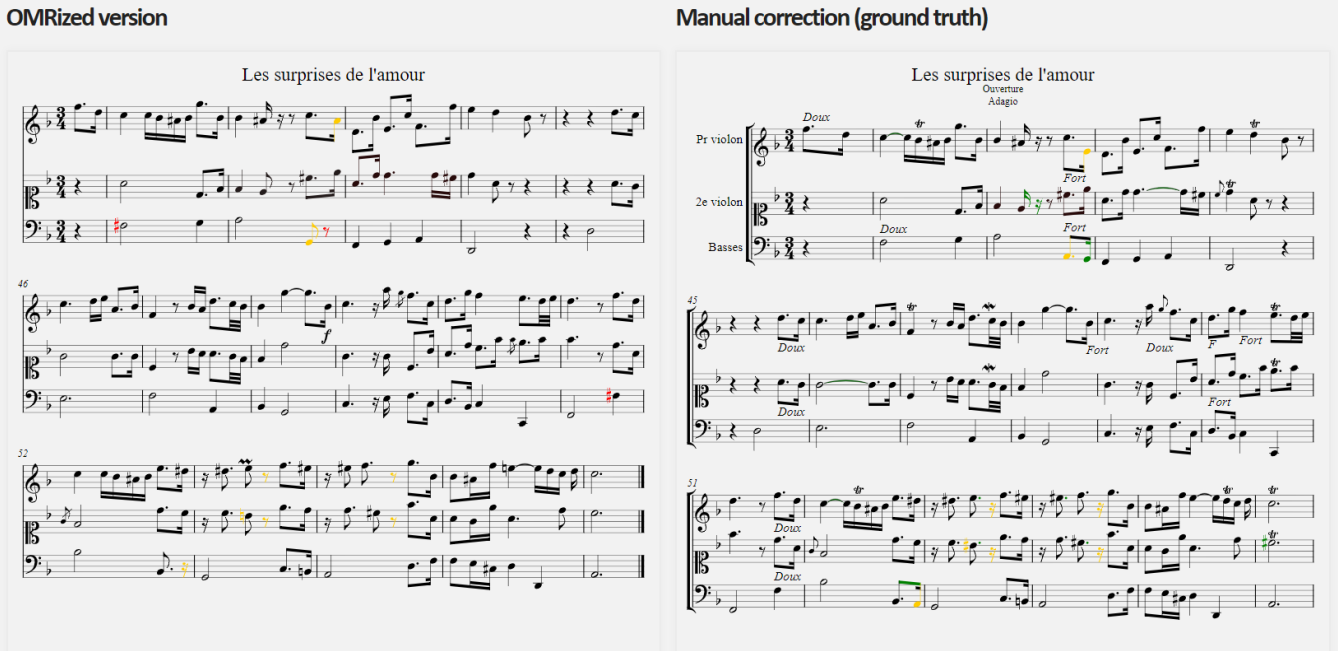

Figure 2: The score diff graphical tool with two scores side-to-side. The differences are annotated with different colors: (red) deletions, (yellow) insertion, (green): modifications. In this example it is used to visually evaluate the result of an OMR transcription.

two sequences of triplets s and s' is the minimal cost of an edition sequence transforming s into s'. It can be computed using the following dynamic programming equations, where $|s|$ denotes the length of s, and $n.s$ represents a sequence made of triple n followed by the subsequence s.

$$
\begin{array}{lcll}
D(\varepsilon,\varepsilon) & = & 0 & \\
D(\varepsilon,n'.s') & = & D(\varepsilon,s')+\delta(\varepsilon\to n') & \text{(ins)}\\
D(n.s,\varepsilon) & = & D(s,\varepsilon)+\delta(n\to\varepsilon) & \text{(del)}\\
D(n.s,n'.s') & = & & \\
\multicolumn{4}{r}{\min\left\{\begin{array}{llll} D(s,s') & + & \delta(n\to n') & \text{(subst)}\\ D(n.s,s') & + & \delta(\varepsilon\to n') & \text{(ins)}\\ D(s,n'.s') & + & \delta(n\to\varepsilon) & \text{(del)}\end{array}\right.}
\end{array}
$$

3 MUSIC NOTATION DIFF

We now present our main contribution, which is based on another representation, more structured, whose purpose is to enable a comparison of scores according to *the way they look*.

This representation (Section 3.1) is based on tree structures akin to the rhythm-trees for rhythm notation [1, 10, 21], but more tightly related to the music notation information, in order to have an unique representation of the graphical content of the XML scores.

We consider an XML score as a nested structure: on the top level we have a list of score parts (usually one for each instrument), each part is a list of measures, each measures is a list of voices and each voices as a list of *general notes* (notes, rests or chords) that occurs sequentially without overlapping. For the sake of presentation we consider from now scores with a single part, where each measure contains a single voice. We explain in Section 4 how we can generalize our technique to a polyphonic score with multiple parts.

The first step is to transform every score into a sequence of trees, one tree (more precisely, one pair of trees, as detailed below) for each bar. Then we proceed in two stages:

(1) *comparison at the bar level* (Section 3.2), with an alignment of identical bars (*i.e.* they have the same trees), with a Longest-Common-Subsequence (LCS) algorithm.
(2) *comparison inside-the-bar* (Sections 3.3, 3.4): a more in-depth processing of unaligned bars, using purposely designed tree-edit distances algorithms that descends into the bar representations in order to identify differences.

3.1 Tree-Based Representation

We present an abstract model of the music notation content of bars in music scores. It is designed as an intermediate structure for our algorithms of *diff computation*, and it is not meant to be a format for data exchange. It can be imported from and exported to XML (see Section 4). This model is based on the following two types of tree representations, corresponding to two aspects of music notation.

3.1.1 Beaming Trees. A *beaming tree* represents the notes in a measure and the beams and ties between them. In particular, we encode in the leaves the information about the note-figures (notehead, dots and ties); the remaining information (*i.e.* beams and flags) are encoded in the tree structure (Figure 3).

Every leaf of a beaming tree represent an *event*: a note, a chord or a rest. In the XML representation, a chord can be considered in a single voice, if all the notes of the chord have the same duration.

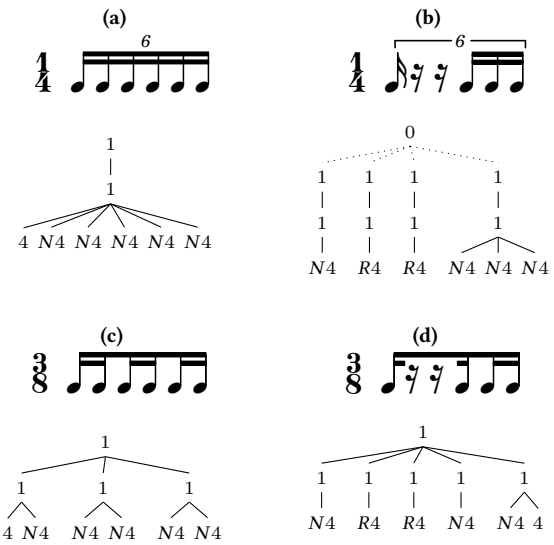

Figure 3: Examples of beaming trees. To simplify the presentation we use a pitch-less notation with notes and rests denoted resp. by N, R. Rests with 2 flags are positioned in the tree like notes with 2 flags, while notes beamed together are represented by a different tree structure.

Each leaf is labeled by a triple containing the following information (see Figure 3 for an illustration):

- *pitches*: a sequence where each element contains a *pitch name* (letter + octave number for notes or *R* for rests), an *alteration value* among 'none' and 𝄫, ♭, ♮, ♯, 𝄪 and a *tie* flag indicating if the corresponding note is tied to an element of the previous event in the tree (following a depth first traversal). The sequence is ordered from the lowest to the highest pitch; in case of a single note or a rest (not a chord), *pitches* is a singleton sequence;
- *notehead*: a value representing a fraction of whole note, *e.g.*, 1, 2, 4, represent respectively 𝅝, 𝅗𝅥, and ♩ for a note and 𝄻, 𝄼, 𝄽 for a rest;
- *dots*: a natural number in $[0, 3]$ that specify the number of dots.

We define the size $\|pitches\|$ of a sequence *pitches* as the sum of the sizes of all its elements, where the size $\|pitch\|$ of an element *pitch* is the number of graphical items it contains, *e.g.*, a note D4 ♯ with a tie will have size of 3, while a rest R will have size of 1.

In the internal nodes, we use the two symbols $'0'$ and $'1'$ for the encoding of *beams*; intuitively a label $'1'$ indicates that there is a beam between the notes in the subtrees under the node. More precisely, let the *weight* of a node n be the number of nodes labelled $'1'$ on the path from n to the root of the tree (including n itself). Let n and n' be events in two successive leaves and ℓ be their least-common-ancestor in the tree (it is always defined and unique). The number of beams between n and n' is either $weight(\ell)$, if in the two paths $\ell \to n$ and $\ell \to n'$ all inner nodes are labeled by $'1'$, or it is $'0'$ otherwise.

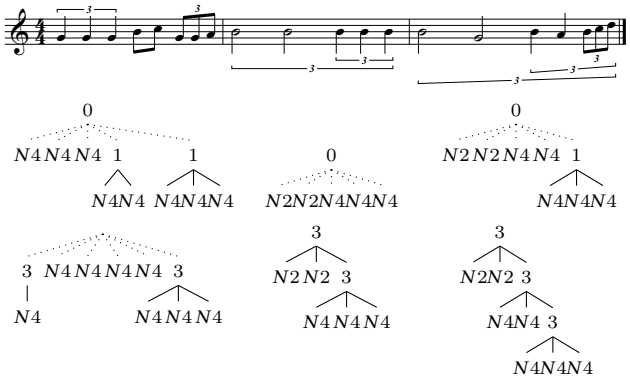

Figure 4: Beaming and tuplet tree for nested triplets (in pitch-less representation). In the first bar, the sequence of the first three notes is labeled by 3 (with a bracket), the sequence of the last three notes is also labeled by 3 (no bracket needed because these three notes are beamed, according to the beaming tree), and the two middle notes do not have a tuplet label.

3.1.2 Tuplet Trees. The *tuplet trees*, illustrated on Figure 4, represent the locations of the numbers and brackets explicitly defining in a score the nature of the tuplets. Every inner node of a tuplet tree is either unlabeled (this is denoted with a label · in figures), or labeled by a pair of integers $i : j$, with $i, j \geq 2$, meaning *'i elements in the time of j'*, or by a single integer i if j can be omitted [11]. The leaves are labeled like for the beaming trees: Every tuplet tree is associated to a companion beaming tree with the same sequence of leaves (the difference between them is only in the structure).

3.2 Longest Common Subsequences of Parts

Assume that we are given two parts extracted from two different scores. We represent each part by a sequence of pairs $p = \langle bt, tt \rangle$, one pair per bar, where bt is a beaming tree and tt is a companion tuplet tree.

Similarly to the line-by-line comparison procedures for text files [13], and considering a bar as the analogous of a line in a text file, we shall align identical bars from the two parts, by computing their longest common subsequence (LCS), whose size is defined by the following recursive Dynamic Programming equations (with similar notations as in Section 2.2):

$$\begin{array}{lcll} LCS(\varepsilon, s') & = & LCS(s, \varepsilon) = 0 & \\ LCS(p.s, p'.s') & = & 1 + LCS(s, s') & \text{if } p \equiv p' \\ & = & \max\big(LCS(p.s, s'), LCS(s, p'.s')\big) & \\ & & & \text{otherwise.} \end{array}$$

In the second line, $p \equiv p'$ means that the trees in pairs $p = \langle bt, tt \rangle$ and $p' = \langle bt', tt' \rangle$ are isomorphic, *i.e.* that $bt = bt'$ and $tt = tt'$. According to the representation presented in Section 3.1, it means the corresponding bars can be considered identical.

The above equation can be adapted in order to compute a maximal *alignment* of identical pairs between the two sequences. Formally, denoting the two sequences by $s = p_1, \ldots, p_m$ and $s' = p'_1, \ldots, p'_n$, the alignment is made of two strictly increasing sequences

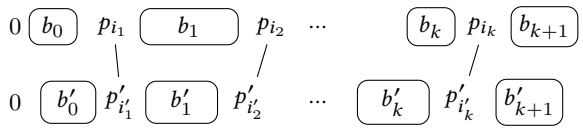

Figure 5: LCS and difference blocks.

of indices of same length, $(i_0 = 0, i_1, \ldots, i_k)$ and $(i'_0 = 0, i_1, \ldots, i'_k)$ such that for all $0 \leq j < k$,

(i) $p_{i_{j+1}} \equiv p'_{i'_{j+1}}$, and

(ii) the pairs in the respective sub-sequences $b_j = p_{i_j+1}, \ldots, p_{i_{j+1}-1}$ and $b'_j = p'_{i_j+1}, \ldots, p'_{i_{j+1}-1}$ are pairwise distinct (each of these sequences might be empty if $i_{j+1} = i_j + 1$ or $i'_{j+1} = i'_j + 1$).

We call each pair $\langle b_j, b'_j \rangle$ as above a *difference block* (see Figure 5) and the objective of the next two subsections is to perform a fine comparison between blocks.

3.3 Inside-the-bars Comparison

In the following, for a tree t (either a beaming tree or tuplet tree), $\|t\|$ denotes the sum of the sizes of labels of all the nodes n in t (size of labels is 1 by default, $\|n\|$ for a leaf as defined in Section 3.1.1, 0 for unlabeled nodes). For a sequence $s = t_1, \ldots, t_k$ of trees, $\|s\| = \sum_{i=1}^{k} \|t_i\|$.

3.3.1 Comparison of Beaming Trees. The following recursive equations define an edit distance (denoted by B) between sequences of beaming trees. The distance between two trees is defined by the particular case of singleton sequences. In these equations, $t.s$ and $t'.s'$ denote two beaming tree sequences, with $t = a(s_0)$ and $t' = a'(s'_0)$, where s_0 and s'_0 might be empty.

We call a tree *atomic* if it is composed of a single leaf node (*i.e.* it represents an event). We call a tree $t = a(s_0)$ *unary* if s_0 is a singleton sequence (*i.e.* the root has a single child).

$$
\begin{array}{lcl@{\quad}l}
B(\varepsilon, s') & = & \|s'\| & B(s, \varepsilon) = \|s\| \\
B(t.s, t'.s') & = & & \\
\multicolumn{4}{l}{\min \left\{ \begin{array}{lcll}
B(s, t'.s') & + & \|t\| & \text{(del-tree)} \\
B(t.s, s') & + & \|t'\| & \text{(ins-tree)} \\
B(s_0.s, t'.s') & + & 1 & \text{(del-node)} \\
\multicolumn{4}{l}{\text{if } t \text{ non-atomic, } t' \text{ atomic, or } t \text{ unary, } t' \text{ not unary}} \\
B(t.s, s'_0.s') & + & 1 & \text{(ins-node)} \\
\multicolumn{4}{l}{\text{if } t \text{ atomic, } t' \text{ non-atomic, or } t \text{ not unary, } t' \text{ unary}} \\
B(s_0.s, s'_0.s') & + & \delta(a, a') & \text{(descend)} \\
\multicolumn{4}{r}{\text{if } t, t' \text{ non-atomic}} \\
B(s, s') & + & A(t, t') & \text{(leaf)} \\
\multicolumn{4}{r}{\text{if } t, t' \text{ atomic}}
\end{array} \right.}
\end{array}
$$

where $\delta(a, a')$ is a defined by $\delta(a, a') = 0$ if $a = a'$, and $\delta(a, a') = 1$ otherwise.

The cases (del-tree) and (ins-tree) correspond to the deletion or insert of a whole subtree. The cases (del-node) and (ins-node) with one unary tree correspond to a difference of one beam between the trees. The other cases descend down to leaves and then compare atomic trees t and t' with the distance $A(t, t')$ defined as follows (p, n, d stand respectively for the attributes *pitches*, *noteheads*, *dots* of Section 3.1):

$$A(t, t') = D\big(p(t), p(t')\big) + \delta\big(n(t), n(t')\big) + |d(t) - d(t')|$$

where the first literal $D\big(pitches(t), pitches(t')\big)$ is the Levenshtein distance defined in Section 2.2, using the following operations costs:

$$
\begin{array}{rcl}
\delta(\varepsilon \to pitch) & = & \delta(pitch \to \varepsilon) = \|pitch\| \\
\delta(pitch \to pitch') & = & \delta(n, n') + \delta(alt, alt') + \delta(tie, tie')
\end{array}
$$

where n, n', *alt*, *alt'* and *tie*, *tie'* are the respective name, alteration and tie values of *pitch* and *pitch'*.

3.3.2 Comparison of Tuplet Trees. Tuplet trees have a simpler structure than beaming trees (*e.g.*, no unary nodes). We compare them with the classical Zhang-Sasha equations [25] for computing a tree-edit-distance (denoted by T), *i.e.* the smallest number of node-edit operations to transform a tuplet tree into another (instead of the algorithm of Section 3.3.1 which is specific to beaming trees).

$$
\begin{array}{lcl@{\quad}l}
T(\varepsilon, s') & = & \|s'\| & T(s, \varepsilon) = \|s\| \\
T(t.s, t'.s') & = & & \\
\multicolumn{4}{l}{\min \left\{ \begin{array}{lcll}
T(s_0.s, t'.s') & + & 1 & \text{(del-node)} \\
T(t.s, s'_0.s') & + & 1 & \text{(ins-node)} \\
T(s, s') + T(s_0, s'_0) + \delta(a, a') & & & \text{(subst-node)}
\end{array} \right.}
\end{array}
$$

where $\delta(a, a')$ is as in Section 3.3.1 for inner nodes and $\delta(a, a') = A(a, a')$ when a and a' are leaf labels.

3.4 Comparison of Difference Blocks

We compute the distance between two difference blocks b, b' using the following recursive equations, similar to the equations of Section 2.2.

We recall that a block is a sequence of pairs (bar representations) of the form $p = \langle bt, tt \rangle$, where bt and tt are respectively a beaming tree and a tuplet tree.

$$
\begin{array}{lcl@{\quad}l}
D(\varepsilon, b') & = & \|b'\| & D(b, \varepsilon) = \|b\| \\
D(p.b, p'.b') & = & & \\
\multicolumn{4}{l}{\min \left\{ \begin{array}{lcll}
D(b, b') & + & \Delta(p, p') & \text{(edit-bar)} \\
D(p.b, b') & + & \|p'\| & \text{(ins-bar)} \\
D(b, p'.b') & + & \|p\| & \text{(del-bar)}
\end{array} \right.}
\end{array}
$$

where, for $p = \langle bt, tt \rangle$ and $p' = \langle bt', tt' \rangle$:

$$\Delta(p, p') = B(bt, bt') + T(tt, tt') - corr(p, p') \qquad (1)$$

is the edit distance between two bar representations; $corr(p, p')$ is a correction of $T(tt, tt')$ to avoid to count the same differences twice (see Section 4).

Finally, the distance between two parts is the sum of distances between all the difference blocks.

4 IMPLEMENTATION & EXPERIMENTS

The construction of the models of Section 2.1 and 3.1 from MEI scores, as well as the implementation of the algorithms proposed, have been performed in Python3, on the top of the Music 21 toolkit [8].

The computations of the string edit distances of Sections 2.2, 3.2 (LCS) and 3.4 are performed with an iterative computation and run in time $O(m*n)$, where m, n is the length of the two sequences.

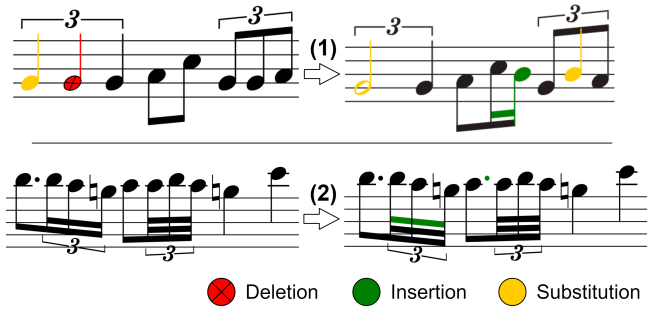

Figure 6: Visualization of differences between two bars computed by the algorithms of Section 3. The upper example (1) is taken from Figure 4 and manually modified to create some differences, while the lower example is a real-life example from an incorrect OMR score import, compared to the correct score.

The computation of the tree-edit distances of Section 3.3, is implemented by recursive functions, with a worse case time complexity in $O(n^4)$. We use a table of solutions in order to avoid to recompute the function on the same inputs and hence reduce the computation time. Nevertheless, efficiency is not a crucial issue in this case, because the difference blocks are typically small (*e.g.,* in OMR evaluation), or such that $b = \varepsilon$ or $b' = \varepsilon$ (insertion or deletion of bars).

A difference in our score is modeled as a triple $(operation, t, t')$ where *operation* is a string identifying the performed operation (e.g. ins-node), and t, t' are the subtrees transformed by that operation (or ε is case of insertion/deletion). In the implementation, the above recursive equations are actually extended in order to compute, beside the edit distance, a list of differences. It means that the functions B, T *etc.* return pairs made of a cost value along with a diff list. To that respect, the operator $+$ in the recursive equations makes the sum of the cost values and the concatenation of *diff lists*, and the min selects the pair with the smallest cost value. The function $corr(p, p')$ in Section 3.4 uses the difference lists returned by B and T in order to detect intersection that may occur, in particular for leaves. These intersections are removed (operator $-$ in (1)) both from the cost value and the diff list. Thus, we avoid accounting twice for the same difference (*e.g.,* a pitch modification must be counted only once, even if it is present in the result of both B and T).

The elements of *diff lists* are pairs of references to elements in the two scores, plus an identifier of the edit operation. To have an unique reference for the elements in the scores, we use the unique id associated to each element in the MEI format of music scores. We store those ids in the leaves of our tree representation, and they are very useful later in order to display the differences between scores. The XML ids are an important feature, not present in the MusicXML format, and the main reason we always work with MEI score files in input (a MusicXML score can be transformed into a MEI score as a preprocessing).

The generalization of our algorithms for single-part, single-voice, single-note comparison to a polyphonic score is implemented as following:

- single-voice → multiple-voices: we first couple the voices considering the ones with shorter distance and then we run our algorithm on each voice independently. If the number of voices of the measure in the two scores is different, we return the eventual voice-insertion and voice-deletion annotation.
- single-part → multiple-parts: we first couple the parts by part-name/instrument and we run our algorithms on each couple of parts independently. In case of added parts, we return the eventual part-insertion and part-deletion annotation. Coupling parts considering the ones with smallest distance is also possible, but it drastically slow down the computation, especially if the number of parts is high (*e.g.,* orchestral scores).

After computing the edit distances and list of differences between two given scores, we display, using Verovio [20], the two scores side-by-side, highlighting the differences.

We run experiments in order to test the usability of our workflow for two scenarios: collaborative edition of scores, and OMR evaluation. For the first use case, we considered made-up scores with some typical modifications (bar insertions, note pitch modifications, *etc.*) in a context of versioning. For the second scenario, we used a corpus produced by the Bibliothèque nationale de France (BNF) composed of 21 *ouvertures* of Jean-Philippe Rameau, each with an *OMR*ized version (from manuscripts) and its manual correction, that we compare with our tool[2].

Our algorithms has shown promising results in both cases, highlighting very fine differences in the scores (*e.g.,* beams and dots), as illustrated on Figure 6. The evaluation of OMR results were not possible for many scores, due to quality issues in the XML files [9], *e.g.,* a tie between two non-consecutive notes, a note duration that exceed the total duration of the bar, *etc.* Our model is designed to handle only correctly encoded scores, and such *faulty* notations result in an error.

5 CONCLUSION

We have presented a procedures to compare XML scores at music notation level that is based on a tree intermediate representation of measures and on a combination of string edit distance and tree edit distance.

In contrast with most approaches in the state of the art, our methodology produces a list of differences that can be highlighted directly on the scores, and alleviates the user from the tedious task of manual scores comparison. Our algorithms gives also a similarity metrics, but this metric is only used to compute the list of differences and it is not useful per se. For this reason, comparison to other approaches (*e.g.,* techniques for melodic similarity) would not be relevant.

Experiments show that our approach allows to correctly displays very fine differences in the scores. Technical improvements are still needed in other stages (XML parsing, error detection/correction, complex score visualization) to improve usability.

[2] see our supplementary material page at https://anonymoususer12042019.github.io/ for details.

REFERENCES

[1] Carlos Agon, Karim Haddad, and Gérard Assayag. 2002. Representation and rendering of rhythm structures. In *Second International Conference on Web Delivering of Music (WEDELMUSIC)*. IEEE, 109–113.

[2] Julien Allali, Pascal Ferraro, Pierre Hanna, Costas Iliopoulos, and Matthias Robine. 2009. Toward a general framework for polyphonic comparison. *Fundamenta Informaticae* 97, 3 (2009), 331–346.

[3] Christopher Antila, Jeffrey Treviño, and Gabriel Weaver. 2017. A hierarchic diff algorithm for collaborative music document editing. In *Third International Conference on Technologies for Music Notation and Representation (TENOR)*.

[4] José F Bernabeu, Jorge Calera-Rubio, José M Iñesta, and David Rizo. 2011. Melodic identification using probabilistic tree automata. *Journal of New Music Research* 40, 2 (2011), 93–103.

[5] Philip Bille. 2005. A survey on tree edit distance and related problems. *Theor. Comput. Sci.* 337, 1-3 (2005), 217–239. https://doi.org/10.1016/j.tcs.2004.12.030

[6] Gregory Cobena, Serge Abiteboul, and Amelie Marian. 2002. Detecting Changes in XML Documents. In *Proceedings 18th International Conference on Data Engineering*. IEEE, 41–52.

[7] Andrea Cogliati and Zhiyao Duan. 2017. A Metric for Music Notation Transcription Accuracy. In *Proceedings of the 18th International Society for Music Information Retrieval Conference (ISMIR)*. 407–413.

[8] Michael Scott Cuthbert and Christopher Ariza. 2010. music21: A toolkit for computer-aided musicology and symbolic music data. In *ISMIR*. International Society for Music Information Retrieval.

[9] Francesco Foscarin, David Fiala, Florent Jacquemard, Philippe Rigaux, and Virginie Thion. 2018. Gioqoso, an online Quality Assessment Tool for Music Notation. In *4th International Conference on Technologies for Music Notation and Representation (TENOR'18)*.

[10] Francesco Foscarin, Florent Jacquemard, Philippe Rigaux, and Masahiko Sakai. 2019. A Parse-based Framework for Coupled Rhythm Quantization and Score Structuring. In *International Conference on Mathematics and Computation in Music*. Springer, 248–260.

[11] Elaine Gould. 2011. *Behind Bars: The Definitive Guide to Music Notation*. Faber Music.

[12] Paul Heckel. 1978. A technique for isolating differences between files. *Commun. ACM* 21, 4 (1978), 264–268.

[13] James Wayne Hunt and M Douglas MacIlroy. 1976. *An algorithm for differential file comparison*. Technical Report. Bell Laboratories Murray Hill.

[14] James W Hunt and M Douglas McIlroy. 1976. An algorithm for differential le comparison. *Computing Science Technical Report* 41 (1976).

[15] Ian Knopke and Donald Byrd. 2007. Towards musicdiff: A foundation for improved optical music recognition using multiple recognizers. *Dynamics* 85, 165 (2007), 121.

[16] Kjell Lemström. 2000. *String Matching Techniques for Music Retrieval*. Ph.D. Dissertation. University of Helsinki, Department of Computer Science.

[17] Andrew Mcleod and Mark Steedman. 2018. Evaluating Automatic Polyphonic Music Transcription. In *Proceedings of the 19th International Society for Music Information Retrieval Conference, ISMIR 2018, Paris, France, September 23-27, 2018*. 42–49.

[18] Marcel Mongeau and David Sankoff. 1990. Comparison of musical sequences. *Computers and the Humanities* 24, 3 (1990), 161–175.

[19] Eugene W Myers. 1986. AnO (ND) difference algorithm and its variations. *Algorithmica* 1, 1-4 (1986), 251–266.

[20] Laurent Pugin, Rodolfo Zitellini, and Perry Roland. 2014. Verovio: A library for Engraving MEI Music Notation into SVG. In *ISMIR*. 107–112.

[21] David Rizo. 2010. *Symbolic music comparison with tree data structures*. Universidad de Alicante.

[22] Perry Roland. 2002. The music encoding initiative (mei). In *Proceedings of the First International Conference on Musical Applications Using XML*, Vol. 1060.

[23] Esko Ukkonen. 1985. Algorithms for approximate string matching. *Information and control* 64, 1-3 (1985), 100–118.

[24] Robert A Wagner and Michael J Fischer. 1974. The string-to-string correction problem. *Journal of the ACM (JACM)* 21, 1 (1974), 168–173.

[25] Kaizhong Zhang and Dennis Shasha. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing* 18, 6 (1989), 1245–1262.