Studying Structural Regularities through Abstraction Trees

Filippo Carnovalini[1,2], Nicholas Harley[2], Steve Homer[2], Antonio Rodà[1], and Geraint A. Wiggins[2,3]

[1] University of Padova, Italy
[2] Vrije Universiteit Brussel, Belgium
[3] Queen Mary University of London, UK
filippo.carnovalini@dei.unipd.it

Abstract. "Structure" is a somewhat elusive concept in music, despite being of extreme importance in a variety of applications. Being inherently a hidden feature, it is not always explicitly considered in algorithms and representations of music. We propose a hierarchical approach to the study of musical structures, that builds upon tree representations of music like Schenkerian analysis, and adds additional layers of abstraction introducing pairwise comparisons between these trees. Finally, these representations can be joined into probabilistic representations of a music corpus. The probability distributions contained in these representation allow us to use concepts from Information Theory to show how the structures we introduce can be applied to musicological and music information retrieval applications.

Keywords: Structure, Schenkerian Analysis, Music Representations

1 Introduction

"Structure" is a term that, even only considering music, can assume a variety of meanings. One common use of this term relates to form: the chaining of different sections to create a longer musical piece where some sections are repeated, with or without variations. Another use is more related to shorter melodic fragments, and relates to how a melody can be divided into periods, phrases, and motifs. In this latter case, a musicologist who wishes to analyze structure will try to divide the music into smaller segments and to find similarities, repetitions, inversions, parallel movements or otherwise links between these segments.

Despite the variety of information that can be gathered by such a process, this kind of analysis is often overlooked in algorithms and representation for computational musicology or music information retrieval. This becomes especially evident when computational systems try to generate novel music after learning some features of music from a given dataset of human compositions [1]. However complex or elegant the model used for the generation, we are still far from obtaining results that are on a par with the starting material. This is generally due to the fact that while these models can capture some aspects of the music they analyze, e.g., typical melodic motifs, they fail to capture the entirety of the hierarchical, structural aspects of music. In many cases, this leads to algorithms that generate music that sounds reasonable for a short time, but seems to ”wander off” as the length of the generated piece increases [2, 3, 5].

In the present work we aim to propose a novel solution to this, using a representation that builds upon existing hierarchical representations of music inspired by Schenkerian Analysis, but that goes further by operating pairwise comparisons between trees reducing small segments of music. These comparisons allow to find the kind of reuse of melodic material that we mentioned before. These structured comparisons are then further abstracted, by considering the comparisons operated on a variety of pieces rather than a single piece. These new representations describe structural regularities within a corpus, and leverage approaches from Information Theory to allow us to isolate more interesting features within the representation and to operate comparisons with other pieces.

1.1 Related Work

This work is linked to a variety of computational musicology applications. Some analysis tools that also abstract tree-like structures based on existing theories of music, such as Schenkerian analysis [12, 13] or GTTM [8, 9], are well known in literature; however, in our proposal the tree representations are not the final goal, but a means of intra-piece comparison, allowing analysis the internal repetition structure. The output is similar to other algorithms meant for form analysis [18], but to our knowledge our approach has never been applied in that field. Finally, Wiggins [19] provides an in-depth theoretical base for the relevance of this approach to music analysis and generation, but does not specify any practical approach to perform the proposed analyses.

2 Representations

The algorithms we describe require the input corpus to be made of monophonic melodies with chord annotations over the melody (lead sheets). We used MusicXML format, but other formats could be appropriate as well.

The first step of the process is to segment the input pieces into segments of equal duration. Depending on the level of detail that is being investigated, a length of one or two measures can be appropriate.

Each segment is then individually analyzed, and from each a tree representing melodic reductions is built, following the algorithm described in [13, 17, 4]. This approach uses a sliding window that passes over the notes in the segment, and every time the window contains two or more notes, one of these is deemed the most important according to the tonality, the metric position, and the current chord. This note is kept and the others are eliminated, and the remaining note is made longer to fill the void left by the other notes. At the end of each iteration a new simplified melody is created, and the window is enlarged. At the end of the last iteration, only one note should be left. By stacking the obtained simplifications, a tree similar to those created by analyses such as the ones contained in GTTM [11] or in Schenkerian Analysis [15] is obtained. For this reason we call this tree *Schenkerian Tree* or simply *Sk_tree*.

Once the Sk_trees are built, it is possible to operate pairwise comparisons between them, comparing their roots and recursively comparing the child nodes. In particular, each node in a Sk_tree either represents a note that is present in the original melody (a leaf node) or a note that was created in the process of iterative simplification described above. In the latter case, this node has two or more children, representing the notes that were present in the previous simplification, one of which is kept and the others are eliminated, and it is possible to consider the musical interval of these children. The comparison between nodes of different Sk_trees depends on the content of this interval. The comparable features of these intervals include difference in number of child nodes, difference in pitch intervals between the children, difference in the direction of the children's intervals, or differences in the way that the Schenkerian reduction was performed (for example if the note that was saved was to the left or to the right of the child interval). Since this new structure is based on differences between different sections, we call it *Difference Tree* or *Diff_tree* Figure 1 shows how a Diff_tree can be built from the comparison of two Sk_trees.

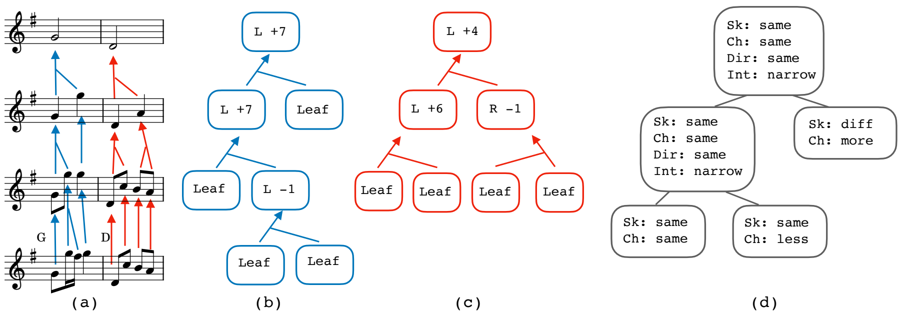

Fig. 1. One example of a simplified diff_tree (d), constructed from two sk_trees (b and c), each representing one of the two measures of the excerpt (a).

For each input piece, a set of $\frac{(n-1)(n)}{2}$ Diff_trees are produced, where n is the number of segments the input piece is divided into. This number is due to the fact that the segments are not compared with themselves nor the segments prior to them, but only with segments that come later in the musical piece, so not all the n^2 possible comparisons are performed. Each Diff_tree is then labelled to indicate which segments are compared in that tree (e.g. if the first and fourth segments are compared, the tree is labeled '0-3'). Once the Diff_trees for a set of pieces are produced, the Diff_trees that share the same label (i.e., that refer to segments in the same position but coming from different pieces) can be joined together into a single tree, that abstracts the general development of that particular comparison. For this reason, we call this representation an *Abstraction Tree* or *Abs_tree*. The procedure works as follows: a new node which will be the root of the Abs_tree is created. Starting from the root of all considered Diff_trees, for each of the possible features, the new node annotates all the possible values that the feature assumes in all the given Diff_trees and the number of occurrences of those val-

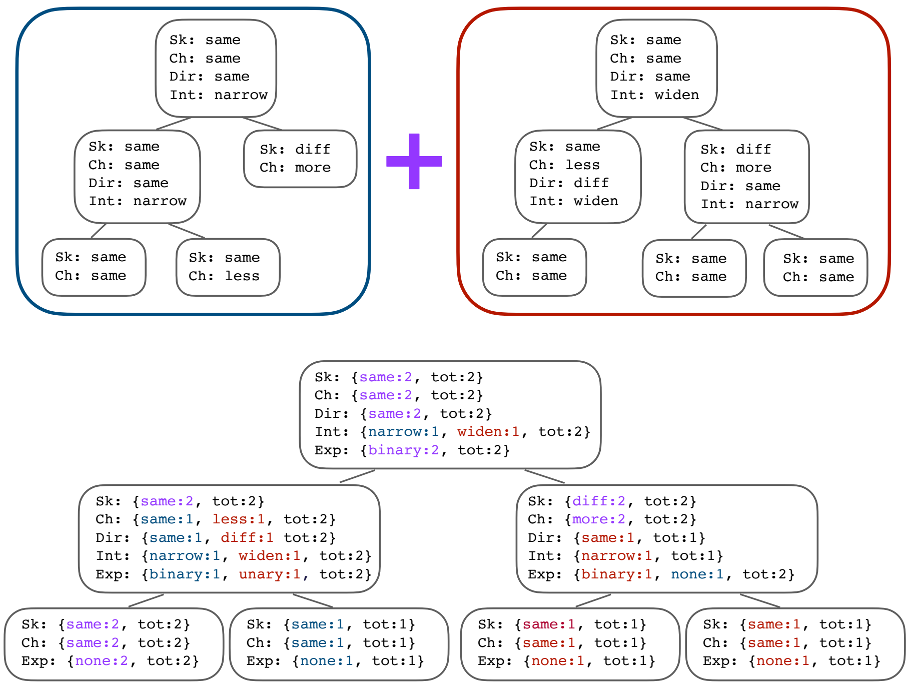
Sk: same Ch: same Dir: same Int: narrow
Sk: same Ch: same Dir: same Int: narrow
Sk: diff Ch: more
Sk: same Ch: same
Sk: same Ch: less
+
Sk: same Ch: same Dir: same Int: widen
Sk: same Ch: less Dir: diff Int: widen
Sk: diff Ch: more Dir: same Int: narrow
Sk: same Ch: same
Sk: same Ch: same
Sk: same Ch: same
Sk: {same:2, tot:2} Ch: {same:2, tot:2} Dir: {same:2, tot:2} Int: {narrow:1, widen:1, tot:2} Exp: {binary:2, tot:2}
Sk: {same:2, tot:2} Ch: {same:1, less:1, tot:2} Dir: {same:1, diff:1 tot:2} Int: {narrow:1, widen:1, tot:2} Exp: {binary:1, unary:1, tot:2}
Sk: {diff:2, tot:2} Ch: {more:2, tot:2} Dir: {same:1, tot:1} Int: {narrow:1, tot:1} Exp: {binary:1, none:1, tot:2}
Sk: {same:2, tot:2} Ch: {same:2, tot:2} Exp: {none:2, tot:2}
Sk: {same:1, tot:1} Ch: {same:1, tot:1} Exp: {none:1, tot:1}
Sk: {same:1, tot:1} Ch: {same:1, tot:1} Exp: {none:1, tot:1}
Sk: {same:1, tot:1} Ch: {same:1, tot:1} Exp: {none:1, tot:1}

Fig. 2. A simplified Abs_tree built from the two Diff_trees on top. For readability, the tree reports frequencies of occurence and the total number of observation rather than the probabilities that need to be computed with the Bayesian Estimator. The colors represent the tree from which each value for each feature comes: blue for left-side tree, red for the right-side one, and purple for those values that are found in both.

ues. The new node also annotates how many children the roots of the given Diff_trees have, as a new separate feature. Then the algorithm repeats recursively as long as at least one Diff_tree node has at least one child node. Once the recursion process is complete, it is possible to compute the probability of each value v for each feature in each node, using the following Bayes Estimator [16]:

$$p(v) = \frac{occ(v) + \beta}{tot + \beta * size} \tag{1}$$

where $occ(v)$ is the number of occurrences of the value v, while tot and $size$ are respectively the total of samples for that feature and the number of possible distinct values for that feature. β was set to 1. All the features for which tot is less than a certain threshold (we used 5 in the experiments described below) can be removed as those features would entail too little information about the corpus. The nodes that remain empty because of this can be removed as well. Figure 2 shows a simplified example of an Abs_tree built from two Diff_trees. This process basically creates a probability distribution for each node, consisting of a set of stochastic variables depending on the features present in the Diff_trees. Because of this the Abs_tree is similar to a Markov Model, but rather than having the probability distributions vary in time depending only on the previous state, the only feature that determines the distribution is the position on the tree. For this reason, it would be misleading to think of an Abs_tree as a chain or an automaton, and it is best to only view it as a static probabilistic description of a musical corpus.

3 Applications

In this section, we demonstrate, through example applications, the utility of the abstract representations we introduced. To do so, we will apply the explained structures to tasks relevant to computational musicology and music information retrieval.

3.1 Regularity Detection within a Corpus

As a first example, we show how the above introduced Abstraction Trees can be inspected to find regularities within a given corpus. As an example corpus, we will use the Leone dataset, a set of twenty-four baroque allemandes [6]. Of these twenty-four we selected the twenty that have sixteen measures in total, to make comparisons easier thanks to the equal length. All the pieces were divided into two-bar segments and the abstraction trees pairwise comparing the segments were built as described above. The procedure produces a large amount of data which is difficult to interpret on its own, but the tools of information theory can help find the most relevant features. For each feature in each node, it is possible to compute the normalized entropy (efficiency) of the probability distribution it describes, which gives a useful indication of the importance of that feature within the corpus. The lower the entropy, the more strictly that feature describes a recurring element in the corpus.

For example, as can be seen in Figure 3, looking at the mean normalized entropy of all the constructed abstraction trees, it becomes evident that the tree comparing segments 0 and 2 (shown in figure) and the one comparing segments 4 and 6 are the most regular ones. In those trees, almost each feature is set to “same”, meaning that there are little or no differences between the above mentioned pairs of segments. Indeed, the phrases in this corpus tend to repeat after 4 measures (the distance between the start of segments 0 and 2), and that is captured by the abstraction tree. Moreover, while the phrases repeat, their ending is varied to make for more definitive phrase endings. This is captured in the abstraction tree comparing segments 1 and 7 (shown in figure) where the left side of the tree shows a repetition like the one described above, but in the right side of the tree the most relevant feature is the one describing the ending grade, which is usually the tonic, as expected from the closing of a musical period.

3.2 Genre Discrimination

The following example uses another metric commonly used in Information Theory. While entropy is related to regularity in a probability distribution, Information Content gives an indication of how unexpected a certain outcome is with respect to a given

Entropy:	Mean
0-1	0,62
0-2	0,23
0-3	0,60
0-4	0,67
0-5	0,59
0-6	0,67
0-7	0,56
1-2	0,62
1-3	0,49
1-4	0,62
1-5	0,57
1-6	0,61
1-7	0,50
2-3	0,60
2-4	0,67
2-5	0,59
2-6	0,67
2-7	0,57
3-4	0,64
3-5	0,61
3-6	0,63
3-7	0,60
4-5	0,65
4-6	0,23
4-7	0,54
5-6	0,67
5-7	0,53
6-7	0,54

```
[0-2]
Root ◊ entropy: 0.14
sk: {'same': 20, 'total': 20, 'entropy': 0.06}
struct: {'same': 20, 'total': 20, 'entropy': 0.07}
end: {'same': 20, 'total': 20, 'entropy': 0.06}
ch_int: {'same': 20, 'total': 20, 'entropy': 0.07}
dir: {'same': 20, 'total': 20, 'entropy': 0.06}
int: {'same': 20, 'total': 20, 'entropy': 0.07}
cont: {'same': 20, 'total': 20, 'entropy': 0.07}
expansion: {'binary': 20, 'total': 20, 'entropy': 0.07}
├──  Right ◊ entropy: 0.13
├    sk: {'same': 20, 'total': 20, 'entropy': 0.06}
├    struct: {'same': 20, 'total': 20, 'entropy': 0.07}
├    end: {'same': 20, 'total': 20, 'entropy': 0.06}
├    ch_int: {'same': 18, 'total': 18, 'entropy': 0.08}
├    dir: {'same': 18, 'total': 18, 'entropy': 0.07}
├    int: {'same': 18, 'total': 18, 'entropy': 0.08}
├    cont: {'same': 18, 'total': 18, 'entropy': 0.08}
├    expansion: {'binary': 16, 'total': 16, 'entropy': 0.09}
│   ├──  Right ◊ entropy: 0.2826050595866655
│   ├    sk: {'same': 16, 'total': 16, 'entropy': 0.07}
│   ├    struct: {'same': 15, 'more': 1, 'total': 16, 'entropy': 0.34}
│   ├    end: {'same': 15, 'total': 15, 'entropy': 0.08}
│   ├──  Left ◊ entropy: 0.23
│   ├    sk: {'same': 16, 'total': 16, 'entropy': 0.07}
│   ├    struct: {'same': 16, 'total': 16, 'entropy': 0.09}
│   ├    end: {'same': 16, 'total': 16, 'entropy': 0.07}
│   ├    ch_int: {'same': 11, 'total': 11, 'entropy': 0.12}
│   ├    dir: {'same': 11, 'total': 11, 'entropy': 0.10}
│   ├    int: {'same': 11, 'total': 11, 'entropy': 0.12}
│   ├    cont: {'same': 11, 'total': 11, 'entropy': 0.12}
│   └    expansion: {'binary': 6, 'unary': 1, 'total': 7, 'entropy': 0.52}
├──  Left ◊ entropy: 0.17
├    sk: {'same': 20, 'total': 20, 'entropy': 0.06}
├    struct: {'same': 20, 'total': 20, 'entropy': 0.07}
├    end: {'same': 20, 'total': 20, 'entropy': 0.06}
├    ch_int: {'same': 20, 'total': 20, 'entropy': 0.07}
├    dir: {'same': 20, 'total': 20, 'entropy': 0.06}
├    int: {'same': 20, 'total': 20, 'entropy': 0.07}
├    cont: {'same': 20, 'total': 20, 'entropy': 0.07}
└    expansion: {'binary': 15, 'ternary': 1, 'total': 16, 'entropy': 0.34}
    ├──  Right ◊ entropy: 0.21
    ├    sk: {'same': 16, 'total': 16, 'entropy': 0.07}
    ├    struct: {'same': 15, 'more': 1, 'total': 16, 'entropy': 0.34}
    ├    end: {'same': 15, 'total': 15, 'entropy': 0.08}
    ├    ch_int: {'same': 9, 'total': 9, 'entropy': 0.13}
    ├    dir: {'same': 9, 'total': 9, 'entropy': 0.12}
    ├    int: {'same': 9, 'total': 9, 'entropy': 0.13}
    ├    cont: {'same': 9, 'total': 9, 'entropy': 0.13}
    ├──  Left ◊ entropy: 0.16
    ├    sk: {'same': 16, 'total': 16, 'entropy': 0.07}
    ├    struct: {'same': 16, 'total': 16, 'entropy': 0.09}
    ├    end: {'same': 16, 'total': 16, 'entropy': 0.07}
    ├    ch_int: {'same': 12, 'total': 12, 'entropy': 0.11}
    ├    dir: {'same': 12, 'total': 12, 'entropy': 0.09}
    ├    int: {'same': 12, 'total': 12, 'entropy': 0.11}
    ├    cont: {'same': 12, 'total': 12, 'entropy': 0.11}
    └    expansion: {'binary': 6, 'total': 6, 'entropy': 0.17}
```

```
[1-7]
Root ◊ entropy: 0.71
struct: {'same': 20, 'total': 20, 'entropy': 0.07}
expansion: {'binary': 20, 'total': 20, 'entropy': 0.07}
├──  Right ◊ entropy: 0.65
├    sk: {'same': 20, 'total': 20, 'entropy': 0.06}
├    struct: {'same': 18, 'less': 2, 'total': 20, 'entropy': 0.38}
├    to: {'5': 1, '1': 14, '6': 2, '7': 1, 'total': 18, 'entropy': 0.49}
├    expansion: {'binary': 17, 'unary': 1, 'ternary': 1, 'total': 19, 'entropy': 0.54}
│   ├──  Right ◊ entropy: 0.43
│   ├    sk: {'same': 18, 'total': 18, 'entropy': 0.07}
│   ├    struct: {'same': 15, 'less': 3, 'total': 18, 'entropy': 0.47}
│   ├    to: {'5': 1, '1': 11, '6': 1, '7': 1, 'total': 14, 'entropy': 0.50}
│   ├──  Left ◊ entropy: 0.27
│   ├    sk: {'same': 19, 'total': 19, 'entropy': 0.06}
│   ├    to: {'1': 6, '7': 1, 'total': 7, 'entropy': 0.32}
│   ├    end: {'diff': 7, 'total': 7, 'entropy': 0.15}
│   └    expansion: {'binary': 6, 'total': 6, 'entropy': 0.17}
├──  Left ◊ entropy: 0.66
├    sk: {'same': 20, 'total': 20, 'entropy': 0.06}
├    struct: {'same': 20, 'total': 20, 'entropy': 0.07}
└    expansion: {'binary': 17, 'ternary': 2, 'total': 19, 'entropy': 0.39}
    ├──  Right ◊ entropy: 0.78
    ├    struct: {'same': 17, 'less': 2, 'total': 19, 'entropy': 0.39}
    ├    expansion: {'binary': 8, 'total': 8, 'entropy': 0.14}
    ├──  Left ◊ entropy: 0.71
    ├    sk: {'same': 19, 'total': 19, 'entropy': 0.06}
    ├    struct: {'same': 17, 'more': 1, 'less': 1, 'total': 19, 'entropy': 0.54}
    └    expansion: {'binary': 8, 'total': 8, 'entropy': 0.14}
```

Fig. 3. A table summing up the mean entropy of each abstraction tree derived from the corpus, and some examples of abstraction trees as a text output of the software. Only the first three levels were kept for readability. The labels of the tree represent the compared segments: for example "0-1" means that the first two bars of a piece are compared to measures 3 and 4, since in this case each segment was two measures long.

probability distribution. Since musical cognition is strongly related to expectation [10], this metric becomes a relevant indicator when analyzing musical pieces [14]. In this experiment, we learnt a set of abstraction trees from the 20 allemandes taken from the corpus mentioned above, and the difference trees from a set of 20 reels from the Nottingham Dataset [7], and from 20 jazz pieces composed between 1921 and 1930 taken from the EWLD corpus [17]. The abstraction trees contain probability distributions for each feature in each node, while difference trees can be considered as outcomes for the same features. This means that for each feature it is possible to compute the information content. To give a single measure of the total information content of a difference tree compared to an abstraction tree, the mean of all the features in a node is computed to give the information content of a single node, and the mean of all the information contents across nodes is computed to give the general information content of a tree. This latter mean is also weighted by the mean entropy of the nodes, and by an added coefficient that makes nodes lower in the tree less important than nodes in the upper part of the tree ($depth\_k$ in the formula below). The total formula is described below, where $p(\mathit{diff\_tree\_feature})$ represents the probability $p(v)$ (computed according to the estimator 1) of the value v found for the considered feature f in the diff_tree node.

$$ic(tree) = \frac{\sum_{node \in tree} ic(node) \frac{1}{ent(node)} * depth\_k(node)}{\sum_{node \in tree} \frac{1}{ent(node)} * depth\_k(node)} \tag{2}$$

$$ic(node) = \sum_{f \in node} \frac{-\log_2(p(\mathit{diff\_tree\_feature}))}{ent(f)} \tag{3}$$

$$ent(node) = \frac{\sum_{f \in node} ent(f)}{number\_of\_features\_in\_node} \tag{4}$$

$$ent(f) = \sum_{v \in alphabet(f)} -\log_2(p(v))p(v) \tag{5}$$

Figure 4 shows the results of the comparisons. Since computing the information content of a piece included in the abstraction tree would be an unfair advantage, similarly to the bias one would get by evaluating a machine learning model using the same dataset that was used for learning the model, an approach similar to a k-fold validation was used. The allemande corpus was split into four parts of 5 pieces, and the information content of each piece was computed with respect to the abstraction trees built solely on the 15 pieces outside the considered allemande's group. This means that there were actually four sets of abstraction trees built each on a different subset of 15 pieces. The values for the other two groups (Jazz and Nottingham) were computed on all the four sets and the mean is reported.

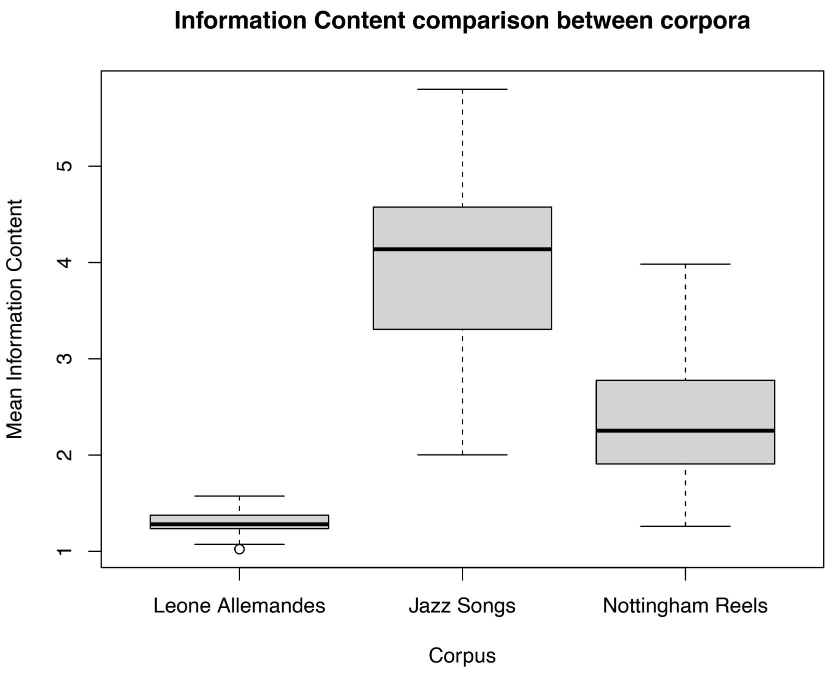

Fig. 4. Comparison of the mean information content computed from each of the three sets of twenty musical pieces. Mean refers to the mean of the trees of a single piece, rather than the mean of an entire corpus.

The results clearly show that this approach is capable of detecting the structural differences between the corpora. The allemandes show a strong structural regularity, that is not found in the other pieces. As expected, the reels from the Nottingham dataset are less unexpected than the jazz pieces, since they too have some structural regularities that are not always found in jazz pieces. It is worth noting that being based on difference trees, what this system captures is the general structure of the piece and how much reuse of melodic material is present, rather than comparing for instance the regularities in the melodies and how typical they are for each genre, possibly making this metric a complementary indicator that could be used in combination with other approaches in genre detection.

4 Conclusions

In this work, we have introduced a novel representation of musical content aimed at encoding in a hierarchical manner features relating to musical structure. The approach builds upon tree-based methods inspired by Schenkerian analysis, but adds additional abstraction layers to describe regularities in a musical corpus rather than in a single piece. The final representation uses probability distributions, that can be analyzed using tools from Information Theory as we show through two examples in the latter part of the paper. The system as described here is capable of detecting regularities in a simple allemande corpus, but the general approach can be adapted to a variety of specific algorithms: the algorithm used for the construction of Schenkerian trees could be changed, as well as the set of features used to compare them and build the Difference trees, potentially adapting to different kinds of music and different analysis needs. One of the biggest drawbacks of the current implementation is that it is based on a fixed window length, making it harder to capture smaller structural features. An algorithm for segmentation could be embedded in the system to detect the best subdivision of a piece, but the general algorithm would need to be modified to adapt to segments of unequal length.

Applying this approach to other structures opens interesting directions for future works, but even keeping the system as presented now it is possible to further investigate its applications. One of the motivations behind this work was to find descriptions of music structure that can be used when generating computer-composed music. In this scenario, Abstraction trees could offer a useful metric to describe how well a generated musical piece respects the typical structure of a certain style by computing the Information Content in comparison with a goal corpus.

While this work is not meant to give a comprehensive descriptor of all musical aspects of a corpus, we believe that this contribution might help formalizing some aspects of music that are sometimes overlooked in favour of more prominent aspects such as melody, rhythm, and harmony.

Acknowledgments. FC is funded by a doctoral grant by University of Padova and by a mobility grant by Fondazione Ing. Aldo Gini. NH, ST and GW received funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme.

References

1. Briot, J.P., Hadjeres, G., Pachet, F.D.: Deep Learning Techniques for Music Generation. Computational Synthesis and Creative Systems, Springer International Publishing, New York, NY (2020). https://doi.org/10.1007/978-3-319-70163-9, https://www.springer.com/gp/book/9783319701622
2. Briot, J.P., Pachet, F.: Deep learning for music generation: challenges and directions. Neural Computing and Applications **32**(4), 981–993 (Feb 2020). https://doi.org/10.1007/s00521-018-3813-6, https://doi.org/10.1007/s00521-018-3813-6
3. Carnovalini, F.: Open Challenges in Musical Metacreation. In: Proceedings of the 5th EAI International Conference on Smart Objects and Technologies for Social Good. pp. 124–125. ACM, Valencia Spain (Sep 2019). https://doi.org/10.1145/3342428.3342678, http://dl.acm.org/doi/10.1145/3342428.3342678
4. Carnovalini, F., Rodà, A.: A Multilayered Approach to Automatic Music Generation and Expressive Performance. In: 2019 International Workshop on Multilayer Music Representation and Processing (MMRP). pp. 41–48. IEEE, Milano, Italy (Jan 2019). https://doi.org/10.1109/MMRP.2019.00016, https://ieeexplore.ieee.org/document/8665367/

5. Carnovalini, F., Rodà, A.: Computational Creativity and Music Generation Systems: An Introduction to the State of the Art. Frontiers in Artificial Intelligence **3**, 14 (Apr 2020). https://doi.org/10.3389/frai.2020.00014, `https://www.frontiersin.org/article/10.3389/frai.2020.00014/full`
6. Carnovalini, F., Rodà, A., Harley, N., Homer, S.T., Wiggins, G.A.: A New Corpus for Computational Music Research andA Novel Method for Musical Structure Analysis. In: Audio Mostly 2021 (AM '21). p. 4. ACM, virtual/Trento Italy (2021). https://doi.org/10.1145/3478384.3478402, `https://doi.org/10.1145/3478384.3478402`
7. Foxley, E.: Nottingham Database (2011), `https://ifdo.ca/~seymour/nottingham/nottingham.html`
8. Hamanaka, M., Hirata, K., Tojo, S.: Implementing methods for analysing music based on lerdahl and jackendoff's generative theory of tonal music. In: Computational Music Analysis, pp. 221–249. Springer, New York, NY (2016)
9. Hamanaka, M., Hirata, K., Tojo, S.: deepgttm-iii: Multi-task learning with grouping and metrical structures. In: International Symposium on Computer Music Multidisciplinary Research. pp. 238–251. Springer, Matosinhos, Porto (2017)
10. Huron, D.: Sweet Anticipation: Music and the Psychology of Expectation. MIT Press (Jan 2008)
11. Lerdahl, F., Jackendoff, R.S.: A generative theory of tonal music. MIT press, Cambridge, MA (1985)
12. Marsden, A., Hirata, K., Tojo, S.: Towards computable procedures for deriving tree structures in music: Context dependency in GTTM and Schenkerian theory. In: Proceedings of the Sound and Music Computing Conference 2013. pp. 360–367. KTH Royal Institute of Technology, Stockholm, Sweden (2013)
13. Orio, N., Rodà, A.: A measure of melodic similarity based on a graph representation of the music structure. In: ISMIR. pp. 543–548. ISMIR, Kobe, Japan (2009)
14. Pearce, M., Wiggins, G.A.: Expectation in melody: The influence of context and learning. Music Perception **23**, 377–405 (2006)
15. Schenker, H.: Free Composition (Der freie Satz). Longman Music Series, Longman, New York, NY, USA (1979)
16. Schürmann, T., Grassberger, P.: Entropy estimation of symbol sequences. Chaos: An Interdisciplinary Journal of Nonlinear Science **6**(3), 414–427 (Sep 1996). https://doi.org/10.1063/1.166191, `http://aip.scitation.org/doi/10.1063/1.166191`
17. Simonetta, F., Carnovalini, F., Orio, N., Rodà, A.: Symbolic Music Similarity through a Graph-Based Representation. In: Proceedings of the Audio Mostly 2018 on Sound in Immersion and Emotion - AM'18. pp. 1–7. ACM Press, Wrexham, United Kingdom (2018). https://doi.org/10.1145/3243274.3243301, `http://dl.acm.org/citation.cfm?doid=3243274.3243301`
18. Velarde, G., Meredith, D.: A wavelet-based approach to the discovery of themes and sections in monophonic melodies. In: International Symposium on Music Information Retrieval: ISMIR. p. 4. ISMIR, Taipei, Taiwan (2014)
19. Wiggins, G.A.: Structure, abstraction and reference in artificial musical intelligence. In: Handbook of Artificial Intelligence for Music. Springer Nature Switzerland AG, Cham (2021), `https://doi.org/10.1007/978-3-030-72116-9_15`