Internship report

TITRE???

Romain VERSAEVEL, M1 Informatique Fondamentale, ENS de Lyon

Tutored by David MEREDITH, Associate professor at Aalborg University

August 6, 2015

Abstract

Contents

1 Introduction **3**

2 Computer music **3**

2.1 Short presentation, history, research areas 3

2.2 Computational music analysis 4

2.3 Several techniques 4

3 Analysing jazz chord sequences **5**

3.1 Motivation 5

3.2 Chord similarities 7

3.3 Compression algorithms 9

3.4 Results 13

4 Other and further work **13**

4.1 Grammatical inference 13

4.2 Segmentation 13

4.3 Improving the compression scheme 13

4.4 The Lr2Cr8 project 14

5 Conclusion **14**

6 Bibliography **14**

1 Introduction

This report was written as part of a twelve-week internship I did during the summer 2015, as part of my Master 1 of Computer Sciences at ENS Lyon. This Internship took place at Aalborg University, Denmark, in the Department of Architecture, Design and Media Technology. I was supervised by Pr. David Meredith, who is Associate Professor there and specializes in Computer music.

The topic of this internship was to ???

The report has two main sections. The first one consists in an introduction to the field of research, *Computer music*, including ??? The second describes my work on ??? and its results. It is completed by a shorter section showing possible improvements and unrelated work I did.

2 Computer music

This section presents the area of *Computer music*. It actually gives an overview of the ??? before focusing on

2.1 Short presentation, history, research areas

The term *Computer music* simply describes any activity that implies both music and computing tools.

Music and mathematics have been linked from the origins; the basis of western music was developed by the Pythagoreans in the 5th century BCE; the French composer Jean-Philippe Rameau used mathematical tools to theorise harmony in the 18th century, in [16] for example. Sound is a physical vibration, and has mathematical properties; and western music is much structured by numbers. Hence, the birth and development of Computer music naturally quickly followed the one of computers. The progress of computation offered to music new tools and new ways to study music; simultaneously, the world of audio moved from an all-analogue to an almost all-numerical. The first pieces composed with the help of computers appeared in the late 50s.

Computer music is therefore a young science based on a very ancient one. In Computer sciences, it is related to *Computational linguistics*; it can be close to Cognitive sciences and often

uses techniques from Machine learning.

Its research areas are numerous and diversified. One can deal with audio (recordings, live performance...) data or symbolic one (scores...); one can study existing pieces or aim at producing new ones; one can improve the laypersons' or the professional musicians' experience. Here is a non-exhaustive list of research areas being part of Computer music:

- automated composition or orchestration ;
- automated live improvisation ;
- computational music analysis ;
- music representation ;
- signal processing...

2.2 Computational music analysis

Why analysing music with computers? As stated above, because computers provide researchers brand new ways of studying music. My analysis hereafter, for instance, requires too many computations for a human being, and on the other hand no musical skills.

As for the general case, analysis can be of audio material or of symbolic representation (or both); its goal can be to acquire knowledge or the ability of creating new pieces (or both). The analysis I give here focuses only on symbolic data and it purpose is rather a learning one.

2.3 Several techniques

This section, purely bibliographical, presents ??? techniques of the state-of-the-art in computational music analysis. It has no ambition of being exhaustive, but to show a wide spectrum of what I studied at the beginning of my internship. More on the outlooks and current improvement of the last technique (???) can be found in ???.

2.3.1 Schenker

2.3.2 FP1

2.3.3 FP2

2.3.4 FP3

2.3.5 FP4

2.3.6 COSIATEC

2.3.7 Grammar induction

2.3.8 Olivier

3 Analysing jazz chord sequences

This section presents my concrete work and contribution. After a presentation of the data motiving this work, jazz lead sheets, I introduce the main tools I used, chord similarity measures and compression algorithms, and finally I give and discuss the results obtained.

3.1 Motivation

The main motivation of my work was a dataset of jazz lead sheets, provided by Sony ???. Before going into more details, I have to introduce basic musical definitions so as to make it more understandable. I try here to give enough explanations for anyone to understand what I dealt with, but also clues for a more interested reader who would like to glimpse a mathematical model of music.

In jazz music, songs are displayed in the form of *lead sheets*, scores giving the base melody with extra indications of chords. An extract of a lead sheet for *What a wonderful world* can by seen on figure 1; the sung melody is written on a staff and the chords are visible in handwritten font.

A *chord* is a set of (at least three) notes; in this context it gives the current mood of the piece and jazz musicians can improvise the accompaniment of the melody with the notes of the chord.

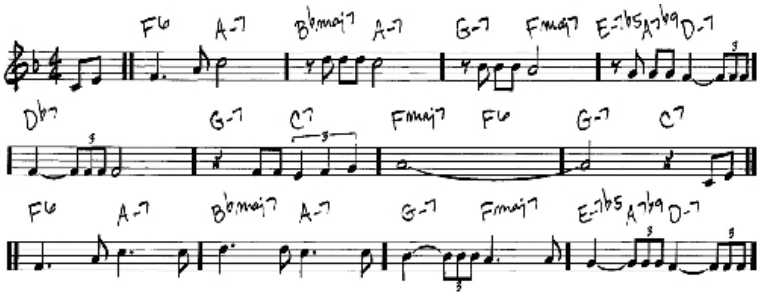

Figure 1: Lead sheet with the beginning of Louis Armstrong's *What a wonderful world*

Chords are represented [1] by a letter from A to G [2] with a possible *accidental* ($\#$ for *sharp*, $\flat$ for *flat*, or nothing), representing the *root note*, and a textual information giving the other notes relatively to the root note. For example, if we consider the $C\#m7$ chord ("C sharp minor seventh"), the notes will be the root note $C\#$, its *minor third* E, its *fifth* $G\#$ and its *minor seventh* B. In jazz there are 313 possible such textual informations [3].

The dataset I worked on contains the lead sheets of 30 jazz pieces by famous artists like Louis Armstrong, Billie Holiday, Charlie Parker... Some files describe the melodies as a sequence of notes (with pitch, onset and duration informations) and other contain the sequences of chords (sometimes with onset and duration informations). I did focus on the chord sequences with no timing information (as if for instance there was exactly one chord per bar). So, my data basically looked like:

A Child Is Born: $B\flat M7;\ E\flat m;\ B\flat M7;\ E\flat m6;\ B\flat M9;\ E\flat m;\ A\ half dim7;\ D7\#9\ldots$

I equally worked on the chord sequences of each song and on the concatenation of all the available sequences. This concatenation does not really make sense as a unique piece, but it is useful to do so in order to have an overview of all the corpus.

So, the motivation of my work was to provide an analysis of this data. The purpose of such an analysis is to gain knowledge about jazz and chord sequences, to understand it better, and also in a second time to use this knowledge to be able to compose similar music that would fit in the corpus.

[1] There are several ways of representing chords; this representation, the *tabular notation*, is the most common in popular music (jazz, pop, rock...). But for instance, superposed notes on a classical music staff would also represent a chord.

[2] French equivalent: from *la* to *sol*.

[3] Actually, one could theoretically form different 56320 chords for a given root note; those 313 correspond to the "harmonious" combinations, which is a subjective notion and thus depends on the music genre.

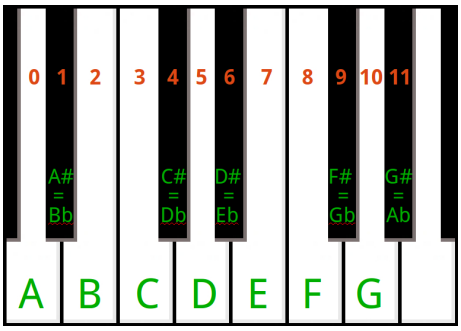

Figure 2: A keyboard with notes names and their equivalent in $\mathbb{Z}/12\mathbb{Z}$

3.2 Chord similarities

This section introduces the concept of chord similarity measures, and defines some measures from the state-of-the-art that I used.

3.2.1 Introduction

There are only 12 possible root notes because two notes separated by an *octave* (concretely, whose frequencies ratio is a power of two) sound likewise, and are thus called the same: C refers to low-pitched as well as to high-pitched sounds. The 12 notes correspond to a division of the octave into twelve equivalent *semitones*; this is called the "well-tempered" scale. So, as shown on figure 2, the keyboard can be mapped to the cyclic group $\mathbb{Z}/12\mathbb{Z}$, and the notes to integers from 0 to 11.

Then chords can be seen as subsets of $\mathbb{Z}/12\mathbb{Z}$. For instance, the previously seen $C\#m7$ chord would be mapped to the set $\{4;7;11;2\}$ (the order does not matter). Moreover the harmonic content in the chord name (here, "$m7$") can be seen as the vector and the root note ("$C\#$") as a starting point. Chords with same labels but different root notes are then the same by translation: $C\#m7$ corresponds to $\{4;7;11;2\}$ and $D\#m7$ to $\{5;8;12;3\} = \{4+1;7+1;11+1;2+1\}$.

It is hence possible to define mathematically distances between chord. However, such a distance much be chosen carefully to be relevant. Canonical distances do not necessarily mirror what is heard. With the L_1 distance ("taxicab metric") on chords of three notes, the chords $C = \{3;7;10\}$ and $Cm = \{3;6;10\}$ would be close since the difference is only of one semitone ($L_1(C, Cm) = 1$); but to the ear they sound very different, one being major and the other minor. On the contrary, one would here something common between $C = \{3;7;10\}$ and $Bm = \{2;5;9\}$:

they are “inversions” of each other.

3.2.2 Definitions of used measures

In all, I used 9 different measures (in addition to the most simple one: equality), some rather elementary, some expressly developed by researchers. Three take as input the two chords to compare, C_1 and C_2, and return a boolean (`true` if and only if the chords are similar). All three define equivalence classes. They are:

- root note equivalence: `true` iff C_1 and C_2 have the same root note;
- translational[4] equivalence: `true` iff C_1 and C_2 have the same harmonic indication (but possibly different root notes);
- PCS-Prime equivalence (see [6]): `true` iff C_1 can be obtained from C_2 by a composition of inversions and translations;
-

The six other are distances[5]: they take as input two chords C_1 and C_2 and return a positive real number (all are scaled so that their image is $[0, 1]$). As I will show in the next part, I did not need this value but to determine if to chord are “close” of “different”, so I used them with a threshold as additional input parameter. The measure will then return `true` if and only if the distance between C_1 and C_2 is greater than the threshold. They are:

- the F1-score;
- Isaacson’s similarity index, defined in [9];
- Lewin’s measure, defined in [12];
- Morris’ measure, defined in [13];
- Rahn’s measure, defined in [15];
- Teitelbaum’s measure, defined in [20]

The definitions of all these measures can be found in appendix. Here I will only present ???

[4] “translation” is the mathematical word; musicians would prefer “transposition”.

[5] I use the word “distance” to emphasize the difference with the first three measures; however all are not distances in a mathematical definition.

3.3 Compression algorithms

The words “analysis” and “compression” will be used here with very close meanings. Indeed, compressing a piece means exhibiting its structure, separating the essential from the redundant. Hence, my analysis of the jazz lead sheets dataset will be described in terms of compression, and evaluated as such.

There are two complementary ways to approach computational music analysis [6]. One sees a piece as a linear sequence of notes (or chords, etc.), as the listener does: he can remember everything he heard but has no way to know what is coming until he actually hears it. The other views the piece from above, completely. Roughly, the first deals with linked lists and the second with arrays. It is the approach of Markov models *versus* the one of formal grammars, Shannon’s entropy *versus* Kolmogorov’s complexity.

My way of compressing simply describes a sequence of chords through the patterns (repeated sub-sequences) it contains, thus removing the redundancy. It is a closer to the second approach, and I developed it a lot more in this direction, but I implemented and tested algorithms for both, in order to be able to compare them. For the former, I used the classical algorithm known as LZ77; for the latter, what I called “diagonal pattern decomposition”. They are presented in this order, after some basic definitions.

3.3.1 Definitions

The compressions are evaluated with their *compression factor* and *loss factor*.

The *compression factor* corresponds to the size of the input data divided by the size of the compressed data. It is expected to be as high as possible (and at least greater than 1). In here, it is considered that the size of a chord is the same as the size of an integer. Indeed, their are $7 \cdot 4 \cdot 313 = 6573$ possible chords, so a chord can be described with $\lceil log_2(6573) \rceil = 13$ bits; and the integer dealt with are lesser than 1469 (the total number of chords in the database), thus defined by $\lceil log_2(6573) \rceil = 11$ bits. Of course, this definition fits the data I used and a different one could be needed for a different data.

There exist compression algorithms *with* and *without loss*. *Without loss* means that the decompression of the compression is equal to the original sequence, while *with loss* it is not necessarily so. The advantage of a compression with loss is that it can achieve better com-

[6] They are of course also relevant in a wider context.

pression factors. I will define here the *recovery factor* as the number of exact chords in the decompression of the compression of a piece, divided by the length of the piece. It is thus a real number between 0 and 1, that we will expect to be as close to 1 as possible. This definition is rather sensitive; indeed, the loss will result of the use of similarity measures, and the original chords will be replaced during the compression by similar ones. However, the only other way to define would be to use precisely the similarity measure ("the two corresponding chords are different by at most x according to the measure") but in my eyes not having an external evaluation would mean have too much faith in the measures; moreover one understands well what it is to be equal or different, but has no precise idea of what a difference of x in a given measure could mean.

3.3.2 Lempel-Ziv 77 (LZ77)

The LZ77 algorithm, designed by A. Lempel and J. Ziv in 1977, introduced in [21], is a very popular compressing algorithm. It takes as input a list of data and outputs a list of triples of the form (a, b, D) meaning "go back a times, copy the next b data, and add D". In the original algorithm, restrained buffer and preview size are given as parameters; however, I chose to let them be unbounded in order to focus on the best possible results. The obtained algorithm is given in 1.

For a better understanding, a complete example of an execution of the algorithm is given in appendix.

With a computation of the longest prefix in $\mathcal{O}\left(|\mathtt{I}|^2\right)$, the overall complexity is $\mathcal{O}\left(|\mathtt{I}|^3\right)$. There are better evaluations of this complexity but that are not of much interest here; knowing that it is polynomial and that the implementation runs fast is enough.

The LZ77 algorithm performs a compression without loss. I modified it in order to have a compression *with loss*. The modification is very simple: when looking for the longest prefix, the algorithm will not only accept identical sequences, but also sequences that are similar up to a certain point. For a given measure $\mathfrak{M}$ and a threshold t (let us assume that the image of $\mathfrak{M}$ is $[0, 1]$ and that higher values correspond to more similar chords), two sequences S_1 and S_2 of the same length n are considered similar if $\forall i \in [1, n], \mathfrak{M}(S_1[i], S_2[i]) \geq t$.

Algorithm 1: LZ77

Input: Queue of Chords $\mathsf{I} = (C_1, \ldots, C_n)$.

Output: Queue of triples $\mathsf{L} = (a_j, b_j, C_{i_j})_j$.

Begin

```
buffer ← empty queue
While I is not empty do
    π ← longest prefix of I in (buffer · I), beginning in buffer
    a ← size(buffer) − (beginning index of π (in buffer)) (0 if none)
    b ← length of π (0 if none)
    For j from 1 to b do
        buffer.push(front(I))
        I.pop()
    L.push(a, b, front(I))
    buffer.push(front(I))
    I.pop()
Return L
```

3.3.3 Diagonal pattern decomposition

My second approach uses *diagonal patterns*. The best way to understand this notion is to visualise it. For a given piece, we write the chord sequence both vertically and horizontally. We can then consider a matrix whose cell (i, j) will correspond to the ith and jth chords of the input sequence. Let us draw this cell in white if those chords are equal, and in black if they are different. We get a binary matrix on which we can see the diagonal patterns, which are diagonal sequences of white cells. They correspond to sequence of chords that appear (at least) twice in the input piece, on two different positions. Such a matrix can be seen on figure 3.

A *pattern* will then precisely be a maximal such sequence, along with its occurrences (positions where it occurs). *Maximal* means that we consider only sequences having an occurrence that cannot be extended. Figure 3 shows the diagonal search for patterns and the maximal pattern $D7GB\flat7E\flat$. $D7G$ is not maximal: every times it occurs it can be extended in the end. On the contrary, $B\flat7E\flat$, which is included in $D7GB\flat7E\flat$, is also a maximal pattern, occurring in the very end. In the piece of figure 3, the diagonal patterns are (by increasing size):

- B, on positions 0 and 11;
- $B\flat7E\flat$, on positions 4, 8 and 13;

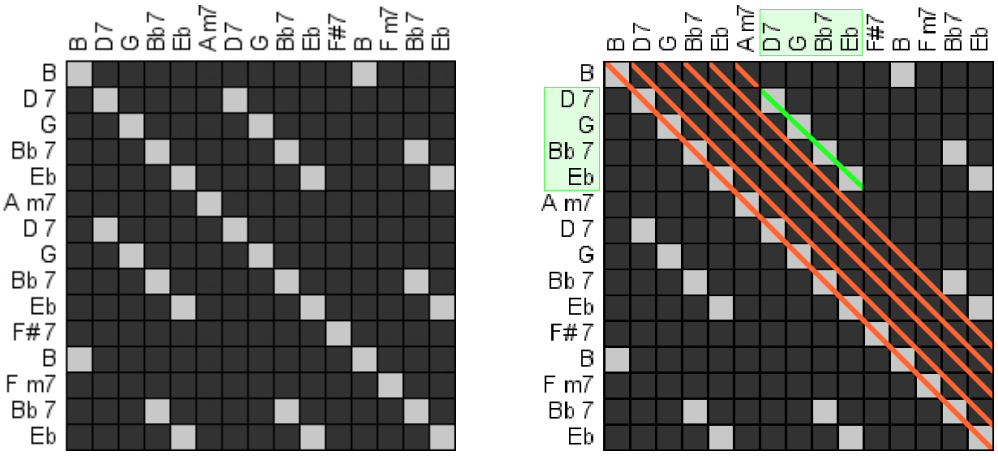

Figure 3: A binary matrix and the diagonal search for patterns

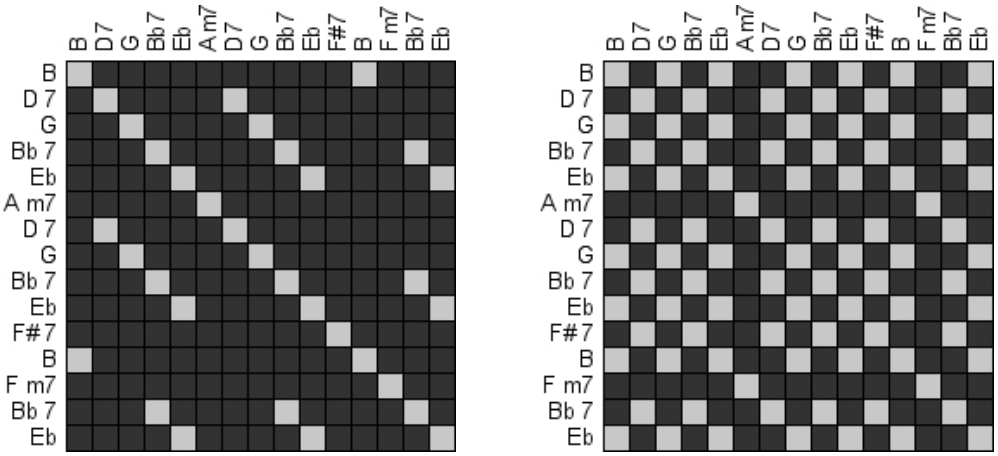

Figure 4: Binary matrices for the same piece, the second using a similarity measure (F1 score)

- $D7GB\flat7E\flat$, on positions 1 and 6.

As for the LZ77, I introduce here similarity measures, which will lead to a compression with loss. Instead of drawing a cell of the matrix in white only if the two corresponding chords are equal, this will be done also if they are similar for a given measure and up to a certain threshold. The matrix has then more white cells, implying more and longer patterns. Figure 4 shows this transformation (with the F1-score and a threshold of 0.9).

Once the list of patterns established, the next step is to select a subset of these which is sufficient to describe the whole piece. In order to be sure this will always be possible (which is not the case in our example!), we add a "pattern" for each single chord appearing in the piece (here: $(B; 0, 11)$, $(D7; 1)$, $(G; 2)\ldots$). This subset should be as "light" as possible, the *weight*

of a pattern being the sum of its length and of its number of occurrences. This problem is a *weighted set cover*, which is NP-complete[7]. So I implemented two heuristics, and the lighter results of the two will be selected. Briefly, their are both greedy algorithms; one aggregates patterns until covering the whole piece and the other removes as many patterns as possible from the exhaustive list. The pseudo-codes of these are given in the appendix.

For the decompression algorithm, the selected patterns are sorted by increasing number of occurrences and are copied in this order (so, a position that is covered by several patterns can be rewritten several times, and only the last, i.e. the corresponding chord of the pattern with most occurrences, will be kept). It is used to compute the *recovery factor*. A pre-treatment is done on the patterns in order to maximize it (to minimize the loss). Indeed, a pattern occurring *similarly* several times can correspond to several different exact sequences, and the recovery factor will depend on the choice of the sequence (which does not impact the compression factor) representing the pattern. For this, I simply compute for every pattern what positions of the reconstruction depend on it, and choose among the possible sequences the one that implies a minimum loss[8].

The complexity of the whole algorithm is $\mathcal{O}\left(|I|^5\right)$[9]. This is quite high. In practice, the number of maximal patterns (which is $\mathcal{O}\left(|I|^2\right)$ in the worst case) seems to be the most determining factor for the running time. On my complete database (approximately 1500 chords), the execution takes between a few seconds and several minutes.

3.4 Results

4 Other and further work

4.1 Grammatical inference

4.2 Segmentation

4.3 Improving the compression scheme

PAPER

[7] See https://en.wikipedia.org/wiki/Set_cover_problem for more explanations on the set cover problem.
[8] This greedy algorithm, knowing the decompression scheme, is obviously optimal for this problem.
[9] See appendix for a brief analysis.

<=> MELODY

4.4 The Lr2Cr8 project

5 Conclusion

6 Bibliography

References

[1] Jean-Julien Aucouturier and Mark Sandler. Finding repeating patterns in acoustic musical signals: Applications for audio thumbnailing. In *Audio Engineering Society Conference: 22nd International Conference: Virtual, Synthetic, and Entertainment Audio*. Audio Engineering Society, 2002.

[2] Stanley F Chen. Bayesian grammar induction for language modeling. In *Proceedings of the 33rd annual meeting on Association for Computational Linguistics*, pages 228–235. Association for Computational Linguistics, 1995.

[3] Stefano Crespi-Reghizzi. The mechanical acquisition of precedence grammars. Technical report, DTIC Document, 1970.

[4] W Bas De Haas, Martin Rohrmeier, Remco C Veltkamp, and Frans Wiering. Modeling harmonic similarity using a generative grammar of tonal harmony. 2009.

[5] Colin De La Higuera. A bibliographical study of grammatical inference. *Pattern recognition*, 38(9):1332–1348, 2005.

[6] Allen Forte. *The structure of atonal music*, volume 304. Yale University Press, 1973.

[7] King-Sun Fu and Taylor L Booth. Grammatical inference: Introduction and survey-part i. *IEEE Transactions on Systems, Man, and Cybernetics*, 1(SMC-5):95–111, 1975.

[8] King-Sun Fu and Taylor L Booth. Grammatical inference: Introduction and survey-part ii. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (3):360–375, 1986.

[9] Eric J Isaacson. Similarity of interval-class content between pitch-class sets: the icvsim relation. *Journal of music theory*, pages 1–28, 1990.

[10] Philip N Johnson-Laird. Jazz improvisation: A theory at the computational level. *Representing musical structure, London*, pages 291–325, 1991.

[11] Kenichi Kurihara and Taisuke Sato. Variational bayesian grammar induction for natural language. In *Grammatical Inference: Algorithms and Applications*, pages 84–96. Springer, 2006.

[12] David Lewin. A response to a response: on pcset relatedness. *Perspectives of new music*, pages 498–502, 1979.

[13] Robert Morris. A similarity index for pitch-class sets. *Perspectives of New Music*, pages 445–460, 1979.

[14] Timothy J O'Donnell, Joshua B Tenenbaum, and Noah D Goodman. Fragment grammars: Exploring computation and reuse in language. 2009.

[15] John Rahn. Relating sets. *Perspectives of New Music*, pages 483–498, 1979.

[16] Jean Philippe Rameau. *Nouveau systême de musique théorique, où l'on découvre le principe de toutes les règles nécessaires à la pratique: pour servir d'introduction au Traité de l'harmonie*. De l, 1726.

[17] Martin Rohrmeier. Towards a generative syntax of tonal harmony. *Journal of Mathematics and Music*, 5(1):35–53, 2011.

[18] Yves Schabes, Michal Roth, and Randy Osborne. Parsing the wall street journal with the inside-outside algorithm. In *Proceedings of the sixth conference on European chapter of the Association for Computational Linguistics*, pages 341–347. Association for Computational Linguistics, 1993.

[19] Mark J Steedman. A generative grammar for jazz chord sequences. *Music Perception*, pages 52–77, 1984.

[20] Richard Teitelbaum. Intervallic relations in atonal music. *Journal of Music Theory*, pages 72–127, 1965.

[21] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.