UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Generative Models of Music for Style Imitation and Composer Recognition

by

Jan Buys
15293262

Honours Project in Computer Science

Final Report

Computer Science Division,
Departement of Mathematical Sciences,
University of Stellenbosch.

Supervisor: Prof. Brink van der Merwe

November 2011

Uittreksel

Ons ondersoek generatiewe modelle van musiek en hul toepassing op die generasie van musiekstukke deur stylimitasie en op komponisherkenning van bestaande musiekstukke. Ons begin by Markov modelle, die algemeenste benadering to algoritmiese komposisie, en ondersoek dan maniere om die onvermoë van Markov modelle om langer musiekstukke wat as geheel aanvaarbare struktuur toon te genereer, te oorkom. Ons ontwikkel 'n rekenaartoepassing wat in staat is om groot versamelings musiekstukke as afrigtingsdata en toetsingsdata te gebruik, om ons modelle te implementeer.

Abstract

We investigate generative models of music and their application to the generation of music pieces by style imitation and composer recognition of existing pieces. We start with Markov models, the most common approach to algorithmic composition, and then investigate ways to overcome the inability of Markov models to generate longer music pieces that exhibits acceptable overall structure. We develop a computer application, capable of using large collections of music pieces as training and testing data, to implement our models.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem statement

> *Music is a hidden arithmetic exercise of the soul, which does not know that it is counting.* – Leibniz

The composition of music is essentially a creative process. However, any piece of music has structure, often a very complex structure. A music piece is usually composed to conform to a given style, by following established rules and principles that give the music certain aesthetic qualities. Such rules are studied by all students undergoing classical Western musical training. However, the complexity of a good music piece cannot be fully described by such rules. The main reason for that is that a music piece should exhibit acceptable structure at a local and global level. There should be a fine balance between various musical qualities in the piece.

Algorithmic composition, i.e., composition by formalizable methods, has a long tradition, and numerous procedures have been investigated [1]. The dominant approach is that of style imitation of existing music pieces. Different classes of algorithms that can achieve this have been studied. All these approaches, however, have limitations, the most common being the inability to generate longer pieces of music that exhibits acceptable overall structure.

The focus of research in music modelling has mostly been on music generation. However, problems related to the analysis of music pieces can be addressed by the same generative models that are used for synthesis. Methods to analyse music pieces with the goal of describing the structure in the music have been studied. Statistical models trained on sets of similar music pieces can be applied to style or composer recognition of unseen music pieces.

1.2 Project aims and methodology

The principle aim of this project was to develop a computer application that takes pieces of classical music as training data for one of a number of models, and use each model to generate music pieces and recognize the composers of music pieces. The generated music should imitate the style and structure of the training data on a local and, to some extend, global level.

To achieve these aims, we investigated current models and developed new models for music generation based on automata and grammars. Firstly, we implemented the standard approach of Markov Models for style imitation. Our implementation addresses some of the problems involved with the handling of irregularities in input music pieces. This enabled us to work with large collections of pieces by classical composers. We extended the model so that it can generate harmony—chords that sound well together with the generated melody.

Next we investigated ways to model similarities between different fragments of music and ways to reproduce those similarities in a generative model. Context-free grammars were implemented for rhythm and pitch sequences. A method to construct clusters of similar musical contours that has been used successfully in Jazz music, was applied to classical music.

During development of the project, we primarily used a corpus of 500 Bach chorales (in MIDI format) as training data for our models. Bach chorales are often used in music modelling (see e.g. [2], [3]). The motivation for their use is their abundance, simplicity and good melodic and harmonic form. We also used a music corpus containing a large number of standard repertoire classical pieces by different composers to generate music in different styles and to test our composer recognition system.

1.3 Related work done at Stellenbosch University

Walter Schulze wrote a masters thesis [4] on music generation at Stellenbosch University, graduating in 2009. He used a Markov model approach similar to the one we implement here for music generation.

In 2010, Herko Lategan did his honours project on algorithmic composition. He developed a digital audio workbench which allows users to make use of Markov based algorithmic composition techniques.

1.4 Report organization

In chapter 2, we discuss the problem of the formalized modelling of music for purposes of algorithmic composition and music analysis. Then, in chapter 3, we

define the generative theoretical models that form the basis of the techniques that we investigate and apply to music. After that, in chapter 4, we do a literature survey on the use of generative models in algorithmic composition and musical information processing. Chapter 5 describes our implementation of a Markov model for melody generation. We also set out the general procedure that we follow to implement generative models. In chapter 6 we use Markov Models to harmonize melodies. Chapter 7 introduces a model that labels the function of melody notes, clusters similar bars of music together and uses that information to generate music. Then chapter 8 gives context-free grammars for melody and rhythm. In chapter 9 we apply generative Markov Models to the problem of composer recognition of unseen music pieces. In chapter 10 we describe Bacchus, the generative music modelling system that we wrote. Finally, in chapter 11, we discuss the evaluation of generated material, make some conclusions on the capabilities of different models and discuss possible future work.

Chapter 2

Problem Description

2.1 Elements of musical notation

A music piece consists of *notes*. A note is a single sound, represented by *pitch*, i.e., how high or low the sound is, and *duration*, i.e., how long the sound is held [4]. Other properties of a note include its volume (loudness) and timbre (the influence of the instrument that the note is played on the sound).

In a standard western classical music piece, the pitch and duration of sequences of notes are governed as follows:

- Pitches are named by their *pitch classes*. There are 12 classes, namely C, C#, D, D#, E, F, F#, G, G#, A, A# and B. These names repeat every 12 semi-tone pitches, partitioning the pitches into 12 equivalence classes. An interval of size 12 is referred to as an *octave*.
- A *scale* is a sequence of pitch classes defined by the starting pitch class of the scale and the intervals between pitches in the scale. The most common scale types are the major and (natural) minor scales.
- The *key signature* of a piece indicates the scale that forms the basis of notes of the piece. However, a piece can also have *accidentals*, notes that are not in the scale of the key signature.
- The *beats* of a music piece are constant time intervals that govern the beginning of notes in the music (although notes can also start *offbeat*). The *tempo* indicates the length of those beats. The *time signature* indicates the meter of the music, i.e., the basic grouping of constant numbers of the beats into *bars*.
- The *rhythm* of a sequence of notes describes the duration of each note in the context of the time signature of the piece.

- Note durations are expressed as a fraction of a "whole note" (usually 4 beats). Typical durations are that of a half note, quarter note (crotchet), eight note (quaver) and sixteenth note (semi-quaver). An exception to the $\frac{1}{2^n}$ format is triplets: Three quaver triplets have the total duration of a quarter note.

In general, a music piece consists of a number of voices, each voice being a single time-dependant sequence of notes. The *melody* of the music piece, the most significant voice, is usually the highest. The *harmony* of the music is the way that different notes sound simultaneously. The harmony can be described by *chords*, predefined combinations of notes in the scale of the music that sound well together.

2.2 Music as a complex phenomena

In a lecture [5] in March 2011 at Stellenbosch University, Prof Paul Cilliers argued that complexity is a fundamental attribute of music: It is not possible to have a model that can sufficiently describe all the aspects of a music piece. A complex system has interconnected parts that as a whole exhibits properties that are not obviously deduced from properties of the individual parts. The most important manifestation of this complexity in music is that some compositions or performances speak to us—the music is very satisfactory to the listener—while others do not. This is explained by the notion of emergence in complexity theory: That a complex system results from simple interactions between simple parts of the system.

What we can take from this in the application of algorithmic composition is as follows: We cannot create a system that is guaranteed to show emergence. However, we can model different components of a system in a way that can lead to emergence, and by careful study we can construct models that will assign high probabilities to pieces that have qualities that occur in emergence, and low probabilities to pieces that do not show potential of emergence.

2.3 Generative grammars

Chomsky [6] refers to the ability of humans to generate an infinite number of sentences as a creative process. A primary goal of linguistics is to formulate a finite grammar (sentence generating device) that accounts for this creativity and is able to generate exactly all the valid sentences in the language. A good grammar will be reasonably simple and will assign a sensible syntactic structure to sentences.

An important attempt to formulate a theory to describe structure in music is that of Lerdahl and Jackendoff in *A Generative Model of Tonal Music* [7]. Their goal is to give a formal description of the musical intuitions of a listener experienced in a musical idiom. As with many other approaches to music modelling,

concepts that originated in linguistic theory are used in the formulation of the theory. However, despite the existence of parallels between language and music, they warn against the literal translation of aspects of linguistic theory into musical terms. The differences in the concepts of structure and meaning in language and music are just too big to justify such parallels.

2.4 Historical development

The first known approach to automatic musical structure generation is that of Guido of Arezzo, around 1000 AD [1]. He generated melodies from text by mapping letters and syllables to tone pitches and melodic phrases. In the 13th century, R Lullus created the *Ars Magna*, a machine capable of generating logical statements. This laid the ground for generative grammars, ways to generate valid statements by algorithmically combining acceptable terms. Algorithmic composition would later extensively use such generative approaches. In the 18th century the *musical dice game* became popular. It involved making random selections from a number of available musical fragments, and concatenating them to form a music pieces of predefined length.

The first fully computer-generated composition was the *Illiac Suite*, produced by L. Heller and L. Isaacson on the ILLIAC computer at the University of Illinois in 1956. This work inspired musicians and computer scientists to investigate approaches to algorithmic composition with computers.

After initial success in in 1950s with statistical, empirical approaches to music generation, driven by advances in machine learning, there developed a sense of frustration with the inability of models to generate even simple satisfactory melodies [8]. This led to a long period where most work on music generation involved a knowledge engineering approach. Though these models delivered good results, the music they generated was too constrained and relied too much on the musical judgment of their creators.

Chomsky's rejection [9] of Markov models for language also contributed to the suppression of work on such models for music. However, it did lead to the formulation of more powerful generative grammars for music.

In the 1990s, following the decisive success of statistical models in speech recognition, there was a dramatic move back to statistical models in linguistics. This was followed by a similar shift is music modelling.

Presently, David Cope is one of the most prominent composers making use of algorithmic composition techniques. His system *Experiments in Musical Intelligence* (EMI) generates music of a high degree of complexity. He uses the approach of musical "recombinancy", which recombines musical fragments found by complex analysis of a given corpus of music [1].

2.5 Approaches to music modelling

In algorithmic composition, one can distinguish between systems of genuine composition and systems of style imitation. Genuine composition uses rules and procedures chosen by the composer or programmer to create new pieces of art. Style imitation examines given music pieces, constructs a model to represent some of the characteristics of these pieces and then uses a stochastic model to generate new music pieces that reflects those characteristics.

Algorithmic composition is predominantly used in the field of style imitation. The main reason for this is that composers seldom publish the formalizable ideas that they do use in their compositions [1]. In most genuine compositions that make use of algorithmic composition techniques, essential decisions are still left to the creativity of the composer. It is therefore difficult to classify a piece as a genuine algorithmic composition.

There are two approaches to generative models for style imitation [2]. In the first approach, *knowledge engineering*, rules and constraints are explicitly encoded in some logic or grammar. In the second approach, *empirical induction*, parameters of a statistical model are determined by an analysis of existing compositions.

The two goals of generative models are analysis and synthesis. In language modelling, the focus was from the beginning mainly on analytic methods. In contrast, in music modelling, research focused initially mainly on synthetic methods. Only recently has analytic topics such as musical classification and phrase structure analysis become popular. However, Conklin [8] argues that we do not need to distinguish between analytic and synthetic methods. The problem of music generation can be made equivalent to that of sampling from a statistical model. This problem can be separated from the problem of formulating and training models.

The concept of a predictive model is used to unify the problems of analysis and synthesis. A predictive model is a statistical model that tries to predict the next note in a melody given a preceding sequence of notes. The predictive power of a model can be measured quantitatively by the notion of entropy. Such models can be compared to see how well they model music pieces in a specific style. It is hypothesized in [2] that highly predictive theories will also generate acceptable original music pieces. Predictive models can also be applied to composer or style recognition.

Chapter 3

Automata and Grammars

3.1 Introduction

A *generative grammar* is a recursive rule system capable of generating well-formed strings in a language [1]. The Chomsky hierarchy distinguishes between four types of generative grammars that show different levels of restriction. A Markov model is equivalent to a regular grammar, the most restrictive type of grammar. The next grammar in the hierarchy, somewhat less restrictive, is the *context-free grammar*.

A grammar can be made probabilistic by adding weights to each of the different production rules for the expansion of non-terminal symbols. These weights can be estimated to maximize the probabilities that the model assign to a given set of sequences.

We now define the models and grammars that we use, and describe ways to represent them.

3.2 Markov chains

A *stochastic chain* describes a sequence of time-dependent random events [1]. An event is represented by one of a finite number of *states*. The *state space* is the set of possible events. A *Markov chain* (MC) is a stochastic chain where the probability of the future state q_{t+1} is dependent only on the current state q_t [1]. Transitions between states are governed by *transition probabilities*. A Markov chain adheres to the first order Markov assumption:

$$P(q_{t+1}|q_t, q_{t-1}, \ldots, q_1) = P(q_{t+1}|q_t),$$

where the state space is $Q = \{q_1, q_2, \ldots, q_t\}$.

When more than one past event is used to calculate transition probabilities, we have a *higher order Markov chain*. The transition probability of a n-th order

Markov chain is dependent on the previous n events. It therefore has the assumption

$$P(q_{t+1}|q_t, q_{t-1}, \ldots, q_1) = P(q_{t+1}|q_t, q_{t-1}, \ldots, q_{\max(t-n+1,1)}).$$

We transform an n-th order MC to an equivalent first order MC by encoding the history of states (the previous up to n states) into the names of the state. Therefore, we replace the state state space Q by the union of $Q^1, Q^2, \ldots, Q^n$, where Q^i is the set of state sequences of length i [10].

Given some sequences of symbols, we can construct an MC, first or higher order, that models the sequences: The state space of the MC is the set of all possible symbols in the sequences of the MC, and the transition probabilities of the MC are determined by calculating the maximum likelihood estimate, using frequency counts on the given data. So in a first order MC, for given sequence s_1^T, the transition probability between states q_i and q_j is:

$$\frac{\#(s_t = q_i, s_{t+1} = q_j)}{\#(s_t = q_i)}$$

The *training* of the model is the process of estimating the parameters of the model (in this case the transition probabilities) from *training data*.

3.3 Finite-state machines

A *finite-state acceptor* (FSA) is a network of states and labeled transitions, with exactly one start state and one final state [11]. A *string* is an ordered sequence of symbols drawn from a finite vocabulary. An FSA accepts string $w_1, w_2, \ldots, w_n$ if there is a *path* from the start state to the final state along transitions labeled $w_1, w_2, \ldots, w_n$. The empty symbol is also a valid transition label, denoting that no new symbol is read from the string. An acceptor with more that one final state can be transformed to have only one final state by adding transitions on empty symbols from the old final states to a single new final state.

A *finite-state transducer* (FST) is similar to an FSA, but each of its transitions have an input label and an output label. An FST therefore transforms an accepted input string into an output string.

A *weighted finite-state acceptor* (WFSA) assigns a weight to each string that it accepts. Every transition is assigned a weight, and the weight of the string is the product of the transition probabilities along the path by which the string is accepted. Similarly, a *weighted finite-state transducer* (WFST) is an FST with probabilities assigned to the transitions. A weighted finite-state acceptor can be represented as a WFST with the same input and output symbols on every transition.

The concept of semiring abstraction let us define automata representations and algorithms over different weight sets and algebraic operations. A *semiring* K consists of a set $\mathcal{K}$ with an associative and commutative operation $\oplus$ and an associative operation $\otimes$, with identities $\overline{0}$ and $\overline{1}$, respectively, such that $\otimes$ distributes over $\oplus$, and $\overline{0} \otimes a = a \otimes \overline{0} = \overline{0}$.

In this project we will either be working in the boolean semiring (corresponding to an unweighted state machine) or the probability semiring, where each transition weight corresponds to the probability of that transition.

Formally, a Weighted finite-state transducer $T = (\Sigma, \Omega, Q, E, i, F, \lambda, \rho)$ over the semiring $\mathcal{K}$ is given by [12]:

- An input alphabet Σ
- An output alphabet Ω
- A finite set of states Q
- A finite set of weighted transitions $E \subseteq Q \times (\Sigma \cup \epsilon) \times (\Omega \cup \epsilon) \times \mathcal{K} \times Q$
- An initial state $i \in Q$
- A set of final states $F \subseteq Q$
- An initial weight λ
- A final weight function ρ

We can represent Markov chains as WFSAs. A WFSA representing a Markov chain has the same states and transition probabilities as the MC. Every transition in the WFSA is labeled with the symbol of the destination state. In the case of higher-order models, the state labels encode the history of the previous n symbols. The label of a transition is the next symbol generated in the string, which becomes the last symbol on the label of the state the transition is going to.

3.4 Hidden Markov models

An *Hidden Markov model* (HMM) is used to model the relation between two sequences, a hidden sequence and an observed sequence. The symbols of the hidden sequence are represented by a discrete number of states. Transition probabilities between these states are defined as with a Markov chain. Every hidden state emits a symbol of the observed sequence, according to a probability distribution specified for each state. The emission probability distributions can be discrete or continuous. An HMM has two special states, the start state and the final state. The start state

does not emit an observed symbol. Transitions from the start state indicate the initial probability distribution of the hidden states. The final state represents the end of the hidden sequence.

Let x_1^T be the sequence of observed symbols and s_1^T the sequence of hidden states. An HMM operates under two fundamental assumptions [13]:

1. The observation independence assumption: An observation is dependent only on the current hidden state.

$$P(x_t|x_{t-1}, x_{t-2}, \ldots x_1, s_t, s_{t-1}, \ldots, s_0) = P(x_t|s_t)$$

2. The first-order Markov assumption:

$$P(s_t|x_{t-1}, x_{t-2}, \ldots x_1, s_{t-1}, s_{t-2}, \ldots, s_0) = P(s_t|s_{t-1})$$

   As with MCs, we can generalize this to higher-order models.

The three main problems related to the use of HMMs and the algorithms to solve them, are [14]:

1. To compute the probability of a given observation sequence: The forward algorithm.

2. To find the optimal hidden state sequence for a given observation sequence: The Viterbi algorithm.

3. Given an observation sequence, to adjust the model parameters to maximize the probability of a observation sequence given the model: Viterbi or Baum-Welch re-estimation.

We can represent a discrete HMM with the composition of two WFSTs. The first is a MC for the hidden states and their transitions. The input and output labels of the transitions both represent the hidden sequence, and the input and output labels at every transition are the same. The second WFST has only one state, which has multiple transitions to itself. Every transition corresponds to an element of the Cartesian product of the hidden and observed symbol alphabets. The input symbol of the transition is a hidden symbol, and the output is an observed symbol. The weights of the transitions from a hidden symbol to different possible observed symbols form the emission probability distribution of the hidden state in the HMM. The left-to-right composition of the first and second WFSTs gives a transducer that takes a hidden sequence as input and gives an observed sequence as output. This representation is described more concretely in the specific models discussed in later chapters.

3.5 Probabilistic context-free grammars

A context-free grammar (CFG) consists of the following components [15]:

1. The *terminal alphabet*, a set of symbols from which the objective sequences are formed.
2. The *non-terminal alphabet*, a set of symbols used to construct sequences of terminal symbols, but not terminal symbols themselves.
3. The *start symbol*, a non-terminal symbol used as the root in generating a sequence.
4. *Productions*, rules that create new sequences by replacing a non-terminal symbol with a sequence of non-terminal and terminal symbols.

A Probabilistic CFG has, in addition, a set of probabilities, each associated to a production rule, such that the sum of the probabilities of all the rules with the same left-hand non-terminal is 1 for every non-terminal symbol.

When strings in a context-free language are given as training data for a CFG, we do not know which production rules are used to generate the string. This is in contrast to a regular grammar, were we can just read off the state transitions in a (deterministic) FSA as we read the input string from left to right. In a CFG, the derivation of a string may be ambiguous: Different sequences of applied production rules may yield the same string.

To train PCFGs, we use a version of the Expectation Maximization algorithm, as described in [16]. We start by giving sensible initial probabilities to all the CFG production rules. Then we determine the possible parse trees of our input sentences, weighted by the probabilities that the PCFG assigns to each parse (expectation step). These parse trees are then used to reestimate the probability of each of the production rules, using maximum likelihood estimation (maximization step). We can repeat these two steps until the change in the total probability of the parse trees falls below a certain threshold.

3.6 Regular tree grammars

A weighted regular tree grammar (RTG) over semiring $\mathcal{W}$ is a 4-tuple $G = (N, \Sigma, P, n_0 i)$ where [17]:

- N is a finite set of non-terminals
- Σ is the ranked input alphabet

- $P = (P', \pi)$, where P' is a finite set of productions, each $p \in P$ of the form $n \to u$, $n \in N$, $u \in T_\Sigma(N)$, and $\pi = P' \to \mathcal{W}$, a weight function of the productions. $T_\Sigma(N)$ is the set of all trees over Σ that may have symbols in N as leaves (that non-terminals will later be replace by applying production rules to them)

- $n_0 \in N$ is the initial non-terminal symbol

We can represent a CFG as a tree grammar [18]. The main difference is that a CFG produces a string, and we replace non-terminals in the string repeatedly until it only contains terminals, while a tree grammar produces a tree that contains all the non-terminals to which production rules were applied in the tree derivation. The set of yields (the strings formed by reading the leave symbols off the trees from left to right) of the trees produced by an RTG is context-free.

3.7 Implementation

In our implementation we use **Carmel** [19], a finite-state transducer package, to process our Markov models. Carmel can train and compose transducers, sample sequences or get sequence probabilities from transducers.

We use **Tiburon** [20], a tree transducer package with similar functionality as Carmel, to implement our context-free grammar and tree transducer models. Tiburon can convert a context-free grammar to an equivalent regular tree grammar or tree transducer to perform parsing and training operations.

Chapter 4

Survey of Generative Models in Music

4.1 Introduction

In this chapter we survey work done on generative models of music. Most of these models were developed for algorithmic composition. We discuss the use of Markov models, which are still the dominant approach to style imitation. We also discuss context-free grammars, the use of generative grammars in music modelling and the related topic of music prediction.

There are many other paradigms of algorithmic composition: See the book by Nierhaus [1] for an overview. These paradigms include transition networks, neural networks, chaos and self-similarity, genetic algorithms, cellular automata and artificial intelligence.

4.2 Markov models

Markov Processes were first used for music generation by Harry Olson around 1950 [1]. Subsequently it has been used in many approaches to style imitation and genuine composition.

Markov models and closely related models remain dominant in statistical music generation. The main reasons are that they are very fast and easy to perform the basic tasks of inducing probabilities, computing the probability of pieces and music generation. The sparse data problem, however, is an important problem that has to be dealt with.

An example of early work done on algorithmic composition is that of Brooks *et al* [21] in 1957. They experimented with modelling melodies with different orders of Markov chains. A melody is represented by a sequence of pitch numbers, one for

every eight note, with a distinction made between a new note attack and a hold on note. They found that one has to find a middle ground between Markov chains of low order, that do not constrain the structure of generated music sufficiently, and Markov chains of high order, that reproduce large fragments of the music pieces used for training.

Conklin and Witten [2] use "Multiple Viewpoint Systems" that combines the representation of independent views of the music being modelled. Context models are used to model sequences of events. A context model is a smoothed higher order Markov chain: The probability distribution of event n in the sequence is given by a weighted linear combination of the probabilities of the event given each of the previous 0 to $n-1$ events in the sequence. The viewpoints modelled include pitch, rhythm, time signature, location of the ends of phrases, the start time and the duration of a piece.

Trivino-Rodriguez and Morales-Bueno [22] use Probabilistic Suffix Automata (PSA) for music generation. A PSA is a variable-order L Markov chain, meaning that its memory length of previous states can vary from 0 up to L. This model does not grow exponentially with size, as standard higher-order Markov chains do, so it is feasible approach to higher-order models. A Multi-attribute Prediction Suffix Graph is used for music prediction.

At Stellenbosch University, Schulze and Van der Merwe [10] investigated mixed order Markov models (equivalent to PSA) and higher order Markov models for music generation. Their evaluation found that, despite mixed order models being able to handle a much longer memory length, music generated from these models are not of a higher quality than that of higher order Markov models.

De la Higuera *et al* [23] learns Stochastic Finite Automata and apply them to musical style recognition. Music pieces are represented by sequences of altering pitch and duration symbols. States in the automata are either pitch or duration states. The MDI learning algorithm is used: Firstly, a prefix tree acceptor is build. Then states are merged iteratively to minimize the number of states and allow previously unseen pieces to be accepted. The probabilities assigned to music pieces are used to classify pieces according to musical style.

A machine learning approach to learning jazz grammars, implemented in ImproVisor system, is proposed in [24]. The melody is abstracted in terms of note categories, note durations and the melodic contour—the interval range of ascending or descending note sequences. Melody fragments with a fixed total duration are represented by S-expressions with these abstractions. These expressions are divided into clusters. A Markov chain of these clusters is then constructed. When this model is used for generation, a sequence of clusters is generated. Then, for each cluster, an abstract melody is sampled from the cluster. The abstract melody is then replaced with a concrete melody by sampling from pitch sequences that satisfies the constraints of the abstract melody. However, those constraints may

have to be relaxed somewhat to allow for the generation of a pitch sequence that satisfies the note category and interval range restrictions over a given sequence of chords.

4.3 Hidden Markov models

The hidden Markov model is a powerful tool in music generation due to its ability to model underlying discrete phenomena of sequences. The clearest example of this in music is harmony.

Allan and Williams [3] use HMMs for the harmonization of given melodies. The visible states of the HMM are melody notes and the hidden states are possible harmonizations for the melody line. They treat different configurations of the same chord—pitch combinations with the same implied harmony—as different harmonizations (hidden states). The harmonization is done on beat-level. A second HMM is used to model ornamentation: To smooth the movement between notes in a line of music, extra notes are added and pitch repetitions are merged.

The MySong automatic accompaniment system [25] uses an HMM approach to generate chords for a piece sung by the user. The system, made commercially available by Microsoft, has model parameters for the style of generated chords. The "jazz factor" and the "happy factor" are parameters that the user can set. The system uses training data from a variety of musical styles. The vocal melody sung by the user is recorded and the most likely key of the melody is identified. The melody is then transposed to C for purposes of harmonization, chord are generated with an HMM, and the generated chords are transposed back to the original key of the melody. The HMM also has chords as hidden states and melody notes as observations.

4.4 Context-free grammars

Keller and Morrison [15] use a hand-crafted context-free grammar to generate jazz improvisations over given chords in their program ImproVisor. The grammar representation of pitches is based on the function of the notes in the given chord: There are 7 terminal symbols that each represent a note category—a possible function of a note played over a given chord. Elements in a generated sequence consist of a terminal pitch-class symbol and a duration specified for that pitch. Additional constraints can be put on the symbols that can be generated. Production rules for a context-free grammar is given, with probabilities that were assigned manually by the authors. After a sequence is generated, concrete pitches are sampled that satisfy the constraints of the terminal symbols and the given sequence of chords.

Gilbert and Conklin [26] proposes a probabilistic context-free grammar for melody in terms of melodic reductions. Pitch is represented by intervals, and production rules for a context-free grammars is defined by replacing one interval by two other intervals. The "New" rule allows for any interval to be inserted, while other rules allow for the substitution of one interval with two intervals that have the same resultant as the original ones. All these intervals are represented by non-terminals, until no futher expansion of intervals is done, when they are replaced by terminals representing the same intervals. The grammar is trained by an expectation-maximization algorithm for probabilistic CFGs.

Bod [27] attempts to model phrase structure in language and music in a uniform way. Given the widespread and successful use of treebanks in natural language processing, he uses the Essen Folksong Collection, a collection of melodies that includes phrase separators in the melodies. When sentences are parsed over a probabilistic tree grammar, there are two goals: The parse trees should be as simple as possible, and the probability of the trees should be as high as possible. Bod finds that the best way to combine these goals in a generative model is to select the parse with the simplest structure from the n most likely trees. The tree structure (that forms a context-free grammar) he proposes for music is relatively simple: The non-terminals are S for song, P for phrase and N for note. The children of the initial non-terminal S are Ps, one of each phrase of the piece. The children of P are Ns, one for each note in the phrase, and the Ns are substituted by terminal note symbols.

SEQUITUR [28] is a linear-time algorithm that infers structure from a sequence of discrete symbols, forming a context-free grammar for the given sequence. It has been shown to give reasonable results when applied to music, correctly identifying cadences. The grammar has the following properties: No pair of adjacent symbols may appear more than once in the grammar, and every rule must be used more than once.

In a recently published article [29], probabilistic tree automata are used to melodic identification. Melodies are represented by a tree structure: The pitches of a melody are the labels of the leaves of the tree. The rhythm is represented by the structure of the tree: Nodes at every level of the tree represent note durations that are halve (or for ternary measures, a third) of the durations of the level above. So the deeper a leave is in a tree, the shorter is its duration. The internal nodes are labeled by bottom-up propagation of the pitch of what is seen as the most important of the child nodes. A tree is constructed in this way for every bar, and the bar trees are linked to a common root node. The concept of an n-gram model over strings is extended to the tree case by a stochastic k-testable tree model. This model is represented by probabilistic tree automata, and such automata can be trained with music pieces of a similar style. The trained PTAs, representing different melody classes, are used to classify given melodies.

4.5 Unrestricted grammars

For Lerdahl and Jackendoff [7] the goal of a generative grammar is not to compose pieces of music (though they acknowledge that a suitable grammar could also be used for that purpose), but to describe the cognition of music, which is a psychological phenomenon. Despite the complexity of music, they believe that obvious forms of organization in music is the basis for understanding the complexity. Generative models are formulated to induce hierarchical structure over the musical surface. The hierarchical components of musical intuition investigated are grouping structure, metric structure, time-span reduction and prolongational structure. A distinction is made between well-formedness rules and preference rules. Firstly, grouping and meter are analysed independently. Meter (rhythmic structure) is a relatively local phenomenon, with structural dependencies usually limited to only a few bars. Grouping structure, describing how sequences of notes are hierarchically grouped together by an experienced listener during perception of the music, is a global phenomenon. The interaction of grouping and rhythm is described by time-span reduction, while prolongational structure models what an experiences listener perceives as tension and relaxation in the music. Transformational rules are applied to describe non-hierarchical events in a music piece.

Cope's EMI system implements complex strategies for the recombination of musical material using an augmented transition network, which has the expressive power of unrestrictive grammars [1]. His system classifies fragments of music into five semantic classes: Statement, preparation, extension, antecedents and consequent. Rules specifies the possible successors of each of the classes.

4.6 Sampling from statistical models

When we generate a music piece with a generative statistical model, we are sampling from the probability distribution that the model represents. The dominant approach to sampling is to take a random walk through the weighted finite state model, from the start state to a final state. However, Conklin [8] points out that the probabilities of such samples may be significantly lower than that of the paths though the model with the highest probabilities. In tasks such as harmonization it is appropriate to get the best sequence from the model. However, in music generation we cannot restrict ourselves to only the few pieces with the highest probabilities. A suggested way to overcome this problem is to use a form Gibbs sampling: Starting with a given music piece, one iteratively choose a random position in the music, modifies a note or a sequence of notes, and accept the change if the modified piece has a sufficiently large probability. However, such a model should also be able to preserve similarities in the music piece.

Chapter 5

Melody Generation with Markov Models

5.1 Implementation

Our system is primarily implemented in Java. However, we use Carmel and Tiburon to perform many of the operations on the automata that we build. A Bash script runs the music generation system. The script has parameters that can be specified through command-line arguments (although all parameters have default values). The script can also be instructed to perform only specified tasks.

Our system follows three steps in the generation process:

1. Analyse given music pieces.
2. Construct a model for the music pieces.
3. Generate new music pieces from the model.

In this chapter we describe the implementation of the Markov model in detail. In later chapters, we describe the models at a higher level, and only discuss implementation details where there are differences to the approach we follow here.

5.2 MIDI files and JMusic

The MIDI file format represent music pieces by event messages about the music, rather that with an audio signal. MIDI is a standard music file format in which a large number of music pieces are available. A symbolic representation of the music can be obtained directly from a MIDI file using an appropriate library. In contrast, with files in audio signal format, those signals must first go through a

transformation process to convert it into a symbolic music representation. We therefore work with MIDI files in our implementation.

We use the JMusic package to read from and write to MIDI files. In JMusic, a music piece is represented by a SCORE. A SCORE consists of a number of PARTS. The SCORE also stores global properties of the music piece, including its time signature, tempo and key signature. Every PART is played by an instrument and consists of a number of PHRASES. The PHRASES of a PART may overlap, and in practice they are different partial voices, and do not (as the name may imply) correspond to any time-based division of the music piece. In our implementation, we treat every PHRASE of every PART of the SCORE as a voice of the music. Every PHRASE consists of a (linear) sequence of NOTES and RESTS. Every NOTE has a pitch and a duration, and each REST has a duration.

5.3 Analysis

The class **Analysis** is used to analyse the training data. We read the MIDI files contained in a specified directory. For each file, we create an object of the **MusicPiece** class that handles the data processing of music pieces in our system. As mentioned above, we use JMusic to handle input from the MIDI files.

We treat every phrase from every part of the music piece as a voice, and represent it by an object of our **Voice** class. We store the pitch sequence and the rhythm (note duration) sequence of a voice. The pitch values are stored as MIDI pitch values, obtained from JMusic. A MIDI pitch value is an integer between 0 and 127 that represents the number of semi-tones the note is higher than the note 5 octaves below middle C. A pitch value of the minimum possible integer value (defined by Integer.MIN_VALUE in Java) indicates a rest. In our implementation we change the rest value to be -1.

We transpose the pitches of the training music pieces to the key of C major or A minor, the keys without any sharps or flats, depending of whether the piece is in a major or minor key. In JMusic format, the key signature is given as the number of sharps (key > 0) or flats (key < 0) in the key. (A key signature cannot have both sharps and flats.) From there we calculate the tonic of the key signature as follows:

$$\text{transposition} = (\text{key} * 7) \bmod 12,$$

where *transposition* is the number of semi-tones the current key is above C major or A minor. The reason for this formula is that key signatures are explained by the *circle of fifths*: Every time a sharp is added to a key signature, the tonic goes up by an interval of a fifth, and every time a flat is added, the tonic goes down by a fifth. A fifth interval is equal to 7 semi-tones, while an octave has 12 semi-tones.

While analysing the rhythm of a voice, we infer the bar structure of the piece. We keep a running total of the note durations during analysis, keeping in mind that an upbeat will cause an incomplete first bar. We insert bar separator symbols into our analysed rhythm sequences. Note duration is given by JMusic as a double value, with 1.0 denoting a crotchet. We found it more convenient to represent the duration internally such that 1.0 denotes a whole note. However, when we write out string sequences to data files, we need to use discrete values. We obtain an integer representation by multiplying the duration by 96 and taking the floor of that value as an integer, such that 96 denotes a whole note. This allows us to store rhythm divisions up to 64th notes, as well as corresponding triplets, without loss of precision.

We write the data obtained by analysis to *.data* files in a format convenient for use in our system. All the files are created in one specified directory. Every data file has two lines of space-separated words for every input music piece, though one of these lines will be empty if only a single sequence is represented. Pitches and chords of major and minor sequences, as well as rhythms of different time signatures, are stored separately. There are data files for the following types of sequences (for the Markov model implementation):

- pitch
- rhythm
- pitch-rhythm
- tempo

The class **Convert** is used to convert different representation formats of values. When we write out sequences to data files, we use the following formatting for symbols: For pitches, "p" followed by the integer pitch value, or "r" for a rest. For rhythms, "s" followed by the integer rhythm value for a non-rest, "r" followed by the integer rhythm value for a rest and "m" for a bar separator.

5.4 Modelling: Building and training automata

We build WFSAs and WFSTs to model the pitch and rhythm of music pieces. We build the automata with our Java classes. Some of the automata are constructed with weights, others are trained using Carmel.

The automata are represented by text files in the format used by Carmel. The format is as follows: The first line contains the symbol of the final state. After that, every line describes a transition between two states. Suppose we are making

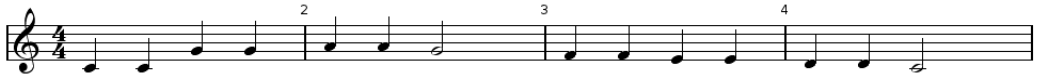

Figure 5.4.1: Example training melody 1

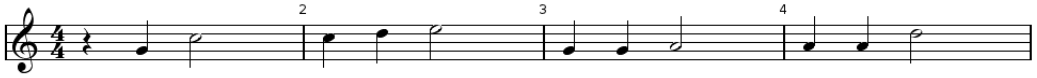

Figure 5.4.2: Example training melody 2

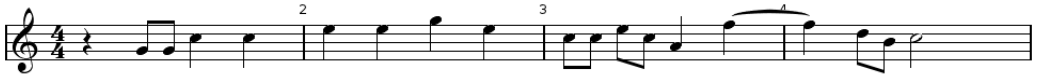

Figure 5.4.3: Example training melody 3

a transition from state 1 to state 2, with "a" as input symbol and "b" as output symbol on the transition, with probability 0.5. The line of the file representing the transition is as follows: (1 (2 a b 0.5)). The first symbol of the first transition in the file is the initial state. The weight and the output symbol is only specified in some types of automata. The input and output labels of transitions are not allowed to be integer values, and "*e*" represents an empty transition (no extra symbol is read from the string).

5.4.1 Higher order Markov chains

The order of the Markov chains used can be set as a parameter. The default value is 3. Suppose we are working with an nth order Markov chain. An *n-gram* is any subsequence of n consecutive symbols. The class **Gram** represents a symbol and the number of times it occurs. The class **NGram** represents the string of an n-gram and a list of Grams of the possible symbols following the n-gram. Note that a symbol may here consist of more that one character.

The class **CountNGrams** extracts n-gram information from the data files and construct WFSAs. For the pitch and rhythm, the following is done: For every sequence, we record the first $n-1$ symbol subsequences, each storing the first i symbols, $i = 1, 2, \ldots, n-1$. Then we record and count the occurrences of all the n-grams in the sequences. We also record the last n-gram of every sequence separately. We write this information to *.abc* files, since we will use it again for the

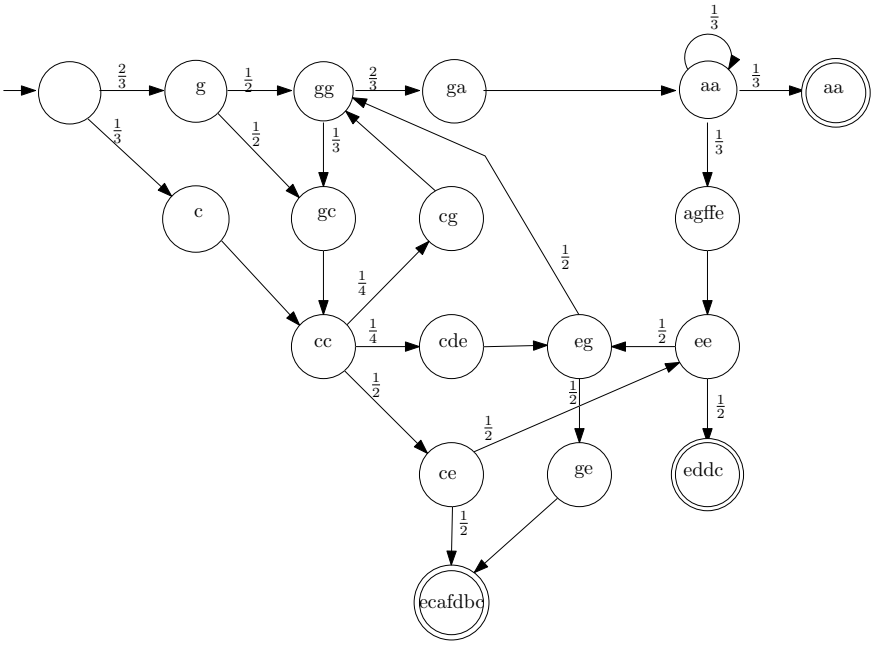

Figure 5.4.4: Example Markov chain for pitch

construction of further automata.

Then, using maximum likelihood estimation for Markov chains, as described above, we can construct an MC and represent it as a WFSA. We use a parameter to indicate if cadences should be enforced, i.e., if only sequences that end in an n-gram that occurs at the end of one of the input sequences should be accepted, or if any n-gram may occur at the end of the sequence.

Next we present an example of second-order MCs for pitch and rhythm build by analysing three 4-bar melodies in C major. The melodies are given in figures 5.4.1 to 5.4.3. Our example has some simplifications to illustrate the model construction better. Pitches are represented by their pitch classes. Rhythm is represented as in standard music notation. Our rhythm model excludes bar separators and rests. Figure 5.4.4 gives the Markov chain for pitches, and figure 5.4.5 gives the Markov chain for rhythm. Normalized transition weights are indicated on the transitions. Where only one transition in possible, no weight is indicated. Note that the empty state is the start start and that states with two concentric circles are final states. To reduce the number of states, subsequences that uniquely follow a symbol are encoded in the same state. Therefore a single transition may result in multiple symbols being generated. In our example the subsequence following the first symbol in the label of a state is generated by that state.

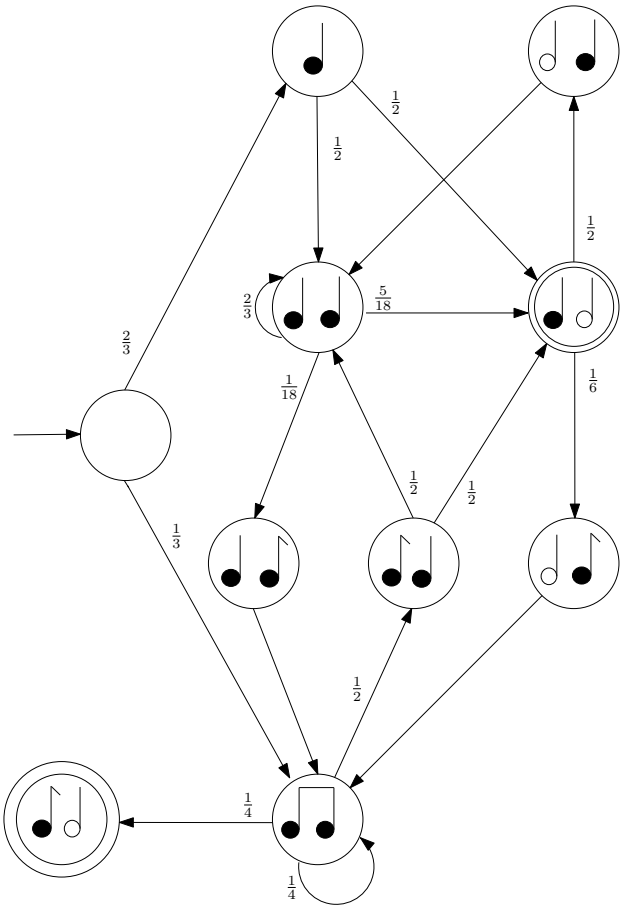

Figure 5.4.5: Example Markov chain for rhythm

5.4.2 Restrictions on generated output sequences

We want to place certain restrictions on the generated music. We do that by constructing FSAs that only accept sequences that adhere to those specific restrictions. By composing such FSAs with the WFSA used to generate pitch or rhythm sequences, we enforce the restrictions.

The class **BarLengthAcceptor** generates an FSA that only accepts rhythm sequences that consist of full bars, with the possible exception of the first and last bars. This exception is to allow for upbeats. In that case, however, the notes must still fit into beats of the bar. The motivation for this restriction is that it enforces structure in the rhythm of the piece. It prevents notes from being hold on over bars. This is sometimes acceptable in music, but as an exception, not as a rule. As bar separators are included in the rhythm sequences used for training, it is also included in the generated rhythms.

The class **NumberOfBarsAcceptor** generates an FSA that accepts rhythm sequences of a given number of bars. As bar separators are already included in the generated rhythm sequences, we only need to count the number of times they occur. Together with BarLengthAcceptor, this will ensure that the generated rhythm has the necessary number of filled bars.

5.4.3 Hidden Markov models

We model the relationship between pitch and rhythm as follows, with a HMM. Given a rhythm sequence, we want to find the probability distribution of pitch sequences occurring with the given rhythm sequence. We can then sample a pitch sequence from that distribution. The probability distribution is determined by a Markov chain for the pitch sequence, and by the probability of a single pitch occurring with a given rhythm. In terms of an HMM, we have the MC for pitch sequences as the hidden states and transition probabilities, and the rhythm sequence as the observed sequence. The emission probability distribution of each state is the probability distribution of the possible durations (rhythms) of a pitch. Given the observed rhythm sequence, we want to sample from the possible hidden state sequences given the observed sequence.

We implement the HMM with transducers in Carmel using noisy channel decoding, as described in the Carmel tutorial [11]. Suppose N is the pitch sequence and R is the rhythm sequence. Then, using Bayes' theorem:

$$P(N,R) = P(N|R) * P(R) = \frac{P(R|N) * P(N)}{P(R)} * P(R)$$

So $P(N|R)$ is directly proportional to $P(N) * P(R|N)$.

The class **PitchToRhythmTransducer** constructs an FST to convert pitch sequences into rhythm sequences. This FST has only one state, and is trained by Carmel, given the pitch and rhythm sequences as training data. The composition of this WFST and a WFSA for pitch can then be used to convert a rhythm sequence into a pitch sequence sampled from the wanted probability distribution. Figure 5.4.6 gives the transitions (self-loops on the only state of the WFST) of the pitch-to-rhythm WFST for the example training melodies given above.

We give a graphical model representation of the melody HMM in figure 5.4.7. This graphical model is simple, but later we will use the same representation for more complex models. In the model, a singly circled node represents a random variable and a double circled node represents a deterministic variable. A shaded circle represents an observed variable, while an unshaded circle represents a latent variable. A directed arrow from node A to node B indicated that variable B is conditionally dependent on variable A.

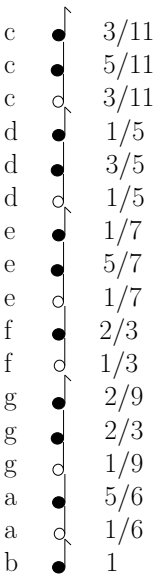

Figure 5.4.6: Example pitch to rhythm transitions

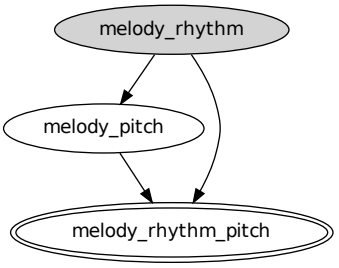

Figure 5.4.7: Graphical model representation for the Markov melody model

5.5 Music generation

We now use the trained transducers and acceptors to generate music pieces. The class **SampleTempo** samples a tempo from the tempos of the pieces in the training data. The sampling is done as follows: The tempos are sorted in ascending order in a list. We choose i randomly in the range of list positions. Then we sample from the uniform distribution between the values at positions i and $i + 1$ in the list.

Using Carmel, we generate a rhythm sequence from the composition of the

rhythm WFSA with the rhythm restriction FSAs described above. Then we generate a pitch sequence from the intersection of the pitch WFSA and the pitch-to-rhythm WFST, with the generated rhythm sequence given as input sequence.

The key signature of the generated music can be specified as a parameter. The information describing the generated music piece, including the pitch and rhythm sequences, are written to a *.bc* file. The class **Synthesize** writes the generated music piece out to a MIDI file, using JMusic.

5.6 Alternative Markov Models

5.6.1 Generation with a Markov chain for pitch-rhythm pairs

We also implement an alternative approach to modelling the relationship between pitch and rhythm. We can encode a melody as a single sequence of symbols, each of which encode the pitch and the duration of one note. We can train an nth order Markov model from such sequences. Then we can generate pitch-rhythm sequences directly from that Markov chain.

Such a model can model the training data slightly closely than an HMM model: When a new pitch and duration is generated, the history used for both choices include the previous n pitches, previous n durations, and the dependency between the new pitch and duration. In contrast, with an HMM, for pitch the previous n pitches is used as history, and for duration the previous n durations and the current pitch is used. However, the penalty for such a model is that the distribution of n-grams of pairs will be more sparse than the distribution of n-grams as modelled by an HMM.

5.6.2 A bar approach to rhythm generation

There is a limited number of acceptable rhythm sequences that have a given, fixed total duration. The bar is a natural division of the rhythm sequences of melodies. So we construct a Markov chain from symbols that each encode the rhythm of a whole bar. This model will be able to generate all rhythmic patterns that occur in as bars in the input data, and will be able to take a longer context into consideration than our standard Markov chain for rhythm.

5.6.3 An interval approach to pitch generation

An alternative way to modelling pitch sequences is by modelling the intervals between pitches. Often in music we have reoccurring patterns that have different

pitch values but the same intervals between pitches. By using intervals instead of absolute pitch values, we can model such dependencies. We will see in some of the models that we describe later how to take advantage of the interval representation approach.

To implement the interval approach, we need to analyse the intervals between pitches. However, we still want a symbol to represent each note in a sequence. Therefore, we choose a fundamental melody note, and the first note in the melody is described by the interval between the fundamental note and this note. The fundamental note is determined as follows: We take the pitch with pitch class C that is nearest to the average of the melody notes of the training data. This will give some meaning to the first interval, and should make it small. The rest of the pitches in the melody are each described by the pitch interval from the previous note to the current note.

The interval between two notes is the number of semitones that the pitch increases (positive) or decreases (negative). Intervals are represented in our data files as follows: "i" followed by an absolute interval represents a nonnegative interval between notes. "j" followed by an absolute interval represents a negative interval between notes. We analyse the intervals and write interval sequences to text files in similar way that we followed for pitches.

After we have trained a Markov chain for intervals, modelled the relationship between intervals and pitches (with class **IntervalToPitchTransducer**), and generated interval sequences, we want to decode the generated interval sequence to a pitch sequence. Class **IntervalToPitchTransducer** constructs a FST that will convert an interval sequence to a pitch sequence. When we construct the transducer, we need to know the chosen fundamental pitch from the training data. Such a transducer will be restricted to give as output pitch sequences that fall within a certain range. The transducer approach will therefore not work for arbitrary interval sequences. However, we assume that the pitch sequence that corresponds to an appropriate interval sequence will fall into the range of pitch sequences in the training data. The advantage of the transducer representation is that we can later compose such an FST with other models.

We can also construct a model that uses the Markov chains for pitches and for intervals. To do so, compose the interval WFSA, the interval-to-pitch FST and the pitch WFSA. By sampling from that transducer we will generate sequences that satisfy the MC probability distribution of the pitches and of the intervals.

5.7 Conclusion

We described the implementation of a Markov model for melody generation. Despite the simplicity of the model we can generate melodies that shows good lo-

cal structure. Due to the emergent properties of music, many of these generated melodies will not sound well, but others will exhibit, to the experienced listener, a form of structure or meaning that goes beyond the local dependencies modelled by the Markov model.

The main steps of the modelling process and its implementation described here, in the context of Markov models, are followed by the models that we describe in the following chapters.

Chapter 6

Markov Modelling of Harmony

6.1 Chord analysis

To model the harmony, we must first analyse the chords in the music piece. An extensive analysis of chord recognition procedures is given by Jiang [30]. We proceed as follows: We try to assign a chord to every beat in the music piece. By a beat we mean one of the divisions of a bar, which length is indicated by the denominator of the time signature. So for 4-4 time, the beat is a crotchet. We use a template-based method: We assign one of a predefined collection of chords to the beat. The possible chords are the empty chord (meaning no chord is classified), the 12 major chords and the 12 minor chords. It is also possible to extend these to other chords such as diminished, augmented and added seventh chords, but these are not very common in classical music and can usually be regarded as similar in function to that of a related major or minor chord.

The chord representation used is as follows: We have an integer vector of 12 elements, one for each of the pitch classes. In each vector entry we store the total duration of notes in the beat that are in a pitch class. We use a scale of 12 subdivisions of a beat, so one note sounding for the full beat value adds 12 to the value of some vector entry. Pitch classes that occur for longer and in more notes in the chord are weighted more in the vector. For our predefined chords, we give the tonic a value of 24, as a note that occurs twice, and the other two tones a value of 12. This corresponds to the standard representation of the chords in four-part harmony.

To do chord classification for a beat we compute the Euclidean distance between the chord representation vector of the beat and each of the template chords. We classify the chord of the beat to the chord template with the least distance to the chord representation vector. However, this distance needs to be greater than the distance of the chord representation vector to the empty chord vector, else no chord

is assigned to the beat.

In order to be able to model the relationship between chords and melody notes, we also choose, for every beat, a representative note from the melody voice. We choose the note that has the longest duration in the beat, the first one occurring breaking any tie. This is the note that is most likely intended to sound together with the chord. If, however, the pitch class of this note does not occur in the classified chord of the beat, we try to choose another note from the melody in that beat which pitch class is in the chord. Otherwise, we remove the chord classification for purposes of training our model. The reason for this is that in music generation we want to choose chords that sound with the melody notes, and when assigning chords we work again with representative notes of the melody. We do not want to train our model to explicitly allow such dissonances, as it will not contribute to generating harmonious music.

The results of our chord classification are satisfactory. Though we did not formally analyse the accuracy, when the melody is reconstructed with the classified chords, the result sounds very similar to the original harmonization.

The representation of chords in our text files is given by "c" followed by a string representation of the chord, or "r" for a rest chord. To get the string representation, for every pitch class that appears in the chord a letter "A" to "L" representing that pitch class is added, in alphabetical order.

6.2 Chord generation

We model the relationship between pitch and chords with an HMM, in a similar way to how we modelled the relationship between pitch and rhythm. Here, for pitch we are working with the representative pitch sequence described above. We find the representative pitch sequence with the class **ConvertRepNotes**.

We want to find the optimal harmonization of our melody, i.e., the chord sequence that will maximize $P(C, N)$, where C is the chord sequence and N the note sequence. Let the chord sequence be modelled by the hidden states of an HMM and the note sequence by the emission sequence. Then, using the Viterbi algorithm, we can find the optimal chord sequence for a given note sequence. We follow broadly the same approach as proposed in [3].

The class **ChordToPitchTransducer** constructs an WFST to convert chord sequences into pitch sequences. We use a form of additive smoothing to let every chord map to every pitch in the range of melody pitches that is is the pitch class of one of the chord notes. We add 1 to the count of each of these pitches. We use these modified counts to compute the transition probabilities of the WFST.

The composition of this WFST and a WFSA for chords forms the hidden Markov model. We then use the Viterbi algorithm to find, for the given representative pitch

sequence, the optimal chord sequence for the probability distribution that the HMM represents.

6.3 Multiple voice generation

The standard harmonization procedure in western classical music is that of four-part harmony. So we extend our chord generation model to generate three accompaniment voices to the melody.

Firstly, we generate a bass voice. In the training data, we identify the base voice as the voice that is, on average, the lowest in a music piece. We construct a Markov model for the representative notes of the bass voices in the training data. We use the class **ChordToBassTransducer** to model the distribution of bass notes that occur with every assigned chord in the training data. We add the extra restriction that we only accept bass notes that are in a pitch class of the chord it is associated with. Implicitly, we also model the concept of chord positions (root position, first inversion, second inversion). The chord position is determined by the pitch class of the chord that is the bass note of a chord configuration. We then use an HMM that have the chord sequence as hidden sequence and the bass note sequence as observed sequence, to generate the bass line (a bass note for every beat).

Then we generate two inner voices that, together with the melody and bass voices, gives the implied harmony of the chord at each beat. We model this in two steps. First, we construct a customized FST that encode all the valid inner voice sequences. The lower of the voices (the third voice) in the input sequence and the higher voice (the second voice) is the output sequence. We accept voice configurations at each beat that satisfy the following restrictions:

1. None of the four voices ever cross each other. Therefore the second voice must always be lower than the melody, and the third voice must be higher that the bass note but not higher than the second voice.

2. All three pitch classes of the chord should be contained in the four notes at the beat. We do not place a restriction on which pitch class may appear twice, but do we allow any pitch class to be left out.

Secondly, we construct a WFSA that will accept inner voices with good voice leading. As we cannot in general identify the second and third voices in the training music pieces, we construct and train the model from all voices in the input pieces. We then compose the inner voice FST from left and right with this automata, and sample the best input/output sequence pair from the composite automata. We therefore find the most acceptable inner voices that will produce the harmony of the generated chords. In this respect, our model is more powerful than the chorale harmonization model proposed in [3].

6.4 Ornamentation

In general, accompaniment voices are not played in blocks, at every beat in the music. Notes repeated on the same pitch may be combined into one longer note, and extra notes can be inserted to improve voice movement (the most common example is to insert a middle note if there is an interval of a third between two notes). We model this ornamentation of the accompaniment voices with an HMM. To do so we first encode the pitch and rhythm of a note sequence in a beat as a single symbol string. The notation we use is as follows:

- For a rest: "r" and the rest duration.
- For a note that is played: "p", the pitch value, "s" and the note duration.
- For a note hold on from the previous beat: "q", the pitch value, "s" and the note duration.

All durations are only for the current beat.

We construct a Markov model of all voices in the training data represented in this format. Then we construct, with the class **RepsToNotesTransducer**, a WFST to model the possible ornamentations associated with each of the representative notes of every voice. We compose these models to form an HMM to convert representative note sequences of the three accompaniment voices into ornamented note sequences. During synthesis we decode the ornamentation sequence notation to pitch-rhythm pairs. We give a graphical model representation of our harmonization model in figure 6.4.1.

6.5 Conclusion

A limitation to our harmonization model is the inability to model the parallel or diverging movement of pairs of voices. An example of the importance of this in music is the harmonic principle that parallel voice movements, in intervals of fifths or octave, should be avoided. The model can also be extended to include the principle that intervals between inner voice notes should be as small as possible.

We saw that to do 4-voice harmonization we had to make a special construction and exploit certain properties of the common harmonization process. In general, we would like to be able to generate n accompaniment voices that all have acceptable voice leading and can simultaneously model dependencies in movement between pairs of voices. To do that, we will need to use a more powerful model than a transducer. Directed graphical models, which generalize the algorithms used in HMMs, should be investigated in future work to overcome these model limitations.

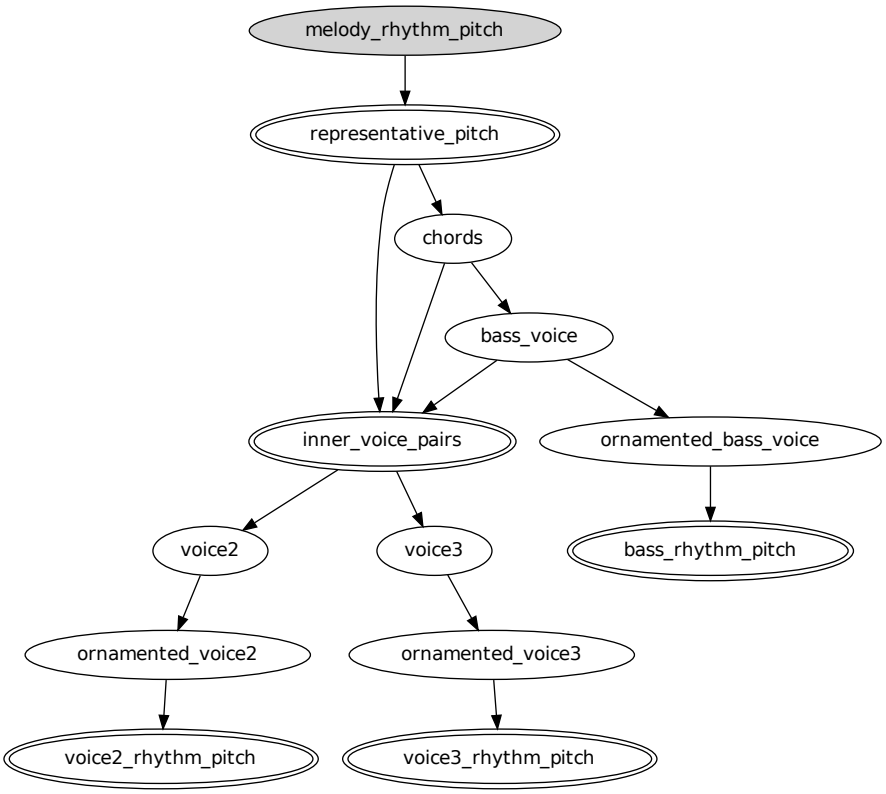

Figure 6.4.1: Graphical model for harmonization

Chapter 7

Abstract Melody Clustering Music Generation

7.1 Abstract melodies

In models for Jazz improvisation proposed in [15] and [24], notes are labeled with tone categories. These categories are based on the relationship between the pitch of a note and the chord that the note is played over. We simplify these categories to tone categories that are appropriate for classical music:

- **C**, a chord tone. The pitch class is in one of the pitch classes of the chord.
- **A**, an approach tone. The note "approaches" a chord tone. It precedes or follows a chord tone, and differs by one or two semi-tones from it.
- **X**, an arbitrary tone. Any note that is not a chord tone or an approach tone.
- **R**, a rest.

Note that the concept of a color tone used in Jazz music is not applicable to classical music.

We want to describe the melody in an abstract way. We use three elements: The note category, the interval between notes, and the duration of each note.

Gillich *et al* [24] use the concept of a *slope*, a subsequence of melody notes that unidirectional, i.e., either all intervals between them are positive (an ascending sequence) or all intervals are negative (a descending sequence). The melody is segmented into slopes, and each slope is describe by the range of intervals in the slope, and the note category and duration of each note. However, in our model we do not work directly with slopes. Rather, for every note we store the interval associated with it (the interval between the previous note and the current note).

Later, in one of our music generation models, we will introduce a mechanism to relax the intervals in an abstract melody.

To construct a grammar for the abstract melodies, we break the melody up into time windows of predefined length. Gillich *et al* found that 4-beat fragments in the 4/4 meter (equivalent to the length of 1 bar) achieve the best balance between continuity and originality. Therefore we see bars as natural divisions of a music piece, and we use bars of abstract melodies to train our model.

7.2 K-means clustering

To express in our model the concept of reoccurring similar fragments of music, we cluster similar abstract melodies together. Note that, in [24] the goal of clustering is to increase the variety in the recombination of melodic ideas when generation is done from a Markov chain. However, we want to go further than that and use the concept to model reoccurring melodic ideas in music pieces.

We use the k-means clustering algorithm (see a description in [13]) to cluster the abstract melody bars into k clusters. For the algorithm, we need to represent each contour as values on an n-dimensional plane, and define a distance metric between such n-tuples. Here we use the Euclidean distance metric and represent each abstract melody by 7 parameters (based on [24]). The note durations are scaled (as double value) such that the length of a beat is 4. The parameters are:

1. The number of notes in the abstract melody.
2. The location of the first (non-rest) note in the bar.
3. The total duration of rests.
4. The *average maximum slope*: For each of the slopes in the abstract melody, we find the absolute maximum interval between notes. Then we take the average of all those maximum values.
5. Whether the first note is at the start of a beat (0) or off-beat (1).
6. The number of times the interval contour changes direction.
7. The *consonance*: For each note, we compute the note duration times a coefficient for the note category. We choose the coefficients as 0.8 for a chord note, 0.6 for an approach note and 0.1 for an arbitrary note. The consonance is the sum of these values. It is an indication of how "pure" the notes are in relation with the chords.

K-means clustering is an EM algorithm. Every abstract melody is represented by a vector of size 7 as described above. The number of clusters is predetermined. We use a "rule of thumb" to choose the number of clusters: $k = \sqrt{\frac{n}{2}}$, where n is the number of distinct abstract melody vectors. As the initial step, we choose for every cluster a random vector as mean element. Then we apply the 2 steps of the EM algorithm iteratively:

1. Expectation step: We assign every vector to a cluster j such that the (Euclidean) distance between the vector and the mean vector of cluster j is a minimum.

2. Minimization step: We update the cluster means by calculating the mean of all the vectors assigned to that cluster.

The objective score is the total distance of all the vectors to their cluster means. We iterate the algorithm until the change in the objective score falls below a certain threshold.

The algorithm, with the vector representation described, shows good convergence and is scalable for different choices of number of cluster. We represent an abstract melody with the class **Contour**. We represent a cluster of abstract melodies with the class **ContourCluster**.

7.3 Markov chains for music generation

The basis for our generation model is a higher order Markov chain of the clusters of abstract melodies. To construct this model, we analyse all the training music pieces to construct abstract melody sequences. Then we cluster the abstract melody bars with the k-means clustering algorithm, using the **ClusterContours** class. The file *slope-clusters.info* is written out. We store, for every cluster, all the abstract bar melodies in that cluster, as well as the number of times each of the abstract melodies appear in the training data. We then re-analyse all the melodies in the training data to find for each melody the cluster sequence of the abstract bar melodies. Then we build a Markov model from those cluster sequences.

The class **SampleFromSlopesTransducer** constructs a WFSA that encodes the abstract melodies for each of the clusters. The probability of each abstract melody for a given cluster is determined by maximum likelihood estimation.

Now, when we use a generated abstract melody to generate a concrete melody, there are two approaches: We can first generate chords independently of the abstract melody and then choose notes that satisfy the note categories of the abstract melody over the chords, and satisfy the intervals in the abstract melody as close as

possible. The second approach is that we generate a concrete melody from the intervals of the abstract melody, disregarding note categories. We can then generate chords to harmonize the abstract melody.

In the approach where chords are generated independently, we construct an FSA that encodes the possible pitch sequences that satisfies the tone category sequence in the abstract melody, over the given sequence of chords. The class **TerminalPitchAcceptor** construct such an FSA for the given abstract melody and chord sequence. The chord sequence is generated by a Markov chain that is constraint by the length of the piece (in number of bars) that we want to generate. A chord is generated for every beat, as in our other chord models.

In general we have the problem that we will not be able to find a pitch sequence from those possible pitch sequences that will satisfy the tone category sequence and the interval sequence in the abstract melody. To address this problem, we allow for the relaxation of intervals in the interval sequence. We construct a single-state WFST that will change the value of an interval with a penalty probability. We only allow an increasing interval to be changed into another increasing interval, and a decreasing interval to be changed into another decreasing interval (to preserve the contour of the abstract melody). We also limit the number of semitones that we can change the interval with to 7. To an interval change of i semitones we give a weight of 2^{-i}.

To generate a pitch sequence that satisfies the above constraints as well as possible, we construct (on the fly) a transducer cascade from the composition of the interval-relaxer WFST, the interval-to-pitch FST and the pitch sequence acceptor based on tone categories. We give the interval sequence of the abstract melody as input to this cascade, and sample as output the best pitch sequence that satisfies these constraints (if such as sequence exists).

In the approach where we do not take chords into consideration, we just need to convert the intervals in the generated abstract melody into pitches, using the interval-to-pitch FST.

In either of the models, we can use the harmonization model described in chapter 6 to generate accompaniment voices for the generated chords. We give graphical model representations for the models in figures 7.3.1 and 7.3.2.

7.4 Other clustering methods

We did also investigate other approaches for clustering. One proposed way is to construct a vector that describes the abstract melody at fixed time steps by the interval change in pitch at each time step. However, in this case we have to modify the concept of the mean of the vectors in a cluster. One approach that we experimented with is to choose as mean of a cluster the vector in the cluster that has the

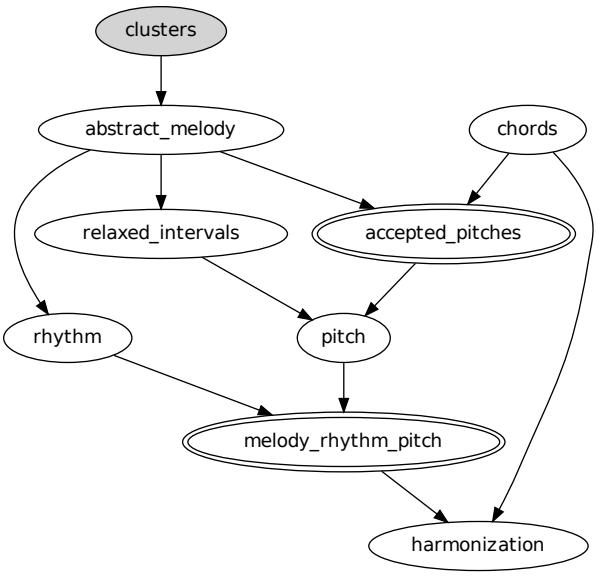

Figure 7.3.1: Graphical model for chord-dependent abstract melody generation

minimum total distance to the other vectors in the cluster. Another approach is to eliminate the calculation of mean vector. Rather, in each iteration of the algorithm, compute the total distance of each vector to all the vectors in each of the clusters, and then assign the vector to the cluster to which it has the least total distance.

In our experiments, however, both these approaches either did not converge properly or assigned almost all the vectors to two or three clusters and only a few to each of the other clusters. There may be situations where we want such a clustering, but in this case we want to spread the abstract melodies as evenly as possible among the different clusters.

7.5 Conclusion

This model remains in essence just a sophisticated Markov model. To show that the model succeeded in modelling the internal similarities in music pieces, we need to see that some clusters appear multiple times in our generated music. The choice of the number of clusters is very important: If there are to many clusters, the chance of having repeated clusters in a generated music piece is very small. If the number of clusters is too small, the abstract melodies inside clusters will differ by too much

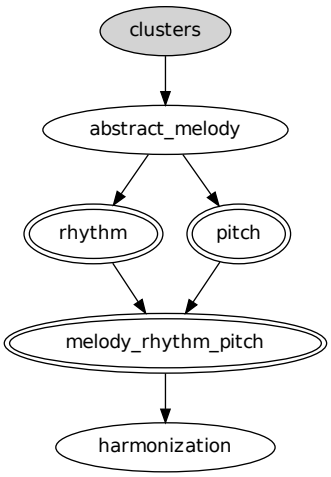

Figure 7.3.2: Graphical model for chord-independent abstract melody generation

to model similarities between bars. However, by choosing appropriate parameters and generating longer pieces of music, we were able to see in generated pieces the non-local dependencies that we set out to model.

Chapter 8

Context-free Grammar Model

8.1 Rhythm model

We constructed a context-free grammar for rhythm that is based on the principle of the hierarchical organization of rhythm: every note can be divided into two (or three) notes that is each half (or a third) of the duration of the original note.

In this CFG the non-terminals are B, the initial non-terminal, M, the non-terminal representing a bar of notes, and non-terminals in the form "S[n]", where "[n]" is the note duration that the non-terminal represents. The terminals are our standard representation for note duration: "s[n]" for a note and "r[n]" for a rest.

Our initial rules are in the form $B \rightarrow M\ M \ldots M$, where the number of M's are the number of bars in the piece. In the grammar that we train we have such a rule for every possible piece length in the training data. When we generate a piece consisting of a certain number of bars, we replace the initial rules with only one initial rule to enforce that number of bars. Note that in this model all our training data must be preprocessed to have only full bars. Therefore we cannot model upbeats with this grammar.

Alternatively, it is possible have rules $B \rightarrow M\ B$ and $B \rightarrow M$ that can model an arbitrary number of bars. However, these rules will create a comb structure that will increase the complexity of our tree. For computational reasons we want to keep our tree structure as simple as possible.

We construct a CFG for every meter of pieces in the training data. M only occurs on the left side of one production rule: $M \rightarrow S[n]$, where n is the bar duration in the time signature we are working in.

The production rules to replace non-terminals that represent duration with more non-terminals that sum to the same total duration are in one of the following forms:

- $S[2 \cdot x] \rightarrow S[x]\ S[x]$

- $S[4 \cdot x] \rightarrow S[x]\ S[2 \cdot x]\ S[x]$
- $S[4 \cdot x] \rightarrow S[3 \cdot x]\ S[x]$
- $S[4 \cdot x] \rightarrow S[x]\ S[3 \cdot x]$
- $S[3 \cdot x] \rightarrow S[2 \cdot x]\ S[x]$
- $S[3 \cdot x] \rightarrow S[x]\ S[2 \cdot x]$
- $S[3 \cdot x] \rightarrow S[x]\ S[x]\ S[x]$

Note that we structure the rules to prevent ambiguous derivations as far as possible. We also allow for the construction of triplets and dotted note durations. To construct the production rules we use the durations of all the notes, not longer than a bar, in the input data, and their reachable note divisions (by recursively applying the rules). For every duration non-terminal we have a rule that substitutes it with the corresponding note duration terminal, and another one for the corresponding rest duration terminal.

8.2 Interval-based pitch model

We base our pitch generation CFG on the one proposed by Gibert and Conklin [26]. Their grammar is based on the concept of melodic reduction: Often in music, a melody can be made more interesting by inserting a note between two existing notes. If we inverse that process, we can reduce a given melody to a compact structure. We want to encode this intuition as a probabilistic CFG. However, we cannot directly have a context-free production that replace a pair of notes with three notes, as context-free production rules may only have one non-terminal on the left-hand side. But if we represent the pitch sequence as a interval sequence, we can define such rules, by replacing an interval with two intervals that sum to the original interval.

At the highest level of our grammar we generate a sequence of uniform non-terminal symbols, each which can be substituted with any non-terminal interval. Each of these non-terminals are then, by applying recursive rules, replaced with an interval sequence that sum to the interval represented by the original non-terminal.

For notational convenience we represent in the chapter interval non-terminals with "I[n]", where n can be positive or negative. However, in our implementation we represent negative intervals with "J[-n]", so that the integer part is non-negative. Our initial non-terminal is S and the initial rules are $S \rightarrow I\ S$ (the "new rule") and $S \rightarrow I$. Therefore we can replace S with an arbitrary sequence of I's, since we do not know beforehand how many I's there will be in the parse of a melody.

Therefore we cannot flatten the tree structure created by the CFG as we did in the case of rhythm. The comb-like structure of the trees is computationally expensive when we later train our model.

We work within a certain range of intervals: [26] proposes -24 to 24, and in our Bach chorale training data that turned our to be the approximate range of intervals in the melodies. We have a rule $I \to I[n]$ for every interval in the range. We can also replace every non-terminal interval with the corresponding terminal for that interval. The terminal interval notation is the same as for earlier models: "i" followed by the interval value is an increasing interval, and "j" followed by the absolute interval value is a decreasing interval.

We have non-terminal interval substitution rules of the forms:

- Repeat rule: $I[n] \to I[n]\ I[0]$
- Neighbour rule: $I[0] \to I[n]\ I[-n],\ |n| \leq 5$
- Passing rule: $I[n] \to I[n1]\ I[n2],\ n1 + n2 = n,\ n1 \cdot n2 > 0,\ |n| \leq 7$
- Escape rule: $I[n] \to I[n1]\ I[n2],\ n1 + n2 = n,\ n1 \cdot n2 < 0,\ |n| \leq 4$

We assign prior weights to each rule, mainly to minimize the number of times the new rule is applied. $S \to I\ S$ has weight 0.1, $S \to I$ has weight 0.9 and every rule $I \to I[n]$ has weight 0.1. The repeat rules and neighbour rules have weight 0.75, while the passing and escape rules have weight 0.5.

8.3 Training the CFGs

We use Tiburon to train our grammars, since it has build-in functionality to train a CFG given the CFG production rules and training sentences. However, if we use this, Tiburon fails to assign meaningful probabilities. We therefore implement the EM algorithm for CFG grammar training as described in [16], using other Tiburon commands.

We iterate over the following steps:

1. Expectation step: For every training sequence, we find the n most likely parse trees over the current PCFG.
2. Maximization step: We reestimate the probability of each rule in the PCFG by maximum likelihood estimation over the parse trees.

To do parsing with Tiburon, we convert the CFG to an equivalent tree-to-string transducer. By composing that transducer on the right with a string (training

sequence), and sampling from the composition, we get the n most likely parses for the string. We store these parses in a temporary file. Next we train the tree-to-string transducer on all of these sentences with the parse trees that we found for them. We can convert the transducer, with re-estimated weights, back to a PCFG.

8.4 Melody generation

We use Tiburon to sample interval and rhythm sequences from the trained CFGs. We described earlier how we limit the rhythm generation to a fixed number of bars. Next, we want to generate an interval sequence that has the same length as the generated rhythm sequence (excluding rests). We convert a copy of the rhythm sequence to a sequence of "s" symbols of the same length. We modify the final trained interval tree-to-string transducer such that the string it produces will consist of "s" symbols, instead of terminal intervals. We compose that transducer with the string of "s" symbols, and we take the yield of a sampled tree from that composed transducer. This gives an interval sequence of the required length. Note that this model does not model the probability distribution of intervals given rhythm.

The interval sequence can be converted to a pitch sequence as before, and we can harmonise the generated melody with our Markov model for harmonization. We give a graphical model representation of the melody generation model in figure 8.4.1.

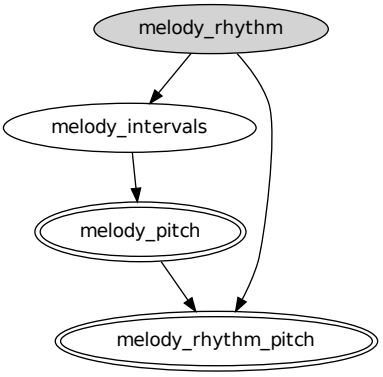

Figure 8.4.1: Graphical model for CFG model

8.5 Conclusion

Context-free grammars for music are able to assign meaningful structure to music pieces. They are also more powerful than Markov models in the formal language description of the music they can generate. An example of an important non-regular structure in music that our interval model can generate, is than of $\{a^n b^n | a = i[n], b = i[-n]\}$.

However, in our music generation experiments, the music that this CFG model generate is not very interesting. It assigns high probabilities to music that shows regularity, but the variation between expected and unexpected motives in a music piece adds a lot of richness to a music piece. A CFG also cannot express well the concept of repetitions of longer phrases in music that it generates.

Chapter 9

Composer Recognition

9.1 The classification problem

Suppose we have training data from k composers $C_1, C_2, \ldots, C_n$, and we construct, for each composer, a generative model for the pieces of the composer. Then, given a music piece for testing, we want to classify that music piece to one of the composers. Bayes' rule gives that:

$$P(C_i|x) \propto p(x|C_i)P(C_i)$$

We assume that the prior probabilities of all the composers are the same (though it would also be possible to let the prior of a composer represent the popularity of the composer or the number of pieces he wrote, relative to the other composers that we model). Therefore the classification problem is equivalent to maximizing $p(x|C_i)$, which is the probability assigned to piece x by the generative model for composer i.

We use the Markov model for melody as generative model for composer recognition. However, our model should, as far as possible, assign a non-zero probability to every piece that it is given. The reason the higher order Markov model we used in our music generation models will not always do so, is due to the sparsity of n-grams in the training data. It is reasonable to assume that all the pitches in the testing melody will occur at least once in the training melodies of the composer model, but we cannot make that assumption for n-grams.

9.2 Katz's back-off model

To overcome the sparse data problem, we use Katz's back-off model [31], one of the most widely used smoothing methods in the construction of language models for speech recognizers.

The main idea is that if a k-gram does not occur in the training data, we *back off* and use the model for the $k-1$-gram that has the same suffix. The back-off process can be done recursively until we find a matching gram. To allow for this process, we need to decrease the probability given to some of the n-grams by maximum likelihood estimation and redistribute the "freed" probability to the back-off transitions. The main equation for the back-off probability is:

$$P_{\text{back-off}}(w_1|w_{i-n+1}\dots w_{i-1}) = \begin{cases} d_{w_{i-n+1}\dots w_i}\frac{c(w_{i-n+1}\dots w_{i-1}w_i)}{c(w_{i-n+1}\dots w_{i-1})} & , c(w_{i-n+1}\dots w_i) > k \\ \alpha_{w_{i-n+1}\dots w_{i-1}} P_{\text{back-off}}(w_i|w_{i-n+2}\dots w_{i-1}) & , \text{otherwise} \end{cases}$$

The default value for k, that we will use, is 0. c is the number of occurrences of the given n-grams. d is the discount coefficient of the n-gram. The model uses Good-Turing estimation to compute the discount coefficient. The Good-Turing discount of the n-gram is

$$d = \frac{c^*}{c}$$

where

$$c^* = (c+1)\frac{N_{c+1}}{N_c}$$

and N_i is the number of n-grams that occur exactly i times in the training data.

To determine the value of α, it is convenient to first define a function β:

$$\beta_{w_1\dots w_{m-1}} = 1 \; - \sum_{w_m:\; c(w_m)>0} P(w_m|w_1\dots w_{m-1})$$

Which is the total weight that is freed. Then we for α we have

$$\alpha_{w_1\dots w_{m-1}} = \frac{\beta_{w_1\dots w_{m-1}}}{\sum_{w_m:\; c(w_m)=0} P(w_m|w_2\dots w_{m-1})}$$

Now, in general, when we compute Good-Turing estimates, there will be cases where, for an n-gram that occurs r times, there will be no n-gram that occurs $r+1$ times. The re-estimated count will then be 0. To address this problem, Katz proposes that we do not reestimate n-grams whose counts are higher than a threshold value r, considering their maximum likelihood estimates as reliable. We choose the threshold as 5. The discount coefficient for re-estimated n-grams is then

$$d_c = \begin{cases} \frac{\frac{c^*}{c} - \frac{(r+1)N_{c+1}}{N_1}}{1-\frac{(r+1)N_{c+1}}{N_1}}, & 1 \le c \le r \\ 1, & c > r \end{cases}$$

We also implemented a simpler back-off model. From every n-gram, we back off to the corresponding 1-gram with a probability of 0.01. The problem with such a model is that there may arise cases where a higher probability is assigned to a path that goes through a back-off state than to a (possible) path that does not include a back-off. In such a case we throw away valuable n-gram information.

9.3 Recognition training and testing

To test our generative model for recognition, we use the standard approach of dividing our music pieces into training and testing data. We do that with the **SplitTrainingTesting** class. The splitting ratio is 70% training data and 30% testing data, and the splitting is done stochastically. For the basic model that we use, we firstly train a melody pitch Markov model with Katz smoothing for the training pieces of each composer. Next we analyse all the testing pieces. For each of the generative models, we use Carmel to compute the likelihood of every testing piece. We use log likelihoods for accurate representation of the probabilities (which can be very small). Then we classify each testing piece to the composer whose generative model assigns the highest probability to the piece.

9.4 Results

We test our model by classifying some classical music pieces.

Our first test is with Bach Chorales and the Händel Messiah. Both are Baroque composers. For a model with major pitch sequences, we use 300 Bach Chorales and 45 pieces from Hädel's Messiah. The Bach Chorales in the testing data are classified very accurately, but the Händel Messiah pieces not so accurate. See table 9.4.1 for the classification results.

Data set	Bach	Händel	Not classified
Bach Chorales	92.13%	7.87%	0%
Händel Messiah	42.86%	50%	7.14%

Table 9.4.1: Bach/Händel composer recognition

In our second experiment we classify data sets of Mozart Symphonies, Beethoven Piano Sonatas and Tchaikovsky's Swan Lake. Mozart and Beethoven are both classical period composers, and Tchaikovsky is a romantic period composer. Note that our model is not able to classify the Mozart Symphonies successfully to Mozart, but it is able to distinguish that it is in by a classical composer and not by a romantic composer. For the major pitch model we use 54 Mozart pieces, 45 Beethoven Pieces and 15 Tchaikovsky pieces. See table 9.4.2 for the classification results.

In our third experiment we classify data sets from the Händel Messiah, Mozart Symphonies and Schubert Piano Sonatas. Händel is Baroque, Mozart is classical and Schubert is romantic. Here our model can more clearly distinguishing between composers of different musical style periods. For the major pitch model we use 45

Data set	Mozart	Beethoven	Tchaikovsky	Not classified
Mozart Symphonies	47.06%	52.94%	0%	0%
Beethoven Piano Sonatas	0%	85.71%	0%	14.29%
Tchaikovsky Swan Lake	0%	0%	75%	25%

Table 9.4.2: Mozart/Beethoven/Tchaikovsky composer recognition

Händel pieces, 54 Mozart pieces and 36 Schubert pieces. See table 9.4.3 for the classification results.

Data set	Händel	Mozart	Schubert	Not classified
Händel Messiah	64.29%	7.14%	28.57%	0%
Mozart Symphonies	17.64%	70.59%	11.65%	0%
Schubert Piano Sonatas	0%	9.09%	90.91%	0%

Table 9.4.3: Händel/Mozart/Schubert composer recognition

9.5 Conclusion

Our experiments have shown that the approach to composer recognition we followed is successful. It is more capable of distinguishing between composers of different styles of composition than between composers of the same style of music. Due to time constraints in this project we did not investigate the classification accuracy of other generative models that may improve the accuracy of our model.

Chapter 10

Bacchus

We describe BACCHUS (Bach-inspired Algorithmic Computer Composer and Harmonizer , University of Stellenbosch), the application we wrote to implement the models described in the previous chapters.

10.1 Java class structure

Table 10.1.1 summarises all the classes in the Java part of our application.

A summary of the package structure of the implementation is given as a directed acyclic graph in figure 10.1.1. An arrow from A to B indicates that classes in package A are imported in classes in package B. *jm* is the external package JMusic.

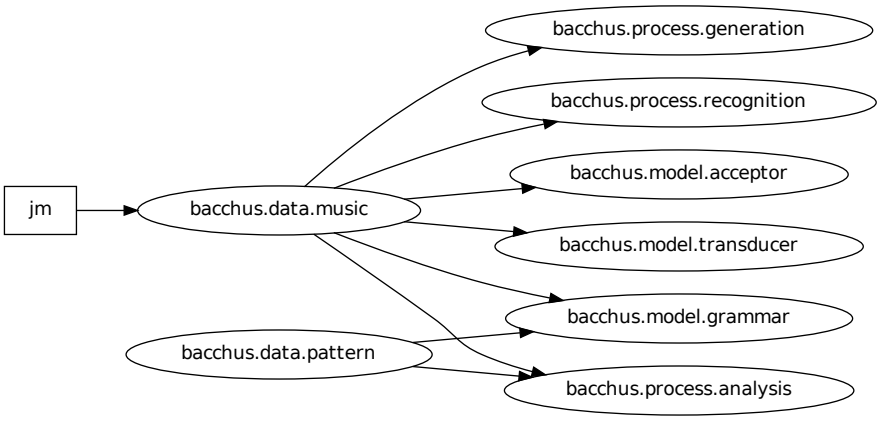

Figure 10.1.1: Java package structure

10.2 Bash Scripts

We briefly describe the functionality of each of the Bash scripts that we use in the execution of the generation and classification tasks of the application.

10.2.1 bacchus-markov

Constructs the Markov models for melody and harmony generation. The usage description of the script is:

```
Usage: bacchus-markov.sh [--analyse|--train|--generate|--all|--help] [OPTION]...
  Options
    --datapath, -dp         data path
    --resultspath, -rp      results path
    --compositionname, -n   composition name
    --meter, -m             time signiture
    --bars, -b              number of bars
    --barrestrict, -br      enforce bars
    --nobarrestrict, -nbr   do not eforce bars
    --wholebarrhythm, -wbr  bar-based rhythm generation
    --standardrhythm, -sbr  standard rhythm generation
    --mcorder, -o           Markov chain order
    --hmcorder, -ho         harmony Markov chain order
    --cadences, -c          enforce cadences
    --no-cadences, -nc      do not enforce cadences
    --hmm                   HMM melody model
    --cmm                   pitch-rhythm pairs model
    --interval, -i          use intervals for pitch generation
    --inner, -inn           generate inner voices
    --noinnner, -nin        no inner voices
    --major, -maj           major key
    --minor, -min           minor key
    --harmonize, -har       harmonize
    --noharmonize, -nhar    do not harmonize
    --key, -k               key signature integer
```

10.2.2 bacchus-clustering

Constructs the abstract melody clustering models for music generation. The usage description of the script is:

```
Usage: bacchus-clustering.sh [--analyse|--train|--generate|--all|--help] [OPTION]..
  Options
    --datapath, -dp         data path
    --resultspath, -rp      results path
    --compositionname, -n   composition name
    --meter, -m             time signiture
    --bars, -b              number of bars
    --mcorder, -o           Markov chain order
    --hmcorder, -ho         harmony Markov chain order
    --cadences, -c          enforce cadences
    --no-cadences, -nc      do not enforce cadences
    --hmm                   HMM melody model
    --inner, -inn           generate inner voices
    --noinnner, -nin        no inner voices
    --major, -maj           major key
    --minor, -min           minor key
    --harmonize, -har       harmonize
    --noharmonize, -nhar    do not harmonize
    --chordsfirst, -cf      chord-dependent generation
    --chordslast, -cl       chord-independent generation
    --key, -k               key signature integer
```

10.2.3 bacchus-tree

Constructs the CFG models for music generation. The usage description of the script is:

```
Usage: bacchus-tree.sh [--analyse|--train|--generate|--all|--help] [OPTION]...
 Options
    --datapath, -dp         data path
    --resultspath, -rp      results path
    --compositionname, -n   composition name
    --meter, -m             time signiture
    --bars, -b              number of bars
    --flatten, -f           flatten the tree represntation
    --interval, -i          model intervals
    --rhythm, -r            model rhythm
    --major, -maj           major key
    --minor, -min           minor key
```

```
    --harmonize, -har       harmonize
    --noharmonize, -nhar    do not harmonize
    --inner, -inn           generate inner voices
    --noinnner, -nin        no inner voices
    --key, -k               key signature integer
```

10.2.4 bacchus-recognize

Constructs the Markov model for composer recognition. The usage description of the script is:

```
Usage: bacchus-recognize.sh [--analyse|--train|--test|--all|--help] [OPTION]...
  Options
    --datapath, -dp     a data path (for every data set to be used)
    --resultspath, -rp  results path
    --meter, -m         timeSigniture
    --major, -maj       major key
    --minor, -min       minor key
```

package	package	class
data	music	Chord
data	music	Convert
data	music	MusicPiece
data	music	Voice
data	pattern	ContourCluster
data	pattern	Contour
data	pattern	Gram
data	pattern	NGram
data	pattern	NGramModel
model	acceptor	BarLengthAcceptor
model	acceptor	InnerVoiceAcceptor
model	acceptor	NumberOfBarsAcceptor
model	acceptor	NumberOfChordBarsAcceptor
model	acceptor	NumberOfSlopesAcceptor
model	acceptor	NumberOfWholeBarsAcceptor
model	acceptor	TerminalPitchAcceptor
model	grammar	IntervalCFG
model	grammar	RhythmCFG
model	transducer	ChordToBassTransducer
model	transducer	ChordToPitchTransducer
model	transducer	IntervalToPitchTransducer
model	transducer	IntervalToRhythmTransducer
model	transducer	PitchToRhythmTransducer
model	transducer	RelaxIntervalsTransducer
model	transducer	RepsToNotesTransducer
model	transducer	SampleFromSlopesTransducer
process	analysis	Analyse
process	analysis	ClusterContours
process	analysis	CountNGrams
process	generation	ConvertPairs
process	generation	ConvertRepNotes
process	generation	SampleTempo
process	generation	Synthesize
process	recognition	Recognize
process	recognition	SplitTrainingTesting

Table 10.1.1: Java classes in package bacchus

Chapter 11

Conclusion

11.1 Evaluation of generated music

The best judge of generated music is the *experienced listener*, described in [7]. A proposed way to evaluate the generate music of a model is to let a person with a considerable amount of knowledge of music listen to some generate pieces, and then judge the success of style imitation of the training data of the model.

A second approach is to perform a partial Turing test, as suggested by [10]. However, we propose that such a Turing test should be performed on a group of people with some experience of listening to music. A good example for such a group of people is a choir.

11.2 Future work

There are several avenues for future work on generative models of music and the implementation of such systems. We mention here a few possibilities.

11.2.1 Music piece processing

Methods for precise preprocessing of the MIDI music pieces should be investigated. In general, we cannot make assumptions about the meaning of parts and phrases in the piece. A proper analysis should be able to merge non-overlapping phrases and address the problem that melody and the base notes may be spread across different parts in the piece.

The key signature of a piece is not always be indicated in its MIDI file. The key can also change during the piece. Probabilistic models for key-finding, such as the one proposed in [32], should be used in music analysis tasks.

11.2.2 Markov modelling

When we want to generate chords independently of melody pitches, a Markov model may not be the most appropriate way to model chords that give structure to the music piece. Models for chords generation should be investigated. One possibility is to investigate a way to use the distance between chords given by the *circle of fifths*.

Not much research have been done on the effect of different methods of sampling from generative models on music generation. As mentioned in chapter 4, sophisticated sampling methods can increase the quality of generated music without hindering the creativity of the model.

11.2.3 Beyond context-free grammars

A big limitation of context-free models is illustrated by the fact that the language $\{ww|w \in \{a,b\}^*\}$ is not context-free. The notion of a repeated phrase in the structure of music pieces is very important, and if we are to model the overall structure of longer music pieces we will have to take that into account.

We propose to use a copying tree transducer to generate such structures. Firstly, we use a Markov model to generate, some phrases of music that are independent of one another, labeled $A, B, C, \ldots$. We encode these generated phrases into a trees: The phrase labels $A, B, C, \ldots$ are children of the root node. The terminal sequence of notes for each phrase becomes the (ordered) children of the phrase label. We give this tree as input to a tree-to-tree transducer that will copy some of the phrases. Each of the possible copying patterns are assigned a probability. Some of the proposed patterns are:

- $A \rightarrow A\ A$
- $A\ B \rightarrow A\ B\ A$
- $A\ B \rightarrow A\ B\ A\ B$
- $A\ B\ C \rightarrow A\ B\ A\ C\ A$

These rules can also be applied recursively by repeatedly applying the transducer to the output tree. In an implementation one would also want to be able to let the user specify the structure with some domain-specific language.

To make this model more powerful, we want to use it to extend our abstract melody model. In our tree representation, we want to put the cluster labels as the children of the phrase labels. The notes in the abstract note sequence then become the children of the cluster labels. We want to use the cluster labels to distinguish between directly copying a phrase and copying it with variation. If we copy a

phrase with variation, we only copy the cluster label sequence of that phrase. Then after we applied the tree transformation, we generate melodies for the parts of the tree that are variations and have cluster labels as leaves.

To use this model to encode local and global dependencies, we will need to encode a Markov model for melody as a tree acceptor that we can compose with the tree transducer that we described.

11.3 Conclusion

As we mentioned in chapter 1, this project builds on the work done by two previous students at Stellenbosch University. The research that we have done in this project makes the following additions to their work:

- We were able construct models from large corpora of music, by automated preprocessing of the music. This allowed use to model the works of well-known composers.
- We introduced a sophisticated model for four-part harmonization. Schulze only generates two-part music: A melody and accompaniment chords. Our model also can also imitate four-part harmonization more closely than the model proposed by Allan and Williams [3].
- We introduced a model that cluster abstract melody patterns and use the clusters to generate music that takes a larger context into consideration. By using transducer cascades, our model is formulated more precisely than that of a similar model proposed recently in [24].
- We proposed a model with tree transducers that has the potential to model the repetition of music phrases, with or without variation, while still modelling local dependencies in the music.
- We applied our Markov models, with some success, to the problem of composer recognition.
- Our extensive literature survey on generative models of music and related concepts shows several paths for future work.

The goal of constructing a generative model for music is a very ambitious one. As we saw in this project, such a model must be able to capture with rules that are not unreasonably complex the creative process of music generation. A model that assigns reasonable structure to a music piece will not necessarily be able to generate interesting music, and vice versa. This is one of the reasons that a context-free grammar often does not generate more acceptable music than a Markov model.

Due to time restrictions we did not implement the tree transducer model we describe above, which is more powerful than a CFG. The tree transducer model can be used to generate music that imitates common overall patterns of organization in a music piece. We can compose this model with a Markov model that model local dependencies. By simultaneous modelling local and global dependencies in a music piece, we can make a contribution to modelling that hidden arithmetic exercise that let us create and enjoy music.

Bibliography

[1] Nierhaus, G.: *Algorithmic Composition: Paradigms of Automated Music Generation.* SpringerWienNewYork, 2009.

[2] Conklin, D. and Witten, I.H.: Multiple viewpoint systems for music prediction. *Journal of New Music Research*, vol. 24, no. 1, 1995.

[3] Allan, M. and Williams, C.K.I.: Harmonizing chorales by probabilistic inference. *Advances in Neural Information Processing Systems*, vol. 17, 2005.

[4] Schulze, W.: *A Formal Language Theory Approach to Music Generation.* Master's thesis, Computer Science, University of Stellenbosch, 2009.

[5] Cilliers, P.: Music and complexity: The aesthetics of emergence. Colloquium, University of Stellenbosch, March 2011.

[6] Chomsky, N.: *Syntactic Structures.* Mouton, 1957.

[7] Lerdahl, F. and Jackendoff, R.: *A Generative Theory of Tonal Music.* MIT Press, 1983.

[8] Conklin, D.: Music generation from statistical models. In: *Proceedings of the AISB 2003 Symposium on Artificial Intelligence and Creativity in the Arts and Sciences.* 2003.

[9] Chomsky, N.: Three models for the description of language. *IEEE Transactions of Information Theory*, vol. 2, no. 3, 1956.

[10] Schulze, W. and Van der Merwe, B.: Music generation with markov models. *IEEE Multimedia*, vol. 18, no. 3, 2011.

[11] Knight, K. and Al-Onaizan, Y.: A primer on finite-state software for natural language processing, 1999. Available at: `http://www.isi.edu/licensed-sw/carmel/carmel-tutorial2.pdf`.

[12] Mohri, M.: Weighted automata algorithms. In: *Handbook of Weighted Automata.* Springer, 2009.

[13] Herbst, B. and Fornberg, B.: Modelling in applied mathematics, 2011. Manuscript: Class notes (Machine Learning).

[14] Rabiner, L.R. and Juang, B.H.: An introduction to hidden markov models. *IEEE ASSP Magazine*, 1986.

[15] Keller, R.M. and Morrison, D.R.: A grammatical approach to automatic improvisation. In: *Proceedings of the 4th Sound and Music Computing Conference*. 2007.

[16] Prescher, D.: A tutorial on the expectation-maximization algorithm including maximum-likelihood estimation and em training of probabilistic context-free grammars. Presented at the 15th European Summer School in Logic, Language and Information, 2003.

[17] May, J.: *Weighted Tree Automata and Transducers for Syntactic Natural Language Processing*. Ph.D. thesis, University of Southern California, 2010.

[18] May, J. and Knight, K.: A primer on tree automata software for natural language processing, 2008. Available at `www.isi.edu/licensed-sw/tiburon/tiburon-tutorial.pdf`.

[19] Graehl, J.: Carmel. 2008. Available at: `http://www.isi.edu/licensed-sw/carmel`.

[20] May, J.: Tiburon. Available at: `http://www.isi.edu/licensed-sw/tiburon/`, 2006.

[21] Brooks, F.P., Hopkins, A.L., Neumann, P.G. and Wright, W.V.: An experiment in musical composition. *IRE Transactions on Electronic Computers*, September 1957.

[22] Trivino-Rodriguez, J.L. and Morales-Bueno, R.: Using multiattribute prediction suffix graphs to predict and generate music. *Computer Music Journal*, vol. 25, no. 3, 2001.

[23] De la Higuera, C., Pait, F. and Tantini, F.: Learning stochastic finite automata for musical style recognition. In: *CIAA 2005*. 2005.

[24] Gillich, J., Tang, K. and Kleer, R.M.: Machine learning of jazz grammars. *Computer Music Journal*, vol. 34, no. 3, 2010.

[25] Simon, I., Morris, D. and Basu, S.: Mysong: Automatic accompaniment generation for vocal melodies. In: *CHI 2008 Proceedings*. 2008.

[26] Gilbert, E. and Conklin, D.: A probabilistic context-free grammar for melodic reduction. In: *Proceedings of the International Workshop on Artificial Intelligence and Music*. 2007.

[27] Bod, R.: A unified model of structural organization in language and music. *Journal of Artificial Intelligence Research*, vol. 17, 2002.

[28] Nevill-Manning, C.G. and Witten, I.H.: Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, vol. 7, 1997.

[29] Bernabeu, J.F., Calera-Rubio, J., Inesta, J.M. and Rizo, D.: Melodic identification using probabilistic tree autamata. *Journal of New Music Research*, vol. 40, no. 2, 2011.

[30] Jiang, N.: *An Analysis of Automatic Chord Recognition Procedures for Music Recordings.* Master's thesis, Saarland University, 2011.

[31] Katz, S.M.: Estimation of probabilities from sparse data for the language model of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 35, no. 3, 1987.

[32] Temperley, D.: *Music and probability.* MIT Press, 2007.