A Formal Language Theory Approach To Music Generation

by

Walter Schulze

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Computer Science at Stellenbosch University

Department of Mathematics, Applied Mathematics and Computer Science,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Prof. AB van der Merwe

December 2009

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature: . . . . . .

Date: . . . . . .

Abstract

We investigate the suitability of applying some of the probabilistic and automata theoretic ideas, that have been extremely successful in the areas of speech and natural language processing, to the area of musical style imitation. By using music written in a certain style as training data, parameters are calculated for (visible and hidden) Markov models (of mixed, higher or first order), in order to capture the musical style of the training data in terms of mathematical models. These models are then used to imitate two instrument music in the trained style.

Uittreksel

Hierdie tesis ondersoek die toepasbaarheid van probabilitiese en outomaatteoretiese konsepte, wat uiters suksesvol toegepas word in die gebied van spraak en natuurlike taal-verwerking, op die gebied van musiekstyl nabootsing. Deur gebruik te maak van musiek wat geskryf is in 'n gegewe styl as aanleer data, word parameters vir (sigbare en onsigbare) Markov modelle (van gemengde, hoër- of eerste- orde) bereken, ten einde die musiekstyl van die data waarvan geleer is, in terme van wiskundige modelle te beskryf. Hierdie modelle word gebruik om musiek vir twee instrumente te genereer, wat die musiek waaruit geleer is, naboots.

Acknowledgements

I want to express my deepest appreciation to my supervisor, Prof. AB van der Merwe, for his guidance, many helpful suggestions, encouragement, making light of seemingly stressful situations, and commitment to relentless proofreading. I would also like to thank my family and friends for their continuous and inspiring support. Also, to the additional proofreaders, I am very grateful for your eye-opening comments, suggestions and commitment to quality. Finally, I would like to thank all the survey participants, everyone who helped to spread the survey to their friends and colleagues, and especially the survey music composers for their valued input.

Dedications

I would like to dedicate this thesis to my loving and supportive parents.

Contents

Chapter 1

Introduction

Music is organised sound - Edgard Varèse

Music is a structured, yet creative medium, which can be considered to be a finite set of frequencies and timing intervals. Composing music consists mainly of the application of compositional rules, however artistic freedom allows a composer to disregard these rules from time to time.

In this thesis we investigate whether a computer can compose music as well as a human, taking into account that each artist is influenced by the styles he/she listens to. We do this by dividing the composition process into several steps, namely chord progression, melodic curve, cadence, etc. as proposed by Högberg (2005). For each of these steps, we calculate parameters for probabilistic automata from music data, and use these probabilistic automata to generate new compositions. We evaluate our system using an online survey which includes a partial Turing test.

The focus of Chapter 5 is on porting Willow, a ranked tree-based system for algorithmic music composition, to MusicXML, an unranked-tree-based system, using XSL (Extensible Stylesheet Language) and DOM (Document Object Model), instead of tree transducers. The MusicXML format is supported by over 85 musical applications (Recordare, 2007). It uses XML (Extensible Markup Language), which is a simple and very flexible text format standardised by the World Wide Web Consortium (2007*a*). XSL is used for defining XML document transformations (World Wide Web Consortium,

2007*b*). These transformations can be compared to tree transducers as both of them transform tree structures.

We propose pluggable artificial intelligence elements to be added to a rule-based system for music composition, by sampling from trained probabilistic automata (Conklin, 2003), as is discussed in Chapter 3. Just as an artist listens to other artists and styles which influence his/her compositions, we would like our system to learn from input compositions, and to generate output compositions in the same style. In Chapter 6 we cover the generation of music from probabilistic automata, and the analysis that is used to calculate their parameters. Similar systems, discussed in Chapter 4, have been proposed and implemented by Simon *et al.* (2008), Conklin and Anagnostopoulou (2001), Dubnov *et al.* (2003), Triviño-Rodriguez and Morales-Bueno (2001), and Pachet (2003).

Our music generation/imitation system (available for download at http://superwillow.sourceforge.net) is evaluated, as discussed in Chapter 7, with a partial Turing test. In an online survey, respondents were asked to identify the human composition, given two compositions, one composed by a human and one generated by our system. Even though our system failed a partial Turing test, promising results were obtained, since 36% of respondents in an online survey incorrectly attributed music composed by our computer system, to a human composer. Finally, Chapter 8 provides concluding remarks and Chapter 9 discusses possible future work.

Chapter 2

Music Theory

2.1 Introduction

This chapter discusses music theory starting with the notion of a musical note. A single musical note is represented by four properties (Ottman, 1983):

- Pitch, how high or low the sound is;
- Duration or note value, how long the sound is held;
- Intensity and loudness of the note;
- Timbre, or the instrument the note is being played on.

Notes are grouped to form chords, which are in turn placed in an ordered sequence to form a chord progression. The ordered sequence of durations of notes in a melody or chords in a chord progression, is the rhythm of the composition. This chapter includes the discussion of chord progression, rhythm, musical notation, intervals, scales, chords, chord inversions, cadences, note duration, time signatures and tempo.

2.2 Pitch

When a string vibrates 261.63 times per second, it has a frequency of 261.63 Hz (Hertz) and a pitch of middle C. This frequency naming convention was

only endorsed by the *International Organization for Standardization* in 1955 (Randel, 2003), and before 1955 there were several popular tuning standards. The standard piano has 88 keys and their frequencies are all related to middle C by the formula $\mathsf{freq(C)} \times 2^{\frac{s}{12}}$, where $\mathsf{freq(C)}$ is the frequency of middle C and s is an integer in the interval $[-39, 48]$. The absolute value of s can also be regarded as the number of semitones, number of half steps or interval size from the specific note to middle C. Each of these pitches has one of the following names: C, C♯, D, D♯, E, F, F♯, G, G♯, A, A♯, B. Every 12 semitones these pitch names repeat and the frequency doubles. This interval size of 12 is referred to as an octave. Octave registers are used to distinguish between pitch names at different frequencies. The octave registers for C are denoted by CC, C, c, c^1, c^2, c^3, c^4, c^5, where c^1 is used for middle C. The lowest note on the standard piano is AAA and has a frequency of 27.5 Hz. This is where humans start to find it hard to distinguish between different pitches (Levitin, 2006). Accidentals modify the pitch of a given note. Sharps (♯) are used to raise the pitch of a given note by one semitone or a half step, and a flat (♭), is used to lower the pitch by one semitone or a half step. This means that C♯ and D♭ are the same for all purposes, except in formal music theory.

2.3 Duration

Rhythm is the variation of duration of a sequence of notes or a series of note values. The duration of a note or note value indicates how long the note is sounded. This is indicated using fractions, for example: whole (𝅝), half (𝅗𝅥), quarter (♩). A dot after a note increases the duration of a note by 50 percent. Thus the note ♩. has a duration of three eighths, as shown in Figure 2.1. When no pitch is sounded for a duration of time, this is called a rest, and it is indicated by using a rest sign which corresponds to the duration, as shown in Figure 2.2.

A measure divides a musical composition into equal time units specified in terms of number of note values. The number of note values is specified by the time signature, which consists of an upper and a lower number. The number of beats per measure is represented by the upper number, while

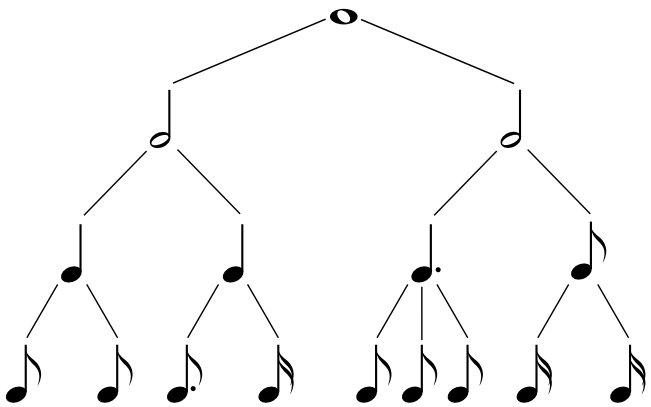


Figure 2.1: Note duration tree

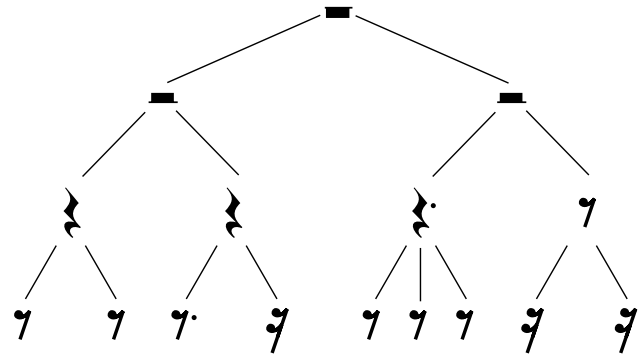


Figure 2.2: Rest duration tree

the duration of a beat is represented by the lower number. For example, if the upper number is two, there are two beats per measure, and if the lower number is 8, the duration of a beat is an eighth. This implies that there will be two eighths per measure. Popular time signatures include:

$\frac{4}{4}$ used in most forms of western classical and pop music;

$\frac{2}{2}$ used for marches and also in fast orchestral music;

$\frac{2}{4}$ used by polkas and sometimes marches;

$\frac{3}{4}$ used for waltzes, scherzi, minuets and some ballads.

The division provided by measures can be bypassed, by letting a note sound or ring from one measure to another.

Tempo of music is described in beats per minute (bpm) and affects the number of seconds a note is played. Suppose that the tempo is 120 bpm and the time signature is $\frac{2}{4}$. Then a beat is a quarter, there are 120 quarters played each minute, and a quarter is played for half a second. This also means that since there are two beats per measure, that the duration of a measure is equal to a second. Furthermore, if the tempo is changed to 60 bpm, then an eighth would be played for half a second. Musicians might play gradually slower towards the end of a composition. This is called a ritardando, and can be used to indicate that the composition is ending.

2.4 Timbre

The timbre of a note is composed of three properties:

- overtone profile;
- attack;
- flux.

When the pitch c^1 is played on a standard piano, one of the strings inside the piano vibrates at several frequencies. The smallest frequency is the defining or fundamental frequency of c^1, namely 261.63 Hz. The other frequencies, known as overtones, are unique for each instrument. Often the frequencies are multiples of the fundamental frequency; for example 523.55 Hz, 784.89 Hz, 1046.52 Hz, etc. Each overtone in the series has its own loudness value relative to the loudness of the other frequencies. These frequency relations are "programmed" in our brains so that *restoration of the missing fundamental* happens when we hear for example a frequency series 220 Hz, 330 Hz, 440 Hz, 550 Hz, which misses the fundamental frequency 110 Hz (Levitin, 2006). Thus although not present, our brain will add the missing frequency.

The attack is the initial frequencies when a note is played. On some music instruments these frequencies have more complex relations than the overtone series. Flux is the way the sound changes once a note starts playing.

Figure 2.3: Clef Signs

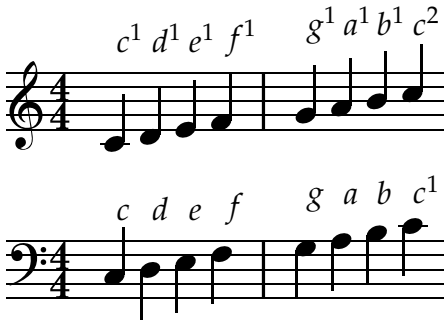


Figure 2.4: Staffs

2.5 Music Notation

Next we discuss music notation. Each pitch is represented on the music staff, which consists of five parallel horizontal lines. Each line represents a specific pitch as shown in Figure 2.4. Clef signs assign a specific pitch to a given line. Figure 2.3 shows the most popular clef signs, the G and F clef. The G clef assigns the G above middle C to the line encircled by its curl, while the F clef assigns the F below middle C to the line between its two dots. The time signature shown right of the clef in Figure 2.4 indicates that there are four quarters in a measure.

2.6 Melody and Harmony

When listening to music, one hears a multitude of sounds at the same time, and also one after the other. Melody can be described as notes heard in succession while harmony as notes heard at the same time. We can also describe melody and harmony as respectively the horizontal and vertical movement of the music.

Scales are collections or subsets of pitches used to compose music. In other

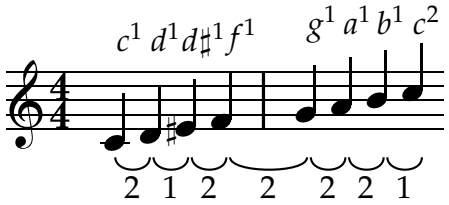


Figure 2.5: C melodic minor (ascending) scale with indicated semitone intervals

words, for a composition, a scale is chosen and then mostly those pitches are used for the composition. In a more complex composition the scale can for example be changed midway through a composition. Even with these exceptions, scales are still important when composing.

Example 2.6.1 (Basic scale examples) **-** The notes in the scales C major, D major, C natural minor, C melodic minor and C pentatonic.

- C major : {C,D,E,F,G,A,B}
- D major : {D,E,F♯,G,A,B,C♯}
- C natural minor : {C,D,D♯,F,G,G♯,A♯}
- C melodic minor (ascending) : {C,D,D♯,F,G,A,B}
- C melodic minor (descending) : {C,D,D♯,F,G,G♯,A♯}
- C pentatonic : {C,D,F,G,A}

In Example 2.6.1 the notes in sample scales are given. Scales are defined by their starting pitch, called the root, and type. C major has a root key of C and is of type major. The type of a scale defines the sequence of intervals. A whole step is equal to two half steps or two semitones and has an interval size of two. The major scale has the following sequence of steps: {whole, whole, half, whole, whole, whole, half}. The melodic minor (ascending) scale has the following sequence of steps: {whole, half, whole, whole, whole, whole, half} and is shown in Figure 2.5. Note that the melodic minor has two forms (ascending and descending) depending on whether the melody is approaching the root note from below or above respectively. There are many other types of scales, for example the harmonic minor and whole tone scales (Randel, 2003).

A melody is a sequence of notes played one after the other. It is not the ab-

solute pitches that identify a melody, but the intervals between them. The melodic contour or pitch profile takes into account only the positive or negative movement of the melody at every interval, in other words whether the next pitch is higher of lower than the previous one. The melodic contour is encoded by Parsons code as follows:

- "u" = up;
- "d" = down;
- "r" = repeat, when the next pitch is equal to the previous one;
- "*" = first pitch.

For example, Twinkle Twinkle Little Star represented by parsons code is: *rururddrdrdrd (Parsons, 1975). In McNab *et al.* (2000) the melodic contour is used to search a database of compositions. They found that using interval sizes, and not just the contour, provided better results for the purpose of music identification, since fewer intervals are required to identify a composition. The interval distance between C and F♯, called a tritone, or in the middle ages the devil's interval, was banned by the Roman Catholic church (Levitin, 2006). When listening to a melody the volume does not impair your ability to recognise the composition. Recognising the melody will also not be affected by changing the root key.

A chord is a group of notes played at the same time. The simplest chord is called a triad, which consists of three notes (see Figure 2.6 [1]). Whereas melody is defined by horizontal intervals, the chord type is defined by vertical intervals. The first (or lowest) note is called the root, followed by the third and fifth interval. The third and fifth intervals are respectively the third and fifth note in the major or minor scale relative to the first (root) note. The fifth interval is the same for the major and minor triad and is called a perfect fifth interval. The third interval for the major and minor triad is called a major third and minor third interval respectively. The minor third interval is three semitones, one less than the major third's four semitones. Another chord, the diminished triad is defined by a minor third and a diminished fifth interval. The diminished fifth interval is six semitones, one

[1] Note that we have dropped the case sensitivity of the minor roman numerals in the rest of the thesis for ease of explanation.

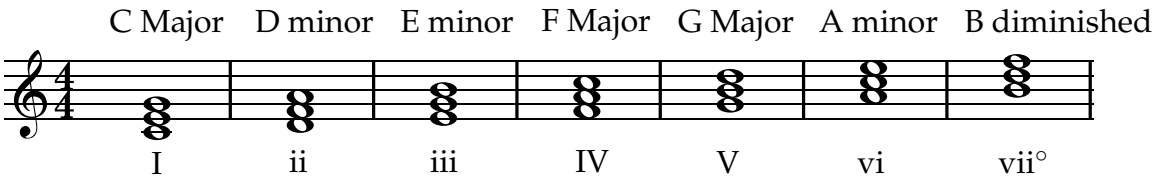


Figure 2.6: Some example triads

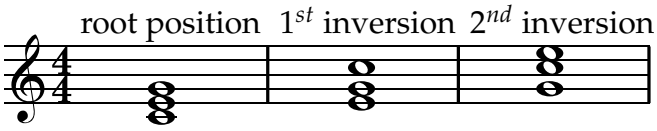


Figure 2.7: C major and its inversions

less than the perfect fifth's seven semitones. Lastly, the augmented triad is defined by the major third and augmented fifth (8 semitones) intervals. There are many other types of chords, for example the dominant seventh, major seventh, dominant eleventh, etc (Ottman, 1983). Contradictory to our previous statement, that the notes of chord are played at the same time, an arpeggio, also called a broken chord, is where the notes of chord are played consecutively. This simplifies the definition of a chord to just a group of notes.

In Figure 2.7, various versions of the C major triad is shown. These triads are referred to as inversions of the C major triad, since the notes representing the C major triad (C,E,G) can be played in any order and still form the C major triad. This implies that a chord type has several sequences of vertical intervals to choose from. A triad without its third is called a power chord. Playing two power chords directly one after the other, is called a parallel fifth, and does not conform to classical music theory rules (Randel, 2003). A parallel motion is when two notes move by the same vertical interval. The parallel fifth is when the vertical interval between the two notes is a fifth. Any combination of vertical intervals can form a chord, but whether a given chord fits in a composition is up to the style of music and the chords between which the given chord is played.

2.7 Chord Progressions and Cadence

A chord progression is an ordered sequence of chords. Chord progressions are usually repeated a few times in a composition. For example, a verse of music in the style of punk might have the chord progression: I $\rightarrow$ V $\rightarrow$ VI $\rightarrow$ IV. This will usually be repeated four times. This repetition is representative of a motif, a repeating and developing melodic or rhythmic idea, or the basic component of the composition. Roman numerals are used to indicate the chord number in the scale of the composition. In Figure 2.6 the triads of the C major scale with their respective numerals are shown.

A cadence or a falling, as it is called in western music, is a certain sequence of intervals or chords that end a phrase (verse, chorus, etc.). When a phrase ends with the chord progression V $\rightarrow$ I or VII $\rightarrow$ I, it is referred to as an authentic or perfect cadence. The chord progression VII $\rightarrow$ I is also representative of resolving dissonance, or harmonic tension. Other types of cadences include (Adams, 2000):

- half cadence: I $\rightarrow$ V
- plagal cadence (Amen cadence): IV $\rightarrow$ I
- deceptive cadence: V $\rightarrow$ VI

Cadences give a definite ending, indicating to the listener that the piece of music is concluding.

2.8 Conclusion

This chapter gave an introduction to music theory, covering the terminology required for this thesis. For our music generation system we consider compositions to be chords for the rhythm guitar or piano and a melody for the lead.

Chapter 3

Automata and Markov Models

3.1 Introduction

Probabilistic finite automata and some of its subclasses are discussed in this chapter. We assume a pre-existing knowledge of basic probability theory and in particular Bayes' theorem (see Appendix C).

3.2 Markov Chains

A Markov chain models a sequence of events by using states and transition probabilities between states. This model adheres to the first order Markov assumption which states:

$$P(q_t|q_{t-1}, q_{t-2}, \ldots, q_1) = P(q_t|q_{t-1}),$$

where $q_1, \ldots, q_t$ is a set of states. Thus in a Markov chain the probability of being in state q_t, at time t, depends only on the previous state, at time t-1. A homogeneous Markov chain, which is defined in Definition 3.2.1, adheres to the stationarity assumption, which states that the transition probabilities of the Markov chain are time-independent. This implies that the probability of moving to a next state, at any time, only depends on the current state.

Definition 3.2.1 (Markov chains) **-** A Markov chain (MC) is a 3-tuple $\langle$ Q,a,π $\rangle$ where:

- Q is the state space,
- a : Q $\times$ Q $\rightarrow$ $[0,1]$ is a mapping defining the probability of each transition,
- and π : Q $\rightarrow$ $[0,1]$ is a mapping defining the initial probability of each state.

The following constraints must be satisfied:

- for $q_i \in Q$, $\sum_{q_j \in Q} a(q_i, q_j) = 1$,
- $\sum_{q \in Q} \pi(q) = 1$.

One approach to estimating the transition probabilities of Markov chains is by calculating the maximum likelihood estimate, using frequency or empirical counts as in the next example.

Example 3.2.1 (Calculating the parameters of a Markov chain) **-** Assume that the training sequences are two melodies (c, d, e, c, d, c, d, e, c, d) and (d, e, d, e, c, d, c, d, e, c). This implies that the state space Q is $\{c, d, e\}$. The transition probabilities are calculated as follows:

$$a(q_i, q_j) = \frac{\#(q_i \rightarrow q_j)}{\sum_{q_k \in Q} \#(q_i \rightarrow q_k)} \quad ,$$

where $\#(q_i \rightarrow q_j)$ is the number of times state q_i is followed by state q_j. Thus:

$$a = \begin{pmatrix} a(c,c) & a(c,d) & a(c,e) \\ a(d,c) & a(d,d) & a(d,e) \\ a(e,c) & a(e,d) & a(e,e) \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ \frac{2}{7} & 0 & \frac{5}{7} \\ \frac{4}{5} & \frac{1}{5} & 0 \end{pmatrix}$$

and

$$\pi = \begin{pmatrix} \pi(c), & \pi(d), & \pi(e) \end{pmatrix} = \begin{pmatrix} \frac{1}{2}, & \frac{1}{2}, & 0 \end{pmatrix}$$

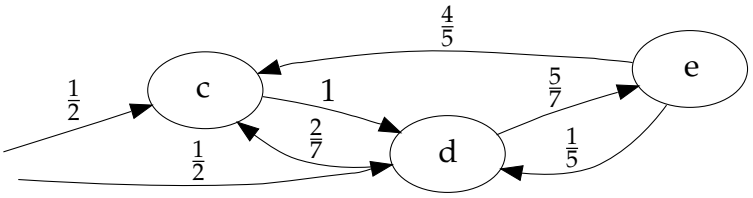


Figure 3.1: A first order Markov chain

We can visualise the Markov chain as a graph where vertices represent states and edges represent transitions as in Figure 3.1.

3.3 Higher Order Markov Chains

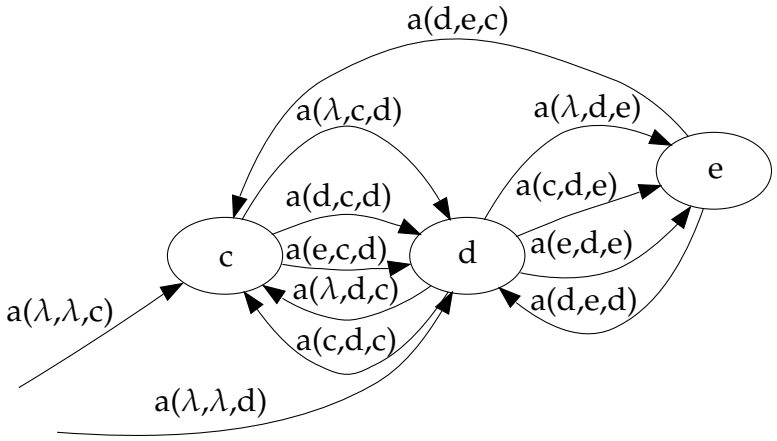
In contrast to Markov chains as defined in Definition 3.2.1, higher order Markov chains have a memory length larger than one. More precisely, an L^{th} order Markov chain operates under the assumption:

$$P(q_t|q_{t-1}, q_{t-2}, \ldots, q_1) = P(q_t|q_{t-1}, \ldots, q_{t-min(t-1,L)}).$$

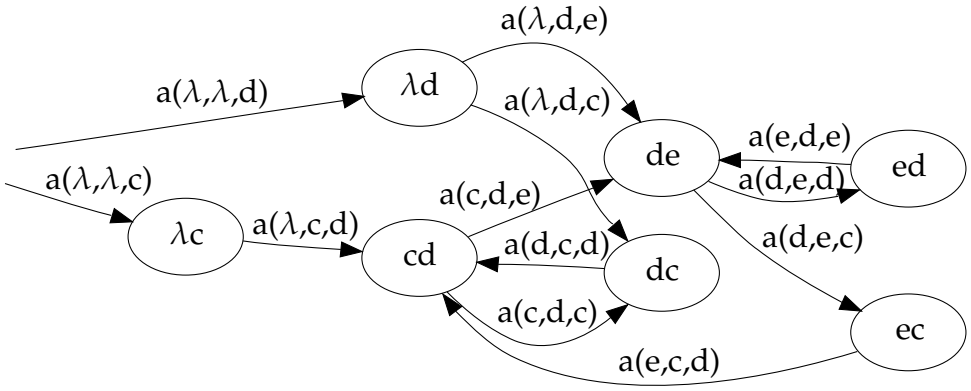
When $t \leq L$, q_t is a startup state, and the excessive states, q_t with (t < 0), are represented by the empty string λ. In the case of L^{th} order Markov chains we use $a(q_{t-L}, \ldots, q_{t-1}, q_t)$, instead of $a(q_{t-1}, q_t)$, to indicate transition probabilities. This results in multiple transition probabilities between states q_{t-1} and q_t, as in Figure 3.2.

The complication introduced by higher order Markov chains can be removed by increasing the number of states and adding a history to the states, as in Figure 3.2. In order to transform an L^{th} order Markov chain to a first order Markov chain, we replace the state space Q by $\cup_{i=1}^{L} Q^i$, where Q^i is all state sequences of length i. More precisely, if q'_t and q'_{t-1} are states in a first order Markov chain, corresponding to a given higher order Markov chain, at times t and t-1 respectively, with $q'_t = (q_{t-L+1}, \ldots, q_t)$ and $q'_{t-1} = (q_{t-L}, \ldots, q_{t-1})$, then we have the following:

$$\begin{aligned} a(q'_{t-1}, q'_t) &:= P(q'_t|q'_{t-1}) \\ &= P(q_{t-L+1}, \ldots, q_t|q_{t-L}, \ldots, q_{t-1}) \\ &= P(q_t|q_{t-L}, \ldots, q_{t-1}) \\ &= a(q_{t-L}, \ldots, q_{t-1}, q_t). \end{aligned}$$



A second order Markov chain



A first order Markov chain

Figure 3.2: A second order Markov chain and its equivalent first order Markov chain

We are thus able to convert a higher order Markov chain to an equivalent first order Markov chain. Next we discuss prediction suffix automata which are equivalent to mixed (variable) order Markov chains.

3.4 Prediction Suffix Automata

Conversion from higher to first order Markov chains, as discussed in the previous section, involves moving the memory from the transitions to the states. Mixed order Markov chains allow transitions of various memory lengths, whereas all transitions of higher order Markov chains have the

same memory length. Prediction Suffix Automata (PSA) represent memory using state labels in a similar fashion to first order Markov chains obtained from translating higher order Markov chains to their equivalent first order Markov chains. Both mixed and higher order Markov chains represent their memory using transitions, whereas prediction suffix automata (PSA) uses state labels. This flexibility of memory lengths in a PSA allows us to avoid the exponential growth associated with higher order Markov chains.

Next we define prediction suffix automata, which are equivalent to mixed order Markov chains.

Definition 3.4.1 (Prediction suffix automata (Ron *et al.*, 1996; Schwardt, 2007)) **-** A prediction suffix automata (PSA) is a 4-tuple $\langle\ \Sigma, \mathrm{Q}, \tau, \mathrm{p}\ \rangle$ where:

- Σ is a finite input alphabet,
- $\mathrm{Q} \subset \Sigma^*$ is a finite set of finite-length strings (so that $\lambda \in \mathrm{Q}$) which is the state space,
- $\tau : \mathrm{Q} \times \Sigma \rightarrow \mathrm{Q}$, is the state transition function,
- $\mathrm{p} : \mathrm{Q} \times \Sigma \rightarrow [0, 1]$, is the next symbol probability distribution.

The following constraints must be satisfied:

- $\sum_{\sigma \in \Sigma} p(q, \sigma) = 1$ for all $q \in Q$;
- the start state q_0 is the empty string λ;
- for all $q \in Q$ and $\sigma \in \Sigma$, $\tau(q, \sigma)$ is equal to the longest suffix of $q\sigma$ that is in Q.

If in a PSA $\langle\ \Sigma, \mathrm{Q}, \tau, \mathrm{p}\ \rangle$ the state space of the PSA contains all states of length L, i.e. if $\Sigma^L \subseteq Q$, we refer to the PSA as an L-PSA. Note that an L-PSA is equivalent to an L^{th} order Markov chain. Both models limit the maximum memory length to L.

Training an L-PSA from training sets is achieved by first training an L-PST (see next section for the definition of a PST) and then converting the L-PST to an equivalent L-PSA. A PSA's transition function specifies the next state, given the previous state and alphabet symbol, whereas a PST needs to calculate the next state. This is done by appending the alphabet symbol to the

right of the current state string and then searching for the longest suffix that is also a state in the PST. This calculation in a PST can take L times longer than when a PSA's transition function is used. The PSA also has extra added prefix states, which connects previously unreachable states to the structure.

In the next section we discuss PSTs, including training and converting a PST to a PSA.

3.5 Prediction Suffix Trees

Definition 3.5.1 (prediction suffix trees (Ron *et al.*, 1996; Schwardt, 2007)) **-** A prediction suffix tree (PST) is a 3-tuple $\langle\ \Sigma, \mathrm{Q}, \mathrm{p}\ \rangle$ where:

- Σ is a finite alphabet;
- Q is the finite state space and $Q \subset \Sigma^*$ is a finite set of finite-length strings with $\lambda \in Q$;
- $\mathrm{p} : \mathrm{Q} \times \Sigma \rightarrow [0, 1]$, is the next symbol probability distribution.

The following constraints must be satisfied:

- $\sum_{\sigma \in \Sigma} p(q, \sigma) = 1$ for all $q \in Q$;
- for all $q \in (Q - \lambda)$ there exists s $\in \Sigma$ and $q' \in Q$ such that q = sq', and q' is the parent of q;
- the root of the tree is labelled by λ.

Ziv and Lempel (1978) developed a variable order algorithm for lossless data compression. LearnPSA is an equivalent lossy compression algorithm. The LearnPSA algorithm (Ron *et al.*, 1996) is used to calculate the state space and probability distributions of a PST. This algorithm finds all the strings with a statistical significance, given certain input parameters for the training sequences. The algorithm's design was motivated by the Probably Approximately Correct (PAC) learning model (Valiant, 1984). Starting at the root node, which is represented by the empty string, the algorithm follows a top-down approach to build a tree to which nodes are added, which appear a significant number of times in the training data and which have a

unique next symbol distribution when compared to shorter suffixes of the same node. The parameters for LearnPSA are listed below:

- Σ is the input alphabet;
- L is the maximum string length allowed to label a state;
- n is the maximum number of states allowed;
- $\delta \in (0, 1)$ is the approximation parameter;
- and $\omega_1, \ldots, \omega_T$ are the training sequences.

The threshold values are calculated as follows:

- $\gamma = \frac{\delta}{48L|\Sigma|}$, is the smoothing factor;
- $P_{min} = \frac{\delta}{2nLlog(1/\gamma)} - \frac{|\Sigma|}{8n}$, is the minimum empirical string probability;
- $\alpha = (1 + \gamma|\Sigma|) \times \gamma$, is the minimum empirical next symbol probability;
- $\beta = 1 + 3\gamma|\Sigma|$, is the minimum empirical next symbol probability ratio.

Next we define the following functions:

- $\eta_i(q, s)$ = the number of times the string $q.s$ appears in the training sequence, ω_i ;
- $\eta_i(q, *) = \sum_{s \in \Sigma} \eta_i(q, s)$;
- $p(q) = \frac{\sum_{i=1}^{T} \eta_i(q,*)}{(\sum_{i=1}^{T} |\omega_i|) - T}$;
- $p(q, s) = \frac{\sum_{i=1}^{T} \eta_i(q,s)}{\sum_{i=1}^{T} \eta_i(q,*)}$,

where $p(q, s)$ represents the frequency probabilities. These functions are calculated for every $q \in \Sigma^{\leq L} = \cup_{i=0}^{L} \Sigma^i$ and $s \in \Sigma$. We need to subtract T from the sum of the length of all training sequences in the denominator of $\frac{\sum_{i=1}^{T} \eta_i(q,*)}{(\sum_{i=1}^{T} |\omega_i|) - T}$, since the last symbol in each training sequence does not have a next symbol. Also, we define parent(q) to be the longest proper suffix of q.

The pseudocode for LearnPSA is given on the next page:

LearnPSA(Σ,L,n,δ,ω)

01 $Q = \{\lambda\}$

Initialise the state space to include the root state λ.

02 $F = \{s \mid s \in \Sigma \; and \; P(s) \geq P_{min}\}$

Initialise the frontier set of states to be considered to include every alphabet symbol with a high enough occurrence rate in the training sequence.

03 *while* $F \neq \varnothing$
04 $\quad q \; = \; F.pop()$

While there are states to be considered, remove the currently considered state from the frontier.

05 $\quad for \; all \; s \in \Sigma$
06 $\quad\quad if \; p(q,s) \geq \alpha$ and $\frac{p(q,s)}{p(parent(q),s)} \geq \beta$
07 $\quad\quad\quad Q = Q \cup q$

If the empirical next symbol probability is significant enough, according to α, and if the state q provides significantly more statistical information about the next symbol than its parent does, q is added to the state space.

08 $\quad\quad\quad F = F \cup (\text{Suffixes}(q) - Q)$
09 $\quad\quad end \; if$

All suffixes of q which are not in the state space or the frontier are now also considered as potential states.

10 $\quad end \; for$
11 $\quad if \; |q| < L \; : \; F = F \cup \{s \cdot q \mid s \in \Sigma \; and \; P(s \cdot q) \geq P_{min}\}$

Consider all children of q with a high enough frequency count.

12 *end while*
13 $p(q,s) = p(q,s) \times (1 - |\Sigma| \times \gamma) + \gamma$

Set the next symbol probability for every state and symbol respectively, while taking a smoothing factor into account.

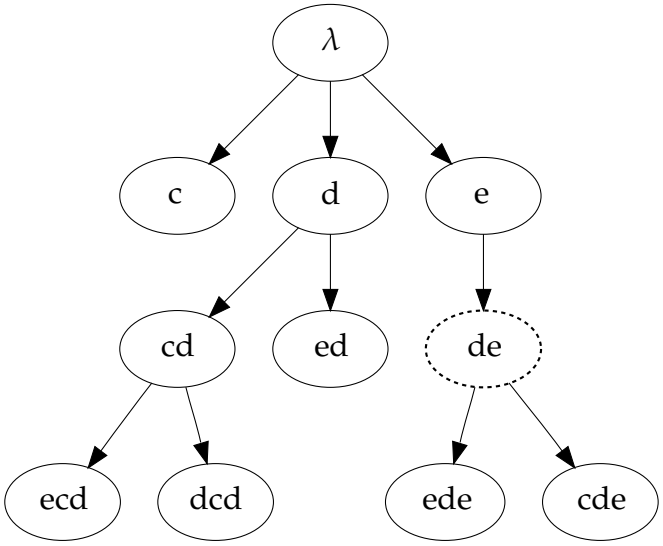


Figure 3.3: A prediction suffix tree

Completing the suffix tree is done by adding all missing parents of the state space. These parents inherit their next symbol probabilities from their respective parents.

Example 3.5.1 - In Figure 3.3 the resulting PST is shown when the input parameters are:

- $\Sigma = \{c, d, e\}$
- $L = 3$
- $n = 10$
- $\delta = 0.1$
- ω_1 = (c, d, e, c, d, c, d, e, c, d)
- ω_2 = (d, e, d, e, c, d, c, d, e, c)

The suffix ec was not added to the tree, since it has the same probability distribution as its parent c. This is the case since c is always followed by d independent of whether its predecessor is e or d. The node de is only added after the completion of the LearnPSA algorithm, in order to complete the constructed tree. Note that internal nodes could have the same next symbol distribution as their parent, in the case where they are used to complete the tree, but this is not true for leaf nodes.

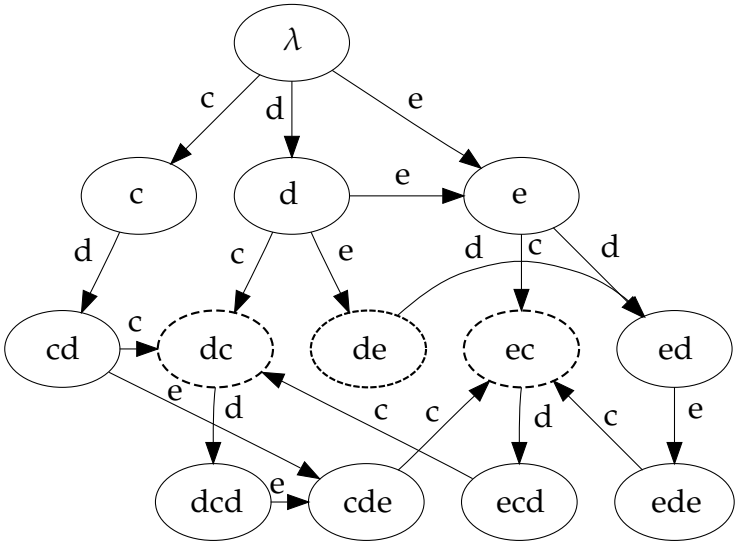


Figure 3.4: A prediction suffix automata corresponding to the PST in Figure 3.3

Converting a PST to a PSA is achieved by adding all missing prefixes to the state space and by constructing the transition function τ. Each state needs all its prefixes to allow the state to be reachable in the PSA. These prefixes inherit their transition probabilities from their longest proper suffix in the state space. Constructing τ involves finding the destination state for each source state and transition symbol. The destination state of a transition is the longest suffix of the string obtained by concatenating the source state and the transition symbol, which is also in the state space. The resulting PSA converted from the PST in Figure 3.3 can be seen in Figure 3.4. Nodes dc and ec are the prefixes added to complete the automaton.

PSAs are in general more compact than higher order Markov chains, since higher order Markov chains require all states of maximum length, where in contrast PSAs only require statistically significant states.

3.6 Hidden Markov Models

Hidden Markov models (HMMs) (Rabiner, 1990) are used to model the relationship between a hidden and an observed sequence. A discrete hidden Markov model is a Markov chain with a discrete probability distribution at each state. These discrete probability distributions define probabilities of

emitting a specific alphabet symbol in a given hidden state. Thus the states of the Markov chain are the hidden states.

Definition 3.6.1 (Discrete hidden Markov models (Dupont *et al.*, 2005)) **-** A discrete HMM is a 5-tuple $\langle \Sigma, Q, a, b, \pi \rangle$ where:

- Σ is a finite alphabet of visible symbols;
- Q is a finite set of hidden states;
- a : $Q \times Q \rightarrow [0,1]$ is a mapping defining the probability of transitions between hidden states;
- b : $Q \times \Sigma \rightarrow [0,1]$ is a mapping defining the emission probability of each visible symbol at a given hidden state, also called a confusion matrix;
- and π : $Q \rightarrow [0,1]$ is a mapping that defines the initial probability of the hidden states.

The following constraints must be satisfied:

- for all $q_i \in Q$, $\sum_{q_j \in Q} a(q_i, q_j) = 1$;
- for all $q \in Q$, $\sum_{\varsigma \in \Sigma} b(q, \varsigma) = 1$;
- $\sum_{q \in Q} \pi(q) = 1$.

We denote the observation and hidden state sequence by $\chi = \chi_1, \ldots, \chi_n$ and $s = s_1, \ldots, s_n$, respectively, where $\chi_i \in \Sigma$ and $s_i \in Q$. We use the following notation:

$$\begin{aligned} \pi(q_i) &= P(s_1 = q_i); \\ a(q_i, q_j) &= P(s_t = q_j | s_{t-1} = q_i); \\ b(q_j, \chi_t) &= P(\chi_t | s_t = q_j). \end{aligned}$$

In the notation above, $P(x)$ is the probability of an event, while $P(x \mid y)$ is the probability of an event x given that event y has occurred (see Appendix C).

In Example 3.6.1 the observation sequence represents the chord progression and the hidden sequence represents the melody of a given composition.

This models the relation between two instruments, one playing chords and the other a chordless melody. In the case where the hidden state sequence and observation sequence are available as training data, empirical counts can be used to calculate the parameters of the model, as for example when using music. When this is not the case, the Baum-Welch forward-backward algorithm is used.

Example 3.6.1 (Training an HMM using empirical counts) **-** The training input is as follows:

$$\begin{array}{c|cccccccccc} \chi_1 & \text{I} & \text{I} & \text{I} & \text{II} & \text{II} & \text{II} & \text{I} & \text{I} & \text{II} & \text{II} \\ s_1 & \text{c} & \text{d} & \text{e} & \text{c} & \text{d} & \text{c} & \text{d} & \text{e} & \text{c} & \text{d} \\ \chi_2 & \text{I} & \text{I} & \text{I} & \text{I} & \text{II} & \text{II} & \text{II} & \text{II} & \text{I} & \text{I} \\ s_2 & \text{d} & \text{e} & \text{d} & \text{e} & \text{c} & \text{d} & \text{c} & \text{d} & \text{e} & \text{c} \end{array},$$

where χ is the observation sequence and s is the hidden state sequence. This implies that $\Sigma = \{\text{I, II}\}$ and $\text{Q} = \{\text{c, d, e}\}$. The hidden state sequence can be used as in Example 3.2.1 to determine the transition probabilities of the underlying Markov chain,

$$a = \begin{pmatrix} 0 & 1 & 0 \\ \frac{2}{7} & 0 & \frac{5}{7} \\ \frac{4}{5} & \frac{1}{5} & 0 \end{pmatrix} \text{ and } \pi = \begin{pmatrix} \frac{1}{2}, & \frac{1}{2}, & 0 \end{pmatrix}.$$

The confusion matrix, b, is also trained using counts as follows:

$$\text{for all } q \in Q \text{ and } \sigma \in \Sigma, \text{ we have that } b(q, \sigma) = \frac{\#(s_t = q \; and \; \chi_t = \sigma)}{\#(s_t = q)}.$$

Thus:

$$b = \begin{pmatrix} b(c, I) & b(c, II) \\ b(d, I) & b(d, II) \\ b(e, I) & b(e, II) \end{pmatrix} = \begin{pmatrix} \frac{2}{7} & \frac{5}{7} \\ \frac{1}{2} & \frac{1}{2} \\ 1 & 0 \end{pmatrix}.$$

A graphical representation of the HMM is shown in Figure 3.5, with the discrete probabilistic distribution at each state displayed as a histogram.

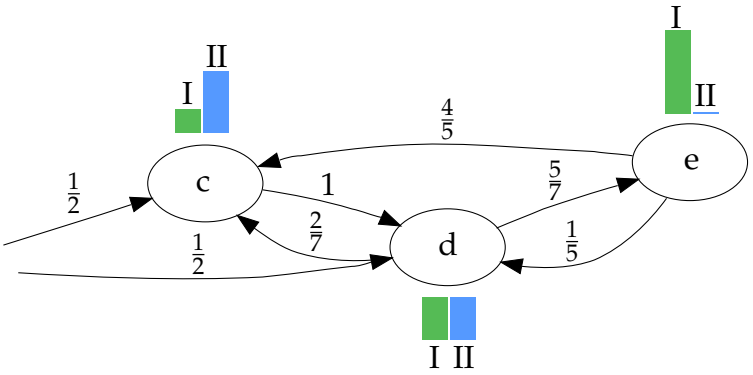


Figure 3.5: A discrete hidden Markov model with histograms defining the emission probabilities

3.7 Hidden Markov Model Algorithms

Next we discuss algorithms applicable to hidden Markov models. The forward algorithm is used to calculate the probability of a given observation sequence. This algorithm is used when multiple models are available, and the model which matches the observation sequence the best needs to be selected. The solution is a dynamic programming algorithm and involves filling in the forward matrix, $\alpha(t, q_i) = P(s_t = q_i, \chi_1^t)$, where χ_1^t represents the observation sequence $\chi_1, \ldots, \chi_t$. Each cell of the forward matrix is equal to the sum of the probabilities of all paths which lead to state q_i and which emits the specified observation sequence from time 1 to time t (see Figure 3.6). The forward matrix is calculated as follows:

$$\alpha(t, q_i) = \begin{cases} \pi(q_i) \times b(q_i, \chi_t) & \text{if } t = 1; \\ \left(\sum_{q_j \in Q} \alpha(t-1, q_j) \times a(q_j, q_i) \right) \times b(q_i, \chi_t) & \text{otherwise.} \end{cases}$$

Thus for a given model, the probability of a specific observation sequence χ_1^T is equal to $\sum_{q \in Q} \alpha(T, q)$. The asymptotic running time of the forward algorithm is $O(|Q|^2 T)$.

Viterbi decoding is used to find the state sequence with the maximum likelihood, given an observation sequence. The dynamic programming solution used for Viterbi decoding is similar to the forward algorithm, but instead of calculating the sum of all paths leading to a hidden state, the most probable path probability leading to a hidden state is determined. The probability

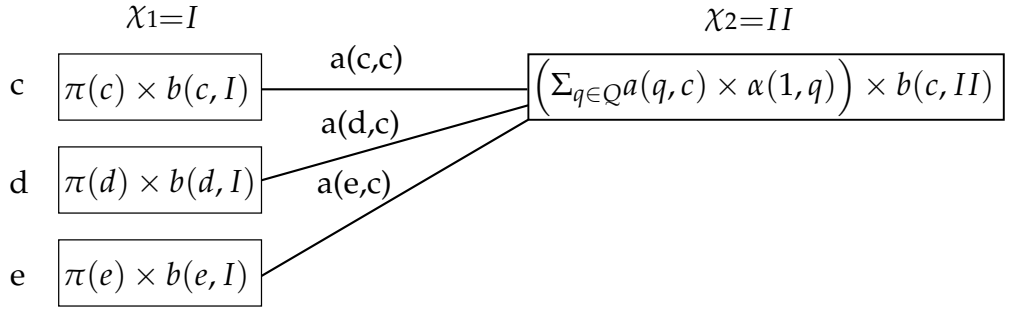


Figure 3.6: The forward algorithm

An application of the forward algorithm with observation sequence χ = (I,II) and state space Q = {a,b,c}.

represented by each cell of the matrix δ is defined as follows:

$$\delta(t, q_i) = P(s_t = q_i, s_{t-1} = s^*_{t-1}, \ldots, s_1 = s^*_1, \chi^t_1),$$

where s^* is the state sequence with the maximum probability, given the observation sequence. These probabilities are calculated recursively as follows:

$$\delta(t, q_i) = \begin{cases} \pi(q_i) \times b(q_i, \chi_t) & \text{if } t = 1; \\ \max_{q_j \in Q}(\delta(t-1, q_j) \times a(q_j, q_i)) \times b(q_i, \chi_t) & \text{otherwise.} \end{cases}$$

While calculating δ, we construct ϕ, which is used to store the state at time (t - 1) in the most probable path to state q_i at time t. We calculate ϕ, for t > 1, as follows:

$$\phi(t) = \text{argmax}_{q_j \in Q}(\delta(t-1, q_j) \times a(q_j, q_i)).$$

After calculating δ and ϕ we use backtracking to find the most probable path ι for a given observation sequence, as shown below:

$$\iota(t) = \begin{cases} \text{argmax}_{q_j \in Q} \delta(t, q_j) & \text{if } t = T; \\ \phi(t+1, \iota(t+1)) & \text{otherwise.} \end{cases}$$

The asymptotic running time of Viterbi decoding is O($|Q|^2$T), which is the same as the running time for the forward algorithm.

Note that Viterbi decoding only finds the most probable hidden state sequence and not a probability distribution over possible hidden state sequences, from which possible sequences of hidden states can be selected by taking the probability distribution into account. The A* search algorithm

(Hart *et al.*, 1968) is an alternative dynamic algorithm which finds the k most probable sequences.

The backward algorithm calculates the probability of emitting a partial observation sequence $\chi_{t+1}^T = \chi_{t+1}, \ldots, \chi_T$, given that the HMM is in the hidden state q_i at time t, and defines β as follows:

$$\beta(t, q_i) = P(\chi_{t+1}^T | s_t = q_i).$$

The backward matrix can be calculated by using a dynamic programming algorithm as shown below:

$$\beta(t, q_i) = \begin{cases} 1 & \text{if } t = T; \\ \sum\limits_{q_j \in Q} (a(q_i, q_j) \times b(q_j, \chi_{t+1}) \times \beta(t+1, q_j)) & \text{otherwise.} \end{cases}$$

Using the transition, confusion, forward and backward matrices we can calculate γ, which is the probability of transitioning from state q_i to q_j at time t, given the full observations sequence $\chi_1^T = \chi_1, \ldots, \chi_T$ (Huang *et al.*, 2001), as shown below for (t > 1):

$$\begin{aligned} \gamma(t, q_i, q_j) &= P(s_{t-1} = q_i, s_t = q_j | \chi_1^T) \\ &= \frac{\alpha(t-1, q_i) \times a(q_i, q_j) \times b(q_j, \chi_t) \times \beta(t, q_j)}{\sum\limits_{q_k \in Q} \alpha(T, q_k)}. \end{aligned}$$

The initial probabilities, at time $t = 1$, is given by:

$$\gamma(t, q_j) = \frac{\pi(q_j) \times b(q_j, \chi_1) \times \beta(t, q_j)}{\sum\limits_{q_k \in Q} \alpha(T, q_k)}.$$

By using γ, we can calculate the probability ω of transitioning to the next hidden state q_j at time t, given the current hidden state q_i at time (t - 1) and the full observation sequence $\chi_1^T = \chi_1, \ldots, \chi_T$, as follows:

$$\begin{aligned} \omega(t, q_i, q_j) &= P(s_t = q_j | s_{t-1} = q_i, \chi_1^T) \\ &= \frac{\gamma(t, q_i, q_j)}{\sum\limits_{q_j \in Q} \gamma(t, q_i, q_j)}. \end{aligned}$$

The Markov chain determined by $\omega(t, *, *)$ can be used to generate multiple possible hidden state sequences, given the observation sequence and the model. This is used in our music generation system to compose a melody given the chords.

3.8 Mixed Order Hidden Markov Models

First order HMMs consist of a first order Markov chain with a probability distribution associated with each state of the Markov chain. More generally, a mixed order hidden Markov model is a mixed order Markov chain with a probability distribution associated with each state of the Markov chain. Instead of using a mixed order Markov chain, as the base of the mixed order hidden Markov model, we use a Prediction Suffix Automata (PSA), since PSAs are equivalent to mixed order Markov chains. Each state of the PSA used in a given mixed order hidden Markov model, is associated with a probability distribution defining its emission probabilities. This probability distribution is the same for all states in the PSA which have the same last symbol in their respective state labels. Note that the mixed order nature of a PSA is kept in the state labels, instead of the transitions as is the case for mixed order Markov chains. This property makes the PSA's transitions first order and allows the algorithms discussed in Section 3.7, to be applied to a our mixed order hidden Markov model. Figure 3.7 shows the PSA in Figure 3.4 with associated probability distributions. The distributions f_c, f_d and f_e are represented by the rows of the confusion matrix.

3.9 Probabilistic Finite Automata

Finally, we discuss Probabilistic Finite Automata (PFA), since all other automata discussed in the previous part of this chapter are PFAs. The class of deterministic probabilistic finite automata (DPFA) is a subclass of the class of PFA, with the property that the transition from a state, given an input symbol, is unique (see Definition 3.9.1).

Note that all prediction suffix automata (PSA) are deterministic. The set of

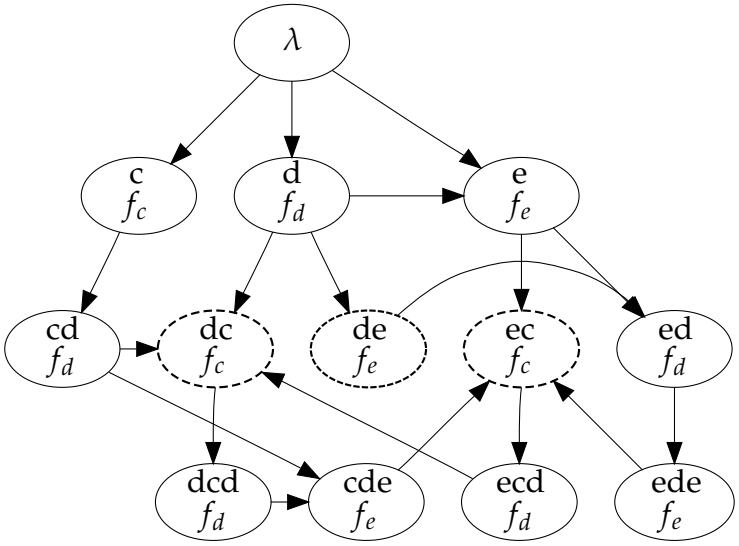


Figure 3.7: A mixed order Markov model

states of a PSA is contained in Σ^*, which is not necessarily the case for (deterministic or non-deterministic) PFAs. This implies that the probability distributions determined by PSAs, prediction suffix trees (PSTs) and Markov chains, are a subset of the probability distributions determined by DPFAs, which is in turn is a subset of the probability distributions determined by PFAs and hidden Markov models (HMMs) (Schwardt, 2007).

Next we give the formal definition of a PFA.

Definition 3.9.1 (Probabilistic finite automata (Thollard *et al.*, 2005)) **-** A probabilistic finite automaton (PFA) is a 5-tuple $\langle$ Q,Σ,τ,π,p $\rangle$ where:

- Σ is a finite input alphabet;
- Q is the state space;
- $\tau \subseteq Q \times \Sigma \times Q$ is a set of transitions;
- π : Q $\rightarrow [0,1]$ defines initial-state probabilities;
- p : $\tau \rightarrow [0,1]$ defines transition probabilities.

The following constraints must be satisfied:

- $\sum_{q \in Q} \pi(q) = 1$;
- for $q_i \in Q$, $\sum_{s \in \Sigma, q_j \in Q} p(q_i, s, q_j) = 1$.

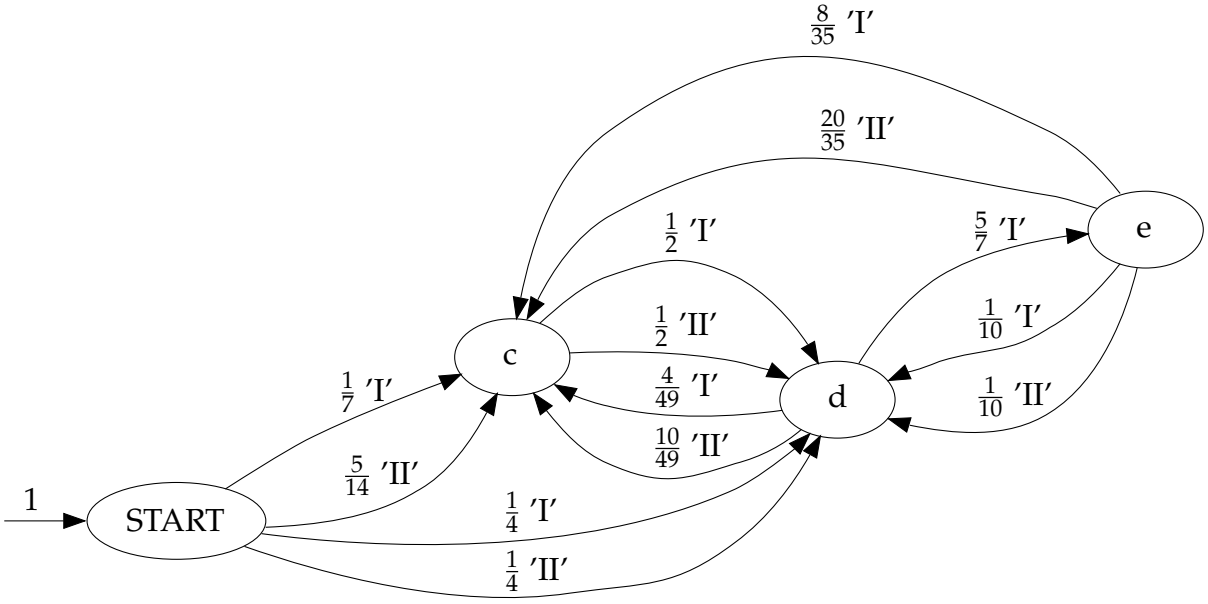


Figure 3.8: A probabilistic finite automaton

Note that a hidden Markov model can be converted to a PFA which determines the same probability distribution. When converting an HMM to a PFA, we use an identical set for Σ and the state space becomes $Q' = Q \cup \{q_{start}\}$. We obtain the transition probability function of the PFA by calculating it over $q_i \in Q'$, s $\in \Sigma$ and $q_j \in Q$ as shown below:

$$p(q_i, s, q_j) = \begin{cases} \pi(q_j) \times b(q_j, s) & \text{if } q_i = q_{start}; \\ a(q_i, q_j) \times b(q_j, s) & \text{otherwise.} \end{cases}$$

We define the initial probabilities of the equivalent PFA as $\pi(q_{start}) = 1$, and all other states have intial probability zero. Figure 3.8 shows a PFA equivalent to the HMM in Figure 3.5.

3.10 Final States

Final or acceptance states in an automaton require the automaton to be in one of these states after the processing of a string, in order to accept the given string. The automata described in the previous sections did not include a final state, since they only modelled strings of a fixed length. This

implies that their distributions were normalised over strings with the same length. For example, all the probabilities of the strings generated from the Markov chain in Figure 3.1, of length two, sums to one, as shown below:

$$\begin{array}{rcl} P(cd) & = & \frac{1}{2} \times 1 \\ P(dc) & = & \frac{1}{2} \times \frac{2}{7} \\ P(de) & = & \frac{1}{2} \times \frac{5}{7} \\ \hline & & 1.0 \end{array}$$

The introduction of a final state normalises the probability distribution of generated strings over all lengths. This is necessary when generating strings of various lengths. The definition of a probabilistic finite automata with a final state is given below.

Definition 3.10.1 (Probabilistic finite automata with final state probabilities (Thollard *et al.*, 2005)) **-** A probabilistic finite automaton with final state probabilities (FPFA) is a 6-tuple $\langle$ Q,Σ,τ,π,p,f $\rangle$ where:

- Σ is a finite input alphabet;
- Q is the state space;
- $\tau \subseteq Q \times \Sigma \times Q$ is a set of transitions;
- π : Q $\rightarrow [0, 1]$ defines initial-state probabilities;
- p : $\tau \rightarrow [0, 1]$ defines transition probabilities;
- f : Q $\rightarrow [0, 1]$ defines the final-state probabilities.

The following constraints must be satisfied:

- $\sum_{q \in Q} \pi(q) = 1$
- for $q_i \in Q$ we have that $f(q_i) + \sum_{s \in \Sigma, q_j \in Q} p(q_i, s, q_j) = 1$

The behaviour of a single acceptance state can be simulated without adjusting the definitions and algorithms described in the previous sections. This is done by appending all training sequences with a final state and emission. This implies an implicit extension of the respective state space and alphabet. In Figure 3.9 an FPFA equivalent to the HMM Figure 3.5 is shown.

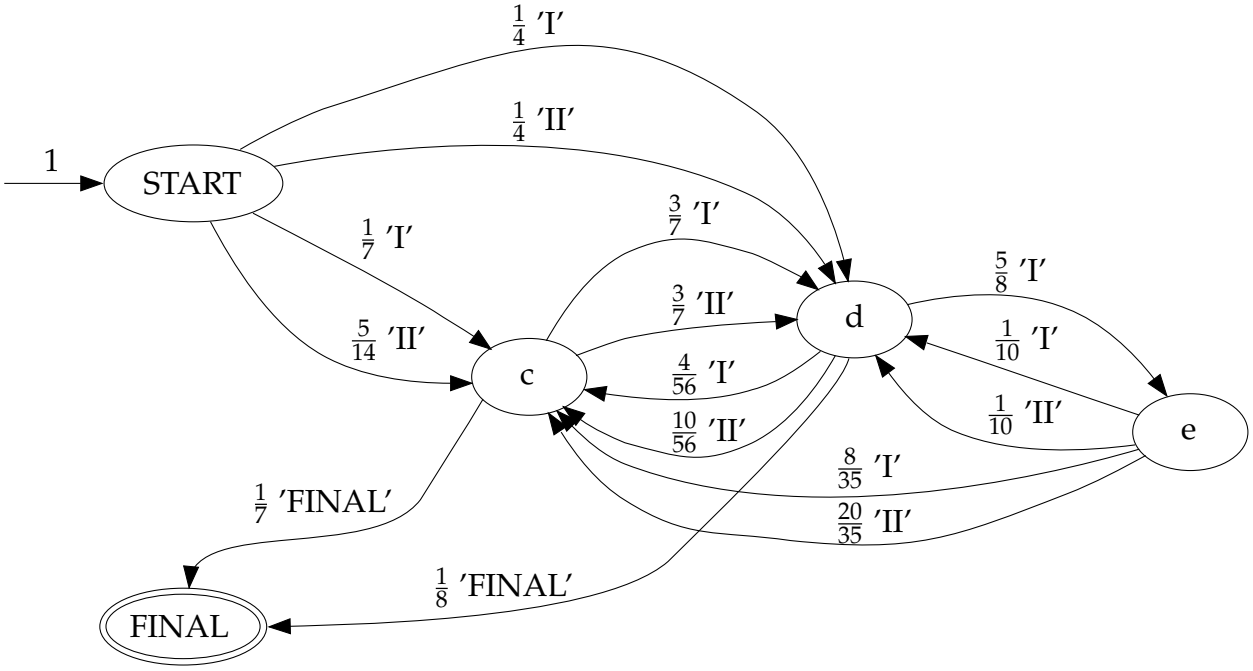


Figure 3.9: A probabilistic finite automaton with a final state

3.11 Conclusion

This chapter provided the required background on probabilistic automata and hidden Markov models. Our music generation/imitation system allows the user to use first, higher and mixed order Markov models. Our first approach to music generation uses first order Markov chains, and is discussed in Chapter 5. In Chapter 6 we discuss how to use hidden Markov models to model the relationship between the melody and harmony of a composition.

Chapter 4

Literature Survey

4.1 Introduction

In this chapter we give survey of techniques used to generate music and how the success of these techniques are measured. Generating music requires some assumptions about the composition process, for example, it could be a random process, a rule-based method or a statistical process. We give an overview of music generation with statistical models (Conklin, 2003) by using sampling methods, in Section 4.2. Music generation using genetic algorithms, agents which interact with each other, and cellular automata, are discussed in Sections 4.3 and 4.4. Next, composing chords for a given melody is described in Section 4.5 and 4.6. Section 4.7 discusses musical style modelling. Lastly, *Willow*, a tree-based music generator, is discussed in Section 4.8, followed by a summary in Section 4.9.

4.2 Overview of Music Generation using Sampling Methods

Music generation with statistical models is done by using sampling. After training analytical models from a composition database, the models can be used for music composition. Context models, including finite state models,

are relatively popular in music generation, since they have the following properties:

- past events are used to predict future events, thus context models are history based;
- the probability of a sequence of events is equal to the the product of the probabilities of the individual events;
- generating music using sampling methods is easy.

Using sparse data to obtain a statistical model could be problematic, since overfitting might for example occur. To solve this problem, a method of viewpoints (Conklin and Anagnostopoulou, 2001), which interpolates between a statistical and rule-based model, is often used. The statistical model is trained using musical pieces of the same style or sometimes only a single piece. The other model is set up with rules based on the style of music.

There are statistical models which are not only history based, such as context-free grammars with a statistical interpretation and non-deterministic top-down tree transducers with added weights, but these statistical models require an existing grammar for a musical style (Högberg, 2005).

A random walk is a stochastical movement between connected states that respects a given set of transition probabilities. One of the shortcomings of a random walk is that the most probable path might not be chosen. This shortcoming is obviously also the reason why a random walk approach can generate many compositions (Ponsford *et al.*, 1999).

Allan and Williams (2005) and Simon *et al.* (2008) used Viterbi decoding to sample from hidden Markov models. Viterbi decoding finds the most probable state sequence in a hidden Markov model. To find the most probable sequence from a non-hidden first order Markov model, a dynamic programming technique similar to Viterbi decoding can be used.

The more complex a model is, the more computationally expensive Viterbi decoding becomes. When Viterbi becomes too expensive, the A* algorithm (Hart *et al.*, 1968) can calculate a specified number of best paths.

Repeating patterns should be discovered and preserved, for instance by using suffix trees, which have been proven to be successful (Conklin and

Anagnostopoulou, 2001; Ziv and Lempel, 1978). We discuss the use of prediction suffix trees for music generation, in Section 4.7.

4.3 Overview of Music Generation using Artificial Agents

4.3.1 Introduction

Rendering of extra-musical behaviour consists of creating artificial agents and mapping their behaviour to musical events, for example a pitch or duration. The musical events generated by agents have no effect on the agents themselves. Thus the agents cannot judge or react to the music. When the behaviour is mapped with some discretion, interesting compositions might be generated (Miranda and Todd, 2003). Improvements can be made by implementing musification rules, for example:

- making certain chord progressions, which are uncommon, impossible (for instance a VII chord is seldomly used in Pop music),
- forcing the drum and piano rhythm to be the same.

4.3.2 Genetic algorithm inspired approaches

Compared to the random approaches, this approach produces music that is in general more pleasing (Miranda and Todd, 2003). Each individual (artificial agent) produces its own music. The survival of a composition is determined by a human or an expert system looking at particular melodic and harmonic developments. As the system evolves, individuals producing better compositions, in terms of the opinion of the judges, are obtained. This implies that more individuals will produce high quality compositions. This system fails if a good human or artificial judge is not appointed. It is important to note that this system still needs human intervention if it is the case that an adequate artificial judge cannot be found.

4.3.3 The cultural approach

In this approach, artificial agents produce their own music signals, which are heard and reacted on by other agents. This might influence the composition they are singing (Todd and Werner, 1999), encourage them to mate or encourage vigilance to defend their own territory (the details of how to achieve this is not specified by Todd and Werner (1999)). Thus the music produced by the agents influences their behaviour. We can thus consider music to have a social role for the agents.

There can be different types of agents, for example composers and critics. Critics can expect certain notes to follow others and might also give points to those composers that are able to surprise them (Miranda, 2002). The critics use an encoded Markov chain for this evaluation. Each critic chooses one mate, but each composer might have more than one mate. The next generation composers and critics are then created by using the qualities that are similar in a critic and the critic's mate.

The cultural approach could also be implemented by using robots with voice synthesisers, hearing devices and a repertoire database. In each round the robots are paired off and for each pair a player and an imitator is identified. The player starts the interaction by randomly choosing a composition from its repertoire and singing it to the imitator. The imitator then chooses the composition closest to it in its repertoire and sings it back. Next the player determines if the composition sung back is the closest to the composition it sung in its own repertoire. If so, it sings the initial composition back to give reassuring feedback. The imitator will then try to be creative, depending on its creative willingness parameter and try to get even closer to the composition. If the composition is not the closest, the player stops interaction without giving reassuring feedback. A system of 5 agents produced an average of 12 compositions per agent in 5000 interactions (Todd and Werner, 1999). Todd and Werner do not discuss the detail of the parameters, including the notion of creative willingness. After various rounds the system settles into an almost fixed repertoire. This is the only system that we are aware of that creates music by allowing music to evolve conceptually in an artificial society.

4.4 Music Generation with Cellular Automata

CAMUS (Miranda and Corino, 2003) is a cellular automata music generator, and consists of two cellular automata algorithms:

- The Game of Life, and
- Demon Cyclic Space.

CAMUS allows the musician to choose his/her own rules for the game of life. The Game of Life produces triples (3 notes). These triples are generated by the x and y coordinates of live cells. The x and y coordinates represent the distances between the notes. The value of the respective cell in the Demon cyclic space determines the instrument playing the notes. Each cell in the Demon cyclic space has a value between 0 and (n-1). In the Demon cyclic space any cell with value k changes its value to (k+1) modulo n, if the cell has a neighbouring cell with a value (k+1) modulo n. The ordering and timing of the triple is determined by the neighbouring cells in the game of life and rules selected/supplied by a musician. Ordering and timing options include: all three notes played at the same time, the second note played first and then the first and third note played together, the three notes played in succession, etc. This approach gives the user an abstract handle on the music generation process.

4.5 Music Generation using Constraints

4.5.1 Introduction

A constraint satisfaction problem consists of a set of constraints which are imposed on a set of variables. Standard constraint satisfaction problem solving is done by using a backtracking algorithm. Such algorithms recursively assign values to variables, and if the newly assigned variable value results in a constraint not being satisfied, the variable value is changed to a new value. In the case where there is no possible satisfying value, the constraint satisfaction algorithm backtracks to the previous variable. Opti-

mising this algorithm involves forcing arc-consistency (Waltz, 1972) by reducing the possible variable value domains. Two variables in a constraint satisfaction problem are arc-consistent if for each possible value of the first variable there exists a possible value of the second variable that will satisfy the constraint. A constraint satisfaction problem is arc-consistent if each allowed ordered variable pair is arc-consistent. Enforcing arc-consistency is done by cycling through all pairs and adjusting the domains to be arc-consistent, until no further adjustments are required. The *Arc Consistency Algorithm 3* (Mackworth, 1977) optimises this process by only reiterating through affected pairs of previous domain adjustments.

4.5.2 Mixing Constraints and Objects to achieve Automatic Harmonisation

BackTalk (Pachet and Roy, 1994) is a finite domain constraint solver, developed in the *SmallTalk* programming language, which uses backtracking and the *Arc Consistency Algorithm 3*. The system *MusES* (Pachet and Roy, 1994) (also developed in *SmallTalk*), is used in conjunction with *BackTalk* and was developed to be applied to the automatic harmonisation problem. *MusES* includes functions for calculating intervals, generating chords in a specific scale and for providing all the scales a specific chord can be played in. There are three types of constraints in the automatic harmonisation problem:

- horizontal distance constraints between notes,
- vertical distance constraints between notes,
- constraints on successive chords.

The constraint satisfaction problem is divided into two stages. The first stage builds a constraint satisfaction problem only for the vertical and horizontal distances between notes. The second stage uses the solutions from the first problem to set up the domains for the chords. These domains are used to solve the constraint satisfaction problem on the successive chords. The values returned by the MusES system are all possible chords to a given melody that adheres to the rules of harmonisation. The system provided chords to 12 and 16 note melodies in 20 and 40 seconds respectively.

4.5.3 Gradus

Gradus (Cope, 2004) composes a second voice accompaniment relative to a first given voice or fixed melody. This is done by composing music which falls inside certain constraints. These constraints or rules are created by analysing a database of 50 compositions. Each of the compositions in the database is a sequence of two notes played simultaneously, and follow the compositional goals set by first species counterpoint (Randel, 2003). First species counterpoint provides a set of goals by which students of counterpoint must compose. These compositional goals include:

- using specific vertical intervals such as thirds, fifths, sixths and octaves,
- avoiding certain parallel motions namely fifths and octaves,
- using only a certain range of horizontal intervals, and
- avoiding continuous same direction horizontal intervals.

The horizontal, vertical, continuous and parallel motions found in the database are used to create rules. The rules are built to allow those motions found and disallow those motions not found.

Gradus uses successful compositions to speed up future compositions. This is done by saving templates to avoid false starts and rules are created to avoid backtracking. Templates consist of three values:

- the interval from the first note of the melody and the seed note (the note before the first note) of the accompaniment;
- the interval between the first and any other note in the melody which has the largest absolute value;
- the interval between the first and the last note of the melody.

Gradus will choose the template which most closely resembles the fixed melody and which has the most successful compositions. The first value of this template is then used to calculate the seed note.

When composing the melody, *Gradus* can move only a maximum step size of two and a minimum of one in any direction. The seed note is used to calculate the possible first notes, from which one is randomly chosen. This

continues for every note, using the previous note as a seed note. Motions could occur which are not allowed by the rules. This will then require the composition to backtrack to the previous note, where a new note will then be chosen. In the case where there are no more notes to choose from, the composition will backtrack again. When backtracking occurs, a rule is created to look one step ahead. These rules consist of three values:

- the vertical interval at the first affected note;
- a list of horizontal intervals for the accompaniment;
- a list of horizontal intervals for the melody.

These rules are created to avoid backtracking and to speed up the composition process.

Gradus can only compose in C major scale, an accompaniment for a given melody and limited first species counterpoints. This severely limits the flexibility of *Gradus*.

4.6 Music Generation with Hidden Markov Models

4.6.1 Introduction

Hidden Markov models are shown, in this section, to successfully model the relation between pitches played by different voices. Allan and Williams (2005) shows how hidden Markov models can be used to harmonise chorales. Simon *et al.* (2008) complements the research by Allan and Williams (2005) by using hidden Markov models in a system, where users provide a melody by singing into a microphone and the system composes complementing chords.

4.6.2 Harmonising Chorales

In western classical music theory training, chorale harmonisation is a traditional exercise. Given a melody, the student is asked to compose three additional lines of music, where each line is a chordless melody. The four lines of music should illustrate that the student has an understanding of the rules of harmonisation.

Allan and Williams (2005) trained a first-order hidden Markov model using harmonisations composed by Johann Sebastian Bach. The data set included 382 chorales, divided between those in major and minor keys. Three fifths of the data was used for training and two fifths for testing. The melody notes and chords were represented by the visible states and hidden states, respectively. The measures were divided in three or four time steps. The Viterbi algorithm was used to predict the chord (three further notes) at each of the time steps, given the model. A second hidden Markov model was used to give a smoother rhythmic form, by linking notes together.

The model was used to calculate the probability of the harmonisations Bach composed, given the respective melody lines. These probabilities were then compared with those calculated by simpler models, such as Markov chains between chord states. It was concluded that the hidden Markov model is superior to the simpler models in harmonising chorales and that a higher-order hidden Markov model would only make the data too sparse.

The generated harmonisations often had large intervals between the bass notes, which is not desirable. These large intervals occur, because the bass line has the most variance with respect to the melody. It was concluded that it is problematic to model the different styles of Bach with a single model. Also, dividing the measures into three or four time steps, implied that only note values as small as quarter notes were taken into account. It was also noted that cadences, which give the generated chorales harmonic closure, were well modelled by the hidden Markov models.

4.6.3 MySong

MySong (Simon *et al.*, 2008) uses a hidden Markov model to generate accompanying chords for an input melody. The input melody is generated by a person singing into a microphone. The chords and melody notes are represented by the hidden and observed states respectively.

The training database includes 298 musical pieces in different styles including Jazz, Rhythm and Blues, Pop, etc. The training data is made less sparse by simplifying chords and transposing all pieces to the key of C. The chords are simplified by casting them to one of five triads namely major, minor, diminished, augmented and suspended.

The hidden states (chords) include a start and end state plus five types of chords for each of the twelve root keys. The hidden Markov model is trained by counting the number of times one state (chord) is followed by another. These values are normalised at the end of training. The observed distribution for each chord is obtained by calculating the total duration each melody note is played with a given chord. The observed distributions are smoothed, by adding each note to each chord for a very short duration, and normalised at the end of training.

There are many scales, but the database is divided into pieces written in major and minor scales respectively, by using a clustering algorithm. The clustering algorithm uses musical heuristics to calculate the initial clusters. A hidden Markov model is calculated for both clusters. Each piece assigns itself to cluster with the hidden Markov model which gives the highest probability for its chord sequence. After all pieces have their newly assigned clusters, the hidden Markov models are recalculated. This process continues until no composition changes cluster.

This process results in two hidden Markov models and one set of observed distributions. The observed distributions are not separated between major and minor modes, since pilot testing showed no significant difference between the major and minor observed distributions. Pitch tracking is used to calculate the pitch of the recorded voice. The pitch tracking method uses an auto-correlation technique which is adapted not to include an octave error correction step. The octave correction is processing intensive and is

assumed to be irrelevant to harmonisation (Simon *et al.*, 2008).

There is no need to extract timing information, since the melody is performed with a computer generated drum beat. All generated chords have the same fixed duration. This implies that for each fixed duration time slice, a chord has to be chosen by the Viterbi algorithm. The log likelihood $L(q,t)$ is calculated for each chord q and each time slice t as shown below:

$$L(q,t) = log \left(\begin{bmatrix} b(q,c) & b(q,c\sharp) & \dots & b(q,a\sharp) & b(q,b) \end{bmatrix} \begin{bmatrix} w(t,c) \\ w(t,c\sharp) \\ \vdots \\ w(t,a\sharp) \\ w(t,b) \end{bmatrix} \right),$$

where $w(t,*)$ is created with a weight element for each musical note, which is calculated by counting the melody notes in the time slice, and $b(q,*)$ is the observed probability vector associated with each chord.

Since there is not a specific correct chord sequence, *Jazz* and *Happy* factors are introduced. The *Jazz* factor is applied to the observed log likelihoods to generate more surprising chord sequences (Simon *et al.*, 2008). The *Happy* factor is applied by giving different weights to the minor and major hidden Markov models as they are added together. Calculating the chord sequence is done for each key and the key with the highest probability is chosen.

Evaluation included 26 recorded vocal melodies of about 13 to 25 seconds, which were transcribed to MIDI, as input melodies. Musical Instrument Digital Interface (MIDI) is an industry-standard protocol used since 1983. Two musical experts were given 5 minutes to compose chords for each of the input melodies using *MySong*, *Band in a Box* (PG Music Inc., 2007) and with manual assignment. Scoring of these compositions was done by 30 volunteer musicians. Each musician received two of the three compositions for each of the 26 melodies. These compositions were randomly selected and equally distributed for each of the six ordered pairings. They then scored these compositions on a scale from 1 to 10, while allowing ties. The mean scores were 6.3, 6.6 and 2.9 for *MySong*, manual assignment and *Band in a Box*, respectively. The number of times one composition type outscored another composition type in a pairing was counted. *MySong* performed better

than manual assignment 95 out of the 264 times with 48 ties. *MySong* performed better than *Band in a Box*, and performed similar to the approach of using manual assignment by experts.

4.7 Music Generation with Prediction Suffix Trees

4.7.1 Introduction

A prediction suffix tree represents a type of lossy compression, since only sequences which are repeated for a significant number of times are considered when building the tree. These significant sequences are called motifs. The prediction suffix tree learning algorithm was developed by Ron *et al.* (1996). The algorithm requires a number of parameters to determine whether a sequence is considered when building the tree (also see Section 3.5). These parameters are:

- the maximum length of a sequence;
- the minimum probability of a sequence occurring in the training data;
- a factor by which the conditional probability of the sequence must be bigger than that of its suffixes.

Ziv and Lempel (1978) first used incremental parsing for lossless compression. A tree is built by parsing a sequence of alphabet symbols. A pointer to the current node, which is initially set to the root node, is used. When parsing the sequence, if the current item in the sequence is a child node of the current node, then the current node pointer is set to the child node. Otherwise the child node representing the current item is created and the current node pointer is moved back to the root node, to start building a new string.

The tree structures created by incremental parsing and the prediction suffix tree training algorithm can both be used to infer a next item, given a context.

4.7.2 Music Generation with Prediction Suffix Trees and Incremental Parsing

Dubnov *et al.* (2003) uses both incremental parsing and prediction suffix trees to generate musical events. Their system also allows constraints on, for example, the minimum context length required to infer the next item. These constraints are implemented using a backtracking algorithm. The system includes a default constraint requiring the maximum context available to be used. This seems to cause infinite loops, which the system detects and breaks free from, by temporarily relieving the constraint.

The tree structures are trained using MIDI files containing live performances. The MIDI files are simplified using five adjustable preprocessing filters. These preprocessing filters include a filter which equalises the attack time of notes played within a specific time threshold of each other, and a filter which reduces the duration value alphabet by equalising duration values which are statistically close. To reduce the overall alphabet size, only the important MIDI events are included in the training. The parameters ignored for training, for instance the velocity value, are still stored in the tree structure. These values provide human characteristics to the generated music. The system also provides crossing over between different styles in the generative process, if common motifs exist. The system can also be used to create multiple instrument pieces with the help of humanly added constraints.

As expected from a lossless compression algorithm, incremental parsing sometimes copies musical sequences. This gives the impression that the style of the training data is too closely reproduced. The prediction suffix tree training algorithm, being a lossy compression algorithm, sometimes produces dissonant notes, which are not representative of the style. This is most likely the result of not enough training data or a too high smoothing factor.

4.7.3 Music Generation with Multi-Attribute Prediction Suffix Graphs

Each music note in a composition consists of multiple attributes including pitch, duration, velocity, etc. Modelling the multiple attributes in a composition using a Markov chain requires a cross product of alphabets. A cross product of alphabets also has an exponential growth effect, in the number of attributes, on the state space. The greater the state space the more data is required to estimate the probability parameters of the model. This problem can be alleviated by providing different memory lengths for each attribute. The state space of mixed order Markov chains need not grow exponentially in memory length as in the case of higher order Markov chains. Triviño-Rodriguez and Morales-Bueno (2001) propose their own model, a multi-attribute prediction suffix graph, which resembles prediction suffix automata. A prediction suffix tree is an example of a multi-attribute prediction suffix graph with a single attribute. Each state in a multi-attribute prediction suffix graph is a vector with a component for each attribute. These components each contain suffixes of attribute values represented as strings. Learning the model involves finding all the states with the maximum specified memory length, that appear more than a calculated bound. Instead of adding states that are found to be significant, as described in Section 3.5, the multi-attribute prediction suffix graph learning algorithm uses a bottom-up approach. The prediction suffix graph is initialised with all states of maximum length and then trimmed, by removing states. As with the prediction suffix tree approach, a parent of a state is preferred if the state does not provide enough extra statistical significance. A parent of a state is a state with a shorter memory length for at least one of the attributes. Lastly, the next symbol probabilities for each attribute of each state are calculated and a smoothing constant is added. The proposed music generation model has attributes for pitch and duration. A multi-attribute prediction suffix graph is trained for each attribute, which is then conditioned on the other attributes.

The prediction suffix graph model was evaluated by a test where 52 human listeners, consisting of beginner and advanced music students, were asked to listen to two compositions, one Bach choral and one computer-composed piece of 10 measures in length. The students were asked to identify which

one of the two was composed by Bach and which by the computer. The listeners chose correctly 55% of the time, which when taking Hoeffding's bound of 12% into calculation, includes the value of 50%. This implies that the prediction suffix graph model passed a partial Turing test.

This system has problems composing music with more than one voice (Triviño-Rodriguez and Morales-Bueno, 2001). If multiple voices were synchronous, the number of attributes could be increased to incorporate another voice, but it is assumed that most compositions are asynchronous. Musical events in a multiple voice composition occurring at different times cannot be modelled by multi-attribute prediction suffix graphs. Furthermore, prediction suffix graph models of music can also incorporate various other attributes, (Conklin and Anagnostopoulou, 2001) such as position in measure and contour. Unfortunately, if too many attributes are modelled, the danger increases of modelling a database of compositions too closely.

4.7.4 The Continuator

The Continuator (Pachet, 2003) is an interactive music style imitator. The system receives a MIDI input stream from the user through a MIDI controller (instrument with MIDI output) which is linked to a MIDI synthesiser, and then sends a MIDI output stream back to the MIDI synthesiser. A prefix tree is built by parsing the input stream from right to left, several times, and adding any new prefixes to the tree. The last element of the input stream is removed for every parse, until there are no elements left. A list of indices are stored in every node of the tree. Each index points to the element following the respective prefix in the input stream. Storing the indices, instead of the probability distributions, saves memory for short sequences, which are typically required for interactive systems. It also allows other attributes, for instance velocity, which are not stored in the tree, to be generated from the original input stream. Generation of the next note only requires walking the prefix tree, and finding the longest available prefix, and then randomly choosing an index from the list. The next element probability is equal to the number of times the index corresponds to the specific element, divided by the number of indices in the list.

The system builds four prefix trees, each requiring less accurate information for generating the next note. The first prefix tree is built using the cross product of the pitch, duration and velocity. The second prefix tree is built using small pitch regions and velocity. Pitch regions group pitches which are close to each other together. The third and fourth model is built using small and large pitch regions respectively. This allows the system to generate notes using as much stylistic information as possible.

It could be reasoned that when a human plays chords it results in imperfect timing. Grouping notes with significant overlaps together fixes this problem. The continuator provides several rhythm generation solutions. The first uses the rhythm provided by the input stream, but after the input stream was parsed, to treat significant overlaps. The second uses a fixed duration for each note. The final approach is to duplicate the exact input rhythm. This approach is not effective when the output sequence is longer than the input sequence. The durations generated often do not fit within the measures determined by the time signature. To fix this, notes are simply truncated when required.

Real time generation is performed in chunks of notes. A few moments before the previously generated note(s) have finished playing, the system generates new note(s) and then the system sleeps for a fixed amount of time.

A fitness function is trained by a database of appropriate chord sequences, allowing the continuator to play with other musicians. The fitness function is a count of how many pitches are the same as those played by the musician, divided by the amount of pitches played by the musician. A constant is used to model the weight of the fitness function versus the Markov probability, when calculating the probability of the next note.

A partial Turing test, which asks listeners to choose between which piece was played by a human and which by the continuator, was passed by the system.

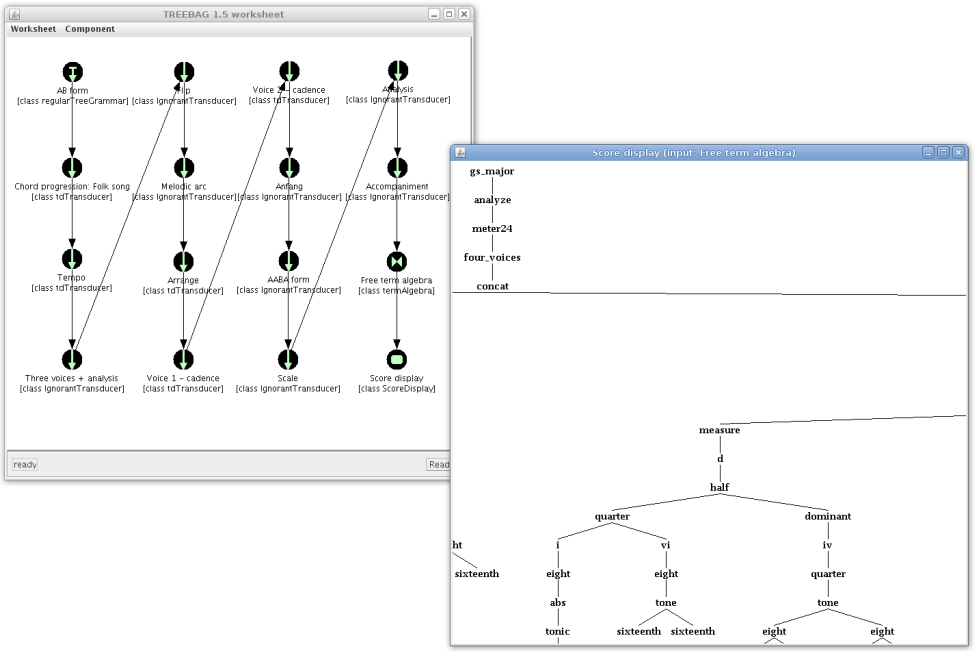


Figure 4.1: A *Willow* worksheet in *Treebag*

4.8 Music Generation with Tree Languages

4.8.1 Introduction

Wind in the Willows (Högberg, 2005), or *Willow* for short, uses tree languages generated by *Treebag*, from regular tree grammars and tree transducers, to generate music. We assume a pre-existing knowledge of tree languages, in particular, regular tree grammars and top-down tree transducers (see Appendix A). *Treebag* is a tree based generator for objects of various types (Drewes, 1998). A worksheet in *Treebag* is a chain of various tree language operations, which include Regular Tree Grammars (RTGs) and Tree Transducers (TDs). *Willow* uses such a worksheet (see Figure 4.1) which starts with a RTG to generate the initial tree. It then uses a chain of TDs to transform this tree into a musical piece. Each of these generation steps will be discussed in the following subsections.

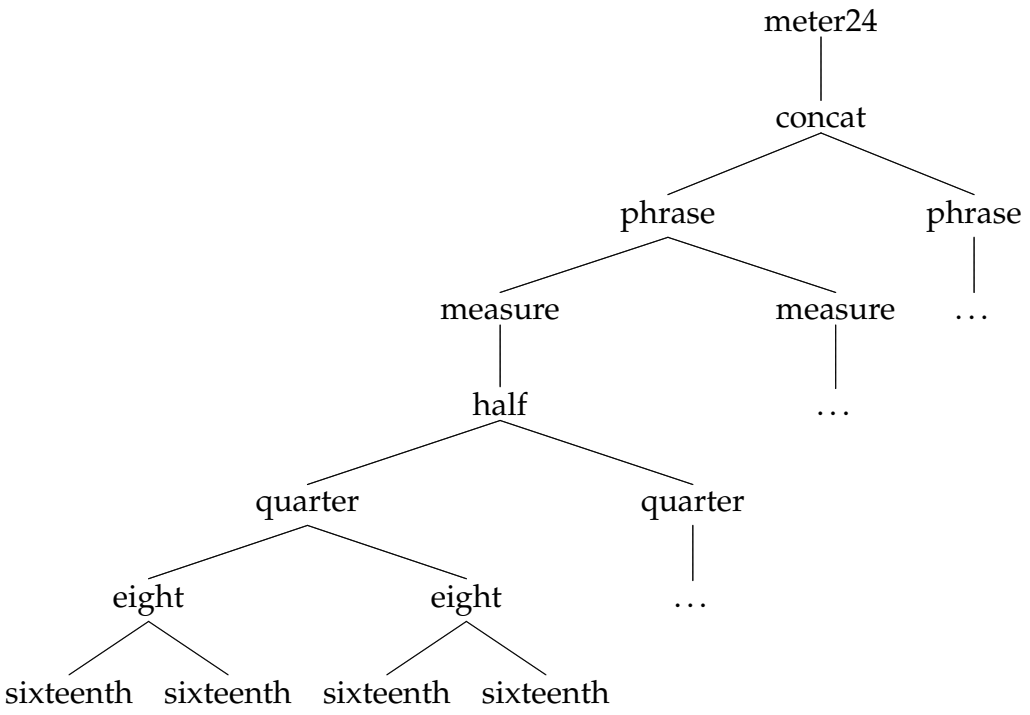


Figure 4.2: Example of a Tree Generated by *Willow's* AB RTG

4.8.2 AB form

The first tree language operation in this chain or assembly line is a RTG, which generates the initial tree. The purpose of this RTG is to generate two phrases, called A and B, respectively. It also chooses the meter or time signature and the number of measures per phrase. An example of a tree generated by this RTG is shown in Figure 4.2. The RTG selects a time signature of 2 quarter beats per measure, and also selects two measures per phrase.

4.8.3 Chord Progression

The chord progression is the first TD applied to the initial tree generated by the previously described RTG. The chord progression TD places chord symbols in a specific order in parts of the tree. The effect of applying a Folk chord progression ($I \rightarrow VI \rightarrow IV \rightarrow V$) TD to a tree is shown in Figure 4.3. This chord progression TD is non-trivial, because of the difficulty of obtaining a relationship between nodes that appear from left to right in a tree. This TD is generated by a script to make it easier for the programmer

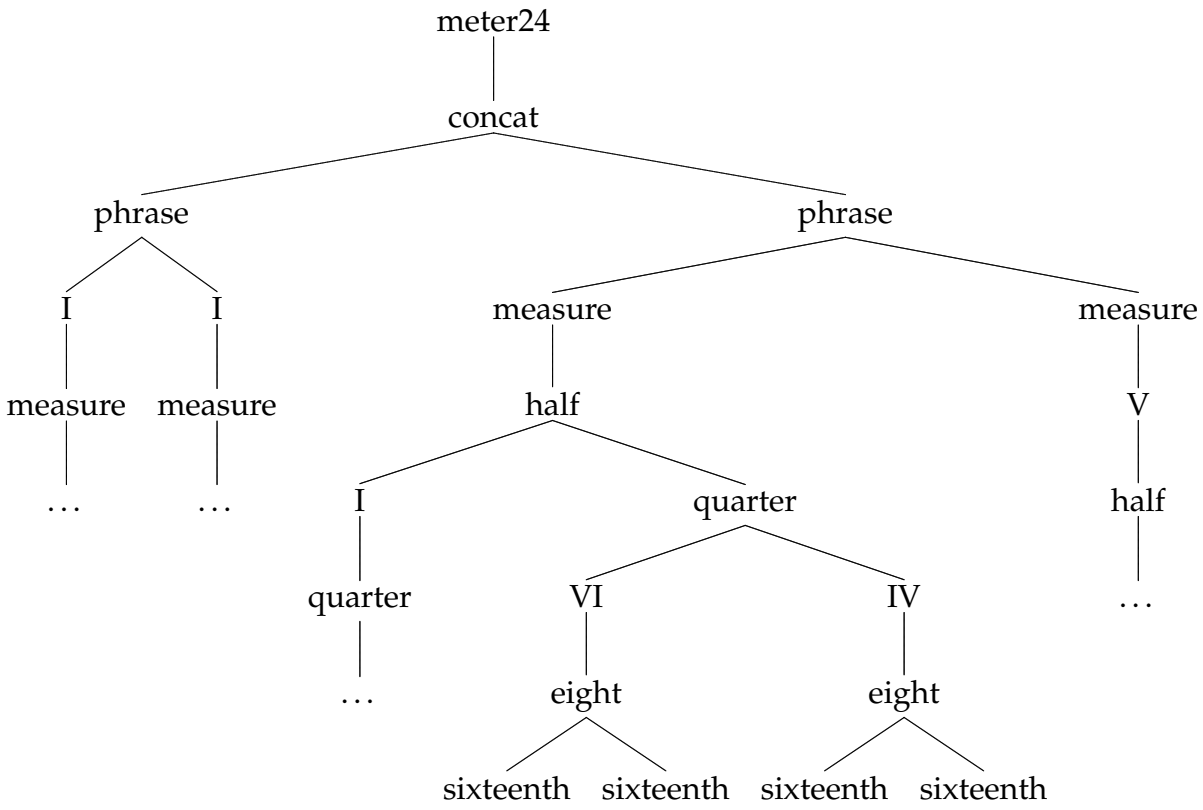


Figure 4.3: Example of a tree transformed by *Willow's* Chord Progression TD

to change the chord progression style.

4.8.4 Tempo

The tempo or tempo progression is achieved in a similar way to the chord progression. It is also generated by a script, and applies various tempos to parts of the tree. An example tempo progression could for example be: $slow \rightarrow fast \rightarrow medium$

4.8.5 Voices

This TD specifies the number of voices (different instruments) in the composition.

4.8.6 Invert Voice Tempo

Two voices, each playing their own melodic line at a fast tempo, will most probably sound cluttered (Högberg, 2005). Inverting one of their tempos will leave the sound less cluttered. The Invert Voice Tempo TD does exactly this.

4.8.7 Melodic Arc

We can think on an intuitive level of the melodic arc as a wave going up and down. The melodic arc TD applies a horizontal movement to parts of the tree. This horizontal movements have embedded rules:

- never jump more than four half steps from one note to the next;
- begin with the root of the first chord and end with the root of the last chord for each phrase.

4.8.8 Arrange

The Arrange TD takes into account the Chord Progression, Tempo Progression and Melodic Arc and then places tones in the tree. For example, the faster the tempo the lower in the tree the notes get placed, ensuring that shorter notes are used in the composition.

4.8.9 Cadence

The Cadence TD applies a cadence to a specified voice. It places a chord progression of a cadence, for example $V \rightarrow I$, at the end of each phrase.

4.8.10 Anfang (beginning)

The Anfang TD ensures that each phrase starts with the root note of the first chord.

4.8.11 AABA form

Initially the tree corresponding to the composition is of the form AB. These phrases could for example be a verse and a bridge. In order to achieve a different form, for example AABA, phrases of the music (or subtrees) have to be copied and moved using a TD.

4.8.12 Scale

Each composition is written in a specific scale. The scale TD determines the scale of the composition. This TD also defines octave registers to adjust the octave of specific voices.

4.8.13 Analysis

The Analysis TD adds a node to the top of the tree which acts as a flag for the post processing step. The post processing step will then provide chord symbols above the appropriate chords in the final score.

4.8.14 Accompaniment

Accompaniment is applied to the possible third voice. The applied voice will play the triad of the chord in the chord progression, one note at a time. A slow tempo might not allow the whole chord to be played. Thus in the case of a slow tempo, only the root note might be played for a full measure.

4.8.15 Post Processing

After applying all the transducers mentioned above, the result is still a tree structure. Converting this tree to music requires some post processing. *Mup* (Arkkra Enterprises, 1992) is an application that can parse a text document into a MIDI (Musical Instrument Digital Interface) file. It handles musical notes as event messages. *Willow* uses a Java program to do this post processing. This Java program parses the tree obtained from applying the various

TDs, and outputs a text file. It also does some final processing on the tree obtained as output from the tree transducers. In the input tree, all the notes are specified after the TDs described above have been applied. These notes specify the duration, the chord in which it is played and sometimes the position in the scale. The scale, melodic arc and chord progression still need to be enforced on the notes. Thus in order to complete the music generation process, these final tree operations first need to be applied. Our music generation system's generation process is also divided into several steps, which is based on this method described by *Willow*.

4.9 Summary

The music generation techniques discussed in this chapter can be divided into the following two broad categories:

- Mapping the behaviour of agents to musical events, as discussed in Section 4.3. This approach is for example used in *CAMUS* (Miranda and Corino, 2003). The agent-based approach requires users to adjust rules, rather than using inference.
- Sampling from statistical models to generate music, as discussed in Section 4.2.

We discussed sampling from hidden Markov models, which is used by Simon *et al.* (2008) and Allan and Williams (2005), and also the use of various types of prediction suffix trees, which is an approach developed by Dubnov *et al.* (2003), Triviño-Rodriguez and Morales-Bueno (2001), and Pachet (2003). The limitation of the prediction suffix tree methods is that they can only compose music consisting of one voice. Two voice harmonisation can be approached from two angles, one being to compose the chords given the melody, as is done by Simon *et al.* (2008), Allan and Williams (2005), Pachet and Roy (2001) and Cope (2004), while the reverse approach (composing the melody given the chords) has only been attempted by Högberg (2005). Harmonising using constraints was attempted by Pachet and Roy (2001) and Cope (2004), and is a unique way to infer compositions from training data, without sampling or using statistical models. Högberg (2005), Allan and

Williams (2005), Miranda and Corino (2003) and Pachet and Roy (2001) all divided their composition process into several steps. This, in many cases, simplifies the state space by not making it the cross product of several alphabets. Our methodology will also:

- use hidden Markov models to harmonise two instrument music;
- use prediction suffix trees to model melody and rhythm;
- divide the composition process into several steps in order to keep the state space manageable.

Chapter 5

Music Generation with XML

5.1 Introduction

The computer music generation system *Willow*, which was developed by Johanna Högberg and described in Högberg (2005), uses a chain of various tree language operations, in fact mainly top-down tree transductions, in order to generate a tree which is interpreted as music. In our implementation the same principles is used, but instead of using tree transductions on ranked trees, we use Extensible Markup Language (XML) transformations. We assume a pre-existing knowledge of XML, in particular the Extensible Stylesheet Language (XSL) and the Document Object Model (DOM) (see Appendix B). These XML operations consists of XSL transformations and the modification of XML documents by using DOM. After each of our steps (see Figure 5.1), a valid MusicXML (Recordare, 2007) file is generated.

Although tree language operations are closely related to XML operations, and can in fact be considered as abstractions of operations on XML documents, most computer scientists are more familiar with XML (Maneth, PhD thesis). Being an industry standard, MusicXML is accessible to musicians by the use of third party music notation programs. In these music notation programs, the MusicXML file can be viewed, played and edited.

Next we briefly describe some of the open source programs and packages that can be used to view and listen to music that is presented in MusicXML

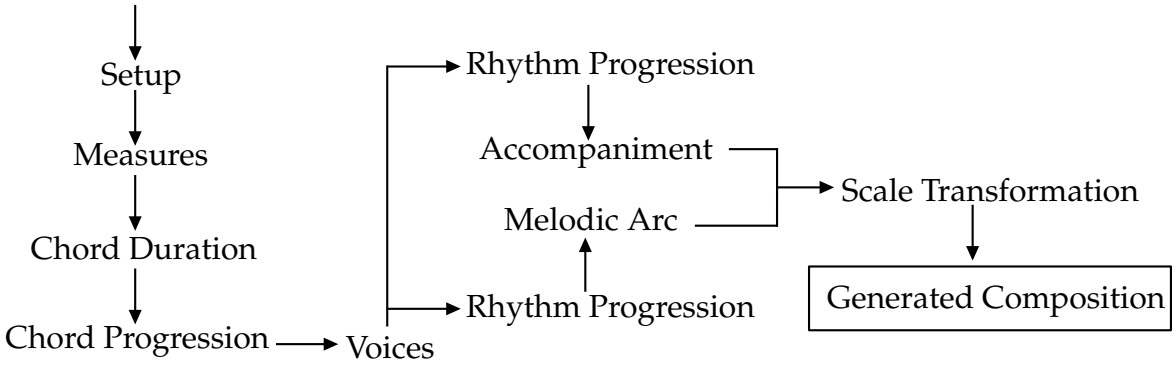


Figure 5.1: Operations

format. The pyScore package (Droettboom, 2005) adds the ability to output MusicXML files in MIDI (Musical Instrument Digital Interface) format. The extended pyScore package (Sinclair *et al.*, 2006) can be used to convert MusicXML to formats usable by the open source music notation software Lilypond (Nienhuys and Nieuwenhuizen, 2008) and Guido (Kilian, 2003). Lilypond can be used to convert MusicXML into standard music notation in PDF (Portable Document Format) or PS (Postscript) format. Guido has similar functionality.

5.2 MusicXML

In this section the structure of a MusicXML document will be explained by means of examples. The following example shows a skeleton layout of a MusicXML file, which is used as the initial file on which all operations are applied.

Example 5.2.1 (Initial MusicXML file) **-** The following MusicXML document consists of a single measure with no notes. All the XML operations described in this chapter are applied to this initial MusicXML file.

```
01 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
02 <!DOCTYPE score-partwise PUBLIC
    "-//Recordare//DTD MusicXML 1.0 Partwise//EN"
    "/musicxml/partwise.dtd">
```

Lines 01 - 02: Note that “partwise.dtd” represents scores by instrument (or part) while “timewise.dtd” represents scores by measure. Representing the score partwise defines the structure with all measures in each part, while representing the score timewise defines all parts in each measure.

```
03 <score-partwise>
```

Line 03: The root element is score-partwise.

```
04   <work></work>
05   <identification>
06     <encoding>
07       <software>superwillow</software>
08       <encoding-description>MusicXML 1.0
09           </encoding-description>
10     </encoding>
11   </identification>
```

Lines 04 - 11: The identification element contains metadata about the score.

```
12   <part-list>
13     <score-part id='P1'>
14       <part-name>Track 1</part-name>
15     </score-part>
16   </part-list>
```

Line 12 - 16: The score-part element contains a list of all the tracks (voices), each represented by an id and a name.

```
17   <part id="P1">
```

Line 17: The first voice starts.

```
18     <measure number="1">
```

Line 18: Next the measure starts.

```
19       <attributes>
```

Line 19: The attributes element is optional for the subsequent measures.

```
20         <divisions>4</divisions>
```

Line 20: The divisions element indicates the number of divisions per quarter note. For example, if divisions element value is “1”, then the duration element value of a quarter note is “1”. Similarly, if the divisions element value is “4”, then a sixteenth note’s duration element value is “1”.

```
21         <time>
22           <beats>4</beats>
23           <beat-type>4</beat-type>
24         </time>
```

Line 21 - 24: The time element specifies the time signature.

```
25       </attributes>
26       <sound pan="8" tempo="120"></sound>
```

Line 25 - 26: The sound element contains playback parameters, for example the panning of the stereo speakers and the tempo of the music.

```
27     </measure>
```

Line 27: When note elements are present, they are contained inside a measure element.

```
28   </part>
29 </score-partwise>
```

In Example 5.2.2 on the next page a typical note element is shown. The pitch element contains the step value which represents the key value, the alter value which has a value of 1 for sharp and 0 for no sharp, and finally the octave value.

Example 5.2.2 (Note Element) **-** An example of a typical note element, which is contained in a measure element.

```
01 <note>
02   <pitch>
03     <step>C</step>
04     <alter>0</alter>
05     <octave>4</octave>
06   </pitch>
07   <duration>16</duration>
08   <type>whole</type>
09   <notations>
10     <technical>
11       <string>5</string>
12       <fret>3</fret>
13     </technical>
14   </notations>
15 </note>
```

The duration value, in Example 5.2.2 above, is used in conjunction with the divisions value as discussed in Example 5.2.1, to represent the duration of a note. Since the value of divisions is 4 in our example, each quarter note has a duration of 4 beats. Thus a whole note, as specified in the type element, has a duration of 16 beats. The notation element is used to specify specific music notation information. In this example, we use tablature notation, which gives the fret and string played on the guitar. There have been other attempts at defining music markup languages, but none of these are as popular as MusicXML. These alternatives include Wedelmusic XML, Standard Music Description Language, eXtensible Score Language and MusiXML (CoverPages, 2006).

5.3 Operations

5.3.1 Setup

The startup MusicXML file contains one measure with no notes, and an instrument setup specification. The first transformations this file go through, are the setup of the tempo and time signature of the composition. These transformations are achieved by using XSLT. The XSLT files are generated by Python scripts. The Python scripts set various variable values in the XSLT file - see for example the explanation of Example 5.3.1. Instead of using Python scripts to generate XSLT, XSLT stylesheets could also call external Python functions, which return the required variable values. Example 5.3.1 shows the core template of the XSL file used for setting the tempo.

Example 5.3.1 (Set Tempo) **-** XSL Template for setting the tempo

```
01 <xsl:template match = "sound">
```

This template will only be applied to the sound element and its children.

```
02   <xsl:element name = "sound">
```

The sound element is recreated.

```
03     <xsl:attribute name = "pan">
```

The pan attribute found in the sound element is recreated.

```
04       <xsl:value-of select="@pan"/>
```

The value of the old pan attribute is retrieved and placed inside the new pan attribute.

```
05     </xsl:attribute>
06     <xsl:attribute name = "tempo">
```

The tempo attribute is recreated.

```
07        <xsl:text>120</xsl:text>
```

The value of the tempo attribute is set. This tempo attribute value is sent to a Python script which generates this XSL template with the required tempo attribute value between the two "xsl:text" tags.

```
08      </xsl:attribute>
09      <xsl:text/>
10    </xsl:element>
11 </xsl:template>
```

5.3.2 Measures

Again an XSL transformation is used to create the number of measures. XSLT does not contain loop functionality, since variable values cannot be changed at runtime. XSLT does however support recursion (since it is a functional language) and a for-each construct which can loop through a set of elements. To achieve the required transformation, we define in XSLT a variable numberofmeasures containing elements named number. Each number element contains an id attribute. The attribute value of id starts at 2 and ends at the number of measures we wish to create. However, cycling through the variable element numberofmeasures is still not possible, since using XPath expressions over variables are not possible. Fixing this, requires the numberofmeasures variable to be transposed to a node-set. This is possible by using EXSLT's (Stewart *et al.*, 2006) node-set function (EXSLT is a community initiative which provides extensions for XSLT). In summary, we cycle through a generated XML tree and in the process we generate additional measures as shown in the next example, Example 5.3.2.

Example 5.3.2 (Measures) **-** Adding measures using an XSL for-each loop

```
01 <xsl:for-each
02        select="exslt:node-set($numberofmeasures)/number">
03    <measure>
04      <xsl:attribute name = "number">
```

```
05       <xsl:number value = "@id"/>
06     </xsl:attribute>
07   </measure>
08 </xsl:for-each>
```

5.3.3 Chord Progression

The chord progression is determined by the user of our software, by specifying a first order Markov model. This is then used in conjunction with DOM in order to modify the current MusicXML file. By using DOM, all notes in the MusicXML file can be retrieved and placed in an array. While walking through the array of notes, we also walk through the chord progression Markov model and apply the chord progression in accordance with the transition probabilities. This cannot be achieved with XSLT, since the number of notes are not known when the XSLT file is being generated.

5.3.4 Chord Duration

Before applying the Chord Progression transformation, the Chord Duration is applied. In popular music, chords are often played for the duration of a whole measure, or sometimes for the duration of half a measure. This is done by placing notes of a specified duration in the measures. These note durations must fit inside measures as specified by the time signature. The durations are again specified by a first order Markov model, using a generated XSL script.

5.3.5 Voices

After the chord progression is applied, we make a copy of the current voice (Part in MusicXML), for each voice in the final composition. This is done with XSLT by changing the part name and id attribute for each specific voice.

5.3.6 Rhythm Progression

The rhythm progression takes each note and divides it into notes of shorter length and the same tone value as the original note. The sum of the lengths of the new notes is equal to the length of the original note. The lengths are determined by using a first order Markov model, similar to the technique that is used to determine the chord progression and chord duration. The only additional constraint is that the sum of the lengths of the new notes must add up to the length of the original note. The rhythm progression function is applied to each voice. Separately specified Markov models, producing different rhythms, are used for each voice.

5.3.7 Melodic Arc

The melodic arc is applied to each note in the chord progression. The note is transposed with a relative distance to its current place in the scale. This is also implemented using a first order Markov model and by using DOM. This function is applied to all voices except the first.

5.3.8 Accompaniment

The first voice is seen as the accompaniment voice. There are a few transformation options:

- Powerchords applies an XSL transformation that replaces all notes with their power chord representation. Power chords are often used in guitar music. It uses the first and third notes of the triad and repeats the root note in the next octave.
- Triadpiano uses XSL to replace all notes with their triad representation.
- Triadone uses DOM to apply a piano that plays the notes of a triad one after the other.

5.3.9 Scale Transformation

This stylesheet transforms the music piece from one scale to another. This is done by specifying types of scales as sets of distances. Similar to the chord progression, the first chord (root chord) is numbered as one and the remainder is numbered relative to the first note as shown in the next example.

Example 5.3.3 (Scale Type Definition) **-** XSL element used to define the major scale type as a set of distances from the root.

```
01 <major>
02   <note num = "1" distance = "1"/>
03   <note num = "2" distance = "3"/>
04   <note num = "3" distance = "5"/>
05   <note num = "4" distance = "6"/>
06   <note num = "5" distance = "8"/>
07   <note num = "6" distance = "10"/>
08   <note num = "7" distance = "12"/>
09 </major>
```

The root note and scale type are specified for the source and destination of the conversion. New scale types can easily be added by adding new distance sets.

5.4 Conclusion

Using XML transformations for each composition step and MusicXML as output, allows the user to view each step in his/her music notation program of choice. This also allows for potential user intervention, allowing the user to edit the MusicXML file between steps. DOM operations were found to be more applicable for the purpose of music generation, as opposed to XSL operations. The theoretical underpinnings of the relation between XML and unranked trees are not discussed in this thesis.

In the next chapter we discuss how to determine probabilities from a data set of music, as opposed to letting the user set the probabilities as in this chapter.

Chapter 6

Style Imitation

6.1 Introduction

The act of musical composition involves a highly structured mental process. Although it is complex and difficult to formalise, it is clearly far from being a random activity - Dubnov *et al.* (2003)

Building probabilistic automata for use in our music generation/imitation system requires the following steps: filtering, preprocessing, extracting and analysing music data.

- A filter is used for filtering out music pieces which we are not capable of analysing, for example a piece with a time signature change.
- The preprocessor is used for correcting assumptions that are found to be incorrect, for example the preprocessor ensures that the duration of the notes of each measure sums to the duration specified by the time signature.
- The extractor collects relevant information such as chord classifications.
- The analyser creates analysis objects including (visible and hidden) Markov models to be used for sampling.

After the music data have been analysed, the analysis objects are integrated into different styles, from which the system can generate compositions and

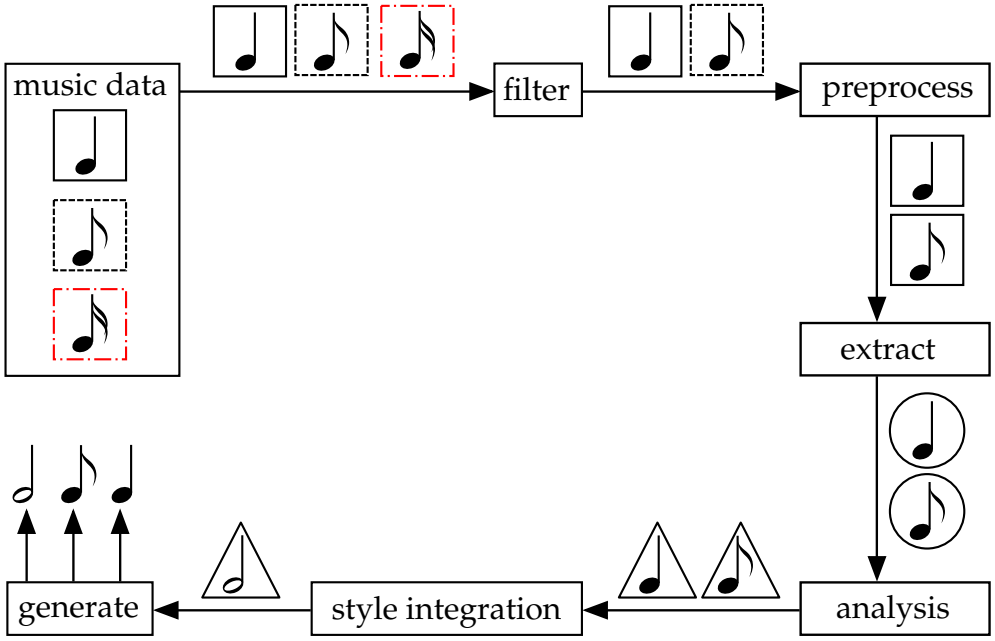


Figure 6.1: Our music generation/imitation system pipeline

imitate these styles, as shown in Figure 6.1.

6.2 Filtering Step

The filtering step is used to ensure that only files which pass the specified requirements are sent to the preprocessing step. In the current implementation the filter ensures that the composition has only one consistent time signature throughout. The filter also verifies that each composition has valid instruments and note values. The instruments and note values which are considered to be valid, are specified in seperate XML files. These XML files can be extended or restricted by a user. Depending on the processing power available, the user might decide, for example, to process only compositions with note values from a whole to an eighth.

6.3 Preprocessor Step

The preprocessor is used to change input files so that they adhere to assumptions on the music data, as required by the other components in the music generation system. These modifications are not supposed to change the way a piece of music is interpreted, but only to make it easier to do so. Presently the preprocessor is only used to fill measures. When transcribing a composition, a person might not add the implied rests at the end of each measure. This is corrected by the preprocessor by adding rests at the end of measures, if required. Although our system generates music without rests, it is still the case that rests need to be accommodated in music data, since it is rare to find a composition without any rests.

6.4 Extraction Step

6.4.1 Introduction

The extraction step obtains the relevant information from the music data by performing the following steps:

- Transposing music data to the key of C while preserving the scale type;
- Obtaining and returning a list of classified chords from the music data;
- Obtaining and returning a list of note durations from the music data.

6.4.2 Transposing the Scale of music data

The key in which a composition is written does not influence the intervals between notes, and thus the relationships, that will be analysed. In other words, whether a composition is written in A or C major will not have any effect on the interval sizes. The type of scale does, however, influence the intervals, and thus compositions in different scale types are organised into different groups. In order to simplify the analysis process, all compositions are transposed to the key of C and, to achieve this, the original key is first obtained.

Definitions of scales, which are not included by default, can be added by a user. Example 6.4.1 shows that each scale type is defined by a set of intervals. Each of these sets of intervals forms a scale which can be transposed to any key.

Example 6.4.1 (Scale Definitions) **-** Scales are defined using intervals.

```
<scales>
  <scale name = 'major'>
    <interval>1</interval>
    <interval>1</interval>
    <interval>1/2</interval>
    <interval>1</interval>
    <interval>1</interval>
    <interval>1</interval>
  </scale>
  <scale ...
  ...
</scales>
```

Finding the scale in which a composition is composed, requires the consideration of accidental notes, and a comparison between all notes in a composition and all possible scales. The chromatic scale contains all pitches, but the chance that a composition is written in the chromatic scale is very low. In general, the scale a composition is composed in contains all pitches that appear frequently in the composition and conversely, most pitches in the scale should appear frequently in the composition. The parameters of our FindPossibleScales algorithm is the set of pitches found in the piece, the frequency count of each of these pitches, and all the scales defined in an XML file. The pitch and weight pairs, of pitches present in the composition, are sorted in ascending order according to weight. A parameter α is used to give leeway and to allow the possibility of accidental notes in the composition. Also, a parameter β is used to narrow down the choice of possible scales for the composition, and to exclude scales which contain more than a certain number of pitches which are not present in the composition.

The value β will most likely rule out the possibility of the chromatic scale, except in the unlikely situation where all pitches in the chromatic scale are used in the composition. The first attempt at finding a scale does not allow any accidentals, and all notes in the selected scale must be present in the composition. When no scale is found, α is adjusted to give leeway, by setting it to the next lowest frequency count. If increasing α does not yield any possible scales, β is increased, by setting it to the index of the current value of α in the sorted set of weights. The selection of the set of possible scales is done by selecting scales that contain all pitches in the composition that have a frequency higher than α. Also, the set of possible scales is reduced to scales for which at most β of their pitches are not present in the composition. If the reduced set of scales is non-empty, we consider it to be the possible scales for the composition, otherwise we increase α or β again. When the non-empty set of possible scales is used, only one of the possible scales must be chosen. In the current implementation, any scale from the set of possible scales, which has a root key with the highest frequency, is chosen from the possible scales. This is not completely correct, but it provides fairly accurate results. Another possible solution is to give more weight to the first and last chord found in a composition.

6.4.3 Extracting Chords

A user is allowed to add chords which are not included by default. All chords are defined in the key of C, as in Example 6.4.2. Each of these chords are transposed to the corresponding chords in other keys.

Example 6.4.2 (Chord Definitions) **-** Chords are defined by type in the key of C. The C major triad (M), consisting of the notes C, E and G, is given below.

```
<chords>
  <chord name = 'M'>
    <key><step>C</step><alter>0</alter></key>
    <key><step>E</step><alter>0</alter></key>
    <key><step>G</step><alter>0</alter></key>
  </chord>
```

```
  <chord ...
  ...
</chords>
```

All chords are retrieved from the pieces of music used as data. Each retrieved chord is compared to each of the previously defined chords, and in this way classified. Some chords contain the same pitches, for example the C augmented triad and the G♯ augmented triad, both contain C, E and G♯. This is obviously problematic. Our system cannot distinguish between the two options and thus randomly chooses the answer, taking into account the root key which had the highest occurrences in the piece. It should also be noted that chords that are arpeggiated are classified as single notes.

6.4.4 Extracting the Rhythm

The rhythm is extracted in a similar way to the method that is used to extract chords. Instead of extracting all pitches played at the same time, the note values (duration of notes) of the pitches are extracted. Note elements contain a child chord element when they are played together with the previous note element in MusicXML, as is illustrated in Example 6.4.3. This implies that all the notes in a chord contain the same type (note value) and duration value. Thus only one duration per chord is needed for the rhythm, since all notes of a chord are sounded together. Durations of note elements with a child chord element are not extracted for the rhythm progression, since we only need one duration for each chord and the first note element of a chord does not contain a chord child element.

Example 6.4.3 (A MusicXML Chord) **-** The chord in this example consists of three notes.

```
<note>
  <pitch>...</pitch>
  <duration>2</duration><type>eighth</type>
  ...
</note>
```

```
<note>
  <chord></chord><pitch>...</pitch>
  <duration>2</duration><type>eighth</type>
  ...
</note>
<note>
  <chord></chord><pitch>...</pitch>
  <duration>2</duration><type>eighth</type>
  ...
</note>
```

6.5 Analysis Step

6.5.1 Introduction

After extracting the relevant information from each MusicXML file used as data, the extracted information is analysed to retrieve the parameters required by the XML operations discussed in Chapter 5. The tempo, scale and time signature are simply retrieved. The *Chord Duration*, *Chord Progression* and *Rhythm Progression* are all represented by first, higher or mixed order Markov chains. The *Melodic Arc* is described by a first, higher or mixed order hidden Markov model. The chords in the *Chord Progression* and *Melodic Arc* are represented by Roman numerals. Generating chords require the roman numerals to represent actual chords which are extracted by the accompaniment analysis. The selection of which voice is the lead and which is the accompaniment is specified by the user, or chosen by voice analysis.

6.5.2 Voice Analysis

Choosing the accompaniment and lead voices for each piece is done by a voice analysis algorithm. This algorithm compresses, for example using *bzip2* (Seward, 2008), each voice and calculates the ratio between the compressed and real size. The voice with the largest ratio (ratio closest to one) is assumed to have the largest information content. This voice is assumed

to be the lead voice or voice representing the melody. Percussion voices are excluded from this analysis. It is preferred that the voices are specified by the user, since using compression to determine the lead and accompaniment is not very accurate. One reason for the lack of accuracy is that the melody might switch between voices.

6.5.3 Chord Duration Analysis

The chord duration analysis is applied to the accompaniment voice chosen by the voice analysis. The parameters of the Markov chain for first, higher or mixed order used to model the chord durations, are calculated by using frequency counts. The training sets are simply lists of durations. These durations are calculated by grouping and summing durations of chords which are the same and which follow each other.

6.5.4 Chord Progression Analysis

The chord progression analysis is applied to the accompaniment voice. The training sets which are lists of roman numerals, are used to calculate the parameters of a first, higher or mixed order Markov chain, by using frequency counts. These roman numerals identify the root key and place of each root key in the scale. A chord is added to the training sets only if it differs from the previous chord.

6.5.5 Accompaniment Analysis

Each unique chord is stored as its root key and type by using its associated roman numeral. In generation, roman numerals are generated as the chord progression. These roman numerals are later replaced by chords from the list, which are associated with each respective roman numeral.

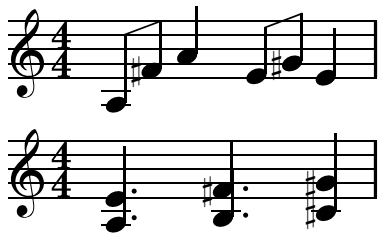
6.5.6 Rhythm Progression Analysis

The rhythm progression analysis is applied to the accompaniment and lead voice. Again the Markov chain parameters of the Markov chain used to model the rhythm progression, are calculated by using frequency counts. The rests and unknown chords are excluded from the training data. This exclusion possibly divides the training set into multiple training sets, but only the first and last set include a start and final state respectively, since this is where they would have been if the training set was not divided. The alphabet of the Markov chain labels states by the duration associated with each note.

6.5.7 Melodic Arc Analysis

The melodic arc analysis calculates the parameters of a hidden Markov model of first, higher or mixed order, used for modelling the melodic arc, by using empirical counts. The observed sequence is described by the accompaniment, while the hidden sequence is described by the melody. The observed states are the roman numerals used by the chord progression. The hidden alphabet is a combination of the index of the melody note in the scale and its relation to the previous note, although it is not really hidden, since the melody notes are available. The relation to the previous note is represented by a plus, minus or an equal sign, depending on whether the current note is higher, lower or equal to the previous note. Each melody note is matched to an accompaniment chord in the observed sequence. The chord that the accompaniment voice was playing, when the melody note was sounded first, is chosen to be the matching chord (see Example 6.5.1).

Example 6.5.1 (Melodic Arc Analysis) **-** This example shows how the music below is analysed.



The table below represents a composition in the scale of E major ($E, F\sharp, G\sharp, A, B, C\sharp, D\sharp$). By using this table, we can determine the matching chord for each note.

Lead	step	A	F	A		E	G	E
	alter	0	1	0		0	1	0
	octave	3	4	4		4	4	4
	note value	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{4}$		$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{4}$
Accompaniment	roman	IV			V			VI
	root key	A			B			C
	type	5			5			5
	note value	$\frac{3}{8}$			$\frac{3}{8}$			$\frac{1}{4}$

The following table represents the hidden and observed sequences retrieved from the above composition.

observed sequence	IV	IV	IV	V	V	VI
hidden sequence	=4	+2	+4	-1	+3	-1

6.5.8 Style Integration

All the analysis objects, namely the various Markov models and lists containing the accompaniment chords and tempos, are exported to XHTML tables for the purpose of easy viewing in a web browser and editing in a text editor. Once all music pieces are analysed, they are combined into analysis objects representing the style of an artist and/or genre. The grouping of the

music pieces, used as data, is left to the end user, and is achieved by the user specifying the file structure used for the music data. This grouping is not necessary, but grouping musical styles together will presumably emphasise certain patterns. Conversely, grouping different styles together could also yield interesting results, but grouping too many styles together could result in a distribution where each note's probability is equal, given any previous note(s). The analysis objects for the various folders cannot always be combined, since the scale, instrument and time signature of each composition need to be taken into account. The analysis objects for the folders are combined under the following conditions:

- *Accompaniment* chord sets are combined if the pieces have the same scale and are played by the same instrument.
- *Chord Progression* and *Melodic Arc* matrices are combined if the pieces have the same scale and are played by the same instrument.
- *Rhythm Progression* and *Chord Duration* chains are only combined if the pieces have the same time signature. The *Rhythm Progression* also needs to be of the same voice.

The tempo object is simply extended to a set of tempos. The analysis objects which are not combined are simply added to a list.

Combining two chains is done by extending the state set to be the union of the two state sets. Finally, the transition probabilities are summed and normalised.

6.5.9 Conclusion

When constructing the training sets for the *Chord Duration* and *Chord Progression*, a new training set is created every time a rest or unidentified chord is encountered. In each analysis phase which uses training sets, only the first and last training set is appended with a start and final state respectively. The analysis objects of each composition, or group of compositions, can be used to generate new compositions by imitating the respective style, which is explained in the next section.

6.6 Music Generation

6.6.1 Introduction

After analysing the compositions, the analysis objects are used to generate new compositions or to imitate the music that was used as training data. This is achieved by using the operation sequence, as discussed in the previous chapter. Generation is not done by using the most probable sequence of notes, but rather by sampling from the distributions obtained from the analysis. First a generation style is chosen by selecting an analysis XHTML file. This could be done randomly or be specified by the user. Then the tempo, time signature, scale and respective instruments are randomly chosen from those found in the selected style.

6.6.2 Rhythm Generation

The Chord Duration and Rhythm Progression are obtained from a Markov chain. This Markov chain generates a note value sequence that sums to a specified duration. This is achieved by extending the state space. In addition to the original PSA state space, it includes a duration left value, which is the duration specified minus the duration of the summed note values in the generated sequence. The sequence is generated in such a way that at the end of the generated sequence, the duration left value is equal to zero.

6.6.3 Carmel

Carmel (Graehl, 2008) is a finite-state transducer package. This package can be used to sample from the distributions created by analysis of the compositions used as data. Rhythm generation, as described above, uses a weighted finite state automaton which is piped through *Carmel* as follows:

```
carmel -OQWEG 1 <wfsa>
```

The weights are in our case probabilities, and we thus supply *Carmel* with probabilistic finite state automata as input. *Chord Progression* uses a probabilistic finite state transducer to transform an input string. Characters in

the input string are all the same, since it is only the length of the string that matters. The number of characters are specific to the length of the chord progression that needs to be generated. In Carmel the calculation is done as follows:

```
echo <input_string> | carmel -OsliG 1 <wfst>
```

The *Melodic Arc* uses noisy channel decoding as described in the tutorial on *Carmel's* website (see Graehl (2008)). The hidden state transitions of the hidden Markov model are represented by a probabilistic finite state automata, while the confusion matrix is described by a probabilistic finite state transducer. We do this calculation in *Carmel* as follows:

```
echo <observed_string> | carmel -IsriG 1 <wfsa> <wfst>
```

In all instances we use the `-G` flag, instead of the `-k` flag, in order to specify that *Carmel* should generate sequences from the distribution determined by the specified automata, instead of calculating the best paths. The fact that we can use *Carmel* for some of our calculations, follows from the fact that hidden Markov models can be converted to equivalent probabilistic finite automata.

6.7 Conclusion

The filter, preprocessor, extractor and analyser can be seen as a pipeline, where only compositions that are appropriate for the preprocessor are allowed through by the filter, and similarly for the other consecutive components.

The music composition software allows the user to choose the order and mixed nature of the Markov chains used, as well as the length of the composition to be generated. We determined experimentally that better results are obtained when smoothing factors are not used. Our system can thus not always generate compositions from the provided music data. Our music generation system allows the user to generate music in any style, given that the user has appropriate music data which adheres to the constraints of the system. These constraints and limitations will be discussed in the next chapter.

Chapter 7

Evaluation

7.1 Introduction

Style imitation can be considered to be an artificial intelligence process, and thus a partial Turing test can be applied as was done by Triviño-Rodriguez and Morales-Bueno (2001), and Pachet (2003). They conducted a survey in which they asked respondents to select the composition that was composed by a human composer, given two compositions, one composed by a human and the other by computer software. We asked a second survey question where the respondent ranked three compositions from favourite to least favourite. The three compositions consisted of a composition composed by a human and two system generated compositions, one being composed using higher and the other mixed order (visible and hidden) Markov models. To ensure fairness, the human composers were given certain constraints which the computer system adheres to. The constraints posed to the human composers are discussed in Section 7.2.2, while the generation of the various computer compositions is discussed in Section 7.2.3.

7.2 Survey Compositions

7.2.1 Introduction

Our evaluation consisted of three surveys, where each survey used music from a different (human) composer (or composers) as data. The following people composed music to be used in the survey:

- the author, Walter Schulze, with 4 years of keyboard playing and 8 years of guitar playing and composition experience;
- Eduard Burger, with 10 years of guitar playing and 8 years of composition experience, preferring the style of metal;
- John Charles Dalton, who has Music Theory grade 8 from the Royal School of Music, Flute Grade 8 and Classical Guitar grade 6;
- affron5 and MindAtrophy, who are two users of the online music composition software, Noteflight.com (2009), with unknown music experience.

By using music composed by these composers, three styles were created:

- Schulze, which consisted of 3 compositions by the author;
- Burger, which consisted of 4 compositions by Eduard Burger; and
- Melted, which consisted of 2 compositions by John Charles Dalton and one each of affron5 and MindAtrophy respectively.

Using such small data sets to infer music styles, could be seen as overfitting of the music data, but it was not possible to amass larger data sets due to time constraints and limits placed on the music accepted as data by our music generation system. It is important to note that when our system generates music from one composition, there might only be one unique path through the various Markov models. This could lead to reproducing the input as output.

Next we discuss the constraints that were placed on every (human) composer.

7.2.2 Composition Constraints

Human composers were asked to compose pieces with two instruments, one playing chords and the other a melody, of more or less 8 measures in length. The Noteflight users were not asked to compose under the given constraints, since their compositions were selected from an online database based on the constraints. The compositions were kept short, since music generated by Markov models do not have an overarching structure. The lack of overarching structure, including phrases and their ordering, is more noticeable for longer compositions. Also, the compositions composed by the human composers had to be in the same scale and time signature. This was required, to ensure that the compositions agree enough to be combined for the purposes of being used as data for our generation system. The constraints placed on the compositions created by the human composers were as follows:

- The time signature should be constant throughout a composition;
- The notes of a chord should be sounded at the same time and not arpeggiated;
- Both instruments should play without any rests;
- The compositions should conclude with a cadence;
- Notes are not allowed to sound from one measure to another;
- Only note durations from a whole to a sixteenth should be used, and durations such as triplets are also not allowed.

The pieces composed were all in the scale of C major and all had a time signature of four quarter beats per measure.

7.2.3 Composition Generation

A number of compositions were generated in each style from which three compositions were chosen. One of the selected compositions was generated using higher order (visible and hidden) Markov models and the other two were generated by using mixed order (visible and hidden) Markov models. One of the mixed order generated compositions was used with one of the

human composition from which it was generated, for the purpose of conducting a partial Turing test. Thus in the survey it was asked which one of these two compositions was composed by a human. The two remaining generated compositions were grouped with another human composition from which they were generated. In the survey it was asked to rank these three compositions from favourite to least liked. The generated compositions were tested against the compositions used as input, since they represent the same style of music. In this way any style bias a respondent might have was eliminated.

The compositions generated for each style consisted of five compositions for each Markov model memory length up to length 10 and type (higher and mixed order). Unfortunately for the Melted style we could only generate higher order up to a memory length of two and mixed order up to a memory length of five, due to sparseness of the music data. Thus for the Melted style we generated 20 compositions of mixed order, which included five compositions for each memory length from two to five, and 20 for higher order, all with a memory length of two. Therefore only 40 compositions were considered in the process of selecting three compositions for the Melted style survey.

This survey was hosted on a website and completed mostly by staff and students from Stellenbosch University and their respective friends and colleagues. When a respondent entered the site, he/she was assigned a random survey. In the next section, the results from the survey are discussed.

7.3 Results

The survey was completed by 440 respondents. In Figure 7.1 the results of the first survey question is shown. This question was setup as a partial Turing test, to discover whether a human could distinguish between a composition generated by our music generation system and a composition composed by a human. The overall results show that more than a third of people completing the survey made the wrong choice, which is promising. The results obtained from the style of Burger were the most promising. The

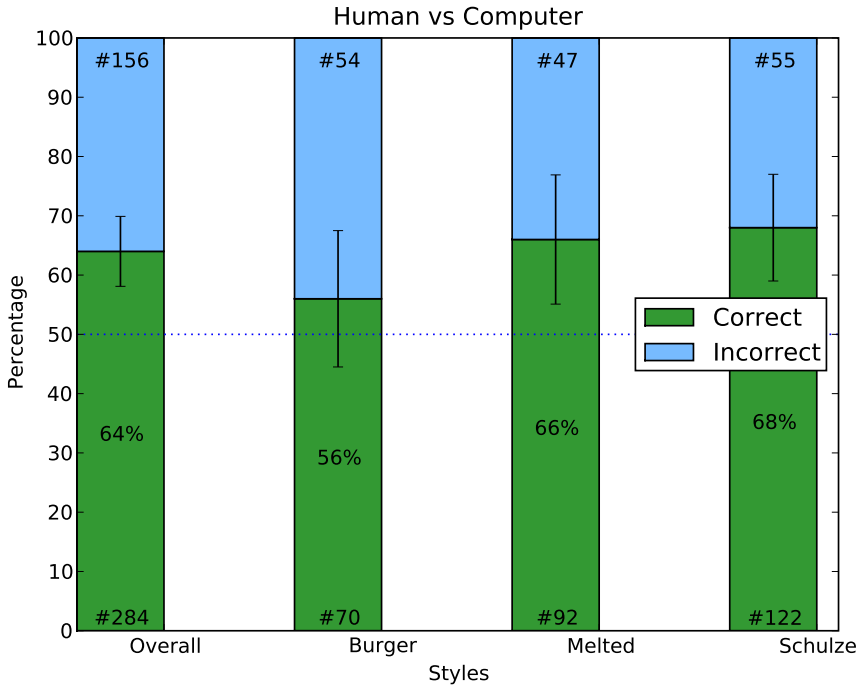


Figure 7.1: Turing Histogram

A Histogram representing the percentage of correct vs incorrect answers, given the question: Which one of the two given compositions was composed by a human and which by a computer.

interval indicated on the histogram bars was calculated by using the normal approximation of the *binomial proportion confidence interval*. This indicates a 99% confidence interval for the average number of people that will answer this question correctly. The style of Burger produced the most promising results and the style of Schulze the worst results. This is slightly surprising, but might be due to more interesting motifs in the compositions in the style of Schulze.

The next survey question asked the respondents to rank three given compositions in the order of preference. The results are shown in Figure 7.2. The aim was to achieve equal ratings for each of the compositions and to show that our music generation system could compose music which is just as pleasant as those composed by a human. Again the overall result shows that this was not completely achieved, but promising results were obtained with the Burger and Melted styles.

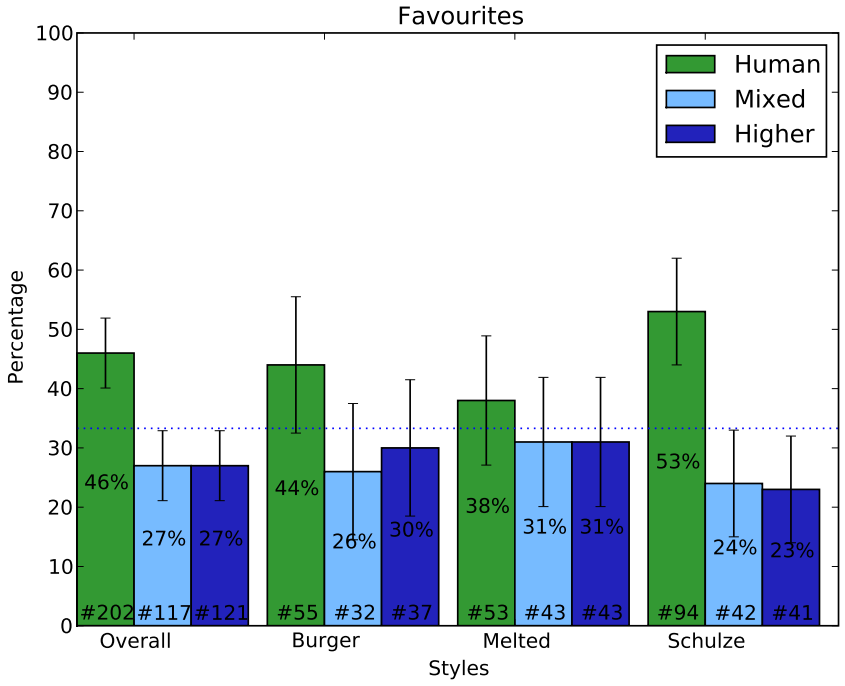


Figure 7.2: Ranking Histogram

A Histogram representing the percentage of respondents that found a specific composition to be their favourite.

In Figure 7.3 we show that in all styles almost half the time one of the computer generated compositions was preferred over the human composition, by the survey respondents.

Finally we did an analysis to correlate the answers of the two survey questions. In Figure 7.4 we show the result of dividing the answers to the survey into four categories. The four categories represent the combinations of whether the human was identified correctly or incorrectly and whether the human or a computer composition was preferred.

This analysis shows that most respondents that identified the human incorrectly preferred a computer composition. This is especially true for the Melted and Burger styles.

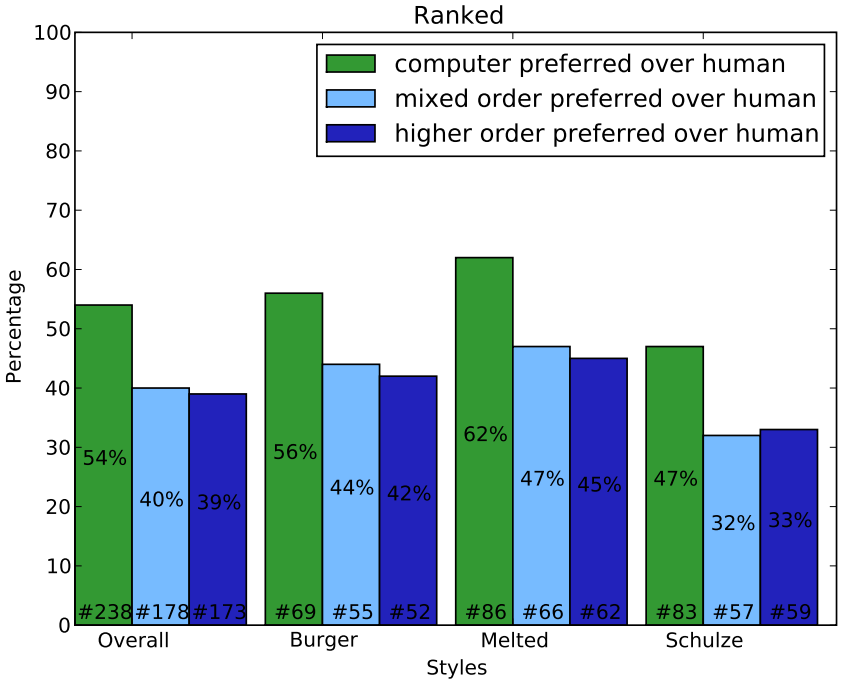


Figure 7.3: Computer Histogram

A Histogram representing the percentage of respondents that preferred at least one of the computer compositions over the human composition.

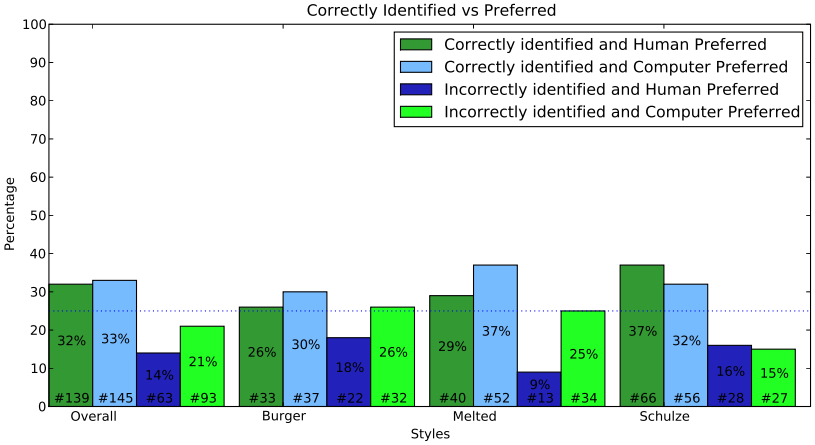


Figure 7.4: Correctly Identified vs Preferred Histogram

The two survey questions are cross correlated into four categories.

7.4 Conclusion

We did not evaluate compositions generated by first order Markov models, since we decided to keep the survey short and a comparison between higher and mixed order was thought to yield more interesting results. Unfortunately the benefit of mixed order over higher order was not pointed out by the survey results. We did find that mixed order was able to produce compositions of a much higher memory length, compared to higher order. The survey data showed that human compositions are still preferred, but also that 68% of respondents either could not distinguish between a human and computer composition, or preferred a computer composition over a human composition. Promising results were achieved for the style of Burger in a partial Turing test and for the styles of Burger and Melted in the ranking test, where mixed, higher and human compositions did almost equally well.

Chapter 8

Conclusion

We investigated whether a computer could compose two-instrument music just as well as a human. Our approach to music generation was modelled on the assumption that artists compose music representative of a certain style and so too it should be possible to train a computer to generate compositions in a certain style. The resulting music generation is a combination of rule-based and statistical sampling methods.

Dividing the music generation process into several different operations reduces the state space and processing power needed for each generation step. Also, the number of compositions the system is able to generate is increased. For instance, the same melody can be mapped to multiple rhythms. Unfortunately, the fact that the different generation steps do not take each other into account results in failed composition attempts, for example, when the *Rhythm Progression* is unable to generate note values that fit into the generated chord durations.

We have reconfirmed that hidden Markov models are well suited to model the relationship between a melody and accompanying chords. As expected, when we used mixed order Markov models, we were able to analyse compositions by using a much longer memory length compared to when we used higher order Markov models. Interestingly enough, based on our opinion, longer memory length did not always have a significant impact on the quality of the compositions that our system produced.

In our survey only 64% of respondents were correct when asked to identify which one of two compositions was composed by a human, given that one was composed by a human and the other by our computer system. Also, only 46% of respondents preferred a human composition over the two computer generated compositions presented with it.

Chapter 9

Future Work

Miles Davis famously described his improvisational technique as parallel to the way that Picasso described his use of a canvas: The most critical aspect of the work, both artists said, was not the objects themselves, but the space between objects. In Mile's case, he described the most important part of his solos as the empty space between notes, the "air" that he placed between one note and the next. Knowing precisely when to hit the next one, and allowing the listener time to anticipate it, is a hallmark of Davis' genius. - Levitin (2006)

This thesis lacks a proper analysis of rhythm. In particular, the following aspects of music generation, involving rhythm, are left as future work:

- extra note attributes, for instance accentuation;
- linked notes and notes which are played over consecutive measures;
- an analysis of the relationship between the rhythms of different voices;
- generation of music with rests, where appropriate; and
- percussion.

These aspects need to be addressed throughout the music generation pipeline, from the filtering step to the generation of new compositions.

Another major shortcoming is the limitation of generating only two voices. This shortcoming is reflected in the lack of accuracy in detecting the lead and accompaniment voices. A dynamic number of voices should be considered when analysing and generating compositions.

Furthermore:

- The theoretical underpinnings of unranked trees in relation to XML is not discussed.
- Analysing scale changes in compositions, and a new algorithm for finding scales. which gives better accuracy and considers the ascending and descending nature of the melodic minor scale, should be implemented.
- Generating accidentals, i.e. notes not in the given scale, should be considered.
- Chord classification is naïve and needs to be improved, for instance arpeggiated chords should be extracted as a chord and not as single notes.
- The note ordering of chords (inversions) should be taken into account, since certain chord inversions may sound more dissonant than others.
- Clustering can be used to group compositions together and to form styles, instead of relying on the user to group them.
- Being able to accommodate fewer restrictions on training data.
- Other music formats, such as MIDI for example, should be supported in the data extraction phase.
- The overhanging structure of a composition, including division into phrases and the repitition of a theme, should be investigated.

Appendices

Appendix A

Tree Languages

A.1 Trees

A Tree is defined to be a connected acyclic graph in graph theory. In Computer Science trees are usually labelled, ordered and rooted. Rooted trees single out a node as being the root and any two connected nodes in the tree have an inherent parent-child relationship. In an ordered tree, the children of each node have a specific ordering.

Next we describe trees that are labelled and ranked. We denote the set of non-negative integers by $\mathbb{N}$. A ranked alphabet Σ is a finite alphabet that is partitioned into disjoint subsets Σ_k, for $k \in \mathbb{N}$. Thus $\Sigma = \cup_{k \in \mathbb{N}} \Sigma_k$ and $\Sigma_i \cap \Sigma_j = \emptyset$ if $i \neq j$. The rank of a node is the number of children of the given node. In labelled ranked trees each node of rank k is labelled by a symbol in Σ_k. Most Extensible Markup Language (XML) documents can be represented by unranked trees, except when links are used.

The tree in Figure A.1 has the signature $\{s : 2,$ 𝅗𝅥 $: 0,$ ♩ $: 0\}$, and it describes one measure in $\frac{4}{4}$ time. This tree represents a quarternote followed by a halfnote followed by a quarternote. Using the notation $a[t_1, \ldots, t_k]$ to denote a tree with root node labelled by the symbol a of rank k, and with subtrees $t_1, \ldots, t_k$, and using simply a if $k = 0$, the tree in Figure A.1 is denoted by $s\left[\, s[$ ♩, 𝅗𝅥 $]$, ♩ $\right]$. In essence one can interpret nodes labelled by s in Figure

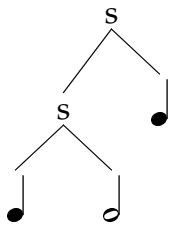


Figure A.1: A sample tree over the ranked alphabet {s: 2, 𝅗𝅥 : 0, ♩ : 0 }

A.1 as a way to convert unranked trees, with notes at the leaves, to ranked binary trees.

Definition A.1.1 (Set of trees over Σ, or T_Σ) **-** (Drewes, 2006) Let Σ be a signature. The set T_Σ of all trees over Σ is the smallest set of strings such that $t_1, \ldots, t_n \in T_\Sigma$ implies that $f[t_1, \ldots, t_n] \in T_\Sigma$, for every $f \in \Sigma^{(n)}$.

Note that all trees consisting of only a root node labelled by a symbol of rank 0 in Σ, is by definition in T_Σ. The trees $t_1, \ldots, t_n$ are called direct subtrees of $f[t_1, \ldots, t_n]$. The generation of trees by grammars are discussed next.

A.2 Regular Tree Grammars

Context-free grammars, the formal equivalent of the Backus-Naur formalism (Ford, 2004), are well-known to most Computer Scientists. The Backus-Naur formalism is often used to describe the syntax of programming languages. A context-free grammar is a finite set of rules that generates a language of strings. This formalism was proposed by Noam Chomsky (Chomsky, 1956). In a similar way, a regular tree grammar (RTG) is a finite set of rules that generates a language of trees (Drewes, 2006).

Definition A.2.1 (Regular Tree Grammars (RTG's)) **-** (Drewes, 2006) A regular tree grammar is a tuple G = (N,Σ,R,S) consisting of

- a finite alphabet N of nonterminals of rank 0;
- a finite output alphabet Σ, disjoint from N, whose elements are called terminals;

- a finite set R of rules of the form $A \rightarrow t$, where $A \in N$ and $t \in T_{\Sigma \cup N}$; and
- an initial nonterminal $S \in N$.

Regular Tree Grammars generate trees by using the rules of the grammar in a sequence of derivation steps.

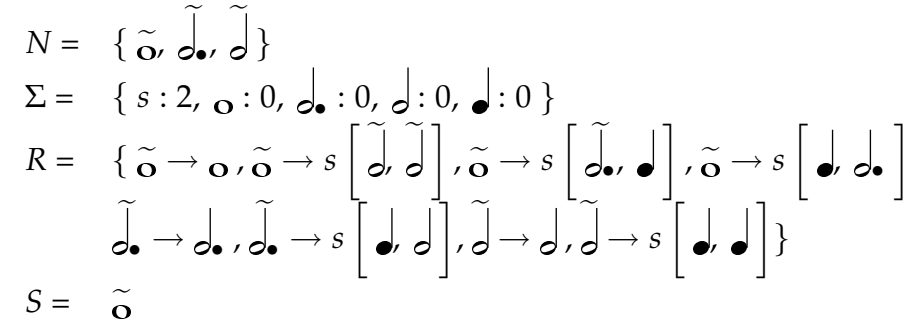
Definition A.2.2 (Regular Tree Grammar Derivations) **-** (Comon *et al.*, 2007) Let G = (N, Σ, R, S) be a regular tree grammar. For trees s, s′ $\in T_{\Sigma \cup N}$, there is a derivation step s $\Rightarrow_G$ s′ (or simply s $\Rightarrow$ s′), if:

- s is a tree which has at least one leaf node labelled by a nonterminal A.
- there is a rule A $\rightarrow t \in R$, and
- s′ is the same tree as s, except that one of the leaf nodes labelled by A is replaced by the tree t.

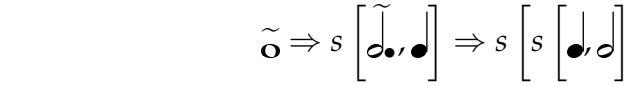
A sequence $t_0 \Rightarrow t_1 \Rightarrow \cdots \Rightarrow t_n$ of n derivation steps ($n \in \mathbb{N}$) is denoted by $t_0 \Rightarrow^n t_n$ and derivations of any length by $t_0 \Rightarrow^* t$. The regular tree language generated by G, denoted by $L(G)$, is the set of trees $\{t \in T_\Sigma | S \Rightarrow_G^* t\}$.

In Example A.2.1 we give an example of a derivation in an RTG.

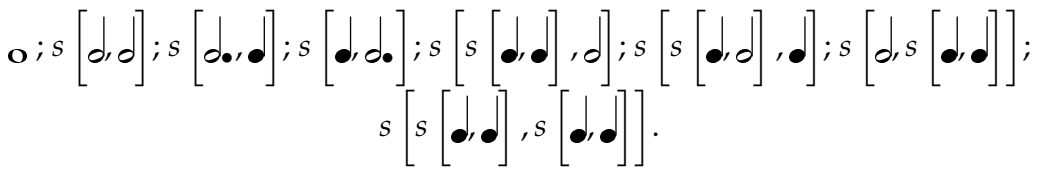
Example A.2.1 (RTG derivations) **-** In this example we give an RTG which generates the tree in Figure A.1.



The RTG above can generate the tree in Figure A.1 as follows:



This is not the only tree that can be generated from this grammar. Here are all the other possibilities:



A.3 Top-Down Tree Transducers

Ranked trees, regular tree grammars and top-down tree transducers play an essential role in the music generation system Willow (Högberg, 2005). Top-down tree transducers are the tree analogues of string transducers. Thus a top-down tree transducer takes a tree as input and produces a tree or nothing as output. Tree transducers (TDs) have both input and output alphabets.

Definition A.3.1 (Tree transducer) **-** (Comon *et al.*, 2007) A top-down tree transducer (TD) is a tuple td = (Q, Σ, Δ, R, q_0), where

- Q is a finite ranked alphabet of states, all of rank one;
- Σ and Δ are finite ranked input and output alphabets, respectively;
- R is a finite set of *rewrite rules* of the form $q[a[x_1, \ldots, x_k]] \rightarrow t$, where $a \in \Sigma_{(k)}$, $q \in Q$, and $t \in T_\Delta(Q(X_k))$;
- $q_0 \in Q$, is the *initial state*.

We briefly explain the notation $T_\Delta(Q(X_k))$ used in the definition above. Firstly, $Q(X_k)$ denotes the set of trees consisting of a state in Q as root node and a variable in X_k as only child, where X_k is the set of variables $\{x_1, \ldots, x_k\}$. A tree t in $T_\Delta(Q(X_k))$ is obtained by taking a tree s in T_Δ and replacing (perhaps) some (or even all) of the leaf nodes of s by trees in $Q(X_k)$.

Next we describe the mechanism with which a top-down tree transducer td=(Q,Σ,Δ,R,q_0) computes output trees from input trees. Let $s \in T_\Sigma$ be an input tree. The computation starts with the tree $q_0[s]$. By $q_0[s]$ we mean the tree with the state q_0 as root and with the tree s as the only child of the root node. Assume that s is given by $a[t_1, \ldots, t_k]$. Next we take any rule

in R of the form $q_0[a[x_1, \ldots, x_k]] \rightarrow t$, where a is the label of the root node of s, and k is the rank of a. If no such rule exists, the transducer does not produce any output when given the input tree s. Note that in the special case where $k = 0$, we have that $t \in T_\Delta$. Since $t \in T_\Delta(Q(X_k))$, the tree contains possibly some of the variables $x_1, \ldots, x_k$ at the leaf nodes. A given variable may also appear at more than one leaf node. When we apply the rule $q_0[a[x_1, \ldots, x_k]] \rightarrow t$ to the tree $q_0 s$, we obtain the tree $t[\![t_1, \ldots, t_k]\!]$. We denote by $t[\![t_1, \ldots, t_k]\!]$ the tree that is obtained by replacing x_i in t by t_k. Note that when we replace x_i by t_i in t, we obtain a tree with a state in Q above each t_i. At each node that is labelled by a state in $t[\![t_1, \ldots, t_k]\!]$, we repeat the rewriting process that was used at the root of $q_0 s$. We repeat this process until we obtain a tree t in T_Δ. We denote the computation that takes s as input and produces t as output by $s \Rightarrow_{td} t$. Also, by $td(s)$ we denote the set $\{t \in T_\Delta | q_0 s \Rightarrow_{td} t\}$. In other words, $td(s)$ is the set of all possible trees in T_Δ that can be obtained if we start with $q_0 s$ and apply the rules in R until we obtain a tree in T_Δ.

In the next example we show how a given transducer transforms the tree in Figure A.1 in such a way that it contains multiple phrases. The parents of the leaf nodes of the trees obtained as output from the tree transducer are also marked by pitch values.

Example A.3.1 - This example gives a non-deterministic total top-down tree transducer which transforms the tree from Figure A.1 to have pitch and multiple phrases. This transducer creates two copies of the input tree with phrase as root. Next it places the pitches C, E or G on the yield of the tree, always starting with C.

$$
\begin{aligned}
\Sigma = \quad & \{\, s:2,\ \mathbf{o}:0,\ \mathbf{d}.:0,\ \mathbf{d}:0,\ \mathbf{q}:0 \,\} \\
\Delta = \quad & \{\, \mathsf{phrase}:2,\ s:2,\ \mathbf{o}:0,\ \mathbf{d}.:0,\ \mathbf{d}:0,\ \mathbf{q}:0,\ c:1,\ e:1,\ g:1 \,\} \\
Q = \quad & \{START, C, E, G\} \\
R = \quad & \{START[x_1] \rightarrow \mathsf{phrase}[C[x_1], G[x_1]]\,, \\
& C[s[x_1,x_2]] \rightarrow s[C[x_1], E[x_2]]\,, \\
& E[s[x_1,x_2]] \rightarrow s[E[x_1], G[x_2]]\,, \\
& G[s[x_1,x_2]] \rightarrow s[G[x_1], G[x_2]]\,, \\
& G[s[x_1,x_2]] \rightarrow s[G[x_1], C[x_2]]\,, \\
& C[\mathbf{o}] \rightarrow c[\mathbf{o}]\,, \quad C[\mathbf{d}] \rightarrow c[\mathbf{d}]\,, \quad C[\mathbf{d}.] \rightarrow c[\mathbf{d}.]\,, \quad C[\mathbf{q}] \rightarrow c[\mathbf{q}]\,, \\
& E[\mathbf{o}] \rightarrow e[\mathbf{o}]\,, \quad E[\mathbf{d}] \rightarrow e[\mathbf{d}]\,, \quad E[\mathbf{d}.] \rightarrow e[\mathbf{d}.]\,, \quad E[\mathbf{q}] \rightarrow e[\mathbf{q}]\,, \\
& G[\mathbf{o}] \rightarrow g[\mathbf{o}]\,, \quad G[\mathbf{d}] \rightarrow g[\mathbf{d}]\,, \quad G[\mathbf{d}.] \rightarrow g[\mathbf{d}.]\,, \quad G[\mathbf{q}] \rightarrow g[\mathbf{q}]\} \\
q_0 = \quad & START
\end{aligned}
$$

A sample transformation of a tree with this top-down tree transducer is shown below:

$$
\begin{aligned}
& s\left[s\left[\mathbf{q}, \mathbf{d}\right], \mathbf{q}\right] \\
\Rightarrow\ & START\left[s\left[s\left[\mathbf{q}, \mathbf{d}\right], \mathbf{q}\right]\right] \\
\Rightarrow\ & \mathsf{phrase}\left[C\left[s\left[s\left[\mathbf{q}, \mathbf{d}\right], \mathbf{q}\right]\right], G\left[s\left[s\left[\mathbf{q}, \mathbf{d}\right], \mathbf{q}\right]\right]\right] \\
\Rightarrow\ & \mathsf{phrase}\left[s\left[C\left[s\left[\mathbf{q}, \mathbf{d}\right]\right], E\left[\mathbf{q}\right]\right], s\left[G\left[s\left[\mathbf{q}, \mathbf{d}\right]\right], C\left[\mathbf{q}\right]\right]\right] \\
\Rightarrow\ & \mathsf{phrase}\left[s\left[s\left[C\left[\mathbf{q}\right], E\left[\mathbf{d}\right]\right], e\left[\mathbf{q}\right]\right], s\left[s\left[G\left[\mathbf{q}\right], G\left[\mathbf{d}\right]\right], c\left[\mathbf{q}\right]\right]\right] \\
\Rightarrow\ & \mathsf{phrase}\left[s\left[s\left[c\left[\mathbf{q}\right], e\left[\mathbf{d}\right]\right], e\left[\mathbf{q}\right]\right], s\left[s\left[g\left[\mathbf{q}\right], g\left[\mathbf{d}\right]\right], c\left[\mathbf{q}\right]\right]\right]
\end{aligned}
$$

A.4 Conclusion

This Appendix gave a short introduction to tree languages. A more detailed introduction can be found in Drewes (2006). We discussed how trees can be generated by regular tree grammars and transformed by top-down tree transducers. Willow (Högberg, 2005) uses tree grammars and tree transducers to implement a rule-based system for algorithmic composition. XML documents, which are used in our music generation system, can often be considered as trees.

Appendix B

The Extensible Markup Language (XML)

B.1 Introduction

XML is a syntax for trees - Wilde and Glushko (2008)

XML plays an essential role in our music generation system since MusicXML files are used as input and produced as output. This Appendix provides the essentials of XML that are required to understand our approach and implementation. XML was designed to simplify the Standard Generalized Markup Language (SGML), a language used to describe custom designed markup languages. XML sacrifices customisability for ease of implementation when compared to SGML (CoverPages, 2002). For instance, SGML can imply some end tags from the Document Type Definition (DTD). Thus SGML does not require each start tag to have a matching end tag, as XML does (Sperberg-McQueen and Burnard, 1994). XML is used to structure, store and send data and uses a sequential notation for trees to do so (Maneth, PhD thesis). The fundamental benefit of XML is the fact that XML documents do not have to be parsed by using a context-free grammar, since XML documents are described with start and end tags (Maneth, PhD thesis). We might suspect that the reason for the excitement around XML is simply that practitioners are catching up with methods of abstraction and

representation via trees that are well-known in academia (Klarlund *et al.*, 2003). Labelled brackets are used to indicate an inner tree node. The trees we discussed in Appendix A are all ranked trees. XML is defined as unranked trees, implying that it does not specify the number of children each type of node has. This simplifies the specification process and makes XML more extensible.

In the next example we give an XML equivalent of the ranked tree in Figure A.1 in Appendix A.

Example B.1.1 (a sample XML document) **-** This example gives an XML document equivalent to the ranked tree described in Figure A.1 in Appendix A.

```
01 <!DOCTYPE measure SYSTEM "measure.dtd">
02 <measure>
03     <s>
04         <s>
05             <quarternote/>
06             <halfnote/>
07         </s>
08        <quarternote/>
09     </s>
10 </measure>
```

In XML it is more appropriate to describe a measure as below, since the rank of nodes are not specified.

```
01 <!DOCTYPE measure SYSTEM "measure2.dtd">
02 <measure>
03     <quarternote/>
04     <halfnote/>
05     <quarternote/>
06 </measure>
```

Each node or rather element, as it is referred to in XML, has an opening `<Node>` and closing `</Node>` tag (labelled bracket) which contains its chil-

dren. A terminal element or an element with empty content can be denoted as `<Node></Node>` or `<Node/>`.

Children of elements could be:

- Elements: For example `<Parent> <Child/> </Parent>`
- Text: For example `<Parent>Child</Parent>`
- Attributes: For example `<Parent Child = "childvalue"/>`

In the next section we discuss validation of XML using DTDs. This will be followed by an explanation of parsing using DOM. Finally, we will discuss transformation of XML documents by using Extensible Stylesheet Language (XSL), and the limitations of XSL.

B.2 Validation

A *well-formed* XML document is a document which conforms to correct XML syntax. If an XML document is not *well-formed*, it cannot be processed by XML extensions such as XSLT, which we discuss later. The syntax of a *valid* XML document is defined by a Document Type Definition (DTD) or XML Schema which are similar to Regular Tree Grammars (RTGs). DTDs are not used to generate XML documents, but rather to verify if XML documents are valid. In Example B.2.2 the `<!DOCTYPE>` tag specifies phrase to be the root element, while SYSTEM and the URL, "phrase.dtd", specify the location of the DTD. Example B.2.1 shows the DTD for the XML document in Example B.2.2.

Example B.2.1 (DTD) **-** phrase.dtd

```
01 <!ELEMENT phrase (measure*)>
```

The first line shows the declaration of the phrase element. It indicates that phrase elements should contain measure elements zero or more times.

```
02 <!ELEMENT measure (note*)>
03 <!ATTLIST measure
```

```
04     number CDATA #REQUIRED
05 >
```

The measure element should also contain note elements zero or more times, but after that measure's attributes are specified using `ATTLIST` (attribute list). It specifies an attribute with the name number and of type `CDATA`. It also adds a further specification, `#REQUIRED`. Character data is denoted by `CDATA` and `#REQUIRED` indicates that the specified attribute should be present in the element.

```
06 <!ENTITY %full-note "(chord?, (pitch | unpitched | rest))">
```

Next `ENTITY` is encountered. This acts like a placeholder to make other element declarations more concise and readable. It can also be used to specify values which will only be specified after parsing. The entity's name is full-note and specifies a possible chord element followed by a pitch, unpitched or rest element.

```
07 <!ELEMENT note (%full-note;, type?)>
```

In the previous line, note uses the full-note entity, in other words the element contains a possible chord element followed by a pitch, unpitched or rest element, followed by a possible type element.

```
08 <!ELEMENT pitch (step, alter?, octave)>
09 <!ELEMENT step (#PCDATA)>
10 <!ELEMENT alter (#PCDATA)>
11 <!ELEMENT octave (#PCDATA)>
12 <!ELEMENT chord EMPTY>
13 <!ELEMENT type (#PCDATA)>
```

In this example unpitched and rest is never specified in the DTD, but the XML document is still *well-formed*, since these elements are not present. The chord element is specified as `EMPTY` and will not contain any other elements or data. The chord element specifies whether a note is part of a chord by its presence. Finally `#PCDATA`, specifies parsed character data, which will be parsed for entities and markup.

Example B.2.2 (Valid XML) **-** A stripped down version of MusicXML using phrase.dtd

```
01 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
02 <!DOCTYPE phrase SYSTEM "phrase.dtd">
03 <phrase>
04     <measure number="1">
05         <note>
06             <pitch><step>C</step><octave>4</octave></pitch>
07             <type>half</type>
08         </note>
09         <note>
10             <pitch><step>E</step>
11               <alter>1</alter><octave>4</octave></pitch>
12             <type>half</type>
13         </note>
14     </measure>
15     <measure number="2">
16         <note>
17             <pitch><step>G</step><octave>4</octave></pitch>
18             <type>whole</type>
19         </note>
20     </measure>
21 </phrase>
```

B.3 Parsing

B.3.1 Document Object Model (DOM)

The Document Object Model (DOM) defines a standard way for accessing and manipulating XML documents (W3Schools, 2007). DOM defines almost everything in an XML document as a node in a tree. This includes the entire document, elements, text, attributes and comments. DOM defines functions that can for example retrieve a handle on an element by the tag name, or

retrieve handles on the parent or child nodes of a specified node. Each node also has certain properties including name, value and type. Using these handles, an XML document can be read or modified in ways similar to any tree data structure.

B.3.2 XPath

XML Path Language (XPath) uses path expressions to select nodes in an XML document. These path expressions are similar to those used in a computer file system.

We can for example select the instrument, in the measure below, with the XPath expression `/measure/instrument`.

Example B.3.1 - A measure element in an XML document

```
01 <measure>
02     <instrument>guitar</instrument>
03     <note pitch = "c">
04         <type value = "quarter"/>
05     </note>
06     <note pitch = "e">
07         <type value = "half"/>
08     </note>
09     <note pitch = "g">
10         <type value = "quarter"/>
11     </note>
12 </measure>
```

We can now retrieve the note `c` with the use of a predicate as follows: `/measure/note[@pitch='c']`. Predicates are contained in square brackets and are in this case used to find a node with a specific value. We can retrieve the quarter notes by using Axes. Axes give a node-set relative to the current node. We can for example find all the parent nodes relative to the type node with the value `quarter` by using the expression:

```
/measure/note/type[@value='quarter']/parent::*
```

XPath also has boolean and mathematical operators and over a hundred functions for strings, numbers, dates, etc. All this makes XPath a very useful tool for finding a node or nodes in an XML document.

B.4 Styling

The Extensible Stylesheet Language (XSL) can describe how XML documents should be displayed by transforming them to HTML, but XSL is even more powerful. In general, it can be used to transform an XML document specified by a schema to another XML document specified by another schema. Thus XSL can, for example, be used to transform an XML document to XHTML, plain text, or can even be used for simple XML queries. Formally XSL consists of three parts: XSL Transformations (XSLT), XML Path Language (XPath) and XSL Formatting Objects (XSL-FO). XSL Formatting Objects is described as an XML vocabulary for specifying formatting semantics (World Wide Web Consortium, 2007*b*). Informally there is a lot of confusion, since XSLT is often referred to as XSL and XPath is often considered as part of XSLT. XSL Transformations (XSLT) can add, remove, change and sort elements and attributes specified by XPath. XSLT transforms XML documents in a similar way in which tree transducers transform (ranked and unranked) trees.

The example stylesheet given in Example B.4.1 transforms the XML document in Example B.3.1 to the XML document in Example B.4.2. This stylesheet in Example B.4.1 can be logically divided into the following parts:

- Lines 3 - 14: Two copies of the measure element are made and given label attributes with a number value.
- Lines 14 - 19: The instrument element and the quarter note elements are copied.
- Lines 20 - 33: Finally, each half note is divided into two quarter notes.

All the tasks are of the form shown below:

```
<xsl:template match = "some xpath expression"/>
  "data transformation"
</xsl:template>
```

The template element uses an XPath expression to select the data (element, character data, etc.) which will be transformed. Inside the template element the actual transformation on the selected data is specified.

Example B.4.1 (XSLT document) **-** Used to style Example B.3.1

Lines 01 - 03: The header of the XSLT document

```
01 <?xml version = "1.0"?>
02 <xsl:stylesheet version = "1.0"
03 xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
```

Lines 04 - 13: We use the copy element to copy the selected measure element. We copy the element twice to make two measure elements and inside each we add a label attribute using the attribute element. Next we use the apply-templates element, which applies all the matching templates to the newly copied measures.

```
04   <xsl:template match = "measure">
05     <xsl:copy>
06       <xsl:attribute name = "label">1</xsl:attribute>
07       <xsl:apply-templates select = "node()|@*"/>
08     </xsl:copy>
09     <xsl:copy>
10       <xsl:attribute name = "label">2</xsl:attribute>
11       <xsl:apply-templates select = "node()|@*"/>
12     </xsl:copy>
13   </xsl:template>
```

Lines 14 - 19: A style has to be applied to instruments and notes other than halfs, otherwise they will not be present in the transformed XML document.

These elements do not have to change, and this is the reason for using the copy-of operation.

```
14   <xsl:template match = "instrument">
15     <xsl:copy-of select = "."/>
16   </xsl:template>
17   <xsl:template match = "note/type[@value = 'quarter']">
18     <xsl:copy-of select = "parent::*"/>
19   </xsl:template>
```

Lines 20 - 33: Finally note elements with a type value of a half are found and replaced by two quarter notes.

```
20   <xsl:template match = "note/type[@value = 'half']">
21     <xsl:element name = "note">
22       <xsl:attribute name = "pitch">
23         <xsl:value-of select = "parent::*/attribute::pitch"/>
24       </xsl:attribute>
25       <type value = "quarter"/>
26     </xsl:element>
27     <xsl:element name = "note">
28       <xsl:attribute name = "pitch">
29         <xsl:value-of select = "parent::*/attribute::pitch"/>
30       </xsl:attribute>
31       <type value = "quarter"/>
32     </xsl:element>
33   </xsl:template>
```

Line 34: The footer of the XSLT document.

```
34 </xsl:stylesheet>
```

Example B.4.2 (XML result) **-** The resulting XML document after styling the XML document in Example B.3.1 with the XSLT stylesheet in Example B.4.1

```
01 <?xml version="1.0"?>
02 <measure label="1">
03     <instrument>guitar</instrument>
04     <note pitch="c"> <type value="quarter"/> </note>
05     <note pitch="e"> <type value="quarter"/> </note>
06     <note pitch="e"> <type value="quarter"/> </note>
07     <note pitch="g"> <type value="quarter"/> </note>
08 </measure>
09 <measure label="2">
10     <instrument>guitar</instrument>
11     <note pitch="c"> <type value="quarter"/> </note>
12     <note pitch="e"> <type value="quarter"/> </note>
13     <note pitch="e"> <type value="quarter"/> </note>
14     <note pitch="g"> <type value="quarter"/> </note>
15 </measure>
```

B.5 Limitations of XSLT

Even with its extensive functionality, there are some transformations XSLT simply cannot perform (Pawson, 1999). Variables are limited to having an initial value at compile-time and cannot be changed at run-time, thus variables are immutable. This stateless execution is synonymous with functional programming, where loops and mutable variables are replaced by recursion and parameters (Harold, 2006). Restrictions that result from immutable variables include:

- XPath expressions cannot be constructed using variable references;
- An XSLT file can include the contents of other XSLT files by using the include operation, but the files to be included cannot be selected by a conditional statement;
- The key parameter of the sort function cannot be selected at run-time.

Some other limitations include:

- Attributes cannot be sorted, since it does not have an order as elements do;
- XSLT was made to generate trees and not tags, thus the generation of opening and closing tags in separate templates is impossible;
- The execution order of templates cannot be specified, since XSLT follows an event-based processing model for its template rules. This is also a side effect of XSLT being a functional programming language, where execution order is of low importance;

B.6 Conclusion

This Appendix gave a short overview of XML. The relationship between XML and trees was briefly discussed. The syntax and validation of XML using DTDs was discussed and explained with examples. Since the MusicXML input files that we use for music generation are not excessively large, the complete XML documents can be in memory and DOM can thus be used for parsing.

Appendix C

Probability Theory

C.1 Definitions

Definition C.1.1 (Conditional Probability) **-** (Ross, 2003) Assume $P(F) > 0$. The conditional probability of event E occurring, given event F has occurred, is denoted by $P(E|F)$ and $P(E|F) = \frac{P(E \cap F)}{P(F)}$.

Definition C.1.2 (Multiplication Rule) **-** (Ross, 2003) For events $E_1, \ldots, E_n$ with $P(E_1 E_2 E_3 \ldots E_n) > 0$, we have that

$$P(E_1 E_2 E_3 \ldots E_n) = P(E_1) P(E_2|E_1) P(E_3|E_1 E_2) \ldots P(E_n|E_1 \cap \ldots \cap E_{n-1}).$$

Definition C.1.3 (Bayes Formula) **-** (Ross, 2003) Assume that $F_1, F_2, \ldots, F_n$ are mutually exclusive events such that $\cup_{i=1}^{n} F_i = S$, where S is the sample space. Then

$$\begin{aligned} P(F_j|E) &= \frac{P(F_j \cap E)}{P(E)} \\ &= \frac{P(E|F_j)P(F_j)}{\sum_{i=1}^{n} P(E|F_i)P(F_i)}. \end{aligned}$$

For more detail on probability theory, consult Ross (2003).

List of Figures

Nomenclature

Abbreviations

ATTLIST	attribute list
bpm	beats per minute
CDATA	character data
DOM	Document Object Model
DPFA	Deterministic Probabilistic Finite Automata
DTD	Document Type Definition
FPFA	Probabilitic Finite Automata with a final state
HMM	Hidden Markov Model
Hz	Hertz
MC	Markov Chain
MIDI	Musical Instrument Digital Interface
PAC	Probably Approximately Correct
PCDATA	parsed character data
PDF	Portable Document Format
PFA	Probabilistic Finite Automata
PS	PostScript
PSA	Prediction Suffix Automata
PST	Prediction Suffix Tree
RTG	Regular Tree Grammar
TD	Tree Transducer
URL	Uniform Resource Locator
UTF	Unicode Transformation Format

wfsa	weighted finite state automata
wfst	weighted finite state transducer
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
XPATH	Extensible Markup Language Path Language
XSL	Extensible Stylesheet Language
XSL-FO	Extensible Stylesheet Language Formatting Objects
XSLT	Extensible Stylesheet Language Transformation

List of References

Adams, R. (2000). Musictheory.net. http://www.musictheory.net.

Allan, M. and Williams, C.K.I. (2005). Harmonising chorales by probabilistic inference. In: Saul, L.K., Weiss, Y. and Bottou, L. (eds.), *Advances in Neural Information Processing Systems*, vol. 17, pp. 25–32. MIT Press, Cambridge, MA.

Arkkra Enterprises (1992). Mup. http://www.arkkra.com/.

Chomsky, N. (1956). Three models for the description of language. *IEEE Transactions on Information Theory*, vol. 2, no. 3, pp. 113–124.

Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S. and Tommasi, M. (2007). Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata. Release October 12th, 2007.

Conklin, D. (2003). Music generation from statistical models. In: *Proceedings of the AISB 2003 Symposium on Artificial Intelligence and Creativity in the Arts and Sciences*, pp. 30–35.

Conklin, D. and Anagnostopoulou, C. (2001). Representation and discovery of multiple viewpoint patterns. In: *International Computer Music Conference*, pp. 479–485. Havana, Cuba.

Cope, D. (2004 September). A musical learning algorithm. *Computer Music*, vol. 28, no. 3, pp. 12–27.

CoverPages (2002). Standard generalized markup language (SGML). http://xml.coverpages.org/sgml.html.

CoverPages (2006). XML and music. `http://xml.coverpages.org/xmlMusic.html`.

Drewes, F. (1998). Treebag - a tree-based generator for objects of various types. Tech. Rep. 1, University of Bremen, Department of Mathematics and Informatics.

Drewes, F. (2006). *Grammatical Picture Generation – A Tree-Based Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

Droettboom, M. (2005). pyScore. `http://pyscore.sf.net/`.

Dubnov, S., Assayag, G., Lartillot, O. and Bejerano, G. (2003). Using machine-learning methods for musical style modeling. *Computer*, pp. 73–80.

Dupont, P., Denis, F. and Esposito, Y. (2005). Links between probabilistic automata and hidden markov models: probability distributions, learning models and induction algorithms. *Pattern recognition*, vol. 38, no. 9, pp. 1349–1371.

Ford, B. (2004). Parsing expression grammars: A recognition-based syntactic foundation. In: *Symposium on Principles of Programming Languages*, pp. 111–122. ACM Press.

Graehl, J. (2008). Carmel. `http://www.isi.edu/licensed-sw/carmel/`.

Harold, E.R. (2006). Tip: Loop with recursion in XSLT. `http://www.ibm.com/developerworks/xml/library/x-tiploop.html`.

Hart, P., Nilsson, N. and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107.

Högberg, J. (2005). Wind in the willows. Report UMINF 05.13, Umeå University.

Huang, X., Acero, A. and Hon, H. (2001). *Spoken language processing: A guide to theory, algorithm, and system development*. Prentice Hall PTR Upper Saddle River, NJ, USA.

Kilian, J. (2003). Guido noteserver. `www.noteserver.org`.

Klarlund, N., Schwentick, T. and Suciu, D. (2003). XML: model, schemas, types, logics, and queries. In: *In Logics for Emerging Applications of Databases*, pp. 1–41. Springer.

Levitin, D.J. (2006 August). *This is Your Brain on Music*. 1st edn. Dutton, a member of Penguin Group (USA) Inc.

Mackworth, A. (1977). Consistency in networks of relations. *Artificial intelligence*, vol. 8, no. 1, pp. 99–118.

Maneth, S. (PhD thesis). *Models of Tree Translation*. Ph.D. thesis, (year unknown).

McNab, R.J., Smith, L.A., Witten, I.H. and Henderson, C.L. (2000 April). Tune retrieval in the multimedia library. *Multimedia Tools and Applications*, vol. 10, no. 2-3, pp. 113 – 132.

Miranda, E.R. (2002). Mimetic development of intonation. In: *International Conference on Music and Artificial Intelligence*, pp. 107 – 118. Springer-Verlag.

Miranda, E.R. and Corino, G. (2003). Digital music: online tutorials on computer music. `http://x2.i-dat.org/~csem/UNESCO/`.

Miranda, E.R. and Todd, P.M. (2003). A-life and musical composition: A brief survey. In: *the IX Brazilian Symposium on Computer Music*. Campinas, Brazil.

Nienhuys, H. and Nieuwenhuizen, J. (2008). Lilypond, music notation for everyone. `http://lilypond.org`.

Noteflight.com (2009). Noteflight. `http://www.noteflight.com/`.

Ottman, R.W. (1983). *Elementary Harmony: Theory and Practice*. 3rd edn. Englewood Cliffs, N.J. : Prentice-Hall.

Pachet, F. (2003). The continuator: Musical interaction with style. *New Music Research*, vol. 32, no. 3, pp. 333–341.

Pachet, F. and Roy, P. (1994). Mixing constraints and objects: a case study in automatic harmonization. In: *Proceedings of TOOLS Europe*, vol. 95, pp. 119–126. Citeseer.

Pachet, F. and Roy, P. (2001). Musical harmonization with constraints: A survey. *Constraints*, vol. 6, no. 1, pp. 7–19.

Parsons, D. (1975 January). *The directory of tunes and musical themes*. Spencer Brown.

Pawson, D. (1999). Things XSLT can't do. `http://www.dpawson.co.uk/xsl/sect2/nono.html`.

PG Music Inc. (2007). Band in a box. `http://www.band-in-a-box.com/`.

Ponsford, D., Wiggins, G. and Mellish, C. (1999). Statistical learning of harmonic movement. *Journal of New Music Research*, vol. 28, no. 2, pp. 150–177.

Rabiner, L. (1990). A tutorial on hidden Markov models and selected applications in speech recognition. *Readings in speech recognition*, vol. 53, no. 3, pp. 267–296.

Randel, D.M. (2003 November). *The Harvard Dictionary of Music*. 4th edn. Belknap Press of Harvard University Press.

Recordare (2007). Recordare: Internet music publishing and software. `http://www.musicxml.org/`.

Ron, D., Singer, Y. and Tishby, N. (1996). The power of amnesia: Learning probabilistic automata with variable memory length. *Machine learning*, vol. 25, no. 2, pp. 117–149.

Ross, S. (2003). *A First Course in Probability*. International edn. Pearson Education.

Schwardt, L. (2007 December). *Efficient Mixed-Order Hidden Markov Model Inference*. Ph.D. thesis, University of Stellenbosch.

Seward, J. (2008). bzip2. `http://www.bzip.org/`.

Simon, I., Morris, D. and Basu, S. (2008 April). Mysong: Automatic accompaniment generation for vocal melodies. In: *CHI 2008*. Florence Italy.

Sinclair, S., Droettboom, M. and Fujinaga, I. (2006). Lilypond for pyScore: Approaching a universal translator for music notation. In: *International Conference on Music Information Retrieval*. Victoria, BC, Canada.

Sperberg-McQueen, C.M. and Burnard, L. (1994). A gentle introduction to SGML. `http://www.isgmlug.org/sgmlhelp/g-index.htm`.

Stewart, C., Bayes, C., Fuller, J., Ogbuji, U., Pawson, D. and Tennison, J. (2006). EXSLT - EXSL:NODE-SET. `http://www.exslt.org/exsl/functions/node-set/`.

Thollard, F., de la Higuera, C. and Carrasco, R. (2005). Probabilistic Finite-State Machines-Part I. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 7, pp. 1013–1025.

Todd, P.M. and Werner, G.M. (1999). Frankensteinian methods for evolutionary music composition. In: Griffith, N. and Todd, P.M. (eds.), *Musical networks: Parallel distributed perception and performance*, pp. 313–339. MIT Press.

Triviño-Rodriguez, J. and Morales-Bueno, R. (2001). Using multiattribute prediction suffix graphs to predict and generate music. *Computer Music Journal*, vol. 25, no. 3, pp. 62–79.

Valiant, L. (1984). A theory of the learnable. *Communications of the ACM*, vol. 27, no. 11, p. 1142.

W3Schools (2007). XML tutorial. `http://www.w3schools.com/xml/default.asp`.

Waltz, D. (1972). *Generating semantic descriptions from drawings of scenes with shadows*. Ph.D. thesis, PhD thesis, AI Lab, MIT, 1972.

Wilde, E. and Glushko, R.J. (2008 7). XML fever. *Communications of the ACM*, vol. 51, no. 7, pp. 40–46.

World Wide Web Consortium (2007*a*). Extensible markup language (XML). `http://www.w3.org/XML/`.

World Wide Web Consortium (2007*b*). The extensible stylesheet language family (XSL). `http://www.w3.org/Style/XSL/`.

Ziv, J. and Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536.