Stylistic Structures: An Initial Investigation of the Stochastic Generation of Tonal Music

CLAUDE ALAMKAN

WILLIAM P. BIRMINGHAM

ELECTRICAL ENGINEERING AND COMPUTER SCIENCE DEPARTMENT
THE UNIVERSITY OF MICHIGAN

MARY H. SIMONI

SCHOOL OF MUSIC
THE UNIVERSITY OF MICHIGAN

9 AUGUST, 1999

TECHNICAL REPORT

CSE-TR-395-99

ABSTRACT

In this paper, we explore the use of a Markov Decision Network to generate music. We show how the network is trained and used to generate compositions of approximately 10 minutes in duration that are representative of the style of the composer whose music was used to train the network.

1 INTRODUCTION

The study of musical style is vital to our understanding of the essence of music. For centuries, musicologists and music theorists have created and refined analytical methodologies that abstract characteristics of music. These analytical methodologies contribute to our understanding of musical style. Computer scientists can develop insight into difficult computational problems by developing models that may be used in the analysis of musical style, and testing these models through the generation of new compositions. For the purposes of this report, we constrain our discussion of musical style to repertoire from the Classical Period of Western Tonal Music (1750-1827).

Our objectives for the research described in this report are the following:

induce a composer's style,

generate new compositions that are consistent with a composer's style.

We represent musical style through *concurrencies*, which are sets of notes, described by pitch and duration that sound simultaneously. Concurrencies form states and transitions in a Markov Decision Network (MDN). The MDN is trained (a joint probability table is constructed) by analyzing selected compositions, called the training set. Each time a composition is analyzed, the MDN captures many interesting stylistic features. The MDN is used after training to generate compositions that have stylistic features induced from the training set. These newly generated compositions are quite often similar to the repertoire from which they were trained.

We believe that composer's style (at least during the period we are interested in) can be approximated as a stochastic process. Thus, the MDN is an attractive technique for capturing musical style, because it is relatively straightforward to train them, and pieces that appear to sound novel while maintain stylistic components expected of a composer are easy to create by unrolling the (trained) MDN. Another advantage of using our approach is that we do not need to use grammars or other predefined structures [4,5] before the system learns about musical styles.

The results from our initial experiments are encouraging. We can train a MDN to learn simple style representation. These networks have generated compositions of approximately 10 minutes in duration that are representative of the style of the composer whose music was used to train the system. Yet, we have found that certain factors sometimes foil the generation process by creating stylistically inconsistent musical passages.

2 CONCURRENCIES

Music of the Classical Period is comprised of notes. Each note has an onset described as a starting time, a pitch, and a duration. We gather the notes that share the same *starting* times into sets called concurrencies. Note in the discussion that follows, we distinguish between two different notions of time:

- Duration: this is the length of time that the note sounds, and is measured in Pulses Per Quarter Note (PPQN), described later in this section.

- Onset time: this is when the note first sounds in the piece. For example, in a note might have an onset time of the third beat of the fourth measure. We mostly ignore onset time when training our network, as described later in this section.

The onset time of a concurrency is defined as the starting time of all the notes that belong to the concurrency; hence, the name *concurrency*. This notion of time allows us to create the total ordering of notes as they occur during a piece defined for natural numbers. By definition, if A and B are two concurrencies, A is inferior to B if and only if the time of concurrency A is inferior to the time of concurrency B. We will write $A < B$ to state that concurrency A is inferior to concurrency B. We define *Time* to be a function from the set C of all the concurrencies to the set N of the natural numbers that returns the onset time of its argument. Thus,

$$A < B \Leftrightarrow Time(A) < Time(B).$$

To account for the duration of notes encoded as MIDI data, we use PPQN (Pulses Per Quarter Note). PPQN creates a simple correspondence between note durations and the set of natural numbers. We assign a quarter note equal to 192 PPQN. Table 1 describes selected note values and their corresponding duration expressed in PPQN.

Table 1: Note Value/PPQN correspondence

Note Value	PPQN
Whole note	768
Half note	384
Quarter note	192
8th note	96
16th note	48
32nd note	24
64th note	12

The example in Figure 1 shows the score of Ludwig van Beethoven's Moonlight Sonata, Op. 14, third movement.

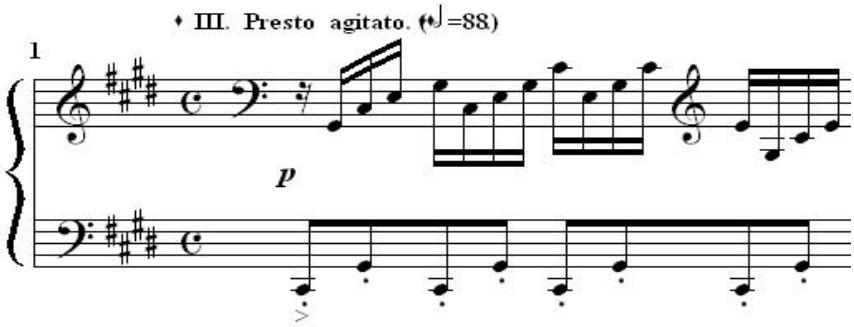

Figure 1: First measure of the third movement of the Moonlight Sonata by Beethoven

To create the concurrencies corresponding to this score, we first identify all the notes that have the same onset time. When forming a concurrency, we do not consider notes that sound simultaneously, such as when one note is struck and sustained during a subsequent note's onset. Thus, these notes will be in different concurrencies. That both notes contribute to the harmony is considered a random event by our system; yet, there will be a link in the MDN between these concurrencies that may, with some chance, produce a piece where the two

notes will sound together. Dividing a piece in this way introduces degrees of freedom to create "novel-" sounding pieces

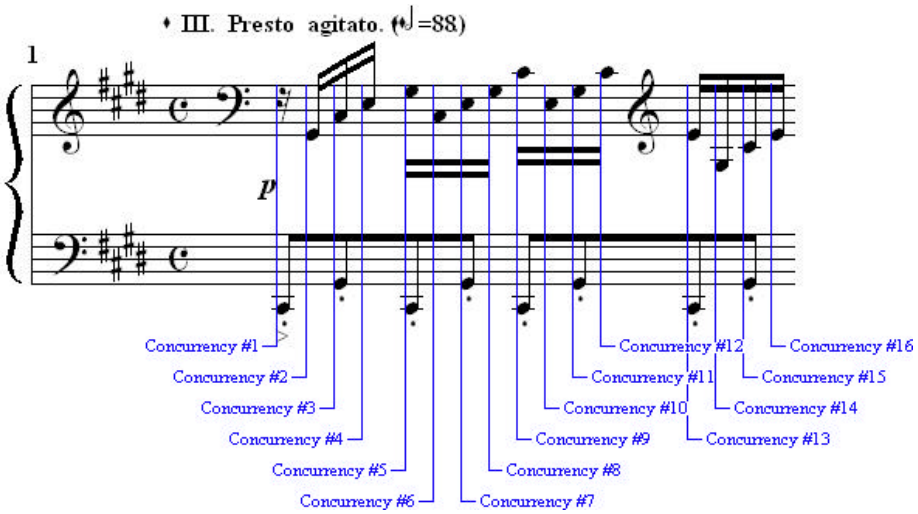

Figure 2: The first measure of the third movement of the Moonlight Sonata with the concurrency division

Figure 2 shows the partition of the notes into concurrencies. For a concurrency to be fully specified, we need the onset time, the pitch, and the duration of each note in the concurrency. Each note name is assigned a seven-bit key number according to the MIDI 1.0 specification, with Middle C (C4) having a key number of 60. The duration is expressed in PPQN.

For our Moonlight Sonata example, the set of concurrencies are given in Table 2.

Table 2: Concurrencies of the first measure of the third movement of the Moonlight Sonata

Concurrency number	Start time in cumulative PPQN	Key numbers and durations expressed as PPQN organized by pairs.
1	0	(Key number = 37, PPQN = 96)
2	48	(Key number = 44, PPQN = 48)
3	96	(Key number = 44, PPQN = 96), (Key number = 49, PPQN = 48)
4	144	(Key number = 52, PPQN = 48)
5	192	(Key number = 37, PPQN = 96), (Key number = 56, PPQN = 48)
6	240	(Key number = 49, PPQN = 48)
7	288	(Key number = 44, PPQN = 96), (Key number = 52, PPQN = 48)
8	336	(Key number = 56, PPQN = 48)
9	384	(Key number = 37, PPQN = 96), (Key number = 61, PPQN = 48)
10	432	(Key number = 52, PPQN = 48)
11	480	(Key number = 44, PPQN = 96), (Key number = 56, PPQN = 48)
12	528	(Key number = 61, PPQN = 48)
13	576	(Key number = 37, PPQN = 96), (Key number = 64, PPQN = 48)
14	624	(Key number = 56, PPQN = 48)
15	672	(Key number = 44, PPQN = 96), (Key number = 61, PPQN = 48)
16	720	(Key number = 64, PPQN = 48)

We can now consider a composition as a set of concurrencies. We call the set of all concurrencies *C*. If a composition has *p* concurrencies and $(c_n)_{n\in\{1,2,\ldots,p\}}$ are the *p* concurrencies of the music, then we denote the composition by the list $(c_1, c_2, \ldots, c_p)$, where the concurrencies $(c_n)_{n\in\{1,2,\ldots,p\}}$ are ordered by onset time. Hence, whenever we talk about a list of concurrencies $(c_1, c_2, \ldots, c_p)$, we implicitly have the following property about the list:

$$\forall n_1 \in \{1,2,...,p\} \quad \forall n_2 \in \{1,2,...,p\} \quad n_1 < n_2 \Rightarrow c_{n_1} < c_{n_2}$$

3 MUSIC GENERATION

All note-based music can be converted to a list $(c_1, c_2, ..., c_p)$ of concurrencies for some positive natural number p and some concurrencies $(c_n)_{n\in\{1,2,...,p\}}$. Conversely, any list $(c_1, c_2, ..., c_p)$ of concurrencies can be "played" as a piece of music. The relationship among the concurrencies capture the melody, harmony and rhythm of a musical work.

To generate a new composition, we select a positive, natural number p and a list $(c_1, c_2, ..., c_p)$ of concurrencies. If done randomly, such a list of concurrencies will have a corresponding musical performance, but such a composition may not exhibit a particular musical style. If we want to generate compositions that are congruent with a particular style, we must have a model for choosing the list of concurrencies comparable to those found by analysis of the compositions in that style.

3.1 PROBABILISTIC HYPOTHESES

With N^* the set of all positive natural number, and C the set of all concurrencies, let $(X_n)_{n\in N^*}$ be a set of random variables with domain $C \cup \{\otimes\}$, and let *Pr* be the joint probability for the sequence of random variables $(X_n)_{n\in N^*}$. The symbol $\otimes$ is added to the set of concurrencies to have a terminal symbol representing the end of a composition.

An experimental value for the set of random variables $(X_n)_{n\in N^*}$ will correspond to a composition. We still have to specify the joint probability distribution *Pr* that we want to use.

3.2 LOCALITY ASSUMPTION

The first assumption we make is that the relationships among concurrencies are local, and that the nature of a concurrency depends only on the past concurrencies.

That the probability only depends on the past concurrencies is given by:

$$\forall n \in N^* \quad \Pr(X_n | X_1, X_2, ..., X_{n-1}, X_{n+1}, ...) = \Pr(X_n | X_1, X_2, ..., X_{n-1})$$

Then, the locality of the relationship can be translated as the Markov chain property:

$$\exists d \in N \quad \forall n \in N^*$$

$$n > d \Rightarrow \Pr(X_n | X_1, X_2, ..., X_{n-1}) = \Pr(X_n | X_{n-d}, X_{n-d+1}, ..., X_{n-1})$$

This assumption reduces the set of possible probability-distribution functions, but still leaves us with a large set of admissible values for *Pr*.

Given the Markov property, we can now compute a new composition recursively:

- First, we choose a starting concurrency c_1 randomly, extract it from an existing composition, or choose it according to some rules.
- Second, from the last d concurrencies, we can use the probability distribution *Pr* to randomly select a new concurrency that we append to the composition.

The problem of generating a new composition in a particular musical style is reduced to finding a probability distribution function *Pr* that satisfies the Markov property derived from the analysis of compositions in that musical style.

We can consider *d* concurrencies without their onset times, which we call *timeless* concurrencies. Timeless concurrencies are important, because they allow us to insert a concurrency anywhere in a piece without being limited by where the concurrency appeared in a piece in the training set.

The set of possible pieces that can be composed from timeless concurrencies is very large. Consider, if we allow the pitches and durations of the notes in these *d* concurrencies to belong to finite sets, respectively, of cardinals *p* and *t*, then, the number of combinations of *d* concurrencies is: $(1+t)^{pd}$

> Proof: Each timeless concurrency may or may not have each of the *p* pitches possible from the 128 MIDI keys. For each pitch that forms a concurrency, a duration is chosen. For example, if a concurrency has both an eighth note and a whole note, one duration must be chosen. This leads to $\sum_{i=0}^{p}\binom{p}{i}t^i$ combinations for a single concurrency, where *i* is the number of pitches present in the concurrency. For *d* concurrencies, the total number is:
>
> $$\left(\sum_{i=0}^{p}\binom{p}{i}t^i\right)^d = \left((1+t)^p\right)^d = (1+t)^{pd}$$

Q.E.D.

If $d=1$ and we constrain the pitch to be chosen from one octave (thus normalizing all notes to one set of 12 pitch classes) and the duration to be an eighth, a quarter, a half or a whole note, the number of combinations exceeds 244 million.

3.3 MUSICAL ASSUMPTIONS

Notice that Concurrency 3 and Concurrency 5 in Figure 2 both contain pitch classes C# and G#. Thus, as far as pitch classes are concerned, these two concurrencies are equivalent. We can also see that both concurrencies have an 8th note and a 16th note. Hence, as far as duration is concerned, these two concurrencies are equivalent.

Since our goal is to generate compositions from a representation of style that we induce, simplifying the choice of the probability-distribution function *Pr* by using our knowledge about music is appropriate. To limit the choices for the probability-distribution function, we take into account -pitch class and duration equivalencies. To do so, we introduce the notion of a *style-descriptor function*, *S*, which is a function from the set of all concurrencies *C* to $N \times N$.

The style descriptor function assigns a pair of natural numbers (M, R) to each concurrency in *C*. *M* represents the pitch-class information of a concurrency, and R represents the duration of a concurrency. We call (M, R) a *style descriptor*.

To compute R, we do the following:

- First, R is set equal to the shortest duration of all the durations of the notes in the concurrency.

- Second, if R is less than or equal to 768, R is rounded up to one of the following values: 48 (16^{th} of note), 96 (8^{th} of note), 192 (quarter note), 384 (half note) or 768 (whole note). Otherwise, R is rounded down to 768.

The reason for R is that by recording the duration of the shortest note, we have an idea of the number of concurrencies per unit time that we must create when a concurrency is used to generate a new composition.

M can be a 12- or 24-bit integer depending on the amount of precision we want to retain from a concurrency. If M is a 12-bit integer, it has a bit for each pitch class. Since there are 12 classes (C, C#, D, D#, E, F, F#, G, G#, A, A# and B), M must have a minimum of 12 bits. If M is a 24-bit integer, it has two bits for each pitch class.

If M is a 12-bit integer, bit zero corresponds to the C pitch class. If a C is present in the concurrency, then bit zero will be set to one; otherwise, it is set to zero. Bit one corresponds to the C# pitch class. If a C# is present in the concurrency, then bit one will be set to one; otherwise, it is set to zero, and so forth for each pitch class. Table 3 gives the correspondence between the index of the bits in M and the set of pitches that can set the bit to one.

Table 3: M value of 12 bits

Bit index	Key numbers	Pitch Class
0	{0, 12, 24, 36, 48, 60, 72, 84, 96, 108, 120}	C
1	{1, 13, 25, 37, 49, 61, 73, 85, 97, 109, 121}	C#
2	{2, 14, 26, 38, 50, 62, 74, 86, 98, 110, 122}	D
3	{3, 15, 27, 39, 51, 63, 75, 87, 99, 111, 123}	D#
4	{4, 16, 28, 40, 52, 64, 76, 88, 100, 112, 124}	E
5	{5, 17, 29, 41, 53, 65, 77, 89, 101, 113, 125}	F
6	{6, 18, 30, 42, 54, 66, 78, 90, 102, 114, 126}	F#
7	{7, 19, 31, 43, 55, 67, 79, 91, 103, 115, 127}	G
8	{8, 20, 32, 44, 56, 68, 80, 92, 104, 116}	G#
9	{9, 21, 33, 45, 57, 69, 81, 93, 105, 117}	A
10	{10, 22, 34, 46, 58, 70, 82, 94, 106, 118}	A#
11	{11, 23, 35, 47, 59, 71, 83, 95, 107, 119}	B

If *M* is a 24-bit integer, bit zero corresponds to all the C pitch classes that are in an octave with an *even* octave number. Bit 12 corresponds to all the C pitch classes that are in an *odd* octave number. Bit one corresponds to the C# pitch classes that are in an octave with an even number, bit 13 to C# pitch classes in octaves with an odd number, etc.

It may seem unusual to separate notes based on the parity of the octave they are in, but this is an efficient way of maintaining a greater amount of information about the pitches. Essentially, a style descriptor function is a projection of the space of possible octaves space into a two-octave space. Thus, by using more bits, the space being projected into becomes larger. Table 4

gives the correspondence between the index of the bits in M and the set of key numbers that can set the bit to one.

Table 4: M value of 24 bits

Bit index	Set of pitches
0	{0, 24, 48, 72, 96, 120}
1	{1, 25, 49, 73, 97, 121}
2	{2, 26, 50, 74, 98, 122}
3	{3, 27, 51, 75, 99, 123}
4	{4, 28, 52, 76, 100, 124}
5	{5, 29, 53, 77, 101, 125}
6	{6, 30, 54, 78, 102, 126}
7	{7, 31, 55, 79, 103, 127}
8	{8, 32, 56, 80, 104}
9	{9, 33, 57, 81, 105}
10	{10, 34, 58, 82, 106}
11	{11, 35, 59, 83, 107}

Bit index	Set of pitches
12	{12, 36, 60, 84, 108}
13	{13, 37, 61, 85, 109}
14	{14, 38, 62, 86, 110}
15	{15, 39, 63, 87, 111}
16	{16, 40, 64, 88, 112}
17	{17, 41, 65, 89, 113}
18	{18, 42, 66, 90, 114}
19	{19, 43, 67, 91, 115}
20	{20, 44, 68, 92, 116}
21	{21, 45, 69, 93, 117}
22	{22, 46, 70, 94, 118}
23	{23, 47, 71, 95, 119}

The value for M can be easily computed using binary and modulo arithmetic. The algorithm for computing a 12-bit M is the following:

```
M ← 0
For all pitches p in the concurrency
 M ← M or (p modulo 12)
```

In the case of a 24-bit *M*, just replace 12 by 24. In Table 5, we show the concurrencies for the Moonlight Sonata example.

We refer to the size of M as being the precision of the style descriptor. Currently, we only consider two precisions: 12 or 24. We can imagine creating a precision of 36 or 48, but we suspect that the information gain from such high precisions does not benefit to the music-generation process.

The style-descriptor function defines an equivalence class on the set *C* of concurrencies.

Finally, we define $S(\otimes)$ to be the pair (0, 0).

Table 5: Style descriptors for the concurrencies of the first measure of the third movement of the Moonlight Sonata with a precision of 12

Concurrency number	Style Descriptor		Concurrency number	Style Descriptor
1	(2, 96)		9	(2, 48)
2	(256, 48)		10	(16, 48)
3	(258, 48)		11	(256, 48)
4	(16, 48)		12	(2, 48)
5	(258, 48)		13	(18, 48)
6	(2, 48)		14	(256, 48)
7	(272, 48)		15	(258, 48)
8	(256, 48)		16	(16, 48)

With this definition of the style descriptor function, the second assumption can be written as:

$$\exists d \in N \quad \forall n \in N^*$$
$$\left(n > d \Rightarrow \Pr\left(X_n \middle| X_{n-d}, X_{n-d+1}, \ldots, X_{n-1}\right) = \Pr\left(X_n \middle| S\left(X_{n-d}\right), S\left(X_{n-d+1}\right), \ldots, S\left(X_{n-1}\right)\right)\right) \wedge$$
$$\left(n \le d \Rightarrow \Pr\left(X_n \middle| X_1, X_2, \ldots, X_{n-1}\right) = \Pr\left(X_n \middle| S\left(X_1\right), S\left(X_2\right), \ldots, S\left(X_{n-1}\right)\right)\right)$$

This function states that the probability of having concurrency X_n when the last d concurrencies for $n > d$ or the $n-1$ first concurrencies for $n \le d$ are known only depends on the equivalence classes of these concurrencies.

4 COMPUTING THE PROBABILITY DISTRIBUTION FUNCTION

With the two assumptions in mind, we now explain the mechanism used to compute a family of probability-distribution functions. Each of these functions will satisfy the Markov chaining property (first assumption) and the equivalence-class property (second assumption).

4.1 GETTING THE PROBABILITY DISTRIBUTIONS BY MEASUREMENTS

In order to have reasonable probability-distribution functions, we use music composed by those recognized to be masters of a particular style. We assume that those composers have developed strong stylistic elements.

Consider a composition $(c_1, c_2, ..., c_p)$. For all sets of d consecutive concurrencies $(c_n, c_{n+1}, ..., c_{n+d-1})$, we can compute the d corresponding style descriptors $(S(c_n), S(c_{n+1}), ..., S(c_{n+d-1}))$. We call a d-tuple of style descriptors a *state*. Thus, from a composition $(c_1, c_2, ..., c_p)$, we will compute the $p-d+1$ states obtained from d consecutive concurrencies.

A transition is a structure with a state, a timeless concurrency, a *count* and a *space*. A count and a space are positive natural numbers. A composition, such as $(c_1, c_2, ..., c_p)$, can provide up to $p-d+1$ different transitions. We build a list, TL, of these transitions by scanning through $(c_1, c_2, ..., c_p)$ and applying the following algorithm:

For all the states of d consecutive concurrencies $(S(c_n), S(c_{n+1}), ..., S(c_{n+d-1}))$, we build a temporary transition *T'* by adding the timeless concurrency c'_{n+d} (or $\otimes$ if $n+d = p+1$), a *count* arbitrarily set to one, and a *space* equal to the difference of the time between c_{n+d} and c_{n+d-1}.If a transition between *T* in the list of transitions, *LT,* with the same state, the same timeless concurrency and the same space exists, then we increment the count of *T* by one. Otherwise, we add transition *T'* to the list of transitions *LT*.

The intuition for using a transition is the following: we want to have a count of the number of times that pieces moved from state $(S(c_n), S(c_{n+1}), ..., S(c_{n+d-1}))$ to concurrency c_{n+d}. If such a *transition* is encountered frequently, we want to record that *transition* as a good one, i.e., highly indicative of a composer's style. Hence, we keep the number of times the *transition* was encountered in a variable called *count*. Since we do not want to discriminate *transitions* by the time they occur in the pieces in the training set, we use the timeless version of concurrency c'_{n+d} of concurrency c_{n+d}.

Nonetheless, we need to record the time interval between c_{n+d} and c_{n+d-1}, because during the process of generating a piece of music, we want to know what duration to use for the timeless concurrency c_{n+d}. Hence, by knowing the concurrency c_{n+d} and the time interval between c_{n+d} and c_{n+d-1} (the *space*), we can recreate a concurrency by adding to the timeless concurrency c_{n+d-1} the time obtained by adding the time of c_{n+d} and the space. We do this via the *Concurrency* function that takes two arguments, a time and a timeless concurrency and returns a concurrency.

In our definition of a transition, we have:

a state that indicates the last *state* to which to append the new concurrency,

a timeless concurrency to provide us with the pitches and durations of the notes to add to the next concurrency,

a count that we use to compute the probability-distribution function based on the frequency of the occurrences, and

a space that allows us to compute an "onset" time for the next concurrency.

We will refer to the timeless concurrency of a transition as the *next concurrency*. For each transition *T* with a state $(S(c_n), S(c_{n+1}),..., S(c_{n+d-1}))$, a next concurrency c_{n+d}, a count *t* and a space *s*, we define the next state to be $(S(c_{n+1}), S(c_{n+2}),..., S(c_{n+d}))$. Note that we can compute the style descriptor of a timeless concurrency, since the style-descriptor function does not use the time of a concurrency as an argument.

From a list of transitions *LT*, we can generate a probability distribution function in the following way:

For each transition *T* with state $(S(c_1), S(c_2),..., S(c_d))$, next concurrency c_{d+1}, count *t* and space *s*, we compute the sum *v* of all the counts of the transitions in *LT* with state $(S(c_1), S(c_2),..., S(c_d))$ and we assert that:

$$\forall n \in N^* \quad n \geq d \Rightarrow$$

$$\Pr\left(X_n = S(Time(X_n) + s, c'_{d+1}) \middle| S(X_{n-d}) = S(c_1), S(X_{n-d+1}) = S(c_2),..., S(X_{n-1}) = S(c_d)\right) = \frac{t}{v}$$

For all the states $(S(c_1), S(c_2),..., S(c_d))$ that do not appear in the list *LT*, we assert that:

$$\forall n \in N^* \quad n \geq d \Rightarrow$$

$$\Pr\left(X_n = \otimes \middle| S(X_{n-d}) = S(c_1), S(X_{n-d+1}) = S(c_2),..., S(X_{n-1}) = S(c_d)\right) = 1$$

With these two rules, we can build a probability-distribution function from any list of transitions. Since we can get lists of transitions by scanning through pieces of music, we are now able to generate networks that we will use to make comparisons and generate new compositions.

5 GENERATING NEW PIECE OF MUSIC

In this section, we present the generation process.

5.1 STARTING THE GENERATION PROCESS

We described in Section 4, how we get the probability-distribution function for the Markov process. This distribution function allows us to randomly select the next state given a current state. In order to get the generation started, we need a start state.

Even though any state taken from any of the training compositions could be a starting state, we decided to only choose from the states that were starting points for the compositions in the training set. Hence, this gives us as many start states as pieces we used. To chose among these possible start states, we use a uniform distribution probability.

There is no penalty for choosing the start state as we did. There is even the advantage that we are guaranteed a reasonable start, which can lead to a more congruent piece than would a randomly chosen start state. The random choice of a state in the entire set, for example, may give the illusion that the piece begins in the "middle."

5.2 RUNNING THE GENERATION PROCESS

When used to generate music, the Markov transition model generates pieces that are "recognizably" in the style of a composer used as a training case. There are, however, problems. A problem that occurs in some pieces is a long, repetitive sequence. This is due to the presence of a path in the state graph with few outgoing edges (edges leading out of the path), and the few outgoing edges have low counts in comparison with the transition within the loop.

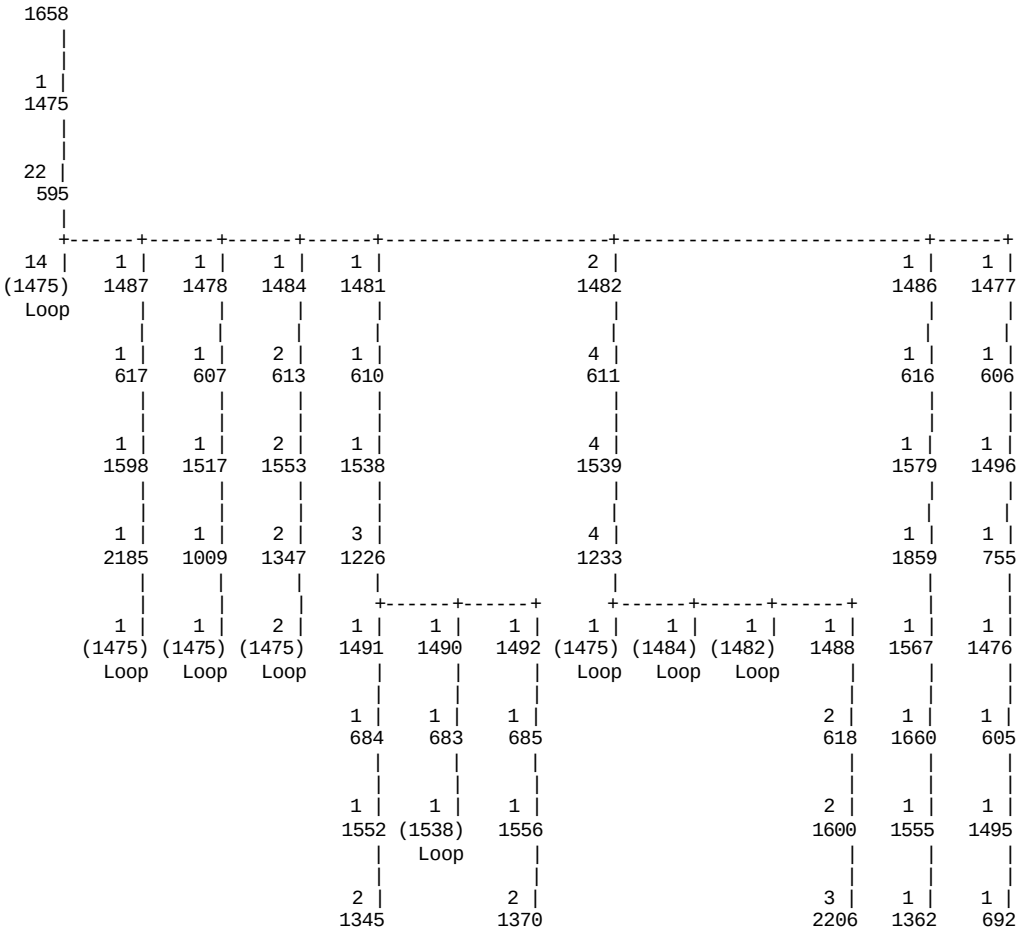

Figure 3: Portion of the state graph for the Moonlight Sonata illustrating the loop effect.[1]

For example, consider Figure 3, the transition process has a probability $\frac{14}{22}=\frac{7}{11}$ of going to state #1475 from state #1475 after only two transitions. After six transitions, the probability of having returned to or of being in state #1475 again is $\frac{14}{22}+\frac{3}{22}+\frac{2}{22}\times\frac{1}{4}=\frac{35}{44}$. Hence, the generation process, following this probability distribution can get to state #1475, and then repeat this state and all the states in the path returning to this state many times.

[1] The edges are oriented top to bottom. The nodes are represented by a state number. The numbers close to the edges and the state numbers are the transition counts. For example, state 595 has edges to eight distinct states: 1475, 1487, 1478, 1484, 1481, 1482, 1486 and 1477. State 1475 is in parenthesis, because it was displayed previously in the graph (above 595). An edge going to a parenthesized state number is to be understood as an edge going upward to a node already printed in the upper part of the graph.

5.3 LOOP-AVOIDANCE ALGORITHM

To avoid such a musically annoying affect, we used a mechanism that prohibits long loops. When we generate a piece, we keep track of the last transition used by the generation process in a list called the *history list*. A transition that has been used in the recent past has its probability lowered to zero. The history list is implemented as a first-in first-out queue with statically defined size *h*. Thus, during the generation we only keep references of the last *h* transitions used.

Consider that we are currently in state S and that there are three transitions from state S: X, Y and Z with respective counts of x, y, z. In the normal case, the respective probability of selecting X, Y or Z to provide the next concurrency is respectively $\frac{x}{x+y+z}$, $\frac{y}{x+y+z}$ and $\frac{z}{x+y+z}$ (1).

If one of these transitions has been used during the last d transitions, then its count is considered zero. For example, if X, has been used recently, but not Y and Z, the new probabilities for X, Y and Z will be 0, $\frac{y}{y+z}$ and $\frac{z}{y+z}$ (2), respectively.

A special case can occur when all the transitions have been used in the last d transitions. In such a case, the probabilities used are those for the normal case (1).

6 RESULTS

The MDN represented by the list of transitions is central to the music-generation process. Thus, we describe important features of the networks we have seen in the experiments.

The quality of the music generated depends on three parameters: *d*, which is the number of style descriptors (concurrencies) we use for the state, *h*, which is the size of the history list used to avoid looping, and the style descriptor precision.

For low values of *d* (less than or equal to four), the generated pieces have sections that are "chaotic" over short spans (relatively few concurrencies). These sections generally intervene between *smooth* portions. This combination of chaotic interspersed among smooth sections is not pleasing, even though within the sections the system does produce music of the style of the composer being imitated. These chaotic portions correspond to places in the state graph where the branching factor explodes and remains higher than the average branching factor for a significant number of transitions. The example from Figure 4 illustrates this phenomenon. The branching factor for state #595 is eight, while the average for the graph is approximately 1.133. Since the probability of getting back to state #595 is high ($\frac{35}{44}$ at least), the branching factor while the generation process remains in these loops in very high. If the states used during these loops originally came from different pieces of music, then numerous transitions from one to another result in a chaotic sounding composition. One way of solving this problem would be to increase *d* when the branching factor tends to be high for some number of transitions. This would have the effect of lowering the branching factor.

For high values of *d* (more or equal to eight), the branching factor is generally very low and there are long paths with no branching opportunities, which corresponds to a deterministic sequence of state as far as the generation process is concerned. These long, branch-free paths come from a single piece and generally (99%) correspond to a highly pronounced melody line. This has the consequence that the generation process seems to copy entire portions of the training compositions. From a point of view of originality, this is not a desired effect. One

way of solving this problem would be to decrease the *d* parameter when the branching factor tends to be too low for a sequence of concurrencies (25 or more transitions). This would have the effect of merging states and possibly increase the branching factor.

In most of the pieces generated, we used $d = 8$, since this value avoids chaotic sections at the cost of have bigger copied portion of music.

Even though going from a precision of 12 to 24 increases the state space by as much as 33% (32% for the Moonlight sonata) and decreases the average branching factor by 20% (from 1.41 to 1.13 for the Moonlight sonata), when listening to pieces generated with different precisions, the difference seem imperceptible.

The key and genre used in the training set are important factors that change the quality of the generated pieces. By genre, we refer to the form and instrumentation of the composition, e.g., whether it is a sonata, a piano concerto, a symphony, an opera, etc. Uniformity helps prevent chaotic-sounding sections.

If the pieces in the training cases are similar (same key, all piano sonatas, for example), then the generated pieces are more musically coherent than pieces trained with very different styles.

The generation process can be used for real-time music generation. It is a very fast process that typically takes less a second to create a 20-minute piece on a AMD-K6 200 MHz machine. Since it takes up to three seconds to analyze a 20-minute piece, the time of the entire process is not a concern.

7 RELATED WORK

Cope [4 and 5] focuses on a grammar-oriented relationship among musical structures. Cope has developed a grammar to parse pieces into elements he can recombine into new compositions. Our approach does not require *a priori* commitment to a particular grammar to parse a piece of music: our concurrencies are a minimal commitment to structure. Rather, we prefer to induce structure from the piece, rather than prescribe or proscribe it with a grammar.

Brooks, Hopkins, Neumann and Wright [3] have tried the probabilistic approach in a very simple manner. They were only considering very simple melody with no harmony (a single pitch at a time). We consider our work an extension of what they did by providing a more complete formalism for music generation and more elaborated generation capabilities than a single pitch at a time.

8 LIMITATIONS AND FUTURE WORK

The first assumption stated that the probability for choosing a concurrency only depended on the previous concurrencies. This assumption prevents the generation process from being guided. One could imagine wanting to end a piece with a certain cadence. The *past-only* assumption placed on the probabilistic model does not allow us such flexibility. Future work will add functionality to bias the probability distribution into accounting for some information about where the piece should go. This can be done by pruning from the state graph all branches and nodes that, if taken, will not allow the generation process to meet some criteria.

Dynamically adapted state size could prevent the appearance of chaotic structure or long "copied" passages (i.e., passages that are mostly direct copies of the training repertoire). Dynamically changing the size of the states means changing the entire state graph during the generation process. Hence, an optimized implementation of transition lists to change the state size without costly overhead will be implemented.

9 CONCLUSION AND SUMMARY

We have developed a representation for music generation by exhibiting a bijection between compositions and lists of concurrencies. With this concurrency structure, we have built a Markov Decision Network that, in spite of a relatively simple representation allows us to extract interesting attributes of musical style.

The MDN has a graph representation showing sparse connectivity and a cycles-oriented structure. The presence of nested short, medium or large cycles gives the graph a fractal-like structure. The graph exhibits in many places long paths with no branch opportunities, while, in some other places, exploding branching factors.

The MDN can be used to generate musical samples. With a relatively short state size, these samples do not make musical sense: they do not have an easily identifiable structure. With longer sizes, the compositions become more coherent, with recognizable musical structures.

10 REFERENCES:

Edward Aldwell, Carl Schachter, *Harmony and Voice Leading*, Second edition 1989

Margaret Boden, *Artificial Genius*, Discover (Oct. 1996), Vol. 17, pp. 104-107

F. P. Brooks, Jr., A. L. Hopkins, Jr., P. G. Neumann, and W. V. Wright, *An Experiment in Musical Composition* (1957) IRE Transactions on Electronic Computers

David Cope, *Computer Modeling of Musical Intelligence in EMI* (1992) Computer Music Journal, Vol. 16, No. 2, Summer 1992

David Cope, *The Search for Adaptations in Song Melodies*, Computer Music Journal, 21:1, pp. 58-67, Spring 1997

Charles Dodge, *Computer music: synthesis, composition, and performance*, New York: Schirmer Books ; London: Prentice Hall International, ©1997

John W Schaffer, *Knowledge-based programming for music research*; Madison, Wis. A-R Editions, ©1997

MIDI: musical instrument digital interface specification 1.0.

11 APPENDIX

Table A1: MIDI key number standard representation

Octave	Note numbers											
	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				